Static Source Code Analysis in Pattern Recognition, Performance Optimization and Software Maintainability

Dénes Bán

Department of Software Engineering
University of Szeged

Szeged, 2017

Supervisor:

Dr. Rudolf Ferenc

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
OF THE UNIVERSITY OF SZEGED

University of Szeged
Ph.D. School in Computer Science

"Failure is not falling down but refusing to get up."
— Chinese Proverb

Preface

When I was in kindergarten, I wanted to be an astronaut. I was so single-minded about it that I recited the planets in order to anyone who would listen and even confronted my teachers about my sign, which I claimed was most decidedly *not* a crescent roll, but a crescent moon. In my last year there, they actually indulged me and changed my sign. Ironically, this was around the time when I realized that it was most unlikely that I would ever become an astronaut.

No problem, Plan B: I am going to be a "computer programmer". I came to this conclusion at the ripe old age of six and believed in it despite anyone calling it just another phase. Fast forward 22 years and here I am, writing my doctoral thesis on the subject. I think I can safely say now that a Plan C will not be necessary.

None of this, of course, would be possible without the help of a lot of people, so acknowledgements are in order. First and foremost, I would like to thank my supervisor, Dr. Rudolf Ferenc for his guidance and insight. He showed me multiple times during our shared research that a bad result or a failed experiment is not the end of a project, only a point where we should change our strategy and carry on. I aim to apply this philosophy to other areas of my life as well. I would also like to thank Dr. Péter Hegedűs for "showing me the ropes", and Dr. István Siket for his ideas about the application of empirical distribution functions and for just always being available when I needed assistance with anything. My sincere thanks to Dr. Tibor Gyimóthy, the head of the Software Engineering Department, for providing me with many interesting research opportunities. Special thanks go to Gergely Ladányi for his invaluable advice and help regarding quality models and data analysis, to Róbert Sipka and Péter Molnár for their tireless work on the dynamic measurements, and to David Curley for his grammatical and stylistic comments. Also, many thanks to my other co-authors, namely Dr. Ákos Kiss, and Gábor Gyimesi, for their contributions.

Last, but certainly not least, I wish to express my gratitude to my amazing wife, Edina, for her constant support and encouragement.

Dénes Bán, 2017

Contents

List of Tables

List of Figures

Listings

To my wife,
who "put me through school"...

"Research is what I'm doing when I don't know what I'm doing."

— Wernher von Braun

1 Introduction

Software rules the world.

As true as this statement already was decades ago, it rings even truer now. When the proportion of the U.S. population using *any* kind of embedded system – including phones, cameras, watches, entertainment devices, etc. – went from its 2011 estimate of 65% to 95% by 2017 purely through phone ownership [43, 70]. When we are at a point where we can even *talk* about "smart cities", let alone build them [5]. When – according to Cisco [15] – connected devices have been outnumbering the population of Earth by a ratio of at least 1.5 since 2015.

This is just exacerbated by the host of embedded systems most people never even consider. An everyday routine contains household appliances like microwave ovens and refrigerators, heating or cooling our spaces, starting or stopping our vehicles; the list goes on. A modern life in this era involves countless hidden, invisible processors, along with the visible ones we have all got so used to. And we have not even mentioned critical applications like flight guidance, keeping patients alive and well in hospitals, or operating nuclear power plants.

All of those need software to run. And all that software needs to be written by someone. There is no question about either the growth of the software industry or the significant acceleration of said growth. The only question is whether or not we can actually keep up [31].

Having established the importance of software development, we can focus on arguably the two most important factors for its success: maintainability and performance. Software systems spend the majority of their lifetime in the maintenance phase, which, on average, can account for 60% of the total costs. Of course, maintaining a codebase does not only mean finding and fixing program faults, as enhancements and constantly changing requirements are far more common [32]. This means that for an efficient maintenance life cycle, a software product has to be quickly analysable, modifiable and testable – among other characteristics.

Just as critical is the issue of software performance and energy efficiency. A software product that fails to meet its performance objectives can be costly by causing schedule

delays, cost overruns, lost productivity, damaged customer relations, missed market windows, lost revenues, and a host of other difficulties [83]. Not to mention that the great amounts of energy consumed by large-scale computing and network systems, such as data centers and supercomputers, have been a major source of concern in a society increasingly reliant on information technology [67].

There are, however, many obstacles in the way of clean, maintainable and high-performance software. Time constraints of the ever-expanding market often make it appear infeasible to consider design best practices when these considerations would push back release times. Similarly, haste and an unwillingness to put in extra effort in advance is what seems to lead to antipatterns and code duplications, harming quality in the long run. Also, inadequate accessibility, tools, and developer support could significantly hinder the full utilization of today's performance optimization opportunities, such as specialized hardware platforms (e.g., GPGPU, DSP, or FPGA) and their corresponding compilers.

Our work aims to assist in both of these areas. Our goals are to:

- draw attention to the importance of the maintenance phase, and illustrate its assets and risks by highlighting the objective, tangible effect design patterns and antipatterns can have on software maintainability; and
- help developers more easily utilize modern accelerator hardware and increase performance by creating an easily usable and extendable methodology for building static platform selection models.

The structure of the thesis follows the same separation and – after a concise overview of the necessary background information in Chapter 2 – dedicates a part for each of these goals.

Part One deals with software maintainability. In Chapter 3, we discuss the connections between design patterns and maintainability. Next, in chapters 4 and 5 we explore the relationships between antipatterns and maintainability from the perspectives of Java and C++. Moreover, in Chapter 4 we also consider the association between antipatterns and program faults.

Part Two deals with software performance. In Chapter 6, we introduce our methodology for building prediction models that can select the best suited hardware platform for a given algorithm. In Chapter 7, we build on this methodology by also estimating how much improvement we can expect once we switch to this platform. Furthermore, in Chapter 8 we examine the maintainability changes caused by source code parallelization.

In Chapter 9, we conclude our discussion and outline possible directions for future work in this area. In addition, we present brief summaries of the thesis in English and Hungarian, which contain the concrete thesis points as well as the author's contributions and supporting publications in appendices A and B, respectively.

"Success depends upon previous preparation, and without such preparation there is sure to be failure."

— Confucius

2 Background

Before diving into the main results of the thesis, there are a few foundational topics worth reviewing as these form the basis of all future discussion. Static source code analysis is the technique providing us with source code metrics and other structural relationship information which, in turn, lead to design pattern and antipattern candidates. Assessing the impact these source code patterns have on maintainability also requires that we be able to objectively measure said maintainability. Such a measurement is similarly important when trying to compare the maintainability of two different versions of the same algorithm. This is where software quality models can help – some of which also warrant an abstract understanding of empirical cumulative distribution functions. Lastly, finding connections between datasets is the domain of statistics and machine learning.

2.1 Static Source Code Analysis

Static analysis is the (automated) examination of a program performed without execution. It can be based on already compiled binaries or, as in our case, the "raw" source code of the subject system. Its goal is generally calculating metrics, highlighting rule violations, generating intermediate representations, facilitating automatic refactorings, etc. The following paragraphs outline the specific steps we used to calculate source code metrics.

First, we converted the source code – through a language specific format and a linking stage – to the LIM model (**L**anguage **I**ndependent **M**odel), a part of the SourceMeter framework [26]. It represents the information obtained from the static analysis in a more abstract, graph-like format. It has different types of nodes that correspond to, e.g., classes, methods, and attributes, while different edges represent the connections among these.

From this data, our metric calculator tool can compute various kinds of source code metrics – e.g., logical lines of code or number of statements in the *size* category, comment lines of code or API documentation in the *documentation* category, and also different complexity, cohesion, and inheritance metrics. Note that the actual metrics

we extracted can change from use case to use case, and from language to language, so the exact list of metrics and their – possibly context dependent – definitions will always appear in their corresponding chapters.

As these values are not a part of the LIM model, the output is pinned to an even more abstract graph – fittingly named "graph". This format literally only has "Nodes" and "Edges", but nodes can have dynamically attached and dynamically typed attributes. Since the LIM model is strict – i.e., it can only have statically typed attributes defined in advance – the "graph" format is more suitable as a target for the results of metrics, pattern matches, etc.

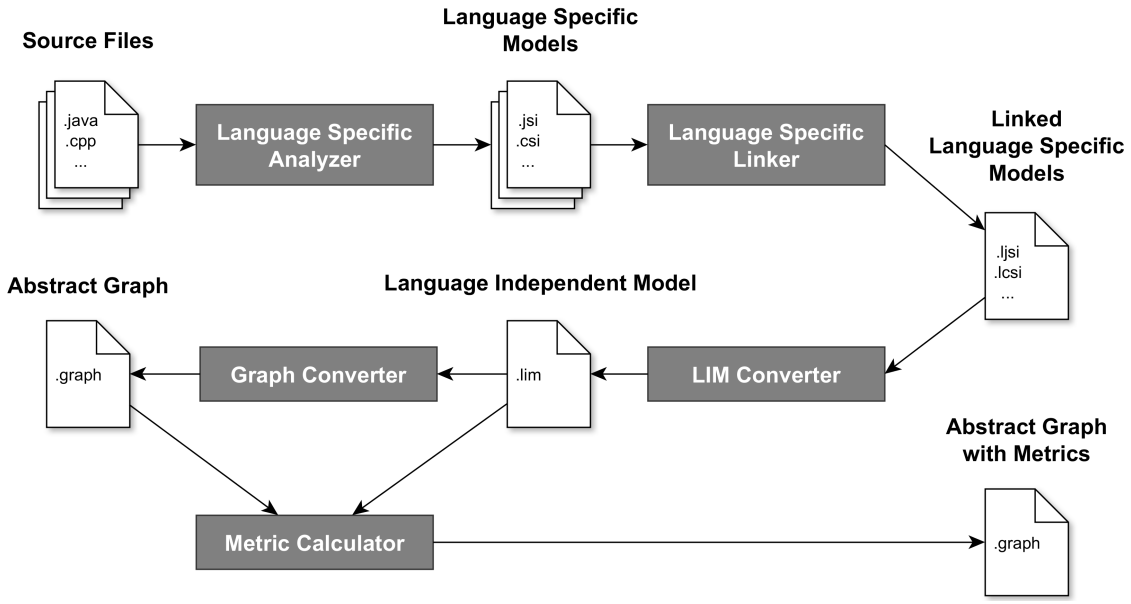

Figure 2.1. General static source code analysis workflow

2.2 Empirical Cumulative Distribution Functions

The metrics we calculate using static source code analysis may be viewed as complete from the perspective of the subject systems, but they cannot be related. They are, in a sense, absolute metric values and we have no way to tell, for instance, what a metric `A` with a value of x and another metric `B` with a value of y *mean* compared to each other. For this reason, it is desirable to normalize each metric value to the $[0, 1]$ interval using a benchmark of similar metrics and empirical cumulative distribution functions (or ECDFs) [89]. This method produces relative numeric values which indicate the ratio of how many of the available data points are smaller than a certain metric. These values are relative because they depend on the context they were evaluated in.

Let $(v_1, v_2, \ldots, v_n)$ be independent and identically distributed random variables with a common distribution function. The empirical distribution function is $\widehat{F}(x) = \frac{1}{n}\sum_{i=1}^{n} I(v_i \leq x)$, where I is the indicator function; namely, $I(v_i \leq x) = 1$ if $v_i \leq x$ and 0 otherwise. For example, the empirical distribution function of variables 1, 1, 1, 1, 2, 2, 4, 4, 5, 5, 6, 6, 8, 9, 13, and 15 can be seen in Figure 2.2.

ECDFs have many advantages in this domain:

- For each given metric, an $\widehat{F}(x)$ ECDF can be calculated objectively.

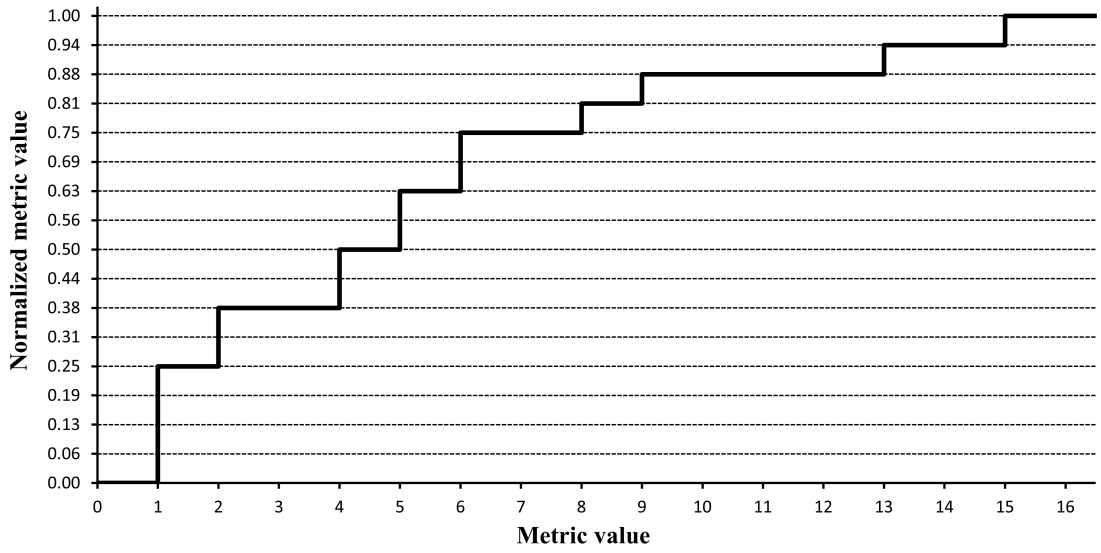

Figure 2.2. Empirical cumulative distribution function

- ECDFs transform metric values into the $[0, 1]$ interval.
- ECDFs can transform "unseen" values as well (i.e., values that are not in the sample).

Note that following the original definition, the normalized metrics will be greater for greater absolute inputs and smaller for smaller ones. If, however, an examined metric is "the smaller the better", we can invert this relation to facilitate a simpler mental model.

2.3 Quality Models

Trying to quantify complex software systems with a single maintainability index is not a new idea. Peercy [69] attempted to characterize subject systems using questionnaires as early as 1981. This, however, was a manual and mostly subjective effort.

Automatic source code analysis and metric extraction later led to metric-based maintainability models. One of the earlier – and more well-known – ones is the Maintainability Index metric (MI) published by Coleman et al. [21], which is a predefined formula that uses specific source code metrics to provide its result.

With the publication of the ISO/IEC 9126 framework [40], the expected structure and aspects of quality (and maintainability) models were more formally defined. It prescribes how to perform a weighted aggregation of objective, low-level source code characteristics so it can obtain increasingly abstract values, thereby providing a high-level overview of the whole system. This aggregation is simply visualized by a graph whose leaf nodes are the source code metrics and the most abstract characteristic (in our case, the maintainability) is the root node.

An example of this approach in practice is given by Antonellis et al. [4]. They use expert opinion-based graph weighting, achieved by using a technique called Analytical Hierarchical Processing. They conclude that this method helps domain experts to find connections between individual metrics and global maintainability as well as identify problematic areas.

Another example of the ISO/IEC 9126 framework in action is the probabilistic quality model published by Bakota et al. [9]. It also aggregates low-level metrics to arrive at the more abstract maintainability, but instead of concrete "goodness values", it makes use of "goodness functions", and the leaf nodes of the dependency graph are treated as random variables. These goodness functions are built by analyzing a benchmark containing over 100 subject systems.

ISO/IEC 25010 [41] – a successor to ISO/IEC 9126 – contains the definition of the main software characteristics and defines which subcharacteristics influence the given main quality characteristics, as shown in Figure 2.3.

Figure 2.3. ISO/IEC 25010 software quality characteristics

As this thesis is concerned with maintainability, we present only its definition and the definitions of its subcharacteristics as they are given in the standard:

- *Maintainability*: This characteristic represents the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements.
- *Analysability*: Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
- *Modifiability*: Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
- *Modularity*: Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
- *Reusability*: Degree to which an asset can be used in more than one system, or in building other assets.

- *Testability*: Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

Additionally, we would like to mention Stability and Changeability. They were subcharacteristics in the ISO/IEC 9126 document which is still the basis of our earlier experiments in Chapter 3. They also appear as manually added intermediate nodes in a later quality model – already based on ISO/IEC 25010 – in Chapter 4.

Most of the software quality models used in this thesis build on the quality characteristics defined by these standards. The computation of the high-level quality characteristics is based on a directed acyclic graph whose nodes correspond to quality properties that can either be internal (low-level) or external (high-level). Internal quality properties characterize the software product from an internal (developer) view and are usually estimated by using source code metrics. External quality properties characterize the software product from an external (end user) view and are usually aggregated somehow by using internal and other external quality properties. The edges of the graph represent dependencies between an internal and an external, or two external properties. The aim is to evaluate all the external quality properties by performing an aggregation along the edges of the graph, called Attribute Dependency Graph (ADG).

The structure of the graph or the aggregation weights of the edges can all be customized for different scenarios. Even the base metrics can be changed, or transformed before the aggregation – e.g., by ECDFs, or the above-mentioned probabilistic "goodness functions".

2.4 Statistical Analysis and Machine Learning

2.4.1 Correlations

A commonly used test to find a (linear) connection between two sets of data is Pearson's correlation coefficient. It shows a measure of how well the two sets are related, where 1 means linear correspondence, -1 is an exact inverse relationship, and 0 expresses no linear connection whatsoever.

If we do not expect the relationship between the inspected values to be linear, – only monotone – we may use the Spearman correlation coefficient instead. Spearman correlation is, in fact, a "traditional" Pearson correlation, only it is carried out on the ordered ranks of the values, not the values themselves. This shows how much the two datasets "move together." The extent of this matching movement is somewhat masked by the ranking, – which can be viewed as a kind of data loss – but it is applicable, e.g., when we are more interested in the *existence* of a relationship than its type.

For pairwise correlations on multiple variables, we used IBM SPSS [22].

2.4.2 Regression Techniques

Regression analysis is a statistical process for estimating the relationships among variables. It seeks to predict how the typical value of the dependent variable changes when any one of the independent variables change. It achieves this by providing an estimate for the dependent variable from a continuous interval. The typical performance measures of these estimations are:

- Pearson's correlation coefficient – how well the predicted values follow the tendency of the real value of the dependent variable.
- Mean absolute error – a quantity used to measure how close predictions are to the eventual outcomes.

The following regression types are used in various parts of this thesis: Linear regression [44], Multilayer perceptron [14], Reduced error pruning tree [24], M5P tree [93], and Sequential minimal optimization regression [82].

2.4.3 Classification Techniques

Classification is the process of identifying to which of a set of categories a new observation belongs. This identification is done based on a training set of data containing observations whose category membership is already known. The result is the category of the instances (instead of a continuous number, as with regression). Note that a good regression model (with a high correlation and a low mean absolute error) is much harder to build than a "simple" classification with a predetermined number of class labels.

The typical performance measures of these categorizations are:

- Correctly classified instances – the ratio of the correctly categorized instances.
- Confusion matrix – a matrix where each column represents the instances in a predicted class, while each row represents the instances in an actual class. The name stems from the fact that it makes it easy to see if the system is confusing two classes.

The following classification algorithms are used in various parts of this thesis: J48 decision tree [77], Naive Bayes classifier [42], Logistic regression [52], and Sequential minimal optimization function [73].

2.4.4 Validation of the Models

The models mentioned in future chapters – unless otherwise stated – were validated with a 10-fold cross-validation [6]. In a 10-fold cross-validation process, the original dataset is randomly partitioned into 10 subsamples, possibly equal in size. Out of the 10 subsamples, 1 subsample is retained as the validation data for testing the model, and the other 9 subsamples are used as training data. The cross-validation process is then repeated 10 times (the number of folds), with each of the 10 subsamples used exactly once as the validation data. The results from the folds are then averaged to produce a single estimate.

The machine learning experiments were all performed with Weka [34].

Part I

Source Code Patterns

"Beauty is the ultimate defence against complexity."
— David Gelernter

3

The Connection between Design Patterns and Maintainability

3.1 Overview

Since their introduction by Gamma et al. [30], there has been a growing interest in the use of design patterns. Object-Oriented (OO) design patterns represent well-known solutions to common design problems in a given context. The common belief is that applying design patterns results in a better OO design, therefore they improve software quality as well [30, 90].

However, there is little empirical evidence that design patterns really improve code quality. Moreover, some studies suggest that the use of design patterns does not necessarily result in good design [62, 95]. The problem with empirical validation is that it is very hard to assess the effect of design patterns on high-level quality characteristics, e.g., maintainability, reusability, understandability, etc. There are some approaches that manually evaluate the impact of certain design patterns on different quality attributes [46].

We also try to reveal the connection between design patterns and software quality but we focus on the maintainability of the source code. As many concrete maintainability models exist, (e.g., [9, 35, 13]) we could choose a more direct approach for the empirical evaluation. To get an absolute measure for the maintainability of a system, we used our probabilistic quality model [9]. Our subject system was JHotDraw 7, a Java GUI framework for technical and structured graphics[1]. Its design relies heavily on some well-known design patterns. Instead of using different design pattern mining tools, we parsed the *javadoc* entries of the system directly to get all the applied design patterns. We analyzed more than 300 revisions of JHotDraw, calculated the maintainability values and mined the design pattern instances. We gathered this empirical data with the following research questions in mind:

[1] `http://www.jhotdraw.org/`

Research Question 1: *Is there a traceable impact of the application of design patterns on software maintainability?*

Research Question 2: *What kind of relation exists between the density of design patterns and the maintainability of the software?*

We achieved some promising results showing that applying design patterns improves the different quality attributes according to our maintainability model. In addition, the ratio of the source code lines taking part in some design patterns in the system has a very high correlation with the overall maintainability in the case of JHotDraw. However, these results are only a small step towards the empirical analysis of design patterns and software quality.

The rest of this chapter is structured as follows. In Section 3.2, we highlight the related work, then in Section 3.3 we present our approach for analyzing the relationship between design patterns and maintainability. Section 3.4 summarizes the empirical results we achieved. Next, Section 3.5 lists the possible threats to the validity of our work. Lastly, we draw our conclusions in Section 3.6.

3.2 Related Work

Although the concept of utilizing design patterns in order to create better quality software is fairly widespread, there is relatively little research that would objectively demonstrate that their usage is indeed beneficial.

Since design patterns and software metrics are both geared towards the same goal, – improving quality – Huston [38] attempted to prove their correlation by representing the system's classes in connection matrices and defining algorithms for applying patterns and evaluating metrics. This approach shows promising results but it is purely theoretical.

In an empirical study, – replicated twice, in 2004 [92] and in 2011 [50] – Prechelt et al. [75] gave groups identical maintenance tasks to perform on two different versions – with and without design patterns – of four programs. Here, the impact on maintainability was measured by completion time and correctness, while we use objective quality metrics and analyze a more complex software system.

In another case study, Vokáč [91] measured the defect frequency of pattern classes versus other classes in an industrial C++ source code for three years and concluded that some patterns – Singleton, Observer – tend to indicate more complex parts than others, e.g., Factory. However, the pattern mining method could have introduced false positives or true negatives, and the defects were also based on subjective reports. In contrast, we rely on the official pattern documentation of the source code and the quality model published in [9].

Khomh and Guéhéneuc [46] used questionnaires to collect the opinions of 20 experts on how each design pattern helps or hinders them during maintenance. They found evidence that design patterns should be used with caution during development because they may actually impede maintenance and evolution. Another experiment, conducted by Ng et al. [65], decomposed maintenance tasks to subtasks and examined the frequency of their use according to the deployed design patterns and whether these patterns were utilized during the change. They statistically concluded that performing whichever task while taking existing patterns into consideration yields less faulty code.

Trying to evaluate the effectiveness of patterns in software evolution, Hsueh et

al. [37] defined both their context and their anticipated changes and then later checked whether they met the expectations. Their conclusion was that although design patterns can be misused, they are effective to some degree in either short or long term maintenance. Aversano et al. [8] also investigated pattern evolution by tracking their modifications and how many other, possibly unrelated modifications they caused. In this study, we do not use questionnaires or evaluate design patterns manually, but rather measure their impact on maintainability directly. Moreover, we focus on their impact on the maintainability of the system as a whole, not only on the evolution of the code implementing design patterns.

In more loosely related research, Prechelt et al. [76] showed that explicit pattern documentation in itself can further help maintenance, and Brito e Abreu and Melo [17] demonstrated the positive effects of object oriented design in general.

3.3 Methodology

To analyze the relationship between design patterns and maintainability, we calculated the following measures for every revision in JHotDraw:

- M_r – an absolute measure of maintainability for the revision r of the system (computed by our probabilistic quality model [9]).
- $TLLOC$ – the total number of logical lines of code in the system (computed by the SourceMeter tool [26]).
- $TNCL$ – the total number of classes in the system.
- PIn_r – the number of pattern instances in revision r of the system.
- PCl_r – the number of classes that play a role in any pattern instances within revision r of the system.
- PLn_r – the total number of logical lines of the classes that play a role in any pattern instances within revision r of the system.
- $PDens_r$ – the pattern line density of the system, defined as the ratio $\frac{PLn_r}{TLLOC}$.

For assessing the maintainability of JHotDraw, we used the node aggregation method mentioned in Section 2.3 with the particular ADG presented in [9].

As for the design pattern related information, instead of applying one of the more general purpose miner tools, (e.g., [23, 94]) we used a more direct approach for extracting pattern instances from different JHotDraw versions. Since every design pattern instance is documented in JHotDraw 7, we could easily build a text parser application to collect all the patterns. This method guarantees that no false positive instances are included and no true negative instances are left out from the empirical analysis. A sample of the design pattern *javadoc* documentation can be seen in Listing 3.1.

The text parser processed the two types of pattern comments that appeared in the source code – the listing below displaying one of them. Then, – using regular expressions – it obtained the names of the patterns and a list of the participants. This list contained the names of both the roles and the classes that fit them. Next, all fully qualified name references were trimmed, – e.g., `foo.Bar` became `Bar` – the lists

Listing 3.1. Design pattern *javadoc* comment

```
/**
...
* <b>Design Patterns</b>
*
* <p><em>Strategy</em><br>
* The different behavior states of the selection tool are implemented by
* trackers. Context: {@link SelectionTool}; State: {@link DragTracker},
* {@link HandleTracker}, {@link SelectAreaTracker}.
...
**/
```

were alphabetically ordered, converted to a unique string and added to a set in order to avoid pattern instance duplication even if a pattern was documented in more than one of its participants' codes. Lastly, we ran the parser on all relevant revisions of JHotDraw 7 to track the changes.

3.4 Results

We analyzed all 779 revisions of the JHotDraw 7 subversion branch[2] and calculated the measures introduced in Section 3.3. The documentation of design patterns is introduced in revision 522, therefore the empirical evaluation has been performed on 258 revisions (between revisions 522 and 779). Some basic properties of the starting and ending revisions of the JHotDraw system can be seen in Table 3.1.

Revision	Lines of code	№ of packages	№ of classes	№ of methods	PIn_r	$\frac{PCl_r}{TNCL}$
522	72,472	54	630	6117	45	11.58%
779	81,686	70	685	6573	54	13.28%

Table 3.1. Basic properties of the JHotDraw 7 system

To be able to answer our **first research question**, we analyzed those particular revisions where the number of design pattern instances had changed. After filtering out the changes that did not introduce or remove real pattern instances, (e.g., comments were added to an already existing pattern instance) five revisions remained. We also confirmed that these change sets did not contain a lot of miscellaneous source code unrelated to patterns, as it is an important prerequisite for being able to clearly distinguish the effect of design pattern changes on maintainability. In all five cases, more than 90% of the code changes were related to the pattern implementations. The tendency of different quality attributes in these revisions can be seen in Table 3.2.

In four out of five cases, there was growth in the pattern instance numbers. In all of those four cases, every ISO/IEC 9126 quality characteristic (including the maintainability) increased compared to the previous revision. This is true even for revision 716, where the pattern line ratio decreased despite the addition of a design pattern. In the case of revision 609, a *Framework* pattern had been removed but the quality

[2]https://jhotdraw.svn.sourceforge.net/svnroot/jhotdraw/trunk/jhotdraw7/

Revision (r)	Pattern Changes	Pattern Line Density ($PDens_r$)	Maintainability (M_r)	Testability	Analysability	Stability	Changeability
531	+3	↗	↗	↗	↗	↗	↗
574	+1	↗	↗	↗	↗	↗	↗
609	−1	↘	—	—	—	—	—
716	+1	↘	↗	↗	↗	↗	↗
758	+1	↗	↗	↗	↗	↗	↗

Table 3.2. Quality attribute tendencies in the case of design pattern changes

characteristics remained unchanged. This is not so surprising since this pattern (which is not part of the GoF patterns [30]) consists of a simple interface alone. Therefore, its removal did not have any effect on the low-level source code metrics our maintainability model is based on.

As a previous work from Bakota et al. [10] shows, a system's maintainability does not improve during development without applying explicit refactorings. Therefore, the application of design patterns can be seen as applying refactorings to the source code. These results support the hypothesis that design patterns do have a traceable impact on maintainability. In addition, our empirical analysis on JHotDraw indicates that this impact is positive.

To shed light on the relationship between design pattern density and maintainability for our **second research question**, we performed a correlation analysis on pattern line density ($PDens_r$) and maintainability (M_r). We chose pattern line density instead of pattern instance or pattern class density because it is the finest grained measure showing the amount of source code related to any pattern instances. Figure 3.1 depicts the tendencies of pattern line density and maintainability.

It is clearly visible that the two curves have a similar shape, meaning that they move closely together. The Pearson correlation analysis of the entire dataset (from revision 522 to 779) shows the same result, the pattern line density and maintainability have a correlation coefficient of **0.89**. This result may indicate that there is a strong relation between the rate of design patterns in the source code and the maintainability. However, this is still an assumption and we cannot generalize these results without performing a large number of additional empirical evaluations.

3.5 Threats to Validity

Like most works, our approach also has some threats to its validity. First of all, when dealing with design patterns, the accuracy of mining is always in question. As there are no provably perfect pattern miner tools, we chose our subject system to be a special one, having all design pattern instances thoroughly documented by its authors. This way we can be sure that all (intentionally placed) design patterns are recognized and no false positive instances are introduced. Of course, it is still possible that some pattern

Figure 3.1. The tendencies of pattern line density and maintainability

comments are missing or our text parser introduces false instances. We reduced this effect by manually inspecting the results of our text parser as well as the source code of JHotDraw.

Another threat to validity is using our chosen, previously published quality model for calculating maintainability values. Although it has gone through some empirical validation in a previous work, we cannot state that the maintainability model we used is fully validated. Moreover, as the ISO/IEC 9126 standard does not define the low-level metrics, the results may vary depending on the quality model's settings (e.g., the chosen metrics and weights given by professionals). These factors are all possible risks, but our first results and continuous empirical validation of the maintainability model demonstrate its applicability and usefulness.

Lastly, the small number of design pattern changes and the fact that less than 300 revisions of a single system have been evaluated threatens the generalizability of our results. It might also be possible that the explored relationship between design patterns and maintainability is just a byproduct of other factors. Our analysis is only a first step towards the empirical validation of this relation. Nonetheless, these first results are already valuable and support the common belief that design patterns do have a positive impact on maintainability.

3.6 Summary

In this chapter, we presented an empirical study of the connection between design patterns and software maintainability. We analyzed nearly 300 revisions of JHotDraw 7, calculated the maintainability values with a probabilistic quality model, and mined the design pattern instances by parsing the comments in its source code. Examining the maintainability values where changes occurred in the number of pattern instances, and by correlation analysis of the design pattern density and maintainability values, we were able to draw some conclusions.

Every ISO/IEC 9126 quality characteristic (including maintainability) increased along with the number of pattern instances. The amount of other, unrelated source code elements involved in these changes were negligible, which indicates that the quality attributes increased due to the introduced patterns. Hence, we could observe a traceable positive impact of design patterns on the maintainability of the subject system.

Another interesting result is that the pattern line density and maintainability values have a very similar tendency. The Pearson correlation analysis of the datasets showed that there is a strong relation between the rate of design patterns in the source code and its maintainability. These observations reinforce the common assumption that using design patterns improves the maintainability of the source code. However, these results should be handled with caution. We analyzed only one system and a relatively few number of pattern instance changes. We are far from drawing general conclusions based on these findings; our work should be considered as a first step towards the empirical validation of the relationship between design patterns and software maintainability.

"Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration."

— Stan Kelly-Bootle

4

The Connection between Antipatterns and Maintainability in Java

4.1 Overview

Antipatterns can be most simply thought of as the opposites of the more well-known design patterns [30]. While design patterns represent "best practice" solutions to common design problems in a given context, antipatterns describe a commonly occurring solution to a problem that generates decidedly negative consequences [19]. Also an important distinction is that antipatterns have a refactoring solution to the represented problem, which preserves the behavior of the code, but improves some of its internal qualities [28]. The widespread belief is that the more antipatterns a software contains, the worse its quality is.

Some research even suggests that antipatterns are symptoms of more abstract design flaws [88, 54]. However, there is little empirical evidence that antipatterns really decrease code quality.

We seek to reveal the effect of antipatterns by investigating their impact on maintainability and their connection to bugs. For the purpose of quality assessment, we again chose a probabilistic quality model [9], which ultimately produces one number per system describing how "good" that system is. The antipattern-related information came from our own, structural analysis based extractor tool using source code metrics computed by the SourceMeter reverse engineering framework [26]. We compiled the data described above for a total of 228 open-source Java systems, 34 of which had corresponding class level bug numbers from the open-access PROMISE [63] database. With all this information, we try to answer the following questions:

Research Question 1: *What kind of relation exists between antipatterns and the number of known bugs?*

Research Question 2: *What kind of relation exists between antipatterns and the maintainability of the software?*

Research Question 3: *Can antipatterns be used to predict future software faults?*

We obtained some promising results showing that antipatterns indeed negatively correlate with maintainability, according to our quality model. Moreover, antipatterns correlate positively with the number of known bugs, and also seem to be good attributes for bug prediction. However, these results are only a small step towards the empirical validation of this subject.

The rest of the chapter is structured as follows. In Section 4.2, we highlight the related work. Then, in Section 4.3 we present our approach for extracting antipatterns and analyzing their relationship with bugs and maintainability. Next, Section 4.4 summarizes the results we achieved, while Section 4.5 lists the possible threats to the validity of our work. Lastly, we conclude the chapter in Section 4.6.

4.2 Related Work

Antipattern Detection The most closely related research to our current work was done by Marinescu. In his publication in 2001 [60], he emphasized that the search for given types of flaws should be systematic, repeatable, scalable, and language-independent. First, he defined a unified format for describing antipatterns, and then a methodology for evaluating those. He showed this method in action using the GodClass and DataClass antipatterns and argued that it could be similarly done for any other pattern. To automate this process, he used his own TableGen software to analyze C++ source code, – analogous to the SourceMeter tool in our case – save its output to a database, and extract information using standard queries.

In one of his works from 2004 [61], he was more concerned with automation and made declaring new antipatterns easier with "detection strategies." In these, one can define different filters for source metrics – limit or interval, absolute or relative – even with statistical methods that set the appropriate value by analyzing all values first and computing their average, mean, etc., to find outliers. Finally, these intermediate result sets can be joined by standard set operations like union, intersection, or difference. When manually checking the results, he defined a "loose precision" value besides the usual "strict" one that did not label a match as a false positive if it *was* indeed faulty, but not because of the searched pattern. The outcome is a tool with a 70% empirical accuracy, that can be considered successful.

These "detection strategies" were extended with historical information by Rapu et al. [78]. Their approach included running the above described analysis on not only the current version of their subject software system, – the Jun 3D graphical framework – but on every fifth revision of it from the start. This way they could extract two more metrics: persistence, that expresses for how much of its "lifetime" was a given code element faulty, and stability, that means how many times the code element changed during its life. The logic behind this was that, e.g., a GodClass antipattern is dangerous only if it is not persistent, – i.e., the error is due to changes, not part of the original design – or not stable – i.e., it really disturbs the evolution of the system. With this method, they managed to halve the candidates in need of manual checking in the case of the above example.

Trifu and Marinescu went further by assuming that these antipatterns are just symptoms of larger, more abstract faults [88]. They proposed grouping several antipatterns – that may even occur together often – and supplementing them with contextual information to form "design flaws." Their main goal was to make antipattern recognition – and pattern recognition in general – more of a well-defined engineering task

rather than a form of art.

Our work is similar to the ones mentioned above in that we also detect antipatterns by employing source code metrics and static analysis. But, in addition, we inspect the correlations between these patterns and the maintainability values of the subject systems, and also consider bug-related information to more objectively demonstrate the common belief that they are indeed connected.

Other Approaches Lozano et al. [54] overviewed a broader sweep of related works and urged researchers to standardize their efforts. Apart from the individual harms antipatterns may cause, they aimed to find out that from exactly when in the life cycle of a software can an antipattern be considered "bad" and – not unlike [88] – whether these antipatterns should be raised to a higher abstraction level. In contrast to the historical information, we focus on objective metric results to shed light on the effect of antipatterns.

Another approach by Mäntylä et al. [58] is to use questionnaires to reveal the subjective side of software maintenance. The different opinions of the participating developers could mostly be explained by demographic analysis and their roles in the company, but there was a surprising difference compared to the metric based results. We, on the other hand, make these objective, metric based results our priority.

Although we used literature suggestion and expert opinion based metric thresholds for this empirical analysis, our work could be repeated – and possibly improved – by using the data-driven, robust, and pragmatic metric threshold derivation method described by Alves et al. [3]. They analyze, weight, and aggregate the different metrics of a large number of benchmark systems in order to statistically evaluate them and extract metric thresholds that represent the best or worst $X\%$ of all the source code. This can help in pinpointing the most problematic – but still manageably few – parts of a system.

If preexisting benchmarks with known antipattern occurrences are available, machine learning becomes a viable option. Khomh et al. [45] built on the methodology of Moha et al. [64] by making the decisions among parts of a complex ruleset more fuzzy with Bayesian networks. Another example was published by Maiga et al. [57], where they used Support Vector Machines to train models based on source code metrics to recognize antipattern instances. Here, however, we build machine learning models just to analyze the connection between the precomputed antipatterns and the maintainability of a given system.

In yet another approach, Stoianov and Şora [84] reduced pattern recognition to the resolution of logical predicates using Prolog. While this may seem radically different, there are similarities with our technique if we treat our metric thresholds and structural checks as the predicates and the programmatic source code traversal as Prolog's internal resolution process.

The Connections between Antipatterns and Maintainability To our knowledge, little research has been done so far on finding an explicit connection between antipatterns and maintainability. One is an investigation by Fontana and Maggioni [27] where they assume the connection and use antipatterns as well as source code metrics to evaluate software quality. Another is an empirical study by Yamashita and Moonen [96] where, after the refactoring of four Java systems, they conclude that antipatterns could provide experts and developers with more insights into maintainability

than source code metrics or subjective judgment alone; however, a combined approach is suggested.

If we broaden our search from maintainability to include other concepts, antipatterns have been linked (among others) to:

- comprehension by Abbes et al. [1], who concluded that, although single instances can be managed, multiple antipattern occurrences could have a significant impact and should be avoided,
- class change- and fault-proneness by Khomh et al. [47], who concluded that classes participating in antipatterns are more change- and fault-prone, and
- unit testing effort by Sabane et al. [80], who concluded that antipattern classes require substantially more test cases and should be tested with additional care.

On the other hand, if we just focus on maintainability, it has been positively linked to design patterns by Hegedűs et al. [103], refactorings by Szőke et al. [86], and version history metrics by Faragó et al. [25].

4.3 Methodology

For analyzing the relationships among antipatterns, bugs, and maintainability, we calculated the following measures for the subject systems:

- an absolute measure of maintainability per system,
- the total number of antipatterns per system, and
- the total number of bugs per system.

For the third research question, we could compile an even finer grained set of data – since the system-based quality attribute is not needed here:

- the total number of antipatterns related to each class in every subject system,
- the total number of bugs related to each class in every subject system, and
- every class level metric for each class in every subject system.

The metric values were extracted by the SourceMeter tool [26], the bug number information came from the PROMISE open bug database [63], and the pattern related metrics were calculated by our own tool described in Section 4.3.2.

4.3.1 Metric Definitions

We used the following source code metrics for antipattern recognition – chosen because of the interpretation of antipatterns described in Section 4.3.2:

- **AD** (**A**PI **D**ocumentation): the ratio of the number of documented public members of a class or package over the number of all of its public members.
- **CBO** (**C**oupling **B**etween **O**bjects): the CBO metric for a class means the number of different classes that are directly used by the class.

- **CC** (**C**lone **C**overage): the ratio of code covered by code duplications in the source code element over the size of the source code element, expressed in terms of the number of syntactic entities (e.g., statements, expressions, etc.).
- **CD** (**C**omment **D**ensity): the ratio of the comment lines of the source code element (CLOC) over the sum of its comment (CLOC) and logical lines of code (LLOC).
- **CLOC** (**C**omment **L**ines **O**f **C**ode): the number of comment and documentation code lines of the source code element; however, its nested, anonymous, or local classes are not included.
- **LLOC** (**L**ogical **L**ines **O**f **C**ode): the number of code lines of the source code element, without the empty and comment lines; its nested, anonymous, or local classes are not included.
- **McCC** (**Mc**Cabe's **C**yclomatic **C**omplexity): the complexity of the method expressed as the number of independent control flow paths in it.
- **NA** (**N**umber of **A**ttributes): the number of attributes in the source code element, including the inherited ones; however, the attributes of its nested, anonymous, or local classes (or subpackages) are not included.
- **NII** (**N**umber of **I**ncoming **I**nvocations): the number of other methods and attribute initializations which directly call the method (or methods of a class).
- **NLE** (**N**esting **L**evel **E**lse-If): the complexity expressed as the depth of the maximum "embeddedness" of the conditional and iteration block scopes in a method (or, the maximum of these for the container class), where in the if-else-if construct only the first if instruction is considered.
- **NOA** (**N**umber **O**f **A**ncestors): the number of classes, interfaces, enums, and annotations from which the class directly or indirectly inherits.
- **NOS** (**N**umber **O**f **S**tatements): the number of statements in the source code element; however, the statements of its nested, anonymous, or local classes are not included.
- **RFC** (**R**esponse set **F**or **C**lass): the number of local (i.e., not inherited) methods in the class (NLM) plus the number of directly invoked other methods by its methods or attribute initializations (NOI).
- **TLOC** (**T**otal **L**ines **O**f **C**ode): the number of code lines of the source code element, including empty and comment lines, as well as its nested, anonymous, or local classes.
- **TNLM** (**T**otal **N**umber of **L**ocal **M**ethods): the number of local (i.e., not inherited) methods in the class, including the local methods of its nested, anonymous, or local classes.
- **Warning P1, P2 or P3**: the number of different coding rule violations reported by the PMD analyzer tool[1], categorized into three priority levels.

[1] http://pmd.sourceforge.net

- **WMC** (**W**eighted **M**ethods per **C**lass): the WMC metric for a class is the total of the McCC metrics of its local methods.

4.3.2 Mining Antipatterns

The process of analyzing the subject source files – up until the point of metric extraction – is discussed in Section 2.1 (and shown in Figure 2.1). Additionally, each antipattern implementation could define one or more externally configurable parameters, mostly used for easily adjustable metric thresholds. These came from an XML-style rule file – called RUL – that can handle multiple configurations and even inheritance. It can also contain language-aware descriptions and warning messages that will be attached to the affected graph nodes.

After all these preparations, our tool could be run on the output LIM and graph of the previous analysis. It is basically a single new class built around the Visitor design pattern [30], which is appropriate as it is a new operation defined for an existing data structure and this data structure does not need to be changed to accommodate the modification. It "visits" the LIM model and uses its structural information and the computed metrics from its corresponding graph nodes to identify antipatterns. It currently recognizes the 9 types of antipatterns listed below. We chose to implement these 9 antipatterns because they appeared to be the most widespread in the literature and, as such, the most universally regarded as a real negative factor. They are described in greater detail by Fowler and Beck [28], and here we will just provide a short informal definition and explain how we interpreted them in the context of our LIM model. The parameters of the recognition are denoted by a starting **$** sign and can be configured in the RUL file mentioned above. Their default values are listed in Table 4.1.

- **Feature Envy** (FE): A class is said to be envious of another class if it is more concerned with the attributes of that other class than those of its own. It is interpreted as a method that accesses at least **$**MinAccess attributes, and at least **$**MinForeign% of those belong to another class.

- **Lazy Class** (LC): A lazy class is one that does not "do much", just delegates its requests to other connected classes – i.e., a non-complex class with numerous connections. It is interpreted as a class whose CBO metric is at least **$**MinCBO, but its WMC metric is no more than **$**MaxWMC.

- **Large Class Code** (LCC): Simply put, a class that is "too big" – i.e., it probably encapsulates not just one concept or it does too much. It is interpreted as a class whose LLOC metric is at least **$**MinLLOC.

- **Large Class Data** (LCD): A class that encapsulates too many attributes, some of which might be extracted – along with the methods that more closely correspond to them – into smaller classes and might be a part of the original class through aggregation or association. It is interpreted as a class whose NA metric is at least **$**MinNA.

- **Long Function** (LF): Similarly to LCC, if a method is too long, it probably has parts that could (or should) be separated into their own logical entities, thereby making the whole system more comprehensible. It is interpreted as a method where the LLOC, NOS or McCC metric exceeds **$**MinLLOC, **$**MinNOS or **$**MinMcCC, respectively.

Antipattern	Parameter	Value
FE	MinAccess	5
FE	MinForeign%	80%
LC	MinCBO	5
LC	MaxWMC	10
LCC	MinLLOC	500
LCD	MinNA	30
LF	MinLLOC	80
LF	MinNOS	80
LF	MinMcCC	10
LPL	MinParams	7
SHS	MinNII	10
TF	RefMax%	10%

Table 4.1. Default antipattern thresholds

- **Long Parameter List** (LPL): The long parameter list is one of the most recognized and accepted "bad code smells" in code. It is interpreted as a function (or method) whose number of parameters is at least **$**MinParams.

- **Refused Bequest** (RB): If a class refuses to use its inherited members – especially if they are marked "protected," through which the parent states that descendants *should* most likely use it – then it is a sign that inheritance might not be the appropriate method of implementation reuse. It is interpreted as a class that inherits at least one protected member that is not accessed by any locally defined method or attribute.

- **Shotgun Surgery** (SHS): Following the "Locality of Change" principle, if a method needs to be modified then it should not cause a demand for many other – especially remote – modifications, otherwise one of those can easily be missed, leading to bugs. It is interpreted as a method whose NII (i.e., the number of the different methods or attribute initializations where this method is called) metric is at least **$**MinNII.

- **Temporary Field** (TF): If an attribute only "makes sense" to a small percentage of the container class then it – and its closely related methods – should be decoupled. It is interpreted as an attribute that is only referenced by at most **$**RefMax% of the members of its container class.

4.3.3 Maintainability Model

We used the ADG presented in Figure 4.1 – which is a further developed version of the ADG published by Bakota et al. [9] – for assessing the maintainability of the selected subject systems. The informal definition of the referenced low-level metrics are described in Section 4.3.1.

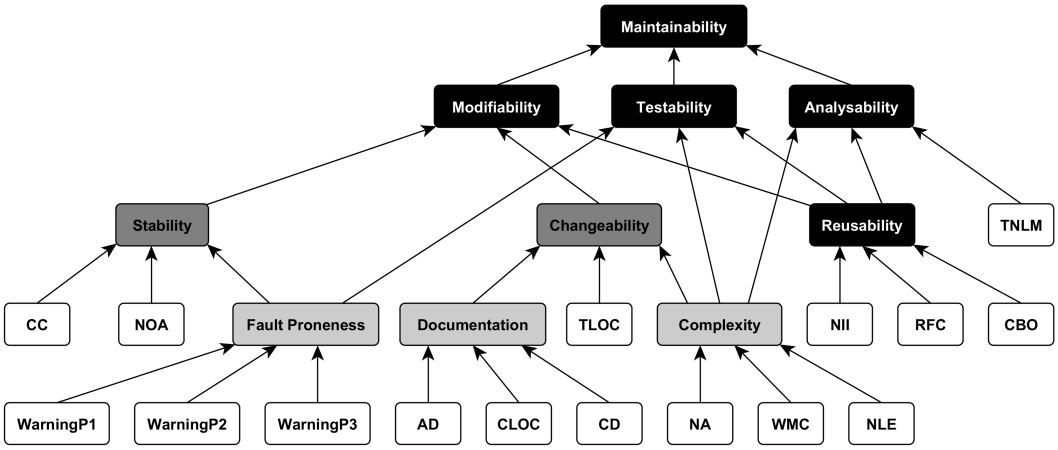

Figure 4.1. The quality model used to calculate maintainability

4.3.4 PROMISE

PROMISE [63] is an open-access bug database whose purpose is to help software quality related research and make the referenced experiments repeatable. It contains the source code of numerous open-source applications or frameworks and their corresponding bug-related data at the class level – i.e., not just system aggregates. We extracted this class level bug information for 34 systems from it that will be used for answering our third research question in Section 4.4.

4.3.5 Machine Learning

Using the class level table of data, we wanted to find out if the numbers of different antipatterns have any underlying structure that could help in identifying which system classes have bugs. As empirically they perform best in similar cases, we chose decision trees as the method of analysis, specifically J48, an open-source implementation of C4.5 [77].

4.4 Results

We analyzed the 228 subject systems and calculated the measures introduced in Section 4.3. These systems are all Java based and open-source – so we could access their source codes and analyze them – but their purposes are very diverse – ranging from web servers and database interfaces to IDEs, issue trackers, other static analyzers, build tools and many more. Note that the 34 systems that also have bug information from the PROMISE database are a subset of the original 228.

For the first two research questions concerned with finding correlation, we compiled a system level database of the maintainability values, the numbers of antipatterns, and the numbers of bugs. As the bug-related data was available at the class level in the corresponding 34 systems, we aggregated those values to fit in on a per system basis. The resulting dataset can be seen in Table 4.2, sorted in ascending order by the number of bugs.

Next, we performed correlation analysis on the collected data. When we only considered the systems that had related bug information, we found that the total number

Name	№ of Antipatterns	Maintainability	№ of Bugs
jedit-4.3	2,351	0.47240	12
camel-1.0	685	0.62129	14
forrest-0.7	53	0.73364	15
ivy-1.4	709	0.49465	18
pbeans-2	105	0.48909	19
synapse-1.0	398	0.58202	21
ant-1.3	933	0.51566	33
ant-1.5	2,069	0.41340	35
poi-2.0	2,025	0.36309	39
ant-1.4	1,218	0.45270	47
ivy-2.0	1,260	0.44374	56
log4j-1.0	224	0.59301	61
log4j-1.1	341	0.56738	86
Lucene	3,090	0.47288	97
synapse-1.1	717	0.56659	99
jedit-4.2	1,899	0.46826	106
tomcat-1	5,765	0.30972	114
xerces-1.2	2,520	0.13329	115
synapse-1.2	934	0.55554	145
xalan-2.4	3,718	0.15994	156
ant-1.6	2,821	0.38825	184
velocity-1.6	614	0.43804	190
xerces-1.3	2,670	0.13600	193
jedit-4.1	1,440	0.47400	217
jedit-4.0	1,207	0.47388	226
lucene-2.0	1,580	0.47880	268
camel-1.4	2,136	0.62682	335
ant-1.7	3,549	0.38340	338
Mylyn	5,378	0.65841	340
PDE_UI	7,523	0.53104	341
jedit-3.2	1,017	0.47523	382
camel-1.6	2,804	0.62796	500
camel-1.2	1,280	0.63005	522
xalan-2.6	11,115	0.22621	625

Table 4.2. The system level metrics of the 34 Java systems

of bugs and the total number of antipatterns per system had a Spearman correlation of **0.55** with a p-value below 0.001. This can be regarded as a significant relation that answers our **first research question** by suggesting that the more antipatterns there are in the source code of a software system, the more bugs it will likely contain. Intuitively this is to be expected, but with this empirical experiment we are a step closer to being able to treat this assumption validated. The discovered relation is, of course, not a one-to-one correspondence, but it illustrates the detrimental effect of antipatterns on source code well.

If we disregard the bug information but expand our search to all the 228 systems we analyzed, we can inspect the connection between the number of antipatterns and the maintainability values we computed. Here we found that there is an even stronger, **-0.62** inverse Spearman correlation, with a p-value – again – less than 0.001. Based on this observation, we can also answer our **second research question**: the more antipatterns a system contains, the less maintainable it will be, meaning that it will most likely cost more time and resources to execute any changes. This result corresponds to the definitions of antipatterns and maintainability quite well, as they suggest that antipatterns – the common solutions to design problems that seem beneficial but cause more harm than good in the long run – really do lower the expected maintainability – a value representing how easily a piece of software can be understood and modified, i.e., the "long run" harm.

This relation is visualized in Figure 4.2. The analyzed systems are sorted in the descending order of the antipatterns they contain, and the trend line of the maintainability value clearly shows improvement.

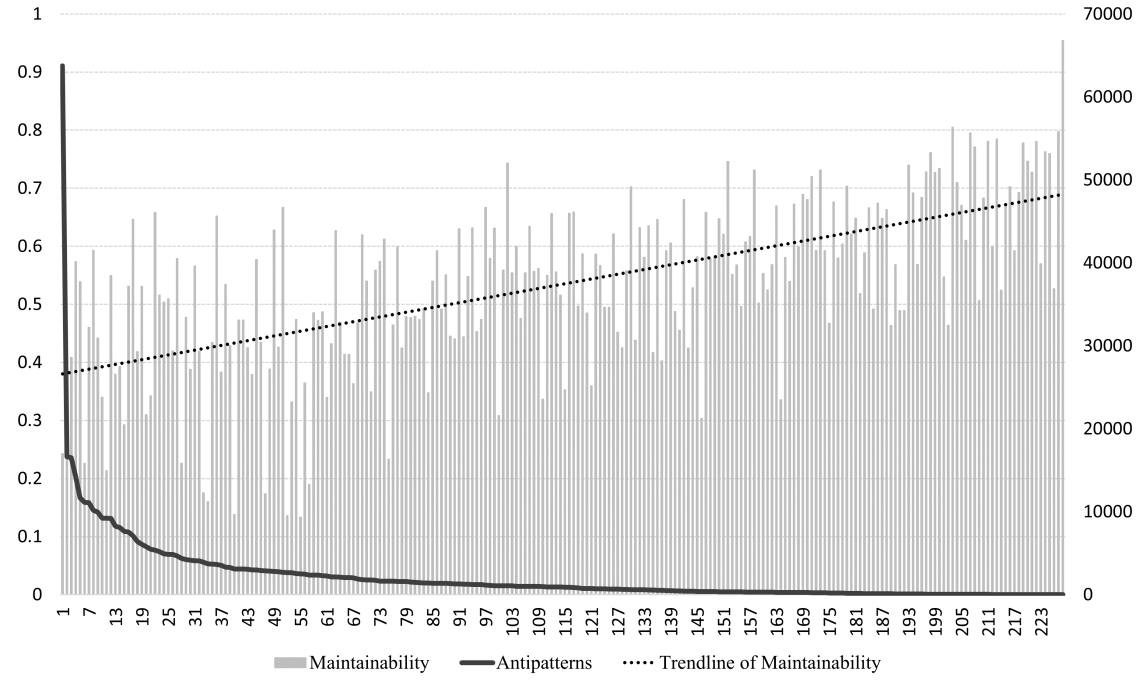

Figure 4.2. The trend of maintainability in the case of decreasing antipatterns

To answer our **third and final research question**, we could compile an even finer grained set of data. Here we retained the original, class level form of bug-related data and extracted class level source code metrics. We also needed to transform the antipattern information to correspond to classes, so instead of aggregating them to

the system level, we kept class antipatterns unmodified, while collecting method and attribute based antipatterns to their closest parent classes. Part of this dataset is shown in Table 4.3.

The resulting class level data was largely biased because – as it is to be expected – more than 80% of the classes did not contain any bugs. We handled this problem by subsampling the bugless classes in order to create a normally distributed starting point. Subsampling means that in order to make the results of the machine learning experiment more representative, we used only part of the data related to those classes that did not contain bugs. We created a *sub*set by randomly *sampling* – hence the name – from the "bugless" classes so that their number becomes equal to the "buggy" classes. This way, the learning algorithm could not achieve a precision higher than 50% just by choosing one class over the other. We then applied the J48 decision tree in three different configurations:

- using only the different antipattern numbers as predictors,
- using only the object-oriented metrics extracted by the SourceMeter tool as predictors, and
- using the attributes of both categories.

We tried to calibrate the decision trees we built to have around 50 leaf nodes and around 100 nodes in total. This is an approximately good compromise between under- and overlearning the training data. The results are summarized in Table 4.4.

These results clearly show that although antipatterns are – for now – inferior to OO metrics in this field, even a few patterns (concerned with single entities only) can approximate their bug predicting powers quite well. We note that it is expected that the "Both" category does not improve upon the "Metrics" category because in most cases – as of yet – the implemented antipatterns can be viewed as predefined thresholds on certain metrics. We conclude that antipatterns can already be considered valuable bug predictors, and with more implemented patterns – spanning multiple code entities – and a heavier use of the contextual structural information, they might even overtake them.

4.5 Threats to Validity

Our approach – naturally – has some threats to its validity. First of all, we reiterate from the previous chapter that when dealing with recognizing antipatterns, – or any patterns – the accuracy of mining is always in question. To make sure that we really extract the patterns we want, we created small, targeted source code tests that checked the structural requirements and metric thresholds of each pattern. To be also sure that we want to match the patterns we *should*, – i.e., those that are most likely to really have a detrimental effect on the quality of the software – we only implemented antipatterns that are well-known and well-documented in the literature. This way, the only remaining threat factor is the interpretation of those patterns to the LIM model.

Then there is the concern of choosing the correct thresholds for the low-level metrics. Although they are easily configurable, – even before every new inspection – in order to have results we could correlate and analyze, we had to use some specific thresholds. These were approximated by literature suggestions and expert opinions, taking

System	Name	№ of Bugs	LLOC	TNOS	...	ALL	FE	LC	LCC	LCD	LF	LPL	RB	SHS	TF
ant-1.3	org.apache.tools.ant.AntClassLoader	2	230	124	...	4	0	0	0	1	0	0	0	0	3
ant-1.3	org.apache.tools.ant.BuildEvent	0	51	21	...	0	0	0	0	0	0	0	0	0	0
ant-1.3	org.apache.tools.ant.BuildException	0	63	27	...	6	0	0	0	0	0	0	0	6	0
ant-1.3	org.apache.tools.ant.DefaultLogger	2	74	30	...	7	0	0	0	0	0	0	6	0	1
ant-1.3	org.apache.tools.ant.DesirableFilter	0	20	11	...	0	0	0	0	0	0	0	0	0	0
ant-1.3	org.apache.tools.ant.DirectoryScanner	0	472	353	...	25	0	0	0	3	0	19	1	2	2
ant-1.3	org.apache.tools.ant.IntrospectionHelper	2	229	142	...	2	0	0	0	1	0	0	0	0	1
ant-1.3	org.apache.tools.ant.Location	0	29	13	...	1	0	0	0	0	0	0	0	0	1
ant-1.3	org.apache.tools.ant.Main	1	364	254	...	3	0	0	0	2	0	0	0	0	1
ant-1.3	org.apache.tools.ant.NoBannerLogger	0	21	8	...	0	0	0	0	0	0	0	0	0	0
ant-1.3	org.apache.tools.ant.PathTokenizer	0	37	16	...	0	0	0	0	0	0	0	0	0	0
ant-1.3	org.apache.tools.ant.Project	1	710	406	...	37	0	0	1	0	2	0	0	9	25
ant-1.3	org.apache.tools.ant.ProjectHelper	3	151	267	...	2	0	0	0	0	1	0	0	0	1
ant-1.3	org.apache.tools.ant.RuntimeConfigurable	2	45	18	...	0	0	0	0	0	0	0	0	0	0
ant-1.3	org.apache.tools.ant.Target	1	103	42	...	1	0	0	0	0	0	0	0	1	0
ant-1.3	org.apache.tools.ant.Task	0	64	19	...	15	0	0	0	0	0	0	8	6	1
ant-1.3	org.apache.tools.ant.TaskAdapter	0	25	13	...	0	0	0	0	0	0	0	0	0	0
ant-1.3	org.apache.tools.ant.taskdefs.Ant	0	133	87	...	0	0	0	0	0	0	0	0	0	0
ant-1.3	org.apache.tools.ant.taskdefs.AntStructure	0	179	134	...	1	0	0	0	1	0	0	0	0	0
ant-1.3	org.apache.tools.ant.taskdefs.Available	0	101	46	...	0	0	0	0	0	0	0	0	0	0

Table 4.3. Part of the compiled class level dataset

Method	TP Rate	FP Rate	Precision	Recall	F-Measure
Antipatterns	0.658	0.342	0.670	0.658	0.653
Metrics	0.711	0.289	0.712	0.711	0.711
Both	0.712	0.288	0.712	0.712	0.712

Table 4.4. The results of the machine learning experiments

into consideration the minimum, maximum, and average values of the corresponding metrics.

Another threat to validity is using our chosen quality model for calculating maintainability values. Despite many empirical validations in previous works, we still cannot state that the maintainability model we used is perfect. Moreover, the ISO/IEC 25010 standard is similarly configurable in its low-level metrics as its predecessor, so the results may vary depending on the quality model's settings. It is also very important to have a source code metric repository with a large enough number of systems to get an objective absolute measure for maintainability.

Our results also depend on the assumption that the bug-related values we extracted from the PROMISE database are correct. If they are inaccurate, that means that our correlation results with the number of bugs are inaccurate too. But as many other works make use of these data, we consider the possibility of this negligible.

Lastly, we have to face the threat that our data is biased or that the results are coincidental. We tried to combat these factors by using different kinds of subject systems, balancing our training data to a normal class distribution before the machine learning procedure, and considering only statistically significant correlations.

4.6 Summary

In this chapter, we presented an empirical analysis of exploring the connections among antipatterns, the number of known bugs, and software maintainability. First, we briefly explained how we implemented a tool that can match antipatterns on a language independent model of a system. We then analyzed more than 200 open-source Java systems, extracted their object-oriented metrics and antipatterns, calculated their corresponding maintainability values using a probabilistic quality model, and even collected class level bug information for 34 of them. By correlation analysis and machine learning methods, we were able to draw interesting conclusions.

At a system level scale, we found that in the case of the 34 systems that also had bug-related information, there is a significant positive correlation between the number of bugs and the number of antipatterns. Also, when we disregarded the bug data but expanded our search to all 228 analyzed systems to concentrate on maintainability, the result was an even stronger negative correlation between the number of antipatterns and maintainability. This further supports what one would intuitively think considering the definitions of antipatterns, bugs, and software quality.

Another interesting result is that the mentioned 9 antipatterns in themselves can quite closely match the bug predicting power of more than 50 class level object-oriented metrics. Although they – as of yet – are inferior, with further patterns that would span over source code elements and rely more heavily on the available structural information, this method has the potential to outperform simple metrics in fault prediction.

As with all similar works, ours also has some threats to its validity, but we feel that it is a valuable step towards empirically validating that antipatterns really do hurt software maintainability and can highlight points in the source code that require closer attention.

"It's the repetition of affirmations that leads to belief."
— Claude M. Bristol

5

The Connection between Antipatterns and Maintainability in C++

5.1 Overview

In this chapter, in order to improve our understanding of the connection between antipatterns and source code maintainability, we intend to (partially) replicate the results of Chapter 4, only in the domain of C++. As our subject systems, we selected 45 evenly distributed sample revisions taken from the `master` and `electrolysis` branches of Firefox between 2009 and 2010 – approximately one revision every two weeks. These revisions provided the basis for both antipattern detection and maintainability assessment. We extracted the occurrences of the same 9 antipattern types we previously discussed and summed the number of matches by type. We also divided these sums by the total number of logical lines of the subject system for each revision to create new, system-level antipattern density predictor metrics.

Next, we computed corresponding maintainability values; still following the principles of the ISO/IEC 25010 standard [41] but this time using a C++ specific quality model. Additionally, we adapted versions of the independent Maintainability Index metric [21] to get a secondary quality indicator.

With these data available, we attempted to answer the following two research questions:

Research Question 1: *How does the number of antipatterns in a given system correlate with its maintainability?*

Research Question 2: *Can the antipattern instances of a system be used to predict its maintainability?*

The chapter is structured as follows: Since the related work we listed in Chapter 4 is still applicable, we start directly with the details and differences of our current methodology in Section 5.2. Next, Section 5.3 discusses the results we obtained, then in Section 5.4 we overview some factors that might threaten the validity of these results. Lastly, in Section 5.5 we draw some pertinent conclusions.

5.2 Methodology

The sequence of steps we took in order to answer our research questions is depicted in Figure 5.1, combining a general static source code analysis with pattern recognition, model evaluation, and machine learning.

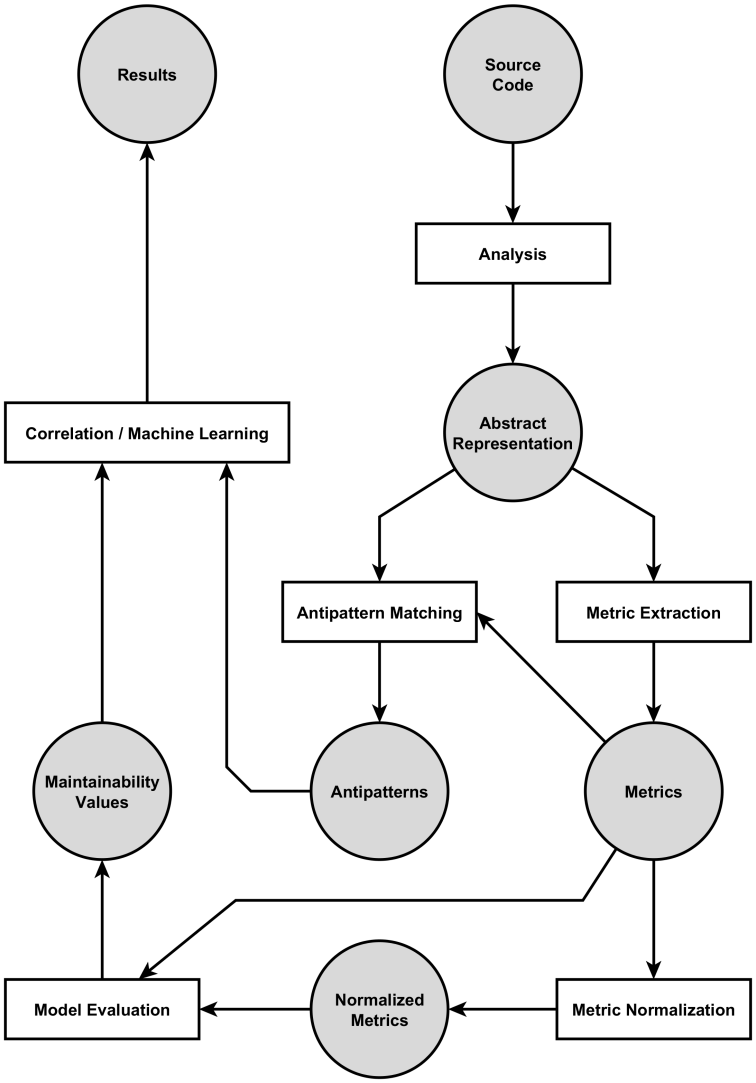

Figure 5.1. The methodology step sequence

The gray circles represent the different artifacts that exist between the steps of the process, while the white rectangles are the steps themselves. The steps that introduce new information, or where relevant changes occurred compared to the methodology in Chapter 4, are explored in their own subsections below.

5.2.1 Static Analysis

The analysis was conducted using a shell script that enumerated the 45 Firefox revisions, checked out the corresponding repositories, (if not yet available) and updated them to the correct commit before initiating a build sequence. The core of the analysis was still performed using the SourceMeter tool [26].

Listing 5.1. The filter file for the analysis

```
-/
+/path/to/repos/
-/config/
-/testing/
-/build/
-/media/
-/security/
-/db/
-/jpeg/
-/modules/
+/path/to/repos/.*/modules/plugin/
+/path/to/repos/.*/modules/staticmod/
```

Note that apart from the simple build script of `make -f client.mk`, our custom analysis configuration contained filters to skip the results of every command that matched the word "conftest" (a so-called hard filter) and to later skip any source code elements whose source code path information matched the filters described in Listing 5.1 (a so-called soft filter). These filters were obtained via manual analysis of the Firefox repositories, pinpointing irrelevant or 3rd party code.

The lines in Listing 5.1 are applied in the order shown, allowing or disallowing a path based on the starting `+` or `-` character. So, for example, the first two lines mean that everything is filtered except for any content coming from the "repos" directory.

It should be added that the 45 Firefox revisions we selected are a subset of the Green Mining Dataset collected by Abram Hindle [36], as it is also our intention to relate antipatterns and software quality to energy and power consumption in the future.

5.2.2 Metric Definitions

After performing our analysis, we extracted the metrics of the global namespace, which represent an aggregated, top-level view of the subject system. These metrics are the following:

- **HVOL** (**H**alstead **VOL**ume): if we let η_1 denote the number of distinct operators, η_2 the distinct operands, N_1 the total number of operators and N_2 the total number of operands, then $HVOL = N_1 + N_2 \cdot log_2(\eta_1 + \eta_2)$. From a C++ perspective, we will treat unary and binary operators (both arithmetic, increment, comparison, boolean, assignment, bitwise, shift and compound), keywords (e.g., return, sizeof, if, else, etc.), brackets, braces, parentheses, semicolons and pointer asterisks as operators, while the corresponding types, names, members, constants and literals will be treated as operands. Although this metric is usually used for single methods, it can be easily generalized to the system level.

- **TCBO** (**T**otal **C**oupling **B**etween **O**bjects): the CBO metric for a class means the number of different classes that are directly used by the class. Usage, among others, includes method calls, parameters, instantiations and attribute accesses as well as returnable and throwable types. **TCBO** is an aggregation of class-level CBOs to the system level, while **AvgCBO** (Average CBO) is defined as the ratio $TCBO/TNCL$.

- **TLCOM5** (**T**otal **L**ack of **CO**hesion in **M**ethods 5): for a class, LCOM5 measures the lack of cohesion, and it is interpreted as how many coherent classes the class could be split into. It is calculated by taking a non-directed graph, where the nodes are the implemented local methods of the class and there is an edge between two nodes if and only if a common attribute or abstract method is used or a method invokes another. The value of the metric is the number of connected components in the graph not counting those which contain only constructors, destructors, getters or setters. TLCOM5 is the sum of LCOM5s, while **AvgLCOM5** (Average LCOM5) is defined as the ratio $TLCOM5/TNCL$.
- **TRFC** (**T**otal **R**esponse set **F**or **C**lass): for a class, RFC is the number of local (i.e., not inherited) methods in the class plus the number of directly invoked other methods by its methods or attribute initializations. For the system, TRFC is the aggregated sum of RFCs, while **AvgRFC** (Average RFC) is defined as the ratio $TRFC/TNCL$.
- **TWMC** (**T**otal **W**eighted **M**ethods per **C**lass): the WMC metric for a class is the total of the McCC (McCabe's Cyclomatic Complexity) metrics of its local methods. For the system, TWMC is the sum of all WMCs, while **AvgWMC** (Average WMC) is defined as the ratio $TWMC/TNCL$.
- **TAD** (**T**otal **A**PI **D**ocumentation): the ratio of the number of documented public members of the system over the number of all of its public members.
- **TCD** (**T**otal **C**omment **D**ensity): the ratio of the comment lines of the system (TCLOC) over the sum of its comment (TCLOC) and logical lines of code (TLLOC).
- **TCLOC** (**T**otal **C**omment **L**ines **O**f **C**ode): the number of comment and documentation code lines in the system, where comment lines are lines that have either a block or a line comment, while a documentation comment line is a line that has (at least part of) a comment that is syntactically directly in front of a member. Note that a single line can be both a logical line *and* a comment line if it has both code and at least one comment.
- **TLLOC** (**T**otal **L**ogical **L**ines **O**f **C**ode): the number of code lines of the system, without the empty and purely comment lines.
- **TNA** (**T**otal **N**umber of **A**ttributes): the number of attributes in the system.
- **TNCL** (**T**otal **N**umber of **C**lasses): the number of classes in the system.
- **TNEN** (**T**otal **N**umber of **E**nums): the number of enums in the system.
- **TNIN** (**T**otal **N**umber of **I**nterfaces): the number of interfaces in the system. Note that although C++ lacks language support for the concept, we will treat classes with only pure virtual methods as interfaces.
- **TNM** (**T**otal **N**umber of **M**ethods): the number of methods in the system.
- **TNPKG** (**T**otal **N**umber of **P**ac**K**a**G**es): the number of namespaces in the system. Note that the word "package" here refers to a generalized object-oriented container concept which, in C++, directly maps to namespaces.

- **TNOS** (**T**otal **N**umber **O**f **S**tatements): the number of statements in the system.

Note that the SourceMeter tool [26] did not have native support for some of the system-level metrics, including the Total and Average versions of CBO, WMC, LCOM5 and RFC, along with the aggregated Halstead Volume. The implementation of these computations was performed specifically for this study.

5.2.3 Metric Normalization

Metric normalizations were performed following the principles outlined in Section 2.2. The only deviations worth mentioning are the exceptions to the "simpler mental model" inversion – or, the "the bigger the better" metrics. These were are all documentation-related, namely TAD, TCD, and TCLOC.

5.2.4 Antipatterns

It should be mentioned that in addition to the single antipatterns from Chapter 4, this time we also collected a SUM value, which is – not surprisingly – defined as the sum of all types of antipatterns in the given subject system. Furthermore, we calculated densities for each absolute antipattern, meaning that for every AP antipattern there is now an $\mathrm{AP}_{\mathrm{DENS}}$ metric available, computed as the ratio $AP/TLLOC$.

5.2.5 Maintainability Models

In order to assess the maintainability of the systems we analyzed, we created an expert opinion-based maintainability model according to the ISO/IEC 25010 standard [41]. The weights of how the subcharacteristics listed in Section 2.3 are to be aggregated – and how they themselves are computed from source code metrics – were derived from the results of a poll.

First, the 10 chosen experts – each of whom is an academic or industrial professional with at least 5 years of experience in software engineering – had to distribute 100 points among the source code metrics (listed in Section 5.2.2) for each subcharacteristic, to express how much they think that metric affects the given subcharacteristic. The results of this step are summarized in Table 5.1.
Next, they had to distribute another 100 points among the subcharacteristics themselves, expressing how much each of them affects the overall Maintainability. The results of this step are summarized in Table 5.2.
Given these weights – and later dividing by 100 – we were able to obtain system-level Maintainability values for each of the given subject revisions in the $[0, 1]$ interval.

In addition, we computed the two "traditional" Maintainability Index metrics [21], interpreting them using our static source code metrics as:

$$MI = 171 - 5.2 \cdot ln(HVOL) - 0.23 \cdot TWMC - 16.2 \cdot ln(TLLOC)$$

and

$$MI_2 = 171 - 5.2 \cdot log_2(HVOL) - 0.23 \cdot TWMC \\ - 16.2 \cdot log_2(TLLOC) + 50 \cdot \sin(\sqrt{2.4 \cdot TCD}).$$

Metric	Analysability	Modifiability	Modularity	Reusability	Testability
HVOL	25	26.6	11	13	25.7
AvgCBO	22	29.6	43	33	24.5
AvgLCOM5	6.5	4.7	8.5	7.5	7.4
AvgRFC	1.5	3	15	7.5	3.3
AvgWMC	10	10	5.5	9	10.8
TAD	7.3	7	3.3	7.9	2.5
TCBO	1	1.3	0	1	3.6
TCD	4.1	1.5	0	2	1.5
TCLOC	0.3	0.5	0	4.5	0
TRFC	0	0.5	0.5	0.5	1
TWMC	1	1	0	0	0
TLLOC	14.5	9.2	0	2.4	8.5
TNA	0	0	0	0	0.2
TNCL	5	3	0	0	5
TNM	1.4	1.3	0	0	3.1
TNOS	0	0	0	0	1
TNPA	0	0	0	1.4	0
TNPCL	0	0	4.6	2.4	0
TNPEN	0	0	0.4	1.3	0
TNPIN	0	0	5.7	3.5	0
TNPKG	0.4	0.8	0	0.2	0
TNPM	0	0	2.5	2.9	1.9

Table 5.1. The results of the subcharacteristic votes

Subcharacteristic	Maintainability
Analysability	28.5
Modifiability	26.5
Modularity	17.1
Reusability	13.7
Testability	14.2

Table 5.2. The results of the Maintainability votes

We also calculated their modified counterparts (MI^* and $MI_2{}^*$), where we changed the Total WMC values to their corresponding averages. We did so to scale each part of the sum to the same magnitude because complexity (WMC) is the only component not inside a logarithm or sine, and the TWMC values dominated every other term of the formulas.

5.3 Results

After all these preliminaries, we are now ready to address our two research questions.

5.3.1 Correlation Results

To address our **first research question**, we decided to calculate the Pearson and Spearman correlations between each antipattern and maintainability measure pair, summarized in tables 5.3 and 5.4, respectively. Note that a single star suffix (*) means that the correlation is statistically significant at the .05 level, while a double star (**) means a significance at the .01 level. Also, to help in quickly parsing these tables, any cell where the correlation coefficient is either positive or non-significant was marked in a light gray background, and a darker gray when it is significantly positive (the worst case from our perspective).

As these tables clearly show, most antipattern-maintainability pairs have a strong, significant inverse connection. There are a few marked correlations, mainly for Modularity and Reusability, but even in these cases the non-significant values are still negative, while the positive values are non-significant and weak. We highlight the correlations between the SUM and SUM_{DENS} antipatterns and our final Maintainability measure as these represent most closely the overall effects of antipatterns on maintainability. The corresponding values are **-0.658** and **-0.692** for Pearson, and **-0.704** and **-0.678** for Spearman correlation, respectively. Thus, in response to the first research question we conclude – based on these empirical findings – that there is a strong, inverse relationship between the number of antipatterns in a system and its maintainability. This supports our initial assumption that the more antipatterns the source code contains, the harder it is to maintain.

5.3.2 Machine Learning Results

To answer our **second research question**, we compiled ten tables applicable for machine learning experiments – one for each maintainability measure. These contained every antipattern type as predictors and the values for their chosen maintainability measures as targets for prediction. We then ran these tables through all five regression techniques mentioned in Section 2.4.2 to see how well they worked in practice. The corresponding correlation coefficients of the resulting models are shown in Table 5.5.

The high values of these coefficients suggest an affirmative answer to our second research question: antipatterns *can* be valuable predictors for maintainability assessment. The models we built weight the antipattern predictors with mostly negative values, but there are numerous positive instances as well. Further analysis of the structure of the models in the case of the Maintainability target revealed that some antipatterns *consistently* appear with negative weights more often than others. Moreover, this ordering of importance largely coincides with the above correlation magnitudes.

5.3.3 Lessons Learned

The most obvious lesson learned, based on these results, is the measurable detrimental effect of antipatterns on maintainability. Moreover, the conclusion we drew from the correspondence between correlation values and negative model weights is that there could also be an *order of importance* among the antipatterns studied here.

The most important ones to avoid appear to be Long Functions, Large Class Codes and Shotgun Surgeries. The frequently suggested refactorings for the first two antipatterns are “Extract Method” and “Extract Class”, respectively. As for Shotgun Surgery,

Antipattern	MI	MI_2	MI^*	MI_2^*	Analysability	Modifiability	Modularity	Reusability	Testability	Maintainability
FE	−.985**	−.985**	−.857**	−.796**	−.830**	−.738**	−.141	−.311*	−.785**	−.657**
FE_{DENS}	−.659**	−.659**	−.303*	−.219	−.548**	−.637**	−.538**	−.570**	−.679**	−.661**
LC	−.825**	−.825**	−.933**	−.968**	−.732**	−.502**	.274	.023	−.519**	−.371*
LC_{DENS}	−.749**	−.749**	−.886**	−.943**	−.666**	−.425**	.327*	.075	−.435**	−.297*
LCC	−.987**	−.987**	−.864**	−.822**	−.862**	−.758**	−.133	−.326*	−.791**	−.674**
LCC_{DENS}	−.670**	−.670**	−.296*	−.249	−.624**	−.700**	−.563**	−.638**	−.711**	−.722**
LCD	−.768**	−.768**	−.555**	−.438**	−.531**	−.554**	−.290	−.314*	−.611**	−.526**
LCD_{DENS}	−.662**	−.662**	−.424**	−.298*	−.422**	−.483**	−.344*	−.325*	−.541**	−.477**
LF	−.988**	−.988**	−.883**	−.830**	−.885**	−.782**	−.174	−.372*	−.830**	−.710**
LF_{DENS}	−.820**	−.820**	−.530**	−.455**	−.783**	−.818**	−.566**	−.688**	−.866**	−.835**
LPL	−.961**	−.961**	−.931**	−.872**	−.781**	−.643**	.014	−.160	−.704**	−.544**
LPL_{DENS}	−.864**	−.864**	−.741**	−.649**	−.652**	−.594**	−.157	−.253	−.673**	−.542**
RB	−.985**	−.985**	−.852**	−.788**	−.820**	−.736**	−.165	−.328*	−.783**	−.662**
RB_{DENS}	−.930**	−.930**	−.714**	−.634**	−.757**	−.734**	−.317*	−.435**	−.786**	−.695**
SHS	−.988**	−.988**	−.850**	−.778**	−.837**	−.767**	−.212	−.375*	−.811**	−.698**
SHS_{DENS}	−.889**	−.889**	−.634**	−.538**	−.745**	−.771**	−.450**	−.548**	−.815**	−.754**
SUM	−.982**	−.982**	−.869**	−.791**	−.814**	−.735**	−.164	−.319*	−.785**	−.658**
SUM_{DENS}	−.910**	−.910**	−.712**	−.606**	−.731**	−.729**	−.342*	−.438**	−.783**	−.692**
TF	−.955**	−.955**	−.835**	−.740**	−.777**	−.723**	−.199	−.330*	−.771**	−.651**
TF_{DENS}	−.882**	−.882**	−.711**	−.592**	−.691**	−.695**	−.317*	−.397**	−.747**	−.653**

Table 5.3. Pearson correlations between antipatterns and maintainability

Antipattern	MI	MI_2	MI^*	MI_2^*	Analysability	Modifiability	Modularity	Reusability	Testability	Maintainability
FE	−.985**	−.985**	−.853**	−.809**	−.852**	−.749**	−.161	−.370*	−.806**	−.652**
FE_{DENS}	−.535**	−.535**	−.257	−.258	−.440**	−.532**	−.550**	−.595**	−.561**	−.609**
LC	−.757**	−.757**	−.936**	−.920**	−.755**	−.538**	.224	−.044	−.572**	−.381**
LC_{DENS}	−.542**	−.542**	−.777**	−.854**	−.517**	−.276	.372*	.123	−.307*	−.133
LCC	−.985**	−.985**	−.873**	−.839**	−.871**	−.754**	−.131	−.354*	−.811**	−.651**
LCC_{DENS}	−.482**	−.482**	−.213	−.258	−.439**	−.543**	−.590**	−.655**	−.550**	−.636**
LCD	−.731**	−.731**	−.484**	−.445**	−.509**	−.551**	−.303*	−.365*	−.590**	−.528**
LCD_{DENS}	−.626**	−.626**	−.355*	−.344*	−.373*	−.431**	−.299*	−.318*	−.474**	−.428**
LF	−.991**	−.991**	−.849**	−.800**	−.902**	−.821**	−.242	−.453**	−.866**	−.728**
LF_{DENS}	−.874**	−.874**	−.622**	−.608**	−.824**	−.837**	−.500**	−.671**	−.876**	−.821**
LPL	−.952**	−.952**	−.926**	−.856**	−.851**	−.696**	.019	−.219	−.750**	−.560**
LPL_{DENS}	−.904**	−.904**	−.707**	−.670**	−.715**	−.654**	−.184	−.338*	−.708**	−.580**
RB	−.976**	−.976**	−.819**	−.793**	−.829**	−.734**	−.167	−.375*	−.794**	−.646**
RB_{DENS}	−.911**	−.911**	−.706**	−.728**	−.735**	−.674**	−.219	−.396**	−.730**	−.614**
SHS	−.985**	−.985**	−.820**	−.768**	−.884**	−.827**	−.291	−.487**	−.871**	−.747**
SHS_{DENS}	−.907**	−.907**	−.694**	−.698**	−.787**	−.773**	−.370*	−.538**	−.812**	−.726**
SUM	−.978**	−.978**	−.806**	−.754**	−.847**	−.786**	−.250	−.444**	−.834**	−.704**
SUM_{DENS}	−.895**	−.895**	−.674**	−.641**	−.732**	−.726**	−.332*	−.476**	−.768**	−.678**
TF	−.945**	−.945**	−.746**	−.704**	−.785**	−.746**	−.277	−.445**	−.796**	−.681**
TF_{DENS}	−.843**	−.843**	−.595**	−.543**	−.675**	−.704**	−.378*	−.484**	−.739**	−.665**

Table 5.4. Spearman correlations between antipatterns and maintainability

	Linear Reg.	MLP	REPTree	M5P	SMO Reg.
MI	.9991	.9969	.9079	.9983	.9993
MI*	.9825	.9968	.8695	.9727	.9971
MI_2	.9991	.9969	.9635	.9983	.9993
MI_2*	.9864	.9689	.9033	.9799	.9858
Analysability	.8210	.9085	.7632	.9097	.9151
Modifiability	.8082	.9223	.7286	.8138	.8348
Modularity	.9082	.8915	.7461	.7589	.8757
Reusability	.8247	.8927	.6777	.6222	.8455
Testability	.8637	.9547	.8564	.8874	.8903
Maintainability	.8513	.9318	.7619	.8179	.8556

Table 5.5. Correlation coefficients of the machine learning models

the main goal is to reduce coupling by moving or extracting methods or fields, or even identifying a common superclass.

Refused Bequests and Temporary Fields seem less dangerous. The former can be fixed with "Replace Inheritance with Delegation" or by extracting an even more abstract superclass to house just the common members, while the latter is often corrected with "Extract Class" – which can coincide with extracting a method object.

And finally, Long Parameter Lists, Feature Envies, Lazy Classes and Large Class Data instances can be more easily tolerated. However, these can also be eliminated using techniques given in [28]. Long Parameter Lists have "Preserve Whole Object" or "Introduce Parameter Object"; Feature Envy has "Move Method" or "Extract Method"; Lazy Classes may vanish if their functionality is inlined or their connections are introduced to each other without the middle man; and lastly, Large Class Data can be solved – again – with "Extract Class".

The key point of these observations is that developers should concern themselves more forcefully with the organization of source code, and not just its behavior, since the work they put in in advance seems to lead to an easier maintenance phase, while the performance overhead introduced by the extra classes and methods is negligible.

5.4 Threats to Validity

There are a few aspects that might possibly threaten the validity of our results. One is that the antipattern matches might not be correct. While finding antipattern instances is far from being a solved problem, we tried to acquire reliable statistics by implementing widely recognized antipatterns with usual/recommended threshold values and previously published tooling support.

Imprecise maintainability scores could also skew our results. To combat this, we decided to utilize static, independent source code metrics and expert opinion-based weight determination, all the while adhering to the guidelines of an international standard.

To ensure that the connections we uncover were not just coincidental, we only included statistically significant correlations in this study. The connections could also

be attributed to the fact that both the maintainability scores and the antipattern instances are – at least partially – based on the same static source code metrics. Despite the overlap, there are important differences, because the two concepts do not rely on the same aggregation level of metrics (method/class or system level) and antipatterns incorporate other structural cues as well. We would also argue that the results *could* be meaningful even if the base set of metrics were identical, given that the mapping of concepts to metrics is plausible.

Lastly, the generalizability of these findings could be largely affected by the number of subject systems analyzed. Although a benchmark made from 45 versions of such a huge and complex software system can hardly be regarded as small, we intend to include more revisions and different applications as well.

5.5 Summary

In this chapter, we analyzed 45 revisions of Firefox and calculated static source code metrics for each of them. Using these metrics, specific threshold parameters, and structural information, we matched the previous 9 types of antipatterns and their respective densities in each revision. Also utilizing these metrics, we calculated maintainability values based on the ISO/IEC 25010 software quality framework. After correlating these two sets of data, we found statistically significant inverse relationships, which we consider another step towards objectively demonstrating that antipatterns have an adverse effect on software maintainability. Moreover, our machine learning experiments indicated that regression techniques can attain high precision in predicting maintainability from antipattern information alone, suggesting that antipatterns can be valuable besides – or even instead of – static source code metrics in software maintainability assessment.

Part II

Performance Optimization

"Performance is the best way to shut people up."
— Marcus Lemonis

6 Qualitative Prediction Models

6.1 Overview

As technological advancements make GPUs – or other alternative computation units – more widespread, it is increasingly important to question whether the CPU is still the most efficient option for running specific applications. In this chapter, we describe a method for deriving prediction models that can select the hardware platform best suited for a given algorithm with regards to one of three different aspects: time, power, or energy consumption. These models are built by applying various machine learning methods where the predictors are calculated from the source code (using static analysis techniques), and the output of the models is the optimal execution platform.

To build the desired prediction models, first we take a number of algorithms – referred to as benchmarks – that have functionally equivalent sequential and parallel (OpenCL and OpenMP-based) implementations. Then, we extract multiple size, coupling and complexity metrics from the main functional parts of every benchmark using static analysis. Next, we collect measurements on the time and power required to run these algorithms on different platforms and assign labels to them based on which platform performed the best. Finally, we apply multiple machine learning methods that use the metrics we calculated to predict the optimal execution platform for a system.

These steps yield models – one for every machine learning approach – that are capable of classifying new systems as well. There are no prerequisites for using these models other than extracting the same static metrics from the source code of the new subject system that were used in the model building phase. With those metrics, one of the previously built models can be utilized to predict the optimal hardware platform for running the subject system.

The chapter is organized as follows: In the next section, we list some works related to ours. Then, in Section 6.3 we describe our methodology in detail. In Section 6.4, we introduce the benchmarks we used, while in Section 6.5 we elaborate on dynamic measurements. Afterwards, in Section 6.6 we discuss the static metric extraction method. In Section 6.7, we show the preliminary results we have achieved. Lastly, in Section 6.8 we draw our conclusions.

6.2 Related Work

As heterogeneous execution environments became more and more prevalent in recent years, it also became increasingly important to study their individual and relative performances. There is a multitude of related work in the area, with fundamentally different approaches.

Some researchers tried to characterize a particular platform alone. For example, Ma et al. [56] focused only on GPUs and built statistical models to predict power consumption. Brandolese et al. [16] concentrated on CPUs by statically analyzing C source code and estimating their execution times. For the OpenMP environment, Li et al. [53] derived a performance model, while Shen et al. [81] compared OpenMP to OpenCL using some of the same benchmark systems we used. Note that although we share some source benchmarks with Shen et al., we focus on predicting performance instead of analyzing the actual, dynamic performance of concrete implementations. For FPGAs, Osmulski et al. [68] introduced a tool to evaluate the power consumption of a given circuit without the need to actually test it. It is also evident from these studies that most of this type of research targets a single aspect (time or power). We, on the other hand, consider multiple platforms and multiple aspects as our goal is to predict the optimal environment from static information alone.

Others are more closely related to our current work as they focus on cross-platform optimization. Yang et al. [97] generalized the expected behavior of a program on another platform by extrapolating from partial execution measurements, while Takizawa et al. [87] aimed at energy efficiency by dynamically selecting the execution environment at run time. Unlike these works, we use dynamic information only for building the prediction models, which then can be used with static data alone.

A subset of these cross-platform works concentrate on compiled or intermediate program representations. Kuperberg et al. [51] analyzed components and platforms separately to avoid a combinatorial explosion. They built parametric models for performance prediction, but these require "microbenchmarks" for each platform and work with Java bytecode only. Marin and Mellor-Crummey [59] also processed application binaries and built architecture-neutral models, which were then used to estimate cache misses and execution time on an unknown platform. One key difference between these studies and our approach is that we use the source code of the training benchmarks and not their compiled forms.

6.3 Methodology

This section contains the detailed description of our concept of a prediction model and how it is built. Using source code metrics produced by static analysis, our model is able to predict the computing unit that allows the fastest or most energy efficient execution of a given program in advance. The model is qualitative, so it does not predict the possible gain of selecting one execution platform over another, only the best platform itself. The model is built following these steps:

- Extract multiple size, coupling, and complexity metrics from the main functional parts of the systems we analyze,
- collect measurements of the time, power, and energy required to run them on different platforms, and

- use various machine learning algorithms to build models that are able to predict the optimal platform for a program with a specific set of metric values.

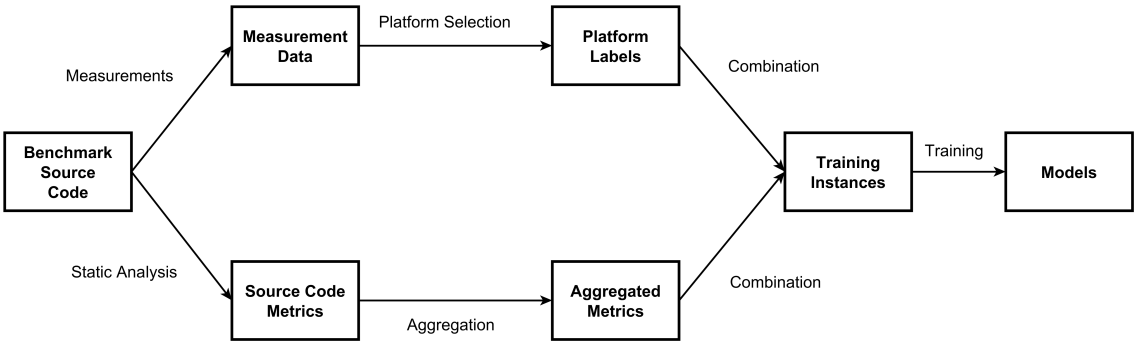

Figure 6.1. The main steps of the model creation process

The steps and intermediate states of our methodology are outlined in Figure 6.1. Each of these steps will be detailed in its own section:

- The selected benchmarks in Section 6.4,
- the dynamic measurements in Section 6.5,
- the static analysis in Section 6.6.1,
- the chosen metrics relevant for representing the encapsulated algorithms in Section 6.6.2,
- the metric aggregation process and its result in Section 6.6.3, where a single set of metrics is collected for every benchmark,
- the platform labeling and the combination of labels and metrics into instances in Section 6.6.4, and finally,
- the model training and its results in Section 6.7.

Once a prediction model is in place, new systems can be analyzed to predict their optimal execution platform. Figure 6.2 depicts the steps of applying a model to a new subject system (unknown to the trained model). To determine the optimal execution platform of a new system, all we have to do is calculate the same source code metrics (via static analysis) that we used for training the model, and let the model decide.

Figure 6.2. Usage of a previously built model on a new subject system

6.4 Benchmarks

For subject systems to train our models on, we used the algorithms found in two self-contained benchmark suites: *Parboil* and *Rodinia*. The Parboil suite [85] provides a combination of sequential, OpenCL, and OpenMP implementations for 11 programs. Rodinia [20] contains 18 benchmark programs with OpenCL and OpenMP implementations, but without the sequential equivalents. In this work, not all of these programs were measured, either because they had only OpenCL or only OpenMP implementations, but not both, or because their input sets were too complex. Note that during metric calculation (see Section 6.6), further systems needed to be skipped either because of a faulty build (inherent include errors) or because a single main file contained the whole logic of the program and therefore it could not be separated from the OpenCL specific overhead, causing large deviations in the computed metrics. The final number of systems that have both metric data and measurements are 7 and 8 for Parboil and Rodinia, respectively.

6.5 Measurements

In order to train our platform prediction models, we needed to obtain dynamic measurements for execution time, power consumption, and energy usage. We compiled the benchmarks with `g++` 4.8.2 using standard `-fopenmp` or `-lOpenCL` flags, and ran them on a platform built from 2 Intel Xeon E5-2695 v2 CPUs (30M Cache, 2.40 GHz), 8 × 8GB of DDR3 1600 MHz memory, a Supermicro X9DRG-QF mainboard, an AMD Radeon R9 290X VGA card, and an Alpha Data ADM-PCIE-7V3 FPGA card. Execution time could have been easily checked using software-based timers only. Power and energy, on the other hand, required a more sophisticated approach. So we additionally applied a universal hardware-extension solution and used our own open-source RMeasure library [48] that provides a unified API, hiding the implementation details.

Section 6.5.1 briefly overviews some of the already available performance and energy consumption measurement methods, while Section 6.5.2 introduces RMeasure and how it incorporates these methods. Next, Section 6.5.3 discusses measurement precision.

6.5.1 Measurement Methods

For the purpose of this overview, we classify methods either as *internal* – if the component under measurement can introspect its own behavior and expose that information, typically via performance counter registers – or as *external* – if some external hardware is needed for the measurement.

The most well-known internal measurement method is Intel's Running Average Power Limit (RAPL) [39] solution introduced in their Sandy Bridge microarchitecture, which gives access to both cycle count and energy consumption data for different physical domains – like sockets, core and uncore elements, and DRAM – through model-specific registers (MSRs). The two major GPU manufacturers, AMD and NVIDIA, both provide libraries and APIs to access similar hardware performance counters in their graphics processors. However, the publicly accessible AMD GPU Performance API [2] provides no access to power or energy consumption counters, while the NVIDIA Management Library (NVML) [66] is able to report the current power draw only for the high-end boards, like the Tesla K10/20/40 cards. Internal methods are not limited

to the x86 world only, as recent ARM cores have built-in performance monitoring units as well. However, up until the latest ARMv8 processors, these were performance-only, with no unified access to power data.

When internal methods are not available, – as it is evident from the above paragraph, this happens mostly for power usage monitoring – external solutions have to be applied. The physics behind most of such external metering methods is similar: a shunt resistor is inserted into the power line of a component, the voltage drop is measured on this resistor, and an instrumentation amplifier is used to make this voltage readable by conventional ADCs (such as the ones used by embedded devices, microcontrollers, or even external test equipments, e.g., oscilloscopes). Knowing the value of the resistor and the voltage of the power rail, the momentary power of the measured component is easily computed with the $P = U_{rail} \cdot (U_{drop}/R_{shunt})$ formula at any given sampling point, while integrating these results over time calculates the energy consumption. Some ARM devices have measurement points, to which an ARM Energy Probe [7] can be attached that works based on this concept and emits measurement result on a USB interface. Some accelerator cards are also instrumented for power measurements using this technique. E.g., the Xilinx Virtex VC709 FPGA development board has shunt resistors inserted into all internal power rails, and the resulting analog values are fed to a DC/DC converter controller chip, which reports power usage information digitally via the external Power Management Bus serial interface.

Since not all computation devices in our platform support a built-in power and energy measurement method, we designed and implemented a universal solution based on the above principles. We designed a printed circuit board (PCB) which can be conveniently placed inside the platform and holds the shunt resistor and amplifier needed for measuring a single computation device or power line. For each computation device, we used one of these circuits. To make the insertion of the circuits into the power lines the least intrusive and also reversible, we did not cut the wires of the power supplies, but we obtained different extension cords and modified them to be used with the measurement PCBs. For both CPU sockets, their 8-pin EPS12V power connectors are intercepted. For the GPU card, as it draws power both from the PCI-Express slot and from an additional PCI-Express power connector, both its rails are routed to a PCB (the former with the help of a PCI riser). Finally, we used a computer-controlled multi-channel measurement device, a PicoScope 4824 oscilloscope, to capture the output of the PCBs over time.

6.5.2 The RMeasure Library

The main goal of our RMeasure performance and energy monitoring library [48] is to provide a unified interface for retrieving performance and energy consumption data about the system, independent of the applied and/or available measurement methods. Thus, the interface handles built-in (e.g., performance counter-based) and external (e.g., shunt & oscilloscope-based) measurements alike, while hiding all implementation details.

The core interface of the library consists of just a few base classes, which represent the concept of a measurement method (e.g., RAPL counter-based or PicoScope-based) and stand for an actual measurement and its results. All supported measurement methods expose what components of the system they can measure and what kind of information they are able to provide. The components of the system are identified by their

HPP-DL component IDs [55]. The HPP-DL path notation provides a manufacturer- and architecture-independent abstraction layer to specify hardware components. The measured information can be an arbitrary combination of the following:

- energy consumption (in Joules),
- minimum, maximum, and average power (in Watts),
- elapsed time (also known as wall-clock time, in seconds), and
- time spent in kernel or in user mode (in seconds).

The API of RMeasure is intentionally simple; however, it can have several components working together under the hood in a full configuration. The main component of RMeasure exposes the public API, but there are certain tasks that needed to be separated from that main part. Specifically, if the external oscilloscope-based measurement method is enabled, the control service of the scope – whose responsibility is to control the oscilloscope via the PicoScope API [72], configure the sample rate and the channels, run in a gap-less continuous streaming mode, and retrieve the raw data – needs to be run on a separate unit, because processing the data requires significant CPU power that could distort the measurements if ran on the measured computer. The RAPL-based internal measurement method also has specific needs, since access to the machine specific registers requires root privileges. Therefore, it was useful for us to organize these into a separate service. The setup of a full measurement configuration is shown in Figure 6.3.

6.5.3 Measurement Precision

Since a service is constantly running in the background on the same computer as the measured code (at least for RAPL counters), it causes additional CPU load and therefore additional power consumption, which can have an effect on the precision of the measurements. To understand the introduced overhead, we took two sets of measurements, one using the service, and another with a slightly modified library setup where no services were running on the measured system. In the latter case, the application directly accessed the RAPL energy counters, thus requiring root permission. According to the results, the overhead on energy consumption, average power and running time were all below 5% on average.

6.6 Metric Extraction

In this section, we briefly describe the process of static analysis for static source code metric calculation. As outlined in Section 6.3, this static source code information is then used to predict the target execution platforms of future subject systems. We list all the selected metrics used in the machine learning algorithms as predictors, and also present how we aggregated the function level metrics to system level.

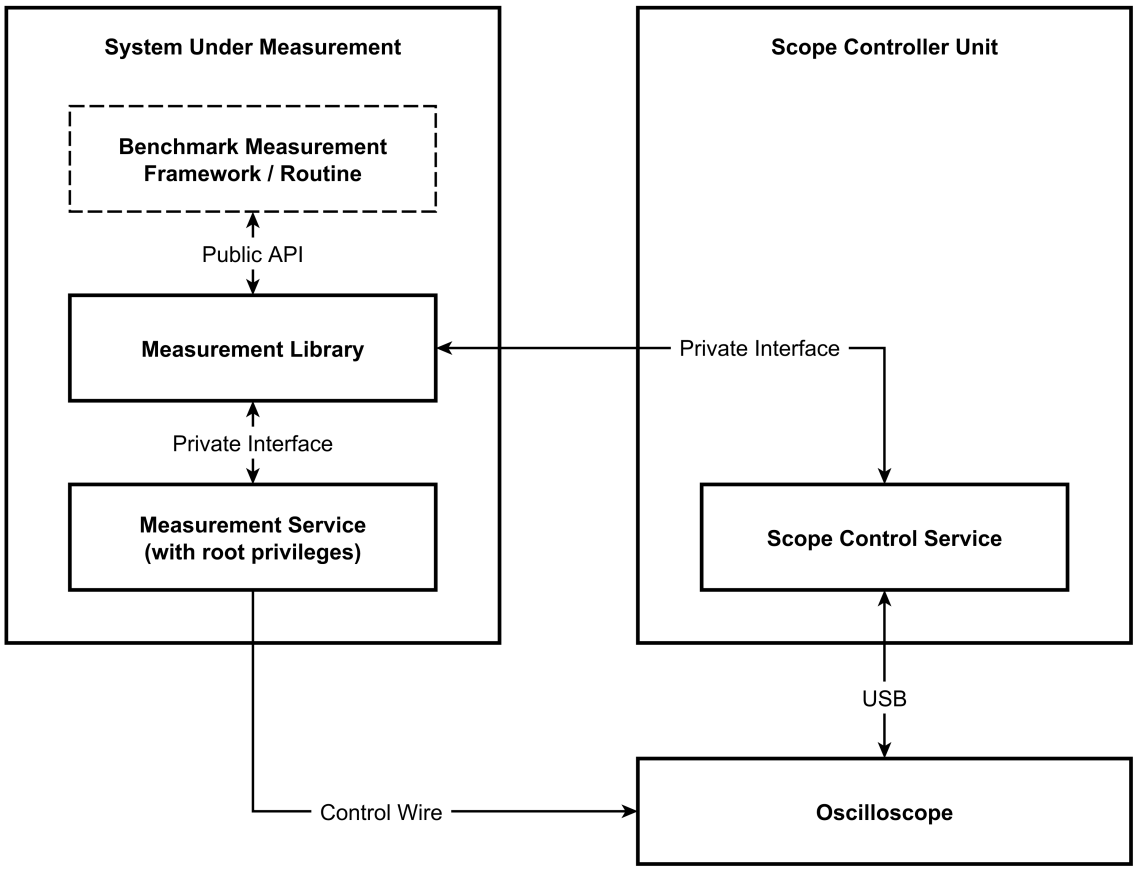

Figure 6.3. RMeasure library overview

6.6.1 Static Analysis

Metric calculation was performed – again – following the guidelines from Section 2.1 for both benchmark suites. Considering the procedural structure of the benchmark systems, we used function level metrics as the basis for further processing.

Note that the precision of the source code metrics could be improved by using block-level extraction, but that would require the manual annotation of every benchmark system (see Chapter 7). Moreover, the current approach does not use any dynamic information from the source code yet, metrics are static, and do not contemplate run time problems such as caching or memory allocation. That is because dynamic information is much more difficult to collect, but it should be definitely considered for further improvement of the prediction models. As a first step, we believe static information offers a good trade-off between efficient data collection and prediction accuracy.

6.6.2 Metric Definitions

The following metrics were computed and used as predictors for the classifications:

- **McCabe's cyclomatic complexity (McCC)** is defined as the number of decisions within the specified function plus 1, where each *if, for, while, do... while* and *?:* (conditional operator) counts once, each N-way *switch* counts N+1 times and each *try block* with N *catch blocks* counts N+1 times. (E.g., *else* does not increment the number of decisions.)

- **Nesting level (NL)** for a function is the maximum of the control structure depth. Only *if, switch, for, while* and *do... while* instructions are taken into account.
- **Nesting level else-if (NLE)** for a function is the maximum of the control structure depth. Only *if, switch, for, while* and *do... while* instructions are taken into account but *if... else if* does not increase the value.
- **Number of incoming invocations (NII)** for a function is the cardinality of the set of all functions that invoke this function.
- **Number of outgoing invocations (NOI)** for a function is the cardinality of the set of all function invocations in the function.
- **Logical lines of code (LLOC)** is the count of all non-empty, non-comment lines in a function.
- **Number of statements (NOS)** is the number of statements inside a given function.

Note that all of these metrics can be statically computed. Nevertheless, they can be used to predict dynamic behavior fairly well, as we will demonstrate in Section 6.7.

6.6.3 Metric Aggregation

The output of the static analysis is a set of metrics for every function in every implementation variant of every benchmark system. To aggregate these metrics into a system-level set for each benchmark system, first we combined the metrics of multiple functions per benchmark implementation. The method we used for aggregation in the current experiment setup is addition, but we already note that different, potentially more complex functions, perhaps even different ones per metric type might be applicable. However, although addition might not always be the best aggregation method for specific metrics (e.g., inheritance depth, or comment density), it is a natural and expressive choice for the metrics we use in this given scenario.

Next, we inspected the differences in the results per implementation variant for a given benchmark system. We noticed that while the sequential and OpenMP variant nearly always yielded the same – or negligibly different – metrics, the OpenCL variant was significantly larger. This turned out to be because:

- the main files (`main.cpp`, `main.cc`, `main.c`) of the OpenCL variants in every benchmark system increased the size and complexity because of the integration characteristics of OpenCL itself (the represented algorithms were not part of the main files), and
- the source code of the OpenCL variant frequently contained OpenCL specific headers and files which implemented functionality that the other variants assumed to be implicitly available.

By filtering out these "unnecessary files", the computed metrics "converged" to a single set and this supports that they really only represent the enclosed algorithm.

Benchmark	McCC	NL	NLE	NII	NOI	LLOC	NOS	TimeLabel	PowerLabel	EnergyLabel
Mri-Q	20	6	6	6	17	129	50	OCL-GPU	OCL-GPU	SEQ-CPU
Mri-Gridding	24	11	11	6	6	135	56	SEQ-CPU	SEQ-CPU	SEQ-CPU
Spmv	5	2	2	2	15	48	15	SEQ-CPU	SEQ-CPU	SEQ-CPU
Lbm	59	35	35	19	25	519	135	OCL-GPU	OCL-GPU	SEQ-CPU
Stencil	8	4	4	2	19	60	18	OCL-GPU	OCL-GPU	SEQ-CPU
Histo	13	5	5	3	10	97	33	SEQ-CPU	SEQ-CPU	SEQ-CPU
Cutcp	53	18	18	9	29	340	157	OCL-GPU	OCL-GPU	SEQ-CPU

Table 6.1. Training instances from the Parboil suite

Benchmark	McCC	NL	NLE	NII	NOI	LLOC	NOS	TimeLabel	PowerLabel	EnergyLabel
Streamcluster	249	66	66	49	160	1263	735	OCL-GPU	OCL-GPU	OMP-CPU
Leukocyte	672	134	134	99	260	2426	1627	OCL-GPU	OCL-GPU	OMP-CPU
Kmeans	100	22	22	9	53	487	240	OCL-GPU	OCL-GPU	OMP-CPU
Nw	21	3	3	3	14	104	58	OMP-CPU	OMP-CPU	OMP-CPU
Bfs	17	5	5	2	13	107	56	OMP-CPU	OMP-CPU	OMP-CPU
Pathfinder	20	6	6	3	10	87	52	OMP-CPU	OMP-CPU	OMP-CPU
Cfd	156	62	54	70	142	1424	776	OCL-GPU	OCL-GPU	OMP-CPU
Lavamd	85	6	6	7	17	370	128	OCL-GPU	OCL-GPU	OMP-CPU

Table 6.2. Training instances from the Rodinia suite

The remaining marginal differences were handled by taking the maximum of the values across the variants.

This way we got one single set of metrics for every benchmark system, capturing its characteristics.

6.6.4 Configuration Selection

After we have obtained measurements for each aspect (time, power, and energy) in each implementation variant (sequential, OpenMP on CPU and OpenCL on GPU) for each benchmark system, the question is *not* how fast (or energy efficient) a given algorithm will be, but on which execution platform will it be the fastest (or most energy efficient). To this end, we assigned three labels to each benchmark system – one for each aspect – denoting the best execution platform for each aspect. The possible platform labels are `SEQ-CPU`, `OMP-CPU` and `OCL-GPU` for the CPU-based sequential, CPU-based OpenMP and GPU-based OpenCL configurations, respectively.

The resulting `.csv` files for the systems in the two benchmark suites can be seen in Table 6.1 and Table 6.2. Note, that while Rodinia (Table 6.2) only has the two possible labels present in its table, Parboil (Table 6.1) could have all three labels, but `OMP-CPU` is not present there because it was never optimal.

These results were then written into `.arff` files with the last three label columns interpreted as nominal values. The `.arff` format (Attribute-Relation File Format) is the internal data representation format of Weka [34]. It is an ASCII text file that describes a list of instances sharing a set of attributes. These attributes can be strings, dates, numerical values and nominal values, the last of which can be used to represent class labels.

Tables 6.1 and 6.2 reveal that the optimal platform for the energy aspect was constant for both benchmark suites, and the optimal platform for the power and time aspects were so strongly correlated that they were always identical in our sample. Because of this, we chose not to consider energy labels, and to merge power and time labels into a single one for further experiments.

6.7 Results

In this section we describe the types of prediction models we built as well as how we built them. We also present the validation results of the models created by different machine learning algorithms. The results were validated with 4-fold cross-validations [6].

6.7.1 Machine Learning

Using the data shown in tables 6.1 and 6.2, we were able to run various machine learning algorithms to build models that can predict the platform labels based on the source code metrics. We performed the machine learning with the wrapper script shown in Listing 6.1.

Listing 6.1. Machine learning Weka script

```
for BENCH in parboil rodinia
do
  java -cp weka.jar weka.core.converters.CSVLoader -N 8 ../java/${BENCH}.csv > ${BENCH}.↩
      arff
  touch ${BENCH}.txt
  for CLASSIFIER in trees.J48 bayes.NaiveBayes functions.Logistic functions.SMO
  do
    for CLASS in 8 # possibly more
    do
      java -cp weka.jar weka.classifiers.${CLASSIFIER} -t ${BENCH}.arff -c ${CLASS} -i -x↩
          4 >> ${BENCH}.txt
    done
  done
done
```

6.7.2 Validation of the Models

Our first experiment – conducted using the J48 decision tree – produced 100% precision in both cases, which is not surprising as there is a clear division between the two possible labels using only a single metric. This means that we can select a metric and a corresponding threshold so that all the systems having a higher metric value than that will fall into one class, while systems with a lower metric value will fall into another class. The learning algorithms can find these values and achieve 100% precision. For Parboil, it was the NOI metric (over value 15 the label is `OCL-GPU`, otherwise it is `SEQ-CPU`), and for Rodinia, it was the NII metric (over value 3 the label is `OCL-GPU`, otherwise it is `OMP-CPU`). These simple separations are illustrated in Table 6.3 and Table 6.4. Note that for Rodinia, every other metric could have provided the same linear separation that NII did.

The final decision trees produced by the J48 algorithm for Parboil (left) and Rodinia (right) can be seen in Figure 6.4.

The Logistic regression model [52] – similarly to the decision tree – is perfectly accurate because of the above-mentioned clear separation based on numeric predictors.

Next, we tried the Naive Bayes classifier that yielded 71.4% precision for Parboil and 100% precision for Rodinia. The confusion matrix for the first case can be seen in Table 6.5. The upper left value shows how many instances were correctly identified as `OCL-GPU` and the upper right value shows the number of `SEQ-CPU` instances that were wrongly classified as `OCL-GPU`. Similarly, the lower right value is the number of

McCC	NL	NLE	NII	NOI	LLOC	NOS	Label
53	18	18	9	**29**	340	157	OCL-GPU
59	35	35	19	**25**	519	135	OCL-GPU
8	4	4	2	**19**	60	18	OCL-GPU
20	6	6	6	**17**	129	50	OCL-GPU
5	2	2	2	**15**	48	15	SEQ-CPU
13	5	5	3	**10**	97	33	SEQ-CPU
24	11	11	6	**6**	135	56	SEQ-CPU

Table 6.3. Clear separation of the Parboil benchmark suite by the NOI metric

McCC	NL	NLE	NII	NOI	LLOC	NOS	Label
672	134	134	**99**	260	2426	1627	OCL-GPU
156	62	54	**70**	142	1424	776	OCL-GPU
249	66	66	**49**	160	1263	735	OCL-GPU
100	22	22	**9**	53	487	240	OCL-GPU
85	6	6	**7**	17	370	128	OCL-GPU
21	3	3	**3**	14	104	58	OMP-CPU
20	6	6	**3**	10	87	52	OMP-CPU
17	5	5	**2**	13	107	56	OMP-CPU

Table 6.4. Clear separation of the Rodinia benchmark suite by the NII metric

correctly classified SEQ-CPUs, while the lower left is the number of OCL-GPUs that were – falsely – classified as SEQ-CPU.

Ultimately, we used a sequential minimal optimization function (SMO). It produced a 71.4% and a 75% precision for Parboil and Rodinia, respectively. The corresponding confusion matrices can be seen in Table 6.5 (Parboil, identical to the Bayes case) and Table 6.6 (Rodinia).

Although these findings can hardly be considered widely generalizable due to the small number of instances, the main result of this study is the streamlined process by which they were produced. With the described infrastructure in place, making the models more precise is largely just a matter of integrating more benchmark source code into the analysis.

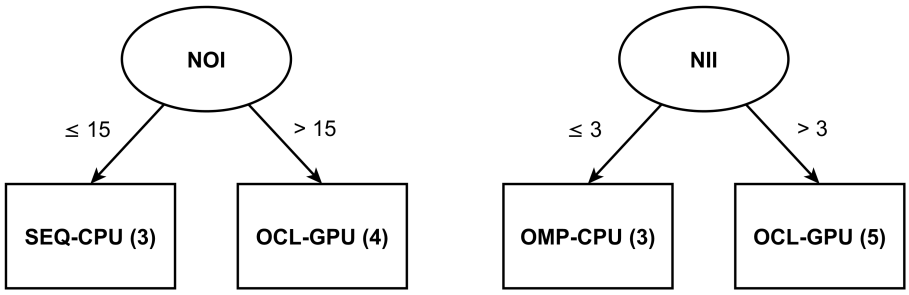

Figure 6.4. The final J48 decision trees for Parboil (left) and Rodinia (right)

		Predicted	
		OCL-GPU	SEQ-CPU
Measured	OCL-GPU	2	2
	SEQ-CPU	0	3

Table 6.5. The Bayes/SMO confusion matrix for Parboil

		Predicted	
		OCL-GPU	OMP-CPU
Measured	OCL-GPU	3	2
	OMP-CPU	0	3

Table 6.6. The SMO confusion matrix for Rodinia

6.8 Summary

The goal of this chapter was to present our work addressing the creation of prediction models that are able to automatically determine the optimal execution platform of a program (i.e., sequential on CPU, OpenCL on GPU, or OpenMP on CPU). To this end, we developed a highly generalizable and reusable methodology for producing such models. Moreover, these models do not depend on dynamic behavior information so they can be easily applied for classifying new subject systems.

Building these models required a set of algorithms that are each implemented on every relevant target platform. After thorough research, we found two independent benchmark suites containing multiple systems that fulfill this criterion. To be able to build the necessary models, we also needed measurements of the time, power, and energy consumption of the algorithms on these platforms. For this, we used a universal solution to measure the power and energy consumption of the hardware components. We then successfully applied our methodology on these systems to create prediction models based on different machine learning approaches, using source code metrics as predictors.

The resulting models are qualitative, which means that they can predict the optimal execution platform, but not how much better it is compared to the other alternatives. Nevertheless, since all the necessary performance information is available, the methodology will be later expanded to produce quantitative models that will make it possible to estimate even the differences.

Overall, we consider the results of this chapter encouraging. Despite the small number of subject systems, we were able to demonstrate that statically computed metrics are appropriate and useful for platform selection. For example, some of the preliminary models we built reached a 100% accuracy in inferring the optimal execution target. The models are promising by themselves, but we feel that our main result here is the methodology behind their creation. We now have a flexible, expandable and configurable infrastructure in place, and the generalizability of its output models depend only on the number of initial benchmark systems we use for training.

"Every line is the perfect length if you don't measure it."

— Marty Rubin

7 Quantitative Prediction Models

7.1 Overview

The previous chapter deals with dynamic platform selection to a certain degree, but our ultimate goal in this domain was creating quantitative models – i.e., not only predicting the optimal platform, but also estimating the expected change in performance. Other obvious opportunities for improvement were extending the small number of benchmarks, capturing even more static qualities of the core algorithms (or "kernels") they contain, and even pinpointing those kernels more precisely.

To this end, we enhanced our former approach to aim for gain ratios. Additionally, we implemented numerous new source code metrics, – relying heavily on the Milepost GCC compiler [29] – refined kernel identification with manual benchmark modifications and block-level metric extraction, and even included FPGAs as a new platform option. Although we also tried to increase the number of benchmarks, this increase was mostly counteracted by the exclusions the introduction of the FPGA platform brought. However, the benchmarks, paired with the extended study dimensions detailed in Section 7.5.1, still lead to significantly bigger learning tables.

The research question we aim to answer is the following:

Research Question: *Can the performance gain of porting an algorithm to another computation element be predicted using only static information?*

In order to encourage further research in this area, we provide the source code of our modified benchmarks [12] along with their static metrics and dynamic measurements [11].

The rest of the chapter is organized in the following manner: As the related work still applies from Chapter 6, the next section immediately focuses on the relevant changes in methodology. Then, Section 7.3 introduces the modified benchmarks, while in Section 7.4 we describe our extensions to the static metric extraction in detail. In Section 7.5 we show the results that we have achieved. Lastly, in Section 7.6 we draw our conclusions.

7.2 Methodology

This section contains the detailed description of the differences and additions to our methodology from Chapter 6 that enable us to build quantitative prediction models. Using these changes, our models will be able to predict – quite adequately – not only the computing unit that allows the fastest or most energy efficient execution of a given program, but also the amount of improvement in terms of performance, power, and energy consumption that can be expected. For even finer grained measurements, we only considered the core of the algorithms, the computing kernels represented in each benchmark program and none of their preparation steps, e.g., OpenCL platform or device initializations, etc. We achieved this by "tagging" the appropriate parts of the benchmarks with a special `STATIC_BEGIN` – `STATIC_END` C/C++ macro pair, which required extensive manual source code comprehension and modification. We also used this tagging approach to separate the dynamic measurements into initialization/cleanup, data transfer, and kernel execution stages with differently parametrized `DYNAMIC_BEGIN` – `DYNAMIC_END` macros.

From that point forward, however, the models are built following the same overall concept we outlined in Section 6.3. The only difference from an abstract perspective is that the computed source code metrics are now combined with gain ratios instead of "best platform" class labels. Each of the notable deviations will be detailed in its dedicated section:

- The benchmark set extension in Section 7.3,
- the updated static analysis in Section 7.4.1,
- the significantly augmented set of metrics in Section 7.4.2,
- the refined metric aggregation process and its result in Section 7.4.3,
- the combination of different dynamic measurements into gain ratios in Section 7.5.1, and finally,
- the model training and its results in sections 7.5.2 and 7.5.3.

7.3 Benchmarks

For this study, the already familiar Parboil and Rodinia benchmark suites were joined by *PolyBench/ACC* [33], which is an extended version of PolyBench [74], and contains 29 programs in multiple implementations. We measured a subset of these three benchmark suites, as some programs were excluded either because of dynamic problems – they were not implemented in all necessary languages or could not be executed on all necessary platforms – or static issues like a faulty build or inherent include errors. The final number of systems that have both metric data and measurements (for both CPU, GPU, and FPGA) are 3 for Parboil (mri-q, spmv, and stencil), 4 for Rodinia (bfs, hotspot, lavaMD, and nn) and 9 for PolyBench/ACC (atax, bicg, convolution-2d, doitgen, gemm, gemver, gesummv, jacobi-2d-imper, and mvt).

7.4 Metric Extraction

7.4.1 Static Analysis

For metric calculation, we – predictably – ran our static code analysis toolchain [26] on all three benchmark suites. However, instead of using method or function level granularity for metrics as "atoms", this time we implemented block level metrics to isolate the characteristics of the kernels and exclude every "wrapper" and "initializer" functionality. We calculated these block level metrics by analyzing only the appropriate source code parts between `STATIC_BEGIN` and `STATIC_END` macros. For static analysis, these macros were resolved to `[[rpr::kernel]]{` and `}` respectively, thereby enclosing the relevant source code in a block that has a REPARA [49] C++ attribute attached to it. The ability to use arbitrary attributes is a new feature in C++11 that makes this "tagging" possible.

Note that the current approach still does not use any dynamic information from the source code yet, metrics are static, and do not contemplate runtime problems such as memory aliasing, caching and memory allocation.

7.4.2 Metric Definitions

The metrics we computed and used as predictors for the classifications and regressions are listed below. It should be noted that the word "block" may refer to either basic blocks (which is a control flow concept) or the above-mentioned tagged source code blocks. To help differentiate between the meanings, we always add a "(tagged)" prefix in the ambiguous cases. Also note that metrics starting with "ft" are adopted directly from the feature list of the Milepost GCC compiler [29].

- **Lines of code (LOC)** is the count of every line in a block.
- **Logical lines of code (LLOC)** is the count of all non-empty, non-comment lines in a block.
- **Nesting level (NL)** for a block is the maximum of the control structure depth. Only *if, switch, for, while and do... while* instructions are taken into account.
- **Nesting level else-if (NLE)** for a block is the maximum of the control structure depth. Only *if, switch, for, while and do... while* instructions are taken into account but *if... else if* does not increase the value.
- **McCabe's cyclomatic complexity (McCC)** is defined as the number of decisions within the specified block plus 1, where each *if, for, while, do... while and ?: (conditional operator)* counts once, each N-way *switch* counts N+1 times and each *try* with N *catch*es counts N+1 times. (E.g., *else* does not increment the number of decisions.)
- **Number of statements (NOS)** is the number of statements inside a block.
- **Number of outgoing invocations (NOI)** for a block is the number of all function invocations inside it.

- **Loop nesting level (LNL)** is the maximum loop depth inside the block. (The same as NL, but without the *if*s, *switch*es, *try*s and ternary operators). We also computed LNL1, LNL2, and LNL3 that contain the number of loops that were at depths 1, 2, and 3, respectively.
- **Number of expressions (EXP)** is the number of expressions in the block.
- **Number of array accesses (ARR)** is the number of array subscript expressions in the block. Also, ARR% is defined as the ratio ARR/EXP.
- **Number of multiplications (MUL)** is the number of multiplications (* or *=) in the block. Also, MUL% is defined as the ratio MUL/EXP.
- **Number of additions (ADD)** is the number of additions (+ or +=) in the block. Also, ADD% is defined as the ratio ADD/EXP.
- **ft1** is the number of basic blocks in the (tagged) block.
- **ft2** is the number of basic blocks with a single successor.
- **ft3** is the number of basic blocks with two successors.
- **ft4** is the number of basic blocks with more than two successors.
- **ft5** is the number of basic blocks with a single predecessor.
- **ft6** is the number of basic blocks with two predecessors.
- **ft7** is the number of basic blocks with more than two predecessors.
- **ft8** is the number of basic blocks with a single predecessor and a single successor.
- **ft9** is the number of basic blocks with a single predecessor and two successors.
- **ft10** is the number of basic blocks with a two predecessors and one successor.
- **ft11** is the number of basic blocks with two successors and two predecessors.
- **ft12** is the number of basic blocks with more than two successors and more than two predecessors.
- **ft13** is the number of basic blocks with number of instructions less than 15.
- **ft14** is the number of basic blocks with number of instructions in the interval [15, 500].
- **ft15** is the number of basic blocks with number of instructions greater than 500.
- **ft21** is the number of assignment instructions in the (tagged) block.
- **ft22** is the number of binary integer operations in the (tagged) block.
- **ft23** is the number of binary floating point operations in the (tagged) block.
- **ft25** is the average number of instructions in basic blocks.

- **ft33** is the number of switch instructions in the (tagged) block.
- **ft34** is the number of unary operations in the (tagged) block.
- **ft40** is the number of assignment instructions with the right operand as an integer constant in the (tagged) block.
- **ft41** is the number of binary operations with one of the operands as an integer constant in the (tagged) block.
- **ft42** is the number of calls with the number of arguments greater than 4.
- **ft45** is the number of calls that return an integer.
- **ft46** is the number of occurrences of integer constant zero.
- **ft48** is the number of occurrences of integer constant one.

Another, slightly different metric is the **Input size**, i.e., the relative size of the input the encapsulated algorithm will process. We categorized input sizes into five possible bins: mini, small, medium, large and extra large. Note that these were already given with our subject benchmarks and as "small" is relative to the algorithm in question, we cannot give exact thresholds.

7.4.3 Metric Aggregation

The output of the static analysis is a set of metrics for every block in the sequential platform version for every benchmark system – except for the input size, which is already system-level. These represent the captured algorithms and are the correct basis for further study because the dynamic measurements also express how much improvement can be expected compared to the sequential platform.

To aggregate these metrics to a system-level set for each benchmark, we combined the metrics of multiple blocks. The method of combination is now customizable per metric and we chose the most naturally expressive option for each:

- addition minus one for McCC (the minus one accounts for the default execution path the separate block gets on its own, and is now not needed),
- maximization for NL, NLE and LNL,
- recalculation for averages (i.e., their numerators and denominators are aggregated separately and the average is computed again at the end), and lastly
- addition for the others, as they are all counts of different occurrences.

This way, we got one single set of metric values for every benchmark, capturing many of its characteristics.

7.5 Results

In this section, we describe the prediction models we built using the static and dynamic data outlined above. We also present the validation results of the models created by different machine learning algorithms.

7.5.1 Training Instances

After we have obtained measurements for each aspect (time, average power, energy), on each platform (sequential, OpenCL on CPU, OpenCL on GPU and OpenCL on FPGA), for each code region (initialization/cleanup, data transfer or kernel execution), and for each input size (mini, small, medium, large or extra large) of each benchmark system, the question is how fast (or energy efficient) a given algorithm will be.

But improvement is a characteristic that is hard to describe in absolute terms because static metrics alone are not expected to fully describe the dynamic behavior of a program. For example, it might happen that two separate programs yield the same source code metric values, but they have significantly different runtimes. If our models learned from one of them that migrating to OpenCL on GPU can produce a shorter runtime, that would not mean anything unless we also knew how much of an improvement that decrease is compared to its original runtime. This is why instead of absolute measures (like seconds or Joules) we used relative values (ratios).

So, after aggregating the source code metrics (detailed in Section 7.4.3) we converted the dynamic measurements to the above-mentioned ratios that could be classes in a machine learning experiment. We did so by dividing the values measured on a parallel computation unit (e.g., the runtime of a kernel on a GPU) by their original, sequential counterparts. Values below one expressed improvement, while values greater than one indicated deterioration. We calculated these ratios for every input size of every benchmark and then combined them with the static metrics to finalize our training databases, each containing over 50 instances.

We also experimented with different measurement aggregation methods that affect *what* exactly do we consider the power/energy consumption of a given program execution. One way is to take only the values of the chosen hardware itself into account (denoted by "Single" in later tables). Another is to always add the CPU's measurements to the total, since there needs to be a CPU in the system to send tasks to the selected accelerator (denoted by "With CPU"). Lastly, we can view the system as a whole and sum the total power/energy consumption that the different hardware components produced (denoted by "All").

The updated tagging provides yet another possible dimension to the study: do we predict the improvement for the kernel only ("Kernel") or for the whole ("Full") program (including initializations and data transfers)? We could also create training datasets for the separated initialization/cleanup ("Init") and data transfer ("Transfer") phases.

This results in a training set for each Phase-Platform-Measurement aggregation method-Aspect tuple. As an example, part[1] of the Kernel-Single-GPU-Time training instances can be seen in Table 7.1.

7.5.2 Machine Learning

Using the datasets like the one shown in Table 7.1, we were able to run various machine learning algorithms to build models that can predict the gain ratios based on the source code metrics.

We have experimented with both classification and regression algorithms. While the regression models were trained for the continuous improvement ratios, the classifi-

[1]The full tables are part of an online appendix [11].

Benchmark	LOC	LLOC	NL	NLE	McCC	...	Input Size	Ratio
poly_atax	16	14	4	2	2	...	1	1250.8500
poly_atax	16	14	4	2	2	...	2	22.9647
poly_atax	16	14	4	2	2	...	3	1.0945
poly_atax	16	14	4	2	2	...	4	1.0689
poly_bicg	17	15	3	2	2	...	1	5287.3750
poly_bicg	17	15	3	2	2	...	2	40.5465
poly_bicg	17	15	3	2	2	...	3	1.5096
poly_bicg	17	15	3	2	2	...	4	1.4827
poly_conv2d	16	12	2	2	2	...	1	1844.5000
poly_conv2d	16	12	2	2	2	...	2	2.2655
poly_conv2d	16	12	2	2	2	...	3	0.2522
poly_conv2d	16	12	2	2	2	...	4	0.7393
poly_conv2d	16	12	2	2	2	...	5	0.6821
...	...	...	...	...	...	...	...	...

Table 7.1. Training instances with Kernel-Single-GPU-Time improvement ratios

cation algorithms required classes. Thus, we have applied a discretizing preprocessing filter to our training data to divide the ratios into 5 (and 3) bins, or "improvement categories." These bins ranged from "large deterioration" to "large improvement" with automatically computed thresholds. This discretization and bin selection represents a middle ground between our previous approach of only choosing the best platform and the regression algorithms that aim to exactly estimate improvement.

7.5.3 Validation of the Models

In the following, we show the results of the experiments where we applied our full set of source code metrics along with the input sizes as predictors. The accuracy values we achieved for the Full, Kernel, Init and Transfer phases are shown in tables 7.2, 7.3, 7.4 and 7.5, respectively. Each of these tables has three layers of headers for the measurement aggregation method (Single, With CPU, All), the target platform (CPU, GPU or FPGA), and the measured dynamic aspect (**T**ime, **P**ower or **E**nergy), while the rows show how each tested algorithm performed on the corresponding problem. The rows are separated into three groups for regression algorithms, 5 bin and 3 bin classifications. All three row groups start with Weka's ZeroR algorithm that can be considered a baseline for the given problem, i.e., algorithms that outperform this accuracy are said to have predictive power in this context. For easier visual parsing, the cells of the tables are colored with five different shades to signal higher precision.

Note that regression cells represent the absolute values of the correlation coefficients from the cross-validations, while the classification values are percentages of the correctly classified instances. (We use absolute values because in this case we are interested in the strength of the correlations, not their direction.) Also note that random choice on a 5 or 3 bin classification would yield 20% or 33.33% accuracy, respectively (which the vast majority of classifiers still outperform), but the baseline can be (and is) worse than random choice as there the model always picks the most represented class in the *training* data, which guarantees nothing in the *test* data. For example, if a dataset with 7 blacks and 3 whites as its classes were separated into training and test datasets where each training data is black and each test data is white, ZeroR would always predict black based on the training data and it would be 0% accurate on the test set. And

although cross-validation repeats this training/test separation n times, the average of the results could still be lower than random choice depending on the separations and the starting distribution of the classes.

Globally, 886 of our 1404 models produced meaningful (i.e., at least 5%) improvement over the baseline performance, and 867 of these used at least two predictor metrics. (This second check was implemented to root out a few models encountered during random manual validation that were simple constants or relied only on InputSize.) Additionally, we collected statistics for the most frequently used metrics in the models. The top ten start with the all important InputSize, – used in 98% of the models – followed by ARR%, LOC, ft25 (average number of instructions in basic blocks), ft48 (number of occurrences of integer constant one), ft7 (number of basic blocks with more than two predecessors), EXP, ARR, LNL1 and MUL, respectively.

To gain further insight into the effect the different dimensions (i.e., phase, aspect, etc.) have on prediction accuracy, we also computed model success distributions for each dimension separately. Note that in order to make this discussion more concise, x/y/z will mean that "out of all possible z models, y managed to outperform the baseline by at least 5%, x of which used at least two predictors from the available set". These could be thought of as "better/good/count".

- Source code phase
  - Initialization/Cleanup: 212/220/351
  - Kernel execution: 224/224/351
  - Data transfer: 206/206/351
  - Full: 225/236/351
- Measurement aggregation method
  - Single: 295/301/468
  - With CPU: 286/292/468
  - All: 286/293/468
- Execution platform
  - CPU: 269/269/468
  - GPU: 325/336/468
  - FPGA: 273/281/468
- Measurement aspect
  - Time: 282/291/468
  - Power: 302/304/468
  - Energy: 283/291/468
- Machine learning technique
  - Regression: 58/77/540
  - 3bin classification: 405/405/432
  - 5bin classification: 404/404/432

	Single									With CPU									All								
	CPU			GPU			FPGA			CPU			GPU			FPGA			CPU			GPU			FPGA		
Algorithm	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**
ZeroR	0.65	0.45	0.65	0.39	0.45	0.42	0.65	0.39	0.65	0.65	0.45	0.65	0.39	0.37	0.40	0.65	0.35	0.65	0.65	0.44	0.65	0.39	0.37	0.40	0.65	0.38	0.65
LinReg	0.65	0.45	0.65	0.39	0.19	0.42	0.65	0.39	0.65	0.65	0.45	0.65	0.39	0.37	0.40	0.65	0.35	0.65	0.65	0.44	0.65	0.39	0.30	0.40	0.65	0.38	0.65
Mult.Perc.	0.01	0.37	0.01	0.12	0.66	0.44	0.09	0.07	0.10	0.01	0.37	0.01	0.12	0.42	0.15	0.09	0.34	0.12	0.01	0.45	0.03	0.12	0.62	0.15	0.09	0.15	0.10
REPTree	0.10	0.45	0.12	0.63	0.26	0.83	0.23	0.36	0.25	0.10	0.45	0.12	0.63	0.39	0.70	0.23	0.05	0.02	0.10	0.39	0.12	0.63	0.26	0.70	0.23	0.08	0.25
M5P	0.30	0.13	0.32	0.65	0.50	0.79	0.47	0.10	0.48	0.30	0.13	0.32	0.65	0.17	0.72	0.47	0.13	0.48	0.30	0.12	0.32	0.65	0.28	0.71	0.47	0.01	0.48
SMOreg	0.06	0.25	0.08	0.15	0.70	0.10	0.04	0.37	0.04	0.06	0.25	0.08	0.15	0.65	0.15	0.04	0.20	0.04	0.06	0.30	0.08	0.15	0.64	0.16	0.04	0.20	0.04
ZeroR	16.28	16.28	16.28	11.11	11.11	11.11	0.00	0.00	0.00	16.28	16.28	16.28	11.11	11.11	11.11	0.00	0.00	0.00	16.28	16.28	16.28	11.11	11.11	11.11	0.00	0.00	0.00
J48	37.21	27.91	58.14	53.33	60.00	53.33	34.48	27.59	37.93	37.21	27.91	58.14	53.33	51.11	60.00	34.48	41.38	48.28	37.21	25.58	58.14	53.33	28.89	60.00	34.48	34.48	41.38
NaiveBayes	25.58	30.23	20.93	22.22	28.89	22.22	17.24	37.93	24.14	25.58	30.23	20.93	22.22	17.78	17.78	17.24	31.03	20.69	25.58	30.23	20.93	22.22	20.00	17.78	17.24	24.14	13.79
Logistic	27.91	23.26	30.23	51.11	31.11	40.00	27.59	37.93	31.03	27.91	23.26	30.23	51.11	33.33	46.67	27.59	31.03	24.14	27.91	27.91	30.23	51.11	33.33	46.67	27.59	24.14	24.14
SMO	39.53	27.91	27.91	40.00	28.89	42.22	27.59	41.38	17.24	39.53	27.91	27.91	40.00	26.67	44.44	27.59	17.24	3.45	39.53	23.26	27.91	40.00	17.78	44.44	27.59	13.79	10.34
ZeroR	23.26	23.26	23.26	22.22	22.22	22.22	34.48	34.48	34.48	23.26	23.26	23.26	22.22	22.22	22.22	34.48	34.48	34.48	23.26	23.26	23.26	22.22	22.22	22.22	34.48	34.48	34.48
J48	65.12	41.86	65.12	71.11	57.78	60.00	55.17	37.93	55.17	65.12	41.86	65.12	71.11	57.78	71.11	55.17	44.83	55.17	65.12	46.51	65.12	71.11	44.44	71.11	55.17	34.48	55.17
NaiveBayes	32.56	37.21	32.56	33.33	40.00	40.00	58.62	41.38	58.62	32.56	37.21	32.56	33.33	44.44	33.33	58.62	41.38	58.62	32.56	37.21	32.56	33.33	46.67	33.33	58.62	37.93	58.62
Logistic	34.88	51.16	34.88	66.67	46.67	60.00	62.07	48.28	62.07	34.88	51.16	34.88	66.67	60.00	66.67	62.07	41.38	62.07	34.88	53.49	34.88	66.67	57.78	66.67	62.07	41.38	62.07
SMO	39.53	46.51	39.53	53.33	60.00	42.22	55.17	55.17	55.17	39.53	46.51	39.53	53.33	46.67	53.33	55.17	55.17	55.17	39.53	44.19	39.53	53.33	53.33	53.33	55.17	41.38	55.17

Table 7.2. Full prediction accuracies

	Single									With CPU									All								
	CPU			GPU			FPGA			CPU			GPU			FPGA			CPU			GPU			FPGA		
Algorithm	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**
ZeroR	0.48	0.46	0.48	0.36	0.46	0.37	0.65	0.41	0.65	0.48	0.46	0.48	0.36	0.42	0.36	0.65	0.41	0.65	0.48	0.47	0.48	0.36	0.44	0.35	0.65	0.41	0.65
LinReg	0.48	0.46	0.48	0.36	0.32	0.37	0.65	0.41	0.65	0.48	0.46	0.48	0.36	0.42	0.36	0.65	0.41	0.65	0.48	0.47	0.48	0.36	0.23	0.35	0.65	0.41	0.65
Mult.Perc.	0.01	0.13	0.02	0.63	0.63	0.56	0.07	0.16	0.16	0.01	0.13	0.02	0.63	0.46	0.67	0.07	0.39	0.13	0.01	0.14	0.02	0.63	0.50	0.61	0.07	0.48	0.06
REPTree	0.08	0.10	0.08	0.74	0.27	0.70	0.19	0.18	0.21	0.08	0.10	0.08	0.74	0.00	0.77	0.19	0.07	0.13	0.08	0.47	0.08	0.74	0.21	0.74	0.19	0.19	0.21
M5P	0.01	0.04	0.01	0.67	0.05	0.67	0.43	0.16	0.44	0.01	0.04	0.01	0.67	0.02	0.64	0.43	0.16	0.44	0.01	0.04	0.01	0.67	0.11	0.67	0.43	0.16	0.44
SMOreg	0.04	0.07	0.05	0.00	0.53	0.00	0.07	0.12	0.07	0.04	0.07	0.05	0.00	0.62	0.01	0.07	0.04	0.06	0.04	0.12	0.05	0.00	0.63	0.01	0.07	0.08	0.06
ZeroR	16.28	16.28	16.28	11.11	11.11	11.11	0.00	0.00	0.00	16.28	16.28	16.28	11.11	11.11	11.11	0.00	0.00	0.00	16.28	16.28	16.28	11.11	11.11	11.11	0.00	0.00	0.00
J48	32.56	18.60	44.19	48.89	35.56	46.67	68.97	20.69	68.97	32.56	18.60	44.19	48.89	22.22	48.89	68.97	17.24	68.97	32.56	34.88	37.21	48.89	48.89	48.89	68.97	48.28	68.97
NaiveBayes	18.60	20.93	13.95	26.67	31.11	22.22	34.48	24.14	34.48	18.60	20.93	13.95	26.67	20.00	33.33	34.48	17.24	34.48	18.60	13.95	6.98	26.67	28.89	33.33	34.48	27.59	34.48
Logistic	41.86	25.58	44.19	33.33	37.78	40.00	41.38	27.59	41.38	41.86	25.58	44.19	33.33	33.33	35.56	41.38	24.14	41.38	41.86	25.58	25.58	33.33	31.11	35.56	41.38	27.59	41.38
SMO	32.56	20.93	20.93	33.33	28.89	40.00	34.48	20.69	34.48	32.56	20.93	20.93	33.33	20.00	42.22	34.48	17.24	34.48	32.56	23.26	20.93	33.33	28.89	42.22	34.48	10.34	34.48
ZeroR	30.23	23.26	30.23	22.22	22.22	22.22	34.48	34.48	34.48	30.23	23.26	30.23	22.22	22.22	22.22	34.48	34.48	34.48	30.23	23.26	30.23	22.22	22.22	22.22	34.48	34.48	34.48
J48	55.81	32.56	55.81	53.33	62.22	53.33	55.17	48.28	55.17	55.81	32.56	55.81	53.33	46.67	48.89	55.17	41.38	55.17	55.81	32.56	55.81	53.33	35.56	48.89	55.17	37.93	55.17
NaiveBayes	37.21	39.53	37.21	44.44	44.44	57.78	34.48	31.03	34.48	37.21	39.53	37.21	44.44	40.00	48.89	34.48	17.24	34.48	37.21	39.53	37.21	44.44	42.22	48.89	34.48	17.24	34.48
Logistic	65.12	37.21	65.12	48.89	55.56	53.33	68.97	37.93	68.97	65.12	37.21	65.12	48.89	57.78	53.33	68.97	27.59	68.97	65.12	37.21	65.12	48.89	53.33	53.33	68.97	34.48	68.97
SMO	53.49	46.51	53.49	48.89	55.56	51.11	44.83	41.38	44.83	53.49	46.51	53.49	48.89	40.00	55.56	44.83	17.24	44.83	53.49	46.51	53.49	48.89	44.44	55.56	44.83	27.59	44.83

Table 7.3. Kernel prediction accuracies

	Single									With CPU									All								
	CPU			GPU			FPGA			CPU			GPU			FPGA			CPU			GPU			FPGA		
Algorithm	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**
ZeroR	0.43	0.41	0.43	0.37	0.46	0.37	0.56	0.42	0.56	0.43	0.41	0.43	0.37	0.35	0.37	0.56	0.51	0.56	0.43	0.41	0.43	0.37	0.34	0.37	0.56	0.53	0.56
LinReg	0.43	0.41	0.43	0.14	0.32	0.07	0.56	0.38	0.56	0.43	0.41	0.43	0.14	0.26	0.13	0.56	0.22	0.56	0.43	0.41	0.43	0.14	0.18	0.13	0.56	0.53	0.56
Mult.Perc.	0.07	0.11	0.06	0.03	0.11	0.02	0.06	0.76	0.07	0.07	0.11	0.06	0.03	0.07	0.03	0.06	0.62	0.06	0.07	0.16	0.07	0.03	0.05	0.02	0.06	0.74	0.06
REPTree	0.42	0.00	0.42	0.07	0.08	0.07	0.56	0.57	0.56	0.42	0.00	0.42	0.07	0.17	0.07	0.56	0.49	0.56	0.42	0.18	0.42	0.07	0.17	0.07	0.56	0.81	0.56
M5P	0.35	0.16	0.35	0.03	0.21	0.01	0.70	0.61	0.70	0.35	0.16	0.35	0.03	0.11	0.04	0.70	0.74	0.70	0.35	0.10	0.35	0.03	0.12	0.04	0.70	0.69	0.69
SMOreg	0.01	0.30	0.01	0.04	0.32	0.10	0.08	0.77	0.08	0.01	0.30	0.01	0.04	0.06	0.09	0.08	0.70	0.08	0.01	0.32	0.01	0.04	0.06	0.09	0.08	0.77	0.08
ZeroR	19.23	19.23	19.23	18.52	18.52	18.52	6.25	0.00	6.25	19.23	19.23	19.23	18.52	18.52	18.52	6.25	9.38	3.13	19.23	19.23	19.23	18.52	18.52	18.52	6.25	9.38	3.13
J48	32.69	28.85	34.62	38.89	44.44	38.89	37.50	28.13	37.50	32.69	28.85	34.62	38.89	20.37	44.44	37.50	9.38	40.63	32.69	34.62	32.69	38.89	22.22	44.44	37.50	25.00	40.63
NaiveBayes	28.85	23.08	25.00	25.93	18.52	16.67	34.38	25.00	34.38	28.85	23.08	25.00	25.93	18.52	20.37	34.38	9.38	28.13	28.85	11.54	23.08	25.93	16.67	24.07	34.38	9.38	28.13
Logistic	42.31	21.15	46.15	35.19	42.59	37.04	50.00	31.25	50.00	42.31	21.15	46.15	35.19	24.07	25.93	50.00	25.00	46.88	42.31	26.92	36.54	35.19	27.78	31.48	50.00	25.00	46.88
SMO	42.31	13.46	50.00	22.22	29.63	20.37	31.25	9.38	31.25	42.31	13.46	50.00	22.22	20.37	22.22	31.25	6.25	50.00	42.31	21.15	57.69	22.22	14.81	20.37	31.25	12.50	50.00
ZeroR	34.62	28.85	28.85	25.93	25.93	25.93	31.25	31.25	31.25	34.62	28.85	28.85	25.93	25.93	25.93	31.25	31.25	31.25	34.62	28.85	28.85	25.93	25.93	25.93	31.25	31.25	31.25
J48	53.85	50.00	59.62	53.70	44.44	62.96	81.25	62.50	81.25	53.85	50.00	59.62	53.70	37.04	53.70	81.25	37.50	81.25	53.85	38.46	59.62	53.70	44.44	57.41	81.25	43.75	81.25
NaiveBayes	46.15	40.38	44.23	27.78	27.78	25.93	59.38	50.00	59.38	46.15	40.38	44.23	27.78	42.59	42.59	59.38	50.00	59.38	46.15	38.46	44.23	27.78	53.70	37.04	59.38	50.00	59.38
Logistic	50.00	53.85	53.85	51.85	61.11	61.11	65.63	65.63	65.63	50.00	53.85	53.85	51.85	53.70	53.70	65.63	50.00	65.63	50.00	48.08	53.85	51.85	57.41	55.56	65.63	46.88	65.63
SMO	63.46	46.15	65.38	44.44	57.41	53.70	65.63	56.25	65.63	63.46	46.15	65.38	44.44	46.30	46.30	65.63	43.75	65.63	63.46	40.38	65.38	44.44	48.15	46.30	65.63	43.75	65.63

Table 7.4. Initialization/cleanup prediction accuracies

	Single									With CPU									All								
	CPU			GPU			FPGA			CPU			GPU			FPGA			CPU			GPU			FPGA		
Algorithm	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**	**T**	**P**	**E**
ZeroR	0.44	0.62	0.44	0.41	0.36	0.38	0.53	0.50	0.53	0.44	0.62	0.44	0.41	0.38	0.40	0.53	0.53	0.54	0.44	0.60	0.44	0.41	0.38	0.40	0.53	0.52	0.53
LinReg	0.02	0.16	0.04	0.05	0.57	0.24	0.53	0.50	0.53	0.02	0.16	0.04	0.05	0.78	0.12	0.53	0.53	0.54	0.02	0.46	0.00	0.05	0.81	0.11	0.53	0.52	0.53
Mult.Perc.	0.05	0.19	0.10	0.02	0.66	0.38	0.22	0.09	0.16	0.05	0.19	0.10	0.02	0.75	0.12	0.22	0.12	0.44	0.05	0.15	0.07	0.02	0.77	0.11	0.22	0.11	0.17
REPTree	0.15	0.61	0.16	0.08	0.56	0.32	0.53	0.46	0.53	0.15	0.61	0.16	0.08	0.20	0.15	0.53	0.53	0.27	0.15	0.63	0.16	0.08	0.15	0.14	0.53	0.52	0.18
M5P	0.19	0.43	0.18	0.14	0.47	0.36	0.04	0.02	0.05	0.19	0.43	0.18	0.14	0.80	0.19	0.04	0.05	0.03	0.19	0.51	0.19	0.14	0.82	0.19	0.04	0.04	0.01
SMOreg	0.31	0.23	0.33	0.17	0.46	0.41	0.02	0.19	0.02	0.31	0.23	0.33	0.17	0.77	0.23	0.02	0.08	0.03	0.31	0.34	0.33	0.17	0.78	0.22	0.02	0.10	0.03
ZeroR	19.23	19.23	19.23	18.52	18.52	18.52	6.25	0.00	6.25	19.23	19.23	19.23	18.52	18.52	18.52	6.25	12.50	6.25	19.23	19.23	19.23	18.52	18.52	18.52	6.25	0.00	6.25
J48	48.08	44.23	48.08	38.89	61.11	50.00	37.50	40.63	37.50	48.08	44.23	48.08	38.89	44.44	44.44	37.50	12.50	40.63	48.08	40.38	48.08	38.89	42.59	44.44	37.50	12.50	40.63
NaiveBayes	13.46	26.92	13.46	7.41	25.93	22.22	28.13	46.88	28.13	13.46	26.92	13.46	7.41	27.78	16.67	28.13	15.63	25.00	13.46	28.85	13.46	7.41	35.19	16.67	28.13	12.50	25.00
Logistic	28.85	30.77	28.85	40.74	38.89	37.04	31.25	37.50	31.25	28.85	30.77	28.85	40.74	44.44	37.04	31.25	18.75	31.25	28.85	32.69	28.85	40.74	40.74	37.04	31.25	25.00	31.25
SMO	28.85	28.85	28.85	42.59	40.74	38.89	18.75	46.88	18.75	28.85	28.85	28.85	42.59	48.15	44.44	18.75	18.75	25.00	28.85	28.85	28.85	42.59	50.00	44.44	18.75	18.75	25.00
ZeroR	28.85	28.85	28.85	25.93	25.93	25.93	31.25	31.25	31.25	28.85	28.85	28.85	25.93	25.93	25.93	31.25	31.25	31.25	28.85	28.85	28.85	25.93	25.93	25.93	31.25	31.25	31.25
J48	51.92	42.31	51.92	61.11	72.22	55.56	50.00	34.38	50.00	51.92	42.31	51.92	61.11	50.00	55.56	50.00	34.38	43.75	51.92	44.23	51.92	61.11	50.00	61.11	50.00	34.38	43.75
NaiveBayes	26.92	38.46	26.92	37.04	44.44	42.59	31.25	43.75	31.25	26.92	38.46	26.92	37.04	40.74	42.59	31.25	31.25	34.38	26.92	38.46	26.92	37.04	40.74	37.04	31.25	31.25	34.38
Logistic	53.85	36.54	53.85	48.15	59.26	42.59	43.75	40.63	43.75	53.85	36.54	53.85	48.15	51.85	42.59	43.75	46.88	53.13	53.85	30.77	53.85	48.15	51.85	48.15	43.75	46.88	53.13
SMO	55.77	30.77	55.77	61.11	62.96	66.67	46.88	40.63	46.88	55.77	30.77	55.77	61.11	61.11	66.67	46.88	40.63	50.00	55.77	34.62	55.77	61.11	61.11	61.11	46.88	40.63	50.00

Table 7.5. Data transfer prediction accuracies

These figures show that every dimension has at least a limited effect on the models. Considering source code phase, kernel execution and the full program are a little easier to estimate than either initialization/cleanup or data transfer. This is to be expected, though, since "kernel" and "full" are the two phases containing the kernels the predictor metrics are based on. Regarding measurement aggregation, concentrating on a single execution platform is proven simpler than accounting for other parts of the hardware system as well. A similar slight edge can be observed for the power aspect, compared to both time and energy, while the GPU platform has an even more pronounced advantage over both CPUs and FPGAs. The most important difference, however, is evident along the machine learning technique dimension. Specifically that barely 10% of the regression models managed to outperform the baseline, while this ratio is over 90% for both classification types. This suggests, not surprisingly, that an exact improvement ratio is much harder to estimate than an interval it will fall in.

Regression models frequently resorted to using only a constant value or a function of a single input metric, which is a clear sign of undertraining, but after disregarding these, we still had a few promising cases. However, these belonged almost exclusively to GPUs. E.g., the highest precision among the "full" regressions – which is also the highest value increase compared to its ZeroR counterpart – is the REPTree model for Single-GPU-Energy estimation. It reaches an absolute .83 correlation coefficient, representing a .41 improvement. The most precise "kernel" regression is also a REPTree – this time for WithCPU-GPU-Energy – with a value of .76, representing another .41 improvement. The pattern of these two tables suggests that REPTree and M5P are more appropriate for time and energy prediction, while Multilayer Perceptron and SMOreg are more successful for average power. This is no longer true for the "initialization/cleanup" phase, where only FPGAs have notable models. M5P seems the most capable for all three aspects, but the best models are the All-FPGA-Power REPTree with a .81 precision, and the Single-FPGA-Power SMOreg with a .35 increase. As for the "data transfer" models, only the GPU-Power columns stand out. The best case scenario here is the All-GPU-Power M5P model with an accuracy of .82, which is a .44 improvement.

Regarding classification models, we no longer see the superiority of GPU prediction. The most easily discernible global observation is that the overwhelming majority is a significant upgrade compared to either the ZeroR reference or a random choice. We can also notice that while regressions were more prone to "column patterns", – i.e., the measurement aggregation method, the platform, or the aspect mattered more in the columns than the algorithms in the rows, leading to higher concentrations of precise models above or below each other – classifications lean towards "row patterns" – i.e., once the source code phase is chosen, higher accuracy correlates more with the algorithm. For the sake of brevity in further model discussion, (vs. x%/y%) will mean "compared to a ZeroR of x% and a random choice of y%".

"Full" classifications are lead by J48 models, which display up to 60% accuracy on 5 bins (vs. 11.11%/20%), once for power and twice for energy. For 3 bins, this value is up to 71.11% (vs. 22.22%/33.33%), but here we note that Logistic regression is a close second. The best "kernel" models come from these two algorithms again; J48 on 5 bins is at times 68.97% (vs. 0%/20%) for FPGA time and energy, while Logistic regression reaches the same 68.97% on 3 bins (vs. 34.48%/33.33%), at the same places. For the "initialization/cleanup" phase, the best choices are SMO on 5 bins for All-CPU-Energy with 57.69% (vs. 19.23%/20%), and J48 on 3 bins for FPGA time and energy

modeling with 81.25% (vs. 31.25%/33.33%). Finally, the most accurate "data transfer" classifications are J48 trees, both times for Single-GPU-Power prediction: 61.11% on 5 bins (vs. 18.52%/20%) and 72.22% on 3 bins (vs. 25.93%/33.33%).

In conclusion, by predicting the improvement category significantly more accurately than either a baseline performance or a random choice, our classification algorithms clearly demonstrated that static metrics have predictive power and skill in this domain. Therefore we can answer our research question in the affirmative.

We would also like to emphasize the fact that every benchmark [12], calculated metric, measurement, machine learning result [11] and even the measurement library [48] is opened to the public so we invite replication or further expansion.

7.6 Summary

The goal of this chapter was to continue the research laid down in Chapter 6 and present our work addressing the creation of prediction models that are able to automatically determine not only the optimal execution platform of a program, (i.e., sequential or OpenCL; CPU, GPU or FPGA) but how much improvement we can expect that way.

We achieved this by changing optimal platform class labels into improvement ratios and trying to predict those directly, using additional – and more precisely calculated – metrics, an extended benchmark set, and even an extra platform. We also studied the available data along more dimensions, aiming to shed light on which has the most impact on prediction accuracy.

Overall, we consider the results of this experiment encouraging. Despite the (still) small number of subject systems, we were able to demonstrate that statically computed source code metrics are appropriate for improvement estimation, not just platform selection. The models represent value above random choice in themselves, but we would like to emphasize again, that it is the approach of their creation that we consider our most generalizable result. It can enable larger scale studies and hopefully lead to more evidence about the connections between static source code metrics and dynamic platform selection.

"There's no such thing as a free lunch."
— Milton Friedman

8 Maintainability Changes of Parallelized Implementations

8.1 Overview

Automatic kernel transformators and other parallelized source code generators aim to make available performance gains more accessible to a wider audience, and in a wider range of scenarios. Additionally, as we saw in the previous chapters, they would also greatly help our current research in extending the available benchmark sets. Is it worth developing such algorithms, however, or should we simply manually maintain a dedicated parallel implementation. To try and explore this question from a source code perspective, we compared the calculated abstract characteristics of the (CPU based) sequential and (accelerator specific) parallel versions of our benchmarks.

After extracting maintainability information from both the "before" and the "after" versions of a hypothetical parallel transformation, we try to answer the following research questions:

Research Question 1: *How does parallelization affect the maintainability of a subject system as a whole?*

Research Question 2: *How does parallelization affect the maintainability of the encapsulated algorithms?*

The rest of the chapter is structured as follows: Section 8.2 places the study among the related work, while Section 8.3 discusses the extra details we had to consider in addition to the methodology explained in Section 7.2. Next, Section 8.4 briefly overviews the results. Lastly, we summarize the chapter in Section 8.5.

8.2 Related Work

The most closely related work, of course, is the REPARA report D7.4: Maintainability models of heterogeneous programming models [79]. In fact, this experiment could

be viewed as a replication of its maintainability change inspection, only using an extended, and more polished benchmark set. Apart from that, however, we are aware of very limited other research explicitly touching on the maintainability of parallelized implementations.

Pflüger and Pfander [71] performed a fine-tuning case study on their SG++ library while trying to preserve source code maintainability. They also concluded – among other lessons learned – that maintainability deterioration is a natural side effect of performance optimization, and that automatic code generation and domain specific languages could help substantially. Another study in this area was done by Brown et al. [18], who examined that starting with a higher abstraction level language and *then* transforming to heterogeneous platforms could yield comparable or even better performance without degrading developer productivity. While these works considered a more subjective measure of maintainability, we aim to quantify its loss between sequential and parallel versions.

8.3 Methodology

8.3.1 Tagging

As our subject systems, we used the benchmarks listed in Section 7.3. The static source code analysis had to be performed on both the sequential and parallel (OpenCL) variants of every benchmark because even though the prediction models required metrics only from the "before" state, we also needed metrics from the "after" state to be able to compare them. This meant that the OpenCL benchmarks had to undergo some artificial modifications before analysis. These modifications included:

- importing the kernel source code from the *.cl files into their corresponding hosts (as additional, dead code blocks, not executed during dynamic measurements),
- defining missing built-in function references,
- handling `__kernel`, `__global`, and other similar tokens that do not have any meaning in plain C++, and
- tagging the relevant source code parts with the help of `STATIC_BEGIN` and `STATIC_END` macros (see Section 7.2).

These changes lead to successful builds, enabling metric calculations.

8.3.2 Maintainability Evaluation

The metrics we calculated for the "after" versions of the benchmarks naturally coincide with the ones extracted for the "before" states, described in Section 7.4. This was followed by a metric normalization stage using ECDFs (see Section 2.2) with all of the benchmark source code metrics providing context.

Then, using these normalized metrics as a base, we performed a weighted aggregation to produce the more abstract scores outlined in Section 2.3, similarly to Section 5.2.5. The maintainability model itself is discussed in detail in the previously referenced REPARA D7.4 report [79]. As we already mentioned, this experiment could

Metric	Analysability	Modifiability	Modularity	Reusability	Testability
LOC	2.43	3.71	1.29	1.43	1.43
LLOC	11.29	12.00	1.71	7.86	9.71
NL	8.57	9.71	2.00	5.14	13.86
NLE	8.71	9.29	2.00	8.14	11.14
McCC	26.71	28.00	2.29	19.29	29.14
NOS	8.86	7.71	2.00	2.43	4.00
NOI	17.86	18.86	85.71	51.29	16.43
LNL	4.43	4.14	1.71	2.43	6.00
LNL1	1.29	0.86	0.00	0.29	1.14
LNL2	1.57	1.00	0.00	0.71	1.29
LNL3	2.57	1.86	0.00	1.00	1.43
EXP	2.86	1.43	1.29	0.00	1.86
ARR	0.71	0.71	0.00	0.00	0.71
ARR%	2.14	0.71	0.00	0.00	1.86
MUL	0.00	0.00	0.00	0.00	0.00
MUL%	0.00	0.00	0.00	0.00	0.00
ADD	0.00	0.00	0.00	0.00	0.00
ADD%	0.00	0.00	0.00	0.00	0.00

Table 8.1. The results of the original subcharacterictic votes

be considered a replication of those results, only on an extended and more fine-tuned benchmark set.

Table 8.1 shows the aggregated scores based on the experts' models from the report. The same process was applied to calculate maintainability from its subcharacteristics, presented in Table 8.2. This means that, according to this specific model,

$$\begin{aligned} Maintainability = {} & 23.57 \cdot Analysability + 10.57 \cdot Modifiability \\ & + 18.14 \cdot Modularity + 30.29 \cdot Reusability + 17.43 \cdot Testability. \end{aligned}$$

Subcharacteristic	Maintainability
Analysability	23.57
Modifiability	10.57
Modularity	18.14
Reusability	30.29
Testability	17.43

Table 8.2. The results of the original Maintainability votes

Benchmark	Analysability	Modifiability	Modularity	Reusability	Testability	Maintainability
mri-q	−0.388	−0.405	−0.448	−0.432	−0.360	−0.407
spmv	−0.667	−0.685	−0.653	−0.676	−0.658	−0.668
stencil	−0.225	−0.237	−0.428	−0.325	−0.199	−0.283
atax	−0.338	−0.354	−0.472	−0.406	−0.298	−0.375
bicg	−0.342	−0.358	−0.471	−0.412	−0.308	−0.379
conv2d	−0.332	−0.346	−0.469	−0.405	−0.296	−0.370
doitgen	−0.372	−0.388	−0.582	−0.476	−0.324	−0.429
gemm	−0.269	−0.283	−0.435	−0.352	−0.237	−0.315
gemver	−0.325	−0.343	−0.494	−0.417	−0.294	−0.375
gesummv	−0.290	−0.304	−0.384	−0.343	−0.262	−0.317
jacobi2d	−0.420	−0.433	−0.560	−0.491	−0.373	−0.456
mvt	−0.339	−0.353	−0.444	−0.396	−0.304	−0.368
bfs	−0.352	−0.367	−0.497	−0.431	−0.319	−0.393
hotspot	−0.226	−0.235	−0.371	−0.308	−0.167	−0.261
lavaMD	−0.271	−0.276	−0.352	−0.315	−0.244	−0.292
nn	−0.429	−0.434	−0.560	−0.485	−0.364	−0.456

Table 8.3. Maintainability changes at the system level

8.4 Results

Using the quality characteristics we calculated, we could investigate our two research questions. The changes in intermediate values (Analysability, Modifiability, Modularity, Reusability, and Testability) and in the final Maintainability score are shown in Table 8.3 for the whole system, and in Table 8.4 for the separated kernel regions.

According to the data in Table 8.3, we can answer our **first research question**: Maintainability experiences a distinct negative change as a result of parallelization. The scores in the table are all negative without exception, and even their absolute values are decidedly large, expressing a significant change. This reinforces our informal "no free lunch" assumption, i.e., the price of a higher performance system seems to be – possibly among other factors – a less maintainable codebase.

When we look at Table 8.4, however, we see a much less pronounced negative effect, which, at times, even turns positive. Although the ratio of the change directions might lean more towards the negative, their absolute values are overall smaller than in the previous table – except maybe Modularity, which is on a similar scale. In answering our **second research question**, it is hard to say anything definitive about the kernels separately. Similarly to the conclusions of the original study [79], we speculate that this is because, even though such a transformation can deteriorate the maintainability of the kernels themselves, its most powerful effect is the boilerplate and added necessary infrastructure it brings to the system as a whole. This can be considered another point in favor of automatic parallel transformations, as that way developers could work on a more maintainable version of the source code, while still being able to reap the benefits of modern accelerators and parallel platforms.

Benchmark	Analysability	Modifiability	Modularity	Reusability	Testability	Maintainability
mri-q	−0.234	−0.240	−0.395	−0.321	−0.225	−0.282
spmv	0.139	0.135	−0.308	−0.069	0.188	0.019
stencil	0.145	0.144	−0.617	−0.205	0.220	−0.059
atax	−0.109	−0.136	−0.435	−0.283	−0.087	−0.208
bicg	−0.200	−0.222	−0.449	−0.329	−0.162	−0.272
conv2d	−0.065	−0.075	−0.431	−0.228	−0.002	−0.161
doitgen	0.147	0.131	−0.653	−0.228	0.226	−0.072
gemm	0.120	0.110	−0.391	−0.123	0.175	−0.019
gemver	−0.161	−0.187	−0.708	−0.429	−0.119	−0.319
gesummv	−0.041	−0.055	−0.341	−0.199	−0.033	−0.131
jacobi2d	−0.035	−0.057	−0.691	−0.347	0.019	−0.220
mvt	−0.148	−0.174	−0.443	−0.302	−0.115	−0.236
bfs	0.067	0.063	−0.408	−0.150	0.095	−0.064
hotspot	−0.035	−0.041	−0.774	−0.390	0.043	−0.235
lavaMD	−0.158	−0.165	−0.434	−0.303	−0.150	−0.239
nn	0.015	0.017	0.007	0.006	0.013	0.012

Table 8.4. Maintainability changes at the kernel level

8.5 Summary

In this chapter, we conducted a replication of the maintainability study by Ferenc et al. [79] on the sequential and parallel versions of the benchmarks from Chapter 7 and observed that the maintainability of parallelized implementations – along with every other quality subcharacteristic – is significantly lower. However, this did not necessarily show – or at least not as strictly – in the kernels themselves, suggesting that the introduced boilerplate is to blame. This maintainability assessment can be considered a step towards justifying and motivating the development of automatic parallel transformations.

"Everything should be made as simple as possible,
but no simpler."

— Albert Einstein

9 Conclusions

In this thesis, we discussed two main topics, these being the effects of source code patterns on software maintainability, and the utilization of static source code metrics in performance optimization.

In the field of *source code patterns*, we focused on the connections among design patterns, antipatterns, software faults, and software maintainability. To briefly summarize our results, we found a compelling proportional relationship between design patterns and maintainability, an inverse connection between antipatterns and maintainability, and a positive correlation between antipatterns and program faults. These findings all coincide with intuitive expectations, only now they are also supported with empirical studies and objective, definite data.

In the field of *performance optimization*, we demonstrated our methodology of creating both qualitative and quantitative platform prediction models, that rely on static information alone. Our main result here is this methodology itself, along with its detailed evaluation and a complementary study demonstrating the detrimental effect of source code parallelization on maintainability.

Future Work

Despite the results we achieved, there are still numerous opportunities for future work.

In the area of source code patterns and software maintainability, we plan to repeat the presented analyses on a larger number of subject systems for increased generalizability. Furthermore, we aim to involve some of the well-known design pattern and antipattern miner tools into matching source code patterns, which would enable us to compare their accuracies. We also intend to calculate maintainability values at lower source code element levels, thereby possibly gaining a more fine-grained view of how pattern and non-pattern elements relate to maintainability. Another goal is to implement further improvements to our antipattern matching tool, such as more extensive structural checks, statistics-based dynamic thresholds, or lexical cues. Besides, our selected Firefox revisions have runtime, power consumption and energy efficiency measurements as part of the Green Mining Dataset [36], and this provides us with the

chance to relate antipatterns or maintainability to those concepts, too.

There are other avenues for improving our performance related research, as well. One of these, of course, is increasing the number of benchmarks on which the models are based. Another factor can be adding even more predictor metrics. Considering the relative importance of our non-standard, low-level source code metrics, we will try to derive even more potentially representative characteristics by manually inspecting the typical properties of the benchmarks. Moreover, we intend to take platform specific configurations and compiler settings into account.

Epilogue

I have always had a particular interest in the quality of the work I do, which my years of research only strengthened. I also advocate putting in work up front, which is evident even in the imbalanced schedule of my publications and credit acquisitions. To me, this aligns well with the philosophy behind design patterns and their future-proof effect, hence my affection for the subject. On the other hand, "what he gains at the toll, he loses at the customs" is a pertinent Hungarian proverb that, I think, sums up antipatterns quite nicely. These lessons – learned and reinforced during my PhD studies – are, in my opinion, much more universally applicable to everyday life than the previous 80 pages of thick technical jargon might suggest.

Another area where my research – particularly, sharing my research – lead to significant personal growth is interpersonal skills. Having to be the smartest man in the room, even if only in a really narrow and specific topic, takes an unexpected amount of courage.

It is my hope that this thesis, as the culmination of my years of effort, will be valuable for someone, somehow. At the very least, citing it might serve as a formal "get out of jail free card" the next time I am caught spending half a workday obsessing over an already functional piece of code only to make it more elegant.

Appendices

A

Summary in English

Software rules the world. As true as this statement already was decades ago, it rings even truer now, when connected devices outnumber the population of Earth by a ratio of at least 1.5. A modern life in this era involves countless hidden, invisible processors, along with the visible ones we have all got so used to. And we have not even mentioned critical applications like flight guidance, keeping patients alive, or operating nuclear power plants. All of these systems need software to run, and we are running out of people to write it. Consequently, sustainable software development has never been more important.

The research behind this thesis aims to facilitate this sustainability by drawing attention to the importance of the maintenance phase, and illustrating its assets and risks by finding objective connections between certain source code patterns and software maintainability. It also seeks to help developers more easily utilize modern accelerator hardware in order to increase performance by creating a readily usable and extendable static platform selection framework. The results we obtained have been grouped into two major thesis points, along the same separation. The relation between these thesis points and their supporting publications is shown in Table A.1.

I. Empirical validation of the impact of source code patterns on software maintainability

The contributions of this thesis point – related to software maintainability – are discussed in chapters 3, 4, and 5.

The Connection between Design Patterns and Maintainability To study the impact design patterns have on software maintainability, we analyzed over 700 revisions of JHotDraw 7 [30]. We chose it especially for its intentionally high pattern density and the fact that its pattern instances were all so thoroughly documented that we could use a *javadoc*-based text parser for pattern recognition. This led to a virtual guarantee of precision regarding the matched design pattern instances, which we paired with the utilization of an objective maintainability model [9]. An inspection of the revisions where the number of pattern instances increased revealed a clear trend of similarly increasing maintainability characteristics. Fur-

thermore, comparing pattern density to maintainability as a whole resulted in a 0.89 Pearson correlation coefficient, which suggested that design patterns do indeed have a positive effect on maintainability.

The Connection between Antipatterns and Maintainability As for the impact of antipatterns, we selected 228 open-source Java systems, along with 45 revisions of the Firefox browser application written in C++ as our subjects for two distinct experiments. In both cases, we matched 9 different, widespread antipattern types through metric thresholds and structural relationships – with additional antipattern densities for C++ [19]. Maintainability calculation remained the same for the Java systems, while the evaluation of Firefox required a C++ specific custom quality model, – based on ECDFs [89] – and we also implemented versions of the "traditional" MI metric [21]. The results of both studies confirm the detrimental effect of antipatterns. The overall Spearman correlation coefficient between antipatterns and maintainability for Java was -0.62, while the C++ analysis provided values for both absolute antipattern instances and antipattern densities, which were -0.66 and -0.69 for Pearson, and -0.7 and -0.68 for Spearman correlation, respectively. Another interesting result is that using antipattern instances as predictors for maintainability estimation produced models with precisions ranging from 0.76 to 0.93.

The Connection between Antipatterns and Program Faults In addition, Chapter 4 contains an experiment that seeks to connect the presence of antipatterns to program faults (or bugs) through the PROMISE open bug database [63]. The study of the 34 systems (from among the 228 Java systems mentioned above) that had corresponding bug information revealed a statistically significant Spearman correlation of 0.55 between antipattern instances and bugs. Moreover, antipatterns yielded a precision of 67% when predicting bugs, being notably above the 50% baseline, and only slightly below the 71.2% of more than five times as many raw static source code metrics, thereby demonstrating their applicability.

The main results of this thesis point are the above-mentioned empirical studies themselves, which support the intuitive expectations about the relations between well-known source code patterns and software maintainability with objective, definite data. To our knowledge, these findings are among the first that were performed on subject systems of such volume, size, and variance, while also avoiding all subjective factors like developer surveys, time tracking, and interviews.

The Author's Contributions

For the research linked to design patterns, the author's main contributions were the implementation of the pattern mining tool, calculating the relevant source code metrics, manually validating the revisions that introduced patterns changes, and reviewing the related literature. On the other hand, the entire antipattern-related research was the author's own work including the preparation and analysis of the subject systems, implementing and extracting the relevant static source code metrics, calculating the corresponding maintainability values, – along with overseeing the creation of the C++ specific quality model – implementing the antipattern mining tool and extracting antipattern matches, considering program fault information, as well as conducting and evaluating the empirical experiments. The publications related to this thesis point are:

- Péter Hegedűs, **Dénes Bán**, Rudolf Ferenc, and Tibor Gyimóthy. Myth or Reality? Analyzing the Effect of Design Patterns on Software Maintainability. In *Advanced Software Engineering & Its Applications (ASEA 2012), Jeju Island, Korea, November 28 – December 2*, pages 138–145, CCIS, Volume 340. Springer Berlin Heidelberg, 2012.
- **Dénes Bán** and Rudolf Ferenc. Recognizing Antipatterns and Analyzing their Effects on Software Maintainability. In *14th International Conference on Computational Science and Its Applications (ICCSA 2014), Guimarães, Portugal, June 30 – July 3*, pages 337–352, LNCS, Volume 8583. Springer International Publishing, 2014.
- **Dénes Bán**. The Connection of Antipatterns and Maintainability in Firefox. In *10th Jubilee Conference of PhD Students in Computer Science (CSCS 2016), Szeged, Hungary, June 27 – 29*, 2016.
- **Dénes Bán**. The Connection of Antipatterns and Maintainability in Firefox. Accepted for publication in the 2016 Special Issue of Acta Cybernetica (extended version of the CSCS 2016 paper above). 20 pages.

II. A hardware platform selection framework for performance optimization based on static source code metrics

The contributions of this thesis point – related to software performance – are discussed in chapters 6, 7, and 8.

Qualitative Prediction Models The goal of our qualitative research was to develop a highly generalizable methodology for building prediction models that are capable of automatically determining the optimal hardware platform (regarding execution time, power consumption, and energy efficiency) of a given program, using static information alone. To achieve this, we collected a number of benchmark programs that contained algorithms implemented for each targeted platform. These benchmarks are necessary for the training of the models, because executing and measuring the different versions of their algorithms on their respective platforms is what highlights their differences in performance. Then, we extracted several low-level source code metrics from these algorithms that would capture their characteristics and would become the predictors of our models. We also developed a universal solution capable of performing accurate cross-platform time, power, and energy consumption measurements for this purpose [48]. Lastly, we applied various machine learning techniques to build the proposed prediction models. A brief empirical validation showed the theoretical usefulness of such models, – demonstrating perfect accuracy at times – but the real result of this study is the methodology that led to them, and could also enable larger scale experiments.

Quantitative Prediction Models Building on the above-mentioned previous work, we extended our methodology to create models that are quantitative, i.e., able to estimate expected improvement ratios instead of just the best platform. Additional refinements included a significantly augmented set of source code metrics, more precise metric extraction, adding new benchmarks, and introducing the FPGA platform as a possible target. As improvement ratios are continuous, they were predicted using regression algorithms, as well as the previously used classification algorithms paired with a discretization filter. While the regression models

rarely proved encouraging, 94% of the classification models outperformed either random choice or the established baseline by at least 5% (up to 49%, at times).

Maintainability Changes of Parallelized Implementations The benchmark source code already available to us also made it possible to look for maintainability changes between the CPU-based original (sequential) and the accelerator hardware specific (parallel) algorithm versions. The only other requirement was for us to compute the same source code metrics for the parallel variants as well, as a suitable maintainability model would be reused from a previous study [79]. The results of comparing the overall maintainability values of the two source code variants clearly indicated that parallelized implementations had significantly lower maintainability compared to their sequential counterparts. This, however, did not appear nearly as strongly in the core algorithms themselves, which suggests that deterioration is mainly due to the added infrastructure (boilerplate code) introduced by the accelerator specific frameworks they employ.

The main results of this thesis point are (a) the empirical proof that static source code metrics are suitable for improvement estimation, and (b) a universal process for creating qualitative and quantitative hardware platform prediction models. A key difference between our approach and other available solutions is that, once they are built, our models operate based on static information alone. Also, their accuracy depends primarily on the number of training benchmarks. These properties make the methodology easy to enhance, and its output models easy to apply.

The Author's Contributions

The author led the effort of collecting and preparing the benchmarks for both static analysis and dynamic measurements. He implemented, extracted, and aggregated both the original and the extended set of source code metrics, and he compiled the final machine learning tables. He formalized and performed the actual experiments, and analyzed their results. The maintainability comparison of benchmark versions and its evaluation is also the author's work. The publications related to this thesis point are:

- **Dénes Bán**, Rudolf Ferenc, István Siket, and Ákos Kiss. Prediction Models for Performance, Power, and Energy Efficiency of Software Executed on Heterogeneous Hardware. In *13th IEEE International Symposium on Parallel and Distributed Processing with Applications (IEEE ISPA-15), Helsinki, Finland, August 20 – 22*, pages 178–183, IEEE Trustcom/BigDataSE/ISPA, Volume 3. IEEE Computer Society Press, 2015.
- **Dénes Bán**, Rudolf Ferenc, István Siket, Ákos Kiss, and Tibor Gyimóthy. Prediction Models for Performance, Power, and Energy Efficiency of Software Executed on Heterogeneous Hardware. Submitted to the Journal of Supercomputing, Springer Publishing (extended version of the IEEE ISPA-15 paper above). 24 pages.

Table A.1 summarizes the main publications and how they relate to our thesis points.

№	[103]	[100]	[98]	[99]	[101]	[102]
I.	◆	◆	◆	◆		
II.					◆	◆

Table A.1. Thesis contributions and supporting publications

B

Magyar nyelvű összefoglaló

Szoftverek uralják a világot. Ez az állítás lehet, hogy már évtizedekkel ezelőtt is fedte volna a valóságot, de napjainkban mindenképp, tekintve hogy a hálózati eszközök legalább másfélszer annyian vannak, mint az emberek. Egy modern élet – a megszokott és jól látható példákon kívül is – tele van láthatatlan, rejtett processzorokkal. Nem is említve azt a sok kritikus rendszert, amikhez nélkülözhetetlenek, mint például a repülés-irányítás, kórházi készülékek, vagy épp egy nukleáris reaktor üzemeltetése. Ezeknek mind szoftverekre van szükségük a működésükhöz. Következésképpen, a fenntartható szoftverfejlesztés fontosabb mint valaha.

A jelen disszertációt megalapozó kutatás ezt a fenntarthatóságot hivatott elősegíteni. Első sorban a szoftverek karbantartási fázisát, illetve az azt segítő vagy hátráltató tényezők fontosságát szeretnénk kihangsúlyozni azzal, hogy objektív összefüggéseket mutatunk be bizonyos forráskód minták és a karbantarthatóság között. Emellett abban is segíteni szeretnénk a fejlesztőket, hogy könnyebben kihasználhassák a performancia növelésére szolgáló modern gyorsító hardverek nyújtotta lehetőségeket azáltal, hogy bemutatunk egy egyszerűen használható és bővíthető statikus platform választó keretrendszert. Az eredményeinket – hasonló csoportosítással – két tézispontba foglaltuk. A tézispontokhoz tartozó publikációkat a B.1. táblázat foglalja össze.

I. A forráskód minták szoftver karbantarthatóságra kifejtett hatásának empirikus validációja

A tézispont témája a szoftver karbantarthatóság, az ide tartozó kutatási eredményeket pedig a 3., 4. és 5. fejezetek tárgyalják.

A tervezési minták és a karbantarthatóság kapcsolata A tervezési minták karbantarthatóságra kifejtett hatásának vizsgálata érdekében a JHotDraw grafikus szoftver több mint 700 revízióját elemeztük. Ezt a rendszert kifejezetten azért választottuk, mert készítői a benne szereplő tervezési mintákat a forráskódban alaposan és következetesen dokumentálták, így az általános felismerő eszközök helyett egy *javadoc* alapú szöveges feldolgozó script-et használhattunk. Ez gyakorlatilag garantálta a kibányászott mintapéldányok precizitását, amit mi egy objektív karbantarthatósági modell használatával egészítettünk ki [9]. Ezután ta-

nulmányoztuk azokat a revíziókat, ahol növekedés történt a rendszerben található tervezési minták számában, és egyértelmű javulást tapasztaltunk a karbantarthatósági értékekben is. Továbbá, a mintasűrűség és a karbantarthatóság átfogó összehasonlítása egy 0,89-es Pearson korrelációs együtthatót eredményezett, ami arra utal, hogy a tervezési minták valóban jótékony hatással vannak a karbantarthatóságra.

Az antiminták és a karbantarthatóság kapcsolata Az antiminták vizsgálatára 228 nyílt forráskódú Java rendszert, valamint a Firefox C++ alapú böngésző 45 revízióját választottuk, 2 különálló kísérletre. Mindkét esetben 9, széles körben elterjedt antiminta típust nyertünk ki metrika határszámok és strukturális kapcsolatok alapján – valamint a C++ esetében antiminta sűrűségeket is. Java rendszerekre továbbra is az előző karbantarthatósági modellünket, míg a Firefox kiértékeléséhez egyedi, ECDF-alapú [89] C++ minőségi modellt és az MI metrika [21] több verzióját használtuk. Mindkét tanulmány az antiminták negatív hatását támasztja alá. Java-ban az antiminták és a karbantarthatóság közötti átfogó Spearman korreláció -0,62 volt, C++-ban pedig az abszolút és sűrűségi antiminta értékek kapcsolata a karbantarthatósággal rendre -0,66 és -0,69 volt Pearson, illetve -0,7 és -0,68 Spearman korreláció esetén. Egy másik érdekes eredmény, hogy a C++-beli antiminta találatokat karbantarthatósági prediktorokként használva 0,76 és 0,93 közötti pontosságú gépi tanulási modelleket tudtunk építeni.

Az antiminták és a programhibák kapcsolata Kiegészítésként a 4. fejezetben az antiminták és a programhibák (vagy „bugok") között is kapcsolatot kerestünk a PROMISE nyílt hiba adatbázis segítségével [63]. A fent említett 228-ból azt a 34 Java rendszert vizsgálva, amikhez hiba információk is tartoztak, statisztikailag szignifikáns 0,55-ös erősségű Spearman korrelációt találtunk az antiminták és a hibák száma között. Ezen felül kimutattuk, hogy az antiminták 67%-os pontossággal tudják előrejelezni a programhibák számát, ami jelentősen jobb az 50%-os alapértéknél, és nem sokkal marad el ötször több statikus forráskód metrika 71,2%-os teljesítményétől sem.

A tézispont fő eredményei maguk a fent említett empirikus tanulmányok, amik a forráskód minták és a karbantarthatóság kapcsolatára vonatkozó intuitív elvárásainkat objektív, kézzel fogható adatokkal támasztják alá. Tudomásunk szerint ezeket az eredményeket elsők között sikerült ilyen nagy mennyiségű, nagy méretű és változatos rendszereken, illetve minden szubjektív tényező – például kérdőívek, időkövetés vagy interjúk – nélkül elérnünk.

A szerző hozzájárulása

A tervezési mintákkal kapcsolatos kutatásban a szerző főként a mintafelismerő eszköz implementációjához, a forráskód metrikák kiszámításához, a mintapéldányok számának változásával járó revíziók kézi ellenőrzéséhez és a kapcsolódó irodalom feldolgozásához járult hozzá. Ezzel szemben mindkét antimintákkal kapcsolatos tanulmány teljes egészében a szerző munkája, beleértve a rendszerek előkészítését és elemzését, a statikus forráskód metrikák implementációját és kinyerését, a karbantarthatósági értékek kiszámítását, – a C++ specifikus minőségi modell készítésével együtt – az antiminták értelmezését, implementálását és beazonosítását, a programhiba információk feldolgozását, valamint az empirikus kísérletek megtervezését és lebonyolítását is. A tézispont a következő publikációkra épül:

- Péter Hegedűs, **Dénes Bán**, Rudolf Ferenc, and Tibor Gyimóthy. Myth or Reality? Analyzing the Effect of Design Patterns on Software Maintainability. In *Advanced Software Engineering & Its Applications (ASEA 2012), Jeju Island, Korea, November 28 – December 2*, pages 138–145, CCIS, Volume 340. Springer Berlin Heidelberg, 2012.
- **Dénes Bán** and Rudolf Ferenc. Recognizing Antipatterns and Analyzing their Effects on Software Maintainability. In *14th International Conference on Computational Science and Its Applications (ICCSA 2014), Guimarães, Portugal, June 30 – July 3*, pages 337–352, LNCS, Volume 8583. Springer International Publishing, 2014.
- **Dénes Bán**. The Connection of Antipatterns and Maintainability in Firefox. In *10th Jubilee Conference of PhD Students in Computer Science (CSCS 2016), Szeged, Hungary, June 27 – 29*, 2016.
- **Dénes Bán**. The Connection of Antipatterns and Maintainability in Firefox. Közlésre elfogadva az Acta Cybernetica 2016-os különkiadásában (a fenti CSCS 2016 publikáció kibővített változata). 20 oldal.

II. Performancia optimalizációt elősegítő, statikus forráskód metrikákon alapuló hardver platform választó keretrendszer

A tézispont témája a szoftver performancia, az ide tartozó kutatási eredményeket pedig a 6., 7. és 8. fejezetek tárgyalják.

Kvalitatív modellek A itt bemutatott kutatásunk fő célja egy általánosítható módszertan kidolgozása volt olyan modellek építéséhez, amik képesek tisztán statikus információk alapján megbecsülni, hogy egy adott programot várhatóan melyik hardver platformon lehet optimálisan végrehajtani – mind a futásidő, mind pedig az energia fogyasztás szemszögéből. Ennek alapjául számos „benchmark" programot gyűjtöttük, bennük olyan algoritmusokkal, amik minden lehetséges célplatformhoz rendelkeztek implementációval. A modelljeink betanításához szükség volt ilyen algoritmusokra, hiszen a performanciabeli eltérésekre úgy világíthatunk rá, ha a különböző verzióikat a kapcsolódó platformjaikon futtatjuk. Ezután számos (alacsony szintű) forráskód metrikát nyertünk ki ezekből az algoritmusokból, amik jól megragadják a jellemzőiket és modelljeink prediktorai lehetnek. Emellett egy olyan általános megoldást is kifejlesztettünk, ami képes pontos, platformfüggetlen idő és energiamérésekre [48]. Végül különböző gépi tanulási módszereket használva megépíthettük a célként kitűzött modelleket. Egy rövid empirikus validáció igazolta a modellek elméleti hasznosságát, – amelyek néhol 100%-os pontosságot is elértek – de a tanulmány igazi eredménye a módszertan, amivel megépítettük őket, és ami nagyobb szabású kísérletekre is lehetőséget adhat.

Kvantitatív modellek Az előző eredményeinkre építve úgy bővítettük a módszerünket, hogy már kvantitatív modellek készítésére is képes legyen, amik nem csak a legjobb platformot becsülik meg, hanem az ott várható teljesítmény növekedés arányát is. Emellett jelentősen megnöveltük a kinyert forráskód metrikáink számát, pontosabb metrika kinyerési stratégiát dolgoztunk ki, új benchmark-okkal bővítettük az elemzett rendszereinket, és lehetséges platformként bevezettük az FPGA-kat is. Mivel a javulási arányok folytonosak, így közelítésükhöz regressziós

algoritmusokat, valamint diszkretizáló előfeldolgozás után osztályozó algoritmusokat is használhattunk. Habár a regressziók ritkán vezettek biztató eredményekhez, az osztályozások 94%-a legalább 5%-kal (és időnként akár 49%-kal) pontosabb tudott lenni a véletlenszerű választásnál és az alapkonfigurációnál is.

A forráskód párhuzamosítás és a karbantarthatóság kapcsolata A rendelkezésre álló benchmark forráskódok lehetővé tették azt is, hogy megvizsgáljuk a CPU alapú eredeti (szekvenciális) és a gyorsító hardver specifikus (párhuzamos) algoritmus verziók közti karbantarthatóság különbségeket. Az egyetlen további előfeltétel az volt, hogy a párhuzamos verziók forráskódjából is kinyerjük a korábbi metrikákat, hiszen a minőségi modellt felhasználhattuk egy korábbi tanulmányból [79]. Az összehasonlítás eredményei azt mutatták, hogy a párhuzamosított implementációk karbantarthatósága jelentősen alacsonyabb, mint a szekvenciális párjaiké. Ez azonban nem volt olyan egyértelműen kimutatható az algoritmusok lényegi részében, ami arra utal, hogy a minőségromlást főként a felhasznált, gyorsító hardver specifikus keretrendszerek által bevezetett extra infrastruktúra (boilerplate) okozza.

A tézispont fő eredményei (a) az empirikus bizonyíték, hogy a statikus forráskód metrikák hasznosak a teljesítmény-javulás előrejelzésében, és (b) egy általános módszertan kvalitatív és kvantitatív hardver platform választó modellek építéséhez. Egy fontos különbség az általunk alkalmazott stratégia és más elérhető megoldások között, hogy a mi modelljeink – megépítésük után – csak statikus információkra hagyatkoznak. Továbbá a pontosságuk leginkább a tanításhoz használt benchmark-ok számának függvénye. Ezek a tulajdonságok teszik a módszerünket egyszerűen továbbfejleszthetővé, a modelljeit pedig egyszerűen alkalmazhatóvá.

A szerző hozzájárulása

A benchmark-ok gyűjtése és előkészítése – mind statikus, mind dinamikus elemzésre – a szerző vezetésével történt. Ő implementálta, nyerte ki, és aggregálta az eredeti és a kibővített forráskód metrikákat, és ő állította össze a gépi tanuláshoz használatos táblázatokat. Ő formalizálta és végezte el az empirikus kísérleteket, és elemezte az eredményeiket. Az algoritmus verziók karbantarthatóságának összehasonlítása és kiértékelése szintén a szerző munkája. A tézispont a következő publikációkra épül:

- **Dénes Bán**, Rudolf Ferenc, István Siket, and Ákos Kiss. Prediction Models for Performance, Power, and Energy Efficiency of Software Executed on Heterogeneous Hardware. In *13th IEEE International Symposium on Parallel and Distributed Processing with Applications (IEEE ISPA-15), Helsinki, Finland, August 20 – 22*, pages 178–183, IEEE Trustcom/BigDataSE/ISPA, Volume 3. IEEE Computer Society Press, 2015.
- **Dénes Bán**, Rudolf Ferenc, István Siket, Ákos Kiss, and Tibor Gyimóthy. Prediction Models for Performance, Power, and Energy Efficiency of Software Executed on Heterogeneous Hardware. Elbírálás alatt a Journal of Supercomputing-nál, Springer Publishing (a fenti IEEE ISPA-15 publikáció kibővített változata). 24 oldal.

A tézispontokat és a kapcsolódó publikációkat a B.1. táblázat összegzi.

№	[103]	[100]	[98]	[99]	[101]	[102]
I.	◆	◆	◆	◆		
II.					◆	◆

B.1. táblázat. A tézispontokhoz kapcsolódó publikációk

Bibliography

[1] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 181–190, Washington, DC, USA, 2011. IEEE Computer Society.

[2] Advanced Micro Devices, Inc. *AMD GPU Performance API – User Guide*, January 2015. v2.15.

[3] Tiago L Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[4] P Antonellis, D Antoniou, Y Kanellopoulos, C Makris, E Theodoridis, C Tjortjis, and N Tsirakis. A data mining methodology for evaluating maintainability according to iso/iec-9126 software engineering–product quality standard. *Special Session on System Quality and Maintainability-SQM2007*, 2007.

[5] H. Arasteh, V. Hosseinnezhad, V. Loia, A. Tommasetti, O. Troisi, M. Shafiekhah, and P. Siano. Iot-based smart cities: A survey. In *2016 IEEE 16th International Conference on Environment and Electrical Engineering (EEEIC)*, pages 1–6, June 2016.

[6] Sylvain Arlot and Alain Celisse. A survey of cross-validation procedures for model selection. In *Statistics Surveys*, volume 4, pages 40–79, 2010.

[7] ARM. *ARM DS-5 Version 5.21 – Streamline User Guide*, March 2015. ARM DUI0482S.

[8] Lerina Aversano, Gerardo Canfora, Luigi Cerulo, Concettina Del Grosso, and Massimiliano Di Penta. An Empirical Study on the Evolution of Design Patterns. In *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, ESEC-FSE '07, pages 385–394, New York, NY, USA, 2007. ACM.

[9] Tibor Bakota, Péter Hegedűs, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimóthy. A Probabilistic Software Quality Model. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, ICSM 2011, pages 368–377, Williamsburg, VA, USA, 2011. IEEE Computer Society.

[10] Tibor Bakota, Péter Hegedűs, Gergely Ladányi, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimóthy. A Cost Model Based on Software Maintainability. In

Proceedings of the 28th IEEE International Conference on Software Maintenance, ICSM 2012, Williamsburg, VA, USA, 2012. IEEE Computer Society.

[11] Dénes Bán, Rudolf Ferenc, István Siket, Ákos Kiss, and Tibor Gyimóthy. Performance, power, and energy prediction models. `http://www.inf.u-szeged.hu/~ferenc/papers/PerformancePowerEnergyModels/`, 2017.

[12] Dénes Bán, Róbert Sipka, and Imre Dobi. Tagged parallel benchmarks. `https://github.com/sed-szeged/TaggedParallelBenchmarks`, 2017.

[13] J. Bansiya and C.G. Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering*, 28:4–17, 2002.

[14] Christopher M Bishop. Neural networks for pattern recognition, 1995.

[15] Cisco Canada Blog. Cisco ioe innovation centre toronto: The future is now, 2015.

[16] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. Source-level execution time estimation of c programs. In *Hardware/Software Codesign, 2001. CODES 2001. Proceedings of the Ninth International Symposium on*, pages 98–103, 2001.

[17] F. Brito e Abreu and W. Melo. Evaluating the impact of object-oriented design on software quality. In *Software Metrics Symposium, 1996., Proceedings of the 3rd International*, pages 90 –99, mar 1996.

[18] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 89–100, Washington, DC, USA, 2011. IEEE Computer Society.

[19] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[20] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

[21] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, Aug 1994.

[22] IBM Corp. Ibm spss statistics for windows.

[23] Jing Dong, Dushyant S. Lad, and Yajing Zhao. DP-Miner: Design Pattern Discovery Using Matrix. In *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, ECBS '07, pages 371–380, Washington, DC, USA, 2007. IEEE Computer Society.

[24] Tapio Elomaa and Matti Kääriäinen. An analysis of reduced error pruning. *CoRR*, abs/1106.0668, 2011.

[25] C. Faragó, P. Hegedus, G. Ladányi, and R. Ferenc. Impact of version history metrics on maintainability. In *2015 8th International Conference on Advanced Software Engineering Its Applications (ASEA)*, pages 30–35, Nov 2015.

[26] Rudolf Ferenc, László Langó, István Siket, Tibor Gyimóthy, and Tibor Bakota. Source meter sonar qube plug-in. In *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, SCAM '14, pages 77–82, Washington, DC, USA, 2014. IEEE Computer Society.

[27] F. A. Fontana and S. Maggioni. Metrics and antipatterns for software quality evaluation. In *Software Engineering Workshop (SEW), 2011 34th IEEE*, pages 48–56, June 2011.

[28] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley object technology series. Addison-Wesley, 1999.

[29] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher K. I. Williams, and Michael O'Boyle. Milepost gcc: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39(3):296–327, Jun 2011.

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Pub Co, 1995.

[31] Inc. Gartner. Gartner says demand for enterprise mobile apps will outstrip available development capacity five to one, 2015.

[32] R. L. Glass. Frequently forgotten fundamental facts about software engineering. *IEEE Software*, 18(3):112–111, May 2001.

[33] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10. IEEE, 2012.

[34] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1), November 2009.

[35] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A Practical Model for Measuring Maintainability. *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, pages 30–39, 2007.

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

[37] Nien-Lin Hsueh, Lin-Chieh Wen, Der-Hong Ting, W. Chu, Chih-Hung Chang, and Chorng-Shiuh Koong. An Approach for Evaluating the Effectiveness of Design Patterns in Software Evolution. In *IEEE 35th Annual Computer Software*

and Applications Conference Workshops (COMPSACW), pages 315 –320, July 2011.

[38] Brian Huston. The Effects of Design Pattern Application on Metric Scores. *Journal of Systems and Software*, pages 261–269, 2001.

[39] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual: Vol. 3B*, January 2015. Order Number 253669.

[40] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality.* ISO/IEC, 2001.

[41] ISO/IEC. ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Technical report, 2010.

[42] George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In *Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345, San Mateo, 1995. Morgan Kaufmann.

[43] Capers Jones and Olivier Bonsignour. *The Economics of Software Quality.* Addison-Wesley Professional, 1st edition, 2011.

[44] John Francis. Kenney and E. S. Keeping. *Mathematics of statistics / by J.F.Kenney.* Van Nostrand N.Y, 2nd ed. edition, 1947.

[45] F. Khomh, S. Vaucher, Y. G. Guéhéneuc, and H. Sahraoui. A bayesian approach for the detection of code and design smells. In *2009 Ninth International Conference on Quality Software*, pages 305–314, Aug 2009.

[46] Foutse Khomh and Yann-Gaël Guéhéneuc. Do Design Patterns Impact Software Quality Positively? In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, CSMR '08, pages 274–278, Washington, DC, USA, 2008. IEEE Computer Society.

[47] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012.

[48] Ákos Kiss, Péter Molnár, and Róbert Sipka. Rmeasure performance and energy monitoring library. `https://github.com/sed-szeged/RMeasure`, 2017.

[49] Ákos Kiss et al. *REPARA deliverable D7.3: System-level quantitative models.* 2016.

[50] J.L. Krein, L.J. Pratt, A.B. Swenson, A.C. MacLean, C.D. Knutson, and D.L. Eggett. Design Patterns in Software Maintenance: An Experiment Replication at Brigham Young University. In *Second International Workshop on Replication in Empirical Software Engineering Research (RESER 2011)*, pages 25 –34, Sept. 2011.

[51] Michael Kuperberg, Klaus Krogmann, and Ralf Reussner. Performance prediction for black-box components using reengineered parametric behaviour models. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering*, CBSE '08, pages 48–63, Berlin, Heidelberg, 2008. Springer-Verlag.

[52] S. le Cessie and J.C. van Houwelingen. Ridge estimators in logistic regression. *Applied Statistics*, 41(1):191–201, 1992.

[53] Dong Li, B.R. de Supinski, M. Schulz, K. Cameron, and D.S. Nikolopoulos. Hybrid mpi/openmp power-aware computing. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[54] Angela Lozano, Michel Wermelinger, and Bashar Nuseibeh. Assessing the impact of bad smells using historical information. In *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, IWPSE '07, pages 31–34. ACM, 2007.

[55] Luis M. Sánchez et al. *Target Platform Description Specification.* REPARA – Reengineering and Enabling Performance and poweR of Applications, 2014. ICT-609666-D3.1.

[56] Xiaohan Ma, Mian Dong, Lin Zhong, and Zhigang Deng. Statistical power consumption analysis and modeling for gpu-based computing, 2009.

[57] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y. G. Guéhéneuc, and E. Aimeur. Smurf: A svm-based incremental anti-pattern detection approach. In *2012 19th Working Conference on Reverse Engineering*, pages 466–475, Oct 2012.

[58] M.V. Mäntylä, J. Vanhanen, and C. Lassenius. Bad smells - humans as code critics. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 399–408, 2004.

[59] Gabriel Marin and John Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 2–13, New York, NY, USA, 2004. ACM.

[60] Radu Marinescu. Detecting design flaws via metrics in object-oriented systems. In *In Proceedings of TOOLS*, pages 173–182. IEEE Computer Society, 2001.

[61] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *In Proc. IEEE International Conference on Software Maintenance*, 2004.

[62] William B. McNatt and James M. Bieman. Coupling of Design Patterns: Common Practices and Their Benefits. In *Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, COMPSAC '01, pages 574–579, Washington, DC, USA, 2001. IEEE Computer Society.

[63] Tim Menzies, Bora Caglayan, Zhimin He, Ekrem Kocaguneli, Joe Krall, Fayola Peters, and Burak Turhan. The promise repository of empirical software engineering data, June 2012.

[64] N. Moha, Y. G. Guéhéneuc, L. Duchien, and A. F. Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, Jan 2010.

[65] T. H. Ng, S. C. Cheung, W. K. Chan, and Y. T. Yu. Do Maintainers Utilize Deployed Design Patterns Effectively? In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 168–177, Washington, DC, USA, 2007. IEEE Computer Society.

[66] NVIDIA Corporation. *NVIDIA Management Library (NVML) – Reference Manual*, March 2014. TRM-06719-001 _vR331.

[67] Anne-Cecile Orgerie, Marcos Dias de Assuncao, and Laurent Lefevre. A survey on techniques for improving the energy efficiency of large-scale distributed systems. *ACM Comput. Surv.*, 46(4):47:1–47:31, March 2014.

[68] Timothy Osmulski, Jeffrey T. Muehring, Brian Veale, Jack M. West, Hongping Li, Sirirut Vanichayobon, Seok-Hyun Ko, John K. Antonio, and Sudarshan K. Dhall. A probabilistic power prediction tool for the xilinx 4000-series fpga. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, IPDPS '00, pages 776–783, London, UK, UK, 2000. Springer-Verlag.

[69] D. E. Peercy. A software maintainability evaluation methodology. *IEEE Transactions on Software Engineering*, SE-7(4):343–351, July 1981.

[70] D.C. Pew Research Center, Washington. Mobile fact sheet, 2017.

[71] D. Pflüger and D. Pfander. Computational efficiency vs. maintainability and portability. experiences with the sparse grid code sg++. In *2016 Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE)*, pages 17–25, Nov 2016.

[72] Pico Technology Ltd. *PicoScope 4000 Series (A API) – Programmers Guide*, 2014. ps4000apg.en r1.

[73] J. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schoelkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1998.

[74] Louis-Noel Pouchet. Polybench: The polyhedral benchmark suite. *URL http://www-roc.inria.fr/ pouchet/software/polybench*, 2011.

[75] L. Prechelt, B. Unger, W.F. Tichy, P. Brössler, and L.G. Votta. A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions. *IEEE Transactions on Software Engineering*, 27:1134–1144, 2001.

[76] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, and W.F. Tichy. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *Software Engineering, IEEE Transactions on*, 28(6):595 –606, jun 2002.

[77] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[78] D. Rapu, S. Ducasse, T. Girba, and R. Marinescu. Using history information to improve design flaws detection. In *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*, pages 223–232, 2004.

[79] Rudolf Ferenc et al. *REPARA deliverable D7.4: Maintainability models of heterogeneous programming models*. 2015.

[80] A. Sabane, M. Penta, G. Antoniol, and Y. G. Guéhéneuc. A study on the relation between antipatterns and the cost of class unit testing. In *Proceedings of the Euromicro Conference on Software Maintenance and Reengineering, CSMR*, March 2013.

[81] Jie Shen, Jianbin Fang, H. Sips, and A.L. Varbanescu. Performance gaps between openmp and opencl for multi-core cpus. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 116–125, Sept 2012.

[82] S.K. Shevade, S.S. Keerthi, C. Bhattacharyya, and K.R.K. Murthy. Improvements to the smo algorithm for svm regression. In *IEEE Transactions on Neural Networks*, 1999.

[83] Connie U. Smith and Lloyd G. Williams. *Software Performance Engineering*, pages 343–365. Springer US, Boston, MA, 2003.

[84] A. Stoianov and I. Şora. Detecting patterns and antipatterns in software using prolog rules. In *Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on*, pages 253–258, May 2010.

[85] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical report, University of Illinois, at Urbana-Champaign, March 2012.

[86] G. Szőke, C. Nagy, P. Hegedűs, R. Ferenc, and T. Gyimóthy. Do automatic refactorings improve maintainability? an industrial case study. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 429–438, Sept 2015.

[87] H. Takizawa, K. Sato, and H. Kobayashi. Sprat: Runtime processor selection for energy-aware computing. In *Cluster Computing, 2008 IEEE International Conference on*, pages 386–393, Sept 2008.

[88] Adrian Trifu and Radu Marinescu. Diagnosing design problems in object oriented systems. In *Proceedings of the 12th Working Conference on Reverse Engineering*, WCRE '05, pages 155–164. IEEE Computer Society, 2005.

[89] A.W. Van Der Vaart. *Asymptotic Statistics*. Cambridge Series in Statistical and Probabilistic Mathematics, 3. Cambridge University Press, 1998.

[90] B. Venners. How to Use Design Patterns - A Conversation With Erich Gamma, Part I. 2005.

[91] M. Vokáč. Defect Frequency and Design Patterns: an Empirical Study of Industrial Code. *IEEE Transactions on Software Engineering*, 30(12):904 – 917, Dec. 2004.

[92] Marek Vokáč, Walter Tichy, Dag I. K. Sjøberg, Erik Arisholm, and Magne Aldrin. A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns - A Replication in a Real Programming Environment. *Empirical Software Engineering*, 9(3):149–195, September 2004.

[93] Y. Wang and I. H. Witten. Induction of model trees for predicting continuous classes. In *Poster papers of the 9th European Conference on Machine Learning*. Springer, 1997.

[94] L. Wendehals. Improving Design Pattern Instance Recognition by Dynamic Analysis. In *Proceedings of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA*, 2003.

[95] Peter Wendorff. Assessment of Design Patterns during Software Reengineering: Lessons Learned from a Large Commercial Project. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, CSMR '01, pages 77–, Washington, DC, USA, 2001. IEEE Computer Society.

[96] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? pages 306–315. IEEE, September 2012.

[97] L.T. Yang, Xiaosong Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 40–40, Nov 2005.

Corresponding Publications of the Author

[98] Dénes Bán. The connection of antipatterns and maintainability in firefox. In *10th Jubilee Conference of PhD Students in Computer Science (CSCS 2016), Szeged, Hungary, June 27 – 29*, 2016.

[99] Dénes Bán. The connection of antipatterns and maintainability in firefox. Accepted for publication in the 2016 Special Issue of Acta Cybernetica (extended version of [98]). 20 pages.

[100] Dénes Bán and Rudolf Ferenc. Recognizing antipatterns and analyzing their effects on software maintainability. In *14th International Conference on Computational Science and Its Applications (ICCSA 2014), Guimarães, Portugal, June 30 – July 3*, pages 337–352. Springer International Publishing, 2014.

[101] Dénes Bán, Rudolf Ferenc, István Siket, and Ákos Kiss. Prediction models for performance, power, and energy efficiency of software executed on heterogeneous hardware. In *13th IEEE International Symposium on Parallel and Distributed Processing with Applications (IEEE ISPA-15), Helsinki, Finland, August 20 – 22*, volume 3, pages 178–183, 2015.

[102] Dénes Bán, Rudolf Ferenc, István Siket, Ákos Kiss, and Tibor Gyimóthy. Prediction models for performance, power, and energy efficiency of software executed on heterogeneous hardware. Submitted to the Journal of Supercomputing (extended version of [101]). 24 pages.

[103] Péter Hegedűs, Dénes Bán, Rudolf Ferenc, and Tibor Gyimóthy. Myth or reality? analyzing the effect of design patterns on software maintainability. In *Advanced Software Engineering & Its Applications (ASEA 2012), Jeju Island, Korea, November 28 – December 2*, pages 138–145. Springer Berlin Heidelberg, 2012.