DEFECT PREDICTION ON THE HARDWARE REPOSITORY

- A CASE STUDY ON THE OPENRISC1000 PROJECT

SILVIU MUSA

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

GRADUATE PROGRAMME IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

YORK UNIVERSITY

TORONTO, ONTARIO

MARCH 2017

Abstract

Software defect prediction is one of the most active research topics in the area of mining software engineering data. The software engineering data sources like the code repositories and the bug databases contain rich information about software development history. Mining these data can guide software developers for future development activities and help managers to improve the development process. Nowadays, the computer-engineering field has rapidly evolved from 1972 until present times to the modern chip design, which looks superficially and very much like software design. Hence, the main objective of this thesis is to check whether it would be possible to apply software defect prediction techniques on hardware repositories. In this thesis, we have applied various data mining methods (e.g., linear regression, logistic regression, random forests, and entropy) to predict the post-release bugs of OpenRISC 1000 projects. We have conducted two types of studies: classification (predicting buggy and non-buggy files) and ranking (predicting the buggiest files). In particular, the classification studies show promising results with an average precision and recall of up to 74% and 70% for projects written in Verilog and close to 100% for projects written in C.

Acknowledgements

I would like to thank my supervisor Prof. Mokhtar Aboelaze for the support, guidance, and feedback which has helped me pursue my MSc. Equally, I would like to thank my co-supervisor Prof. Zhen Ming (Jack) Jiang for extensively using his time to provide me the best of guidance and support during this project. I am very grateful to Mr. Jeremy Bennett (Open Cores community) for providing all the needed additional information. My inexpressible appreciation goes to my family and friends. Their unconditional love and support have always been the main source of motivation in my personal and professional life. I would also like to thank the entire Graduate Studies Department of the Electrical Engineering and Computer Science at York University for making my master studies a wonderful experience.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

The hardware development complexity is exponentially increasing nowadays; a single scale of chips may contain multiple billions of transistors [25]. The growing design complexity requires efficient hardware quality assurance (QA) techniques that differ from the software QA. A major difference between software and hardware QA process is that the hardware QA is not done sequentially, the different functional blocks behave simultaneously making the QA process more difficult. Generally, two thirds of the total hardware design budget is used for design QA [2, 21]. Meanwhile, there is an increasing number of projects that were written using a hardware description language (e.g., VHDL). Hence, it would be worthwhile to investigate whether some of the software QA techniques would be applicable to the process of hardware QA.

One of the active research areas in software engineering is software defect prediction. Software Defect Prediction (SDP) is the line of research that is concerned with building prediction models, which leverage software metrics to predict defect-prone areas within a software system. The typical metrics include code complexity metrics (e.g., lines of code) or historical code change metrics (e.g., code churns). Together with the traditional software testing approach, the data obtained from SDP can be used to further improve the quality of various software systems. It would be worthwhile to investigate whether the software prediction techniques can be applied to detect potential issues in the hardware code.

With respect to applying SDP on hardware repositories, there is only one recent work [21] which predicts hardware defects on one large-scale commercial product. However, there are no published works on predicting defects on open source hardware repositories. Hence, this would be the focus of this thesis.

This thesis contains the defect prediction work done over a very popular open hardware project and open source development OpenRISC1000. Our goal was to perform an empirical investigation on this hardware project studying the relationship between various metrics (e.g., code complexity metrics and historical change metrics) and the post-release bugs. By applying the SDP techniques on hardware repositories, we were able to find the defect prone software modules. We focused on the post-release bugs rather than pre-release bugs or all bugs because of the post-release bugs, which are bugs discovered after the systems are released into the field, are more important with respect to the customer experience.

Thesis Organization

This thesis is organized as follows: Chapter 2 presents the OpenRISC1000 project and the background information about the SDP process. Chapter 3 provides an overview of our case study. Chapter 4 describes our experiments and discusses our results. Chapter 5 concludes this thesis and presents some future work.

Chapter 2
Background and Related Works

This chapter is organized as follows. First, we will describe the open source OpenRISC1000 project in Chapter 2.1. Then we will provide an overview of the related works in the area of SDP in Chapter 2.2.

2.1 OpenRISC1000 Overview

Since 1960, the embedded computers have evolved continuously as a result of the advances in the design and manufacture of microelectronic components and devices. Starting 1980, there is a new architectural approach regarding microprocessors, favouring a reduction in design complexity [12, 23].

The Reduced Instruction Set Computing (RISC) is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions, rather than a more specialized one, often found in other types of architectures. The most important characteristics of this family of processors are: one cycle execution time, pipelining and a large number of registers [2].

Almost in the same period of time (mid 80's) when the RISC architecture was designed, there was a movement in terms of open source and free software systems. The Free Software Foundation was born aiming to create the environment for less restrictive software regarding freedom to access and open of suffocating "usage protective measures" [1]. Fifteen years later, the philosophy was transferred to the discipline of hardware design. As a result, a large number of soft cores are currently available as open hardware. Cortex-M1, OpenRISC1000, OpenSPARC, LEON, Lattice Micro32, RISC-V are only a few of names from the list. The most popular of open source hardware systems are presented below.

Processor	Developer	Processor	Developer
AEMB	Shawn Tan	**Nios I, Nios II**	Altera
ARC	ARC International	**OpenFire**	Virginia Tech
Cortex-M1	ARM	**OpenRISC**	OpenCores
ERIC5	Entner Electronics	**OpenSPARC**	SUN
eSI-RISC	EnSillica	**T1**	SUN
JOP	Martin Schoeberi	**PacoBlaze**	Pablo Bleyer
Lattice Micro32	Lattice	**pAVR**	Doru Cuturela
LEON2	ESA	**PicoBlaze**	Xillinx
LEON 3/4	Aeroflex Gaisler	**RISC-V**	UC-Berkeley
MOL86	MicroCore Labs	**SecretBlaze**	U of Montpellier
Navre	Seb Bourdeauducq		

Table 1: Soft core processors [24]

In 1990, one of the most prominent projects aiming to develop an open source processor architecture was initiated by a group of students. These students had the goal to create a RISC processor design, including specifications and implementation. The name of the project was OpenRISC1000 and, at the same time, an online open source hardware design community called Open Cores was created.

The Open Source Hardware Community, known as Open Cores Community has more than 150,000 registered users [3]. There are 20 programmers assigned to be the maintainers of the project and the most popular names are: Damjan Lampret, Julius Baxter, Jeremy Bennett, and Stefan Kristiansson.

In the absence of a widely accepted open source hardware license, the components produced by the Open Cores initiative used initially several different software licenses. The Open Cores portfolio consists of multiple design elements from central processing units, memory controllers, peripherals, motherboards, and other components. The cores are implemented in the hardware description languages like Verilog, VHDL which may be synthesized to either silicon or gate arrays. Among the components created by Open Cores contributors are: OpenRISC1000 - a highly configurable RISC central processing unit, Amber (processor core) - an ARM-compatible RISC central processing unit, a ZilogZ80 clone, USB 2.0 controller, Tri Ethernet controller, 10/100/1000 Mbit. From the multitude of Open Cores community achievements, the OpenRISC1000 project has the largest popularity [13].

2.1.1 OpenRISC1000 Architecture

OpenRISC1000 is an open source 32-bit processor IP core that has been widely used in many academic and technologic projects [5]. Its nickname is OR1K and the project is one important component of ORP “Open RISC reference platform”.

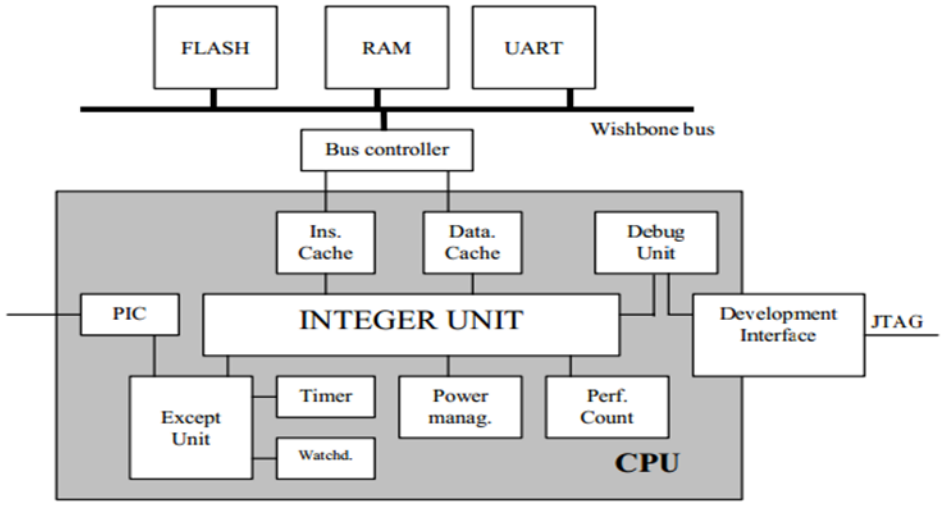

Figure 1: OpenRISC1000 – Component blocks diagram [13]

OpenRISC1000 provides a large area of implementations at a multitude of price per performance levels for a large range of industrial and telecommunication applications [20]. The basic implementations of the processor will only occupy 70% of a 50,000-gate Xilinx Virtex FPGA board, running at 80 MHz, reaching 80 MIPS.

The microprocessor block components are described in detail by Pablo Sanchez and Eugenio Villar in their paper "Using Open Source Cores in Real Applications" as being a 32/64-bitload microprocessor with store RISC architecture that has been designed with the emphasis on performance, simplicity, low power requirements, scalability and versatility [24]. The main CPU unit is implemented in 15,400 System C code lines (Figure 1) and has the following components described below according to the official manual [24].

The Integer Unit is the main part of the CPU designed to decode each instruction, to obtain the operation code and the operands. After each operation is performed, the integer unit writes the existing results. The integer unit is designed to allow the utilization of a higher clock frequency. The System C description of these units has 4,100 code lines.

Data and instruction caches are separate modules according to the hardware architecture of the design and are highly configurable. Cache size can be set from2 KBto8 KB and data block size can be set to 16, 32 or 64 bytes. Least Recently Used (LRU) is the algorithm used and it proved to be very efficient. This unit is described in 5,900 System C code lines.

Exception management unit is designed to calculate the address where the exception handler routine is placed and to decide which information about the status of the core must be stored for later restoring of the execution. This unit is described in 600 System C code lines.

Debug unit and development interface is an optional block that provides the possibility to create hardware breakpoints based on comparison conditions with stored or loaded values, data and instruction memory addresses. This unit is very closely related to the development interface which allows the debugging process to be completely in-system. Through the development interface, the debugging software can analyze the status of the CPU the memory content and trace information. The debug and development interface description takes 2,600 System C code lines.

Programmable interrupt controller allows the connection of 32 external masked interrupt lines through the interrupt line provided by the architecture. The OpenRISC1000 architecture only provides an insufficient number of interrupt thus a programmable interrupt controller (PIC) has to be implemented.

Tick timer facility provides the software with a precise clock reference. The tick timer generates an interrupt when the count reaches a programmed value.

Performance counters unit keeps a count of the number of times that a certain event has occurred. These events can be: instruction fetches, load and store accesses, cache misses and watch points.

Power management unit helps to better administer the power consumption of the core. This unit can perform modifications of the system clock frequency; it can shut down modules or can force the CPU to enter sleep mode.

Watchdog unit is used to prevent the CPU from entering into an endless loop or an erroneous routine from which the system cannot recover from.

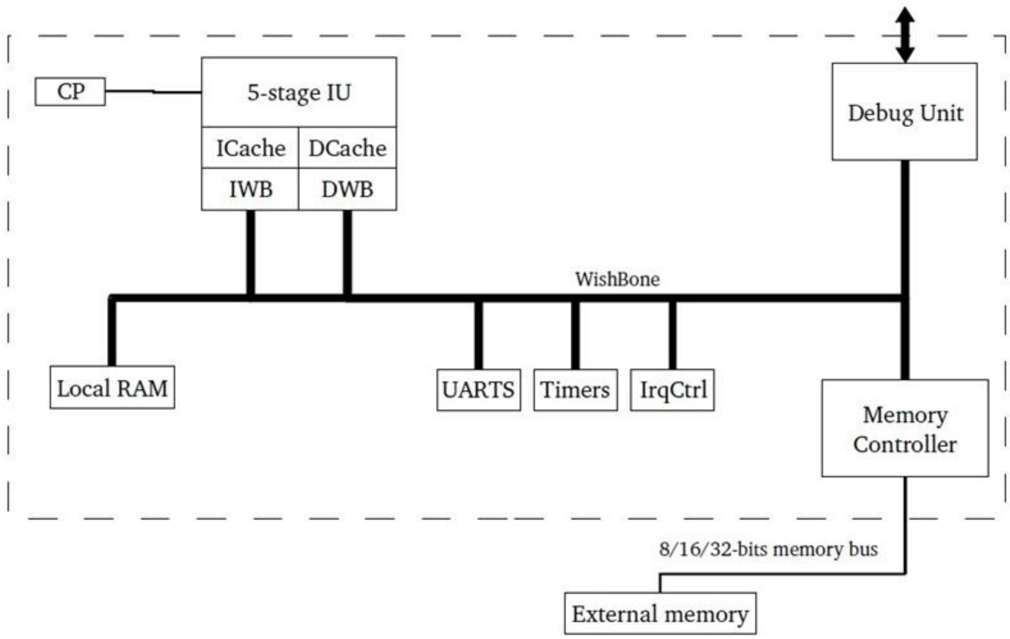

Figure 2: Functional Blocks for OR1200RTL [14]

The functional blocks are presented in the Figure 2. All of these blocks were designed with the focus on several principles such as ergonomics, efficiency, and lower power consumption [17].

The project started in 1998 aiming to develop reusable IP cores processors in Verilog/VHDL and since then each project has had several hundreds of code commits. First edition was launched in 2000 and since then up to the present times has been regularly updated to reflect the state of the latest implementations. The statistics provided by the Open Cores community show a total of 1,093 downloads per year. The project is known now as Design proven, ASIC proven, FPGA proven, but it is well known that still suffers historically from a lack of testing [24]. Knowing the increasing trend of design reuse, the developed IP cores have a high probability to be used in the research industry for the embedded control systems and fast FPGA prototyping and they can be implemented as well for day by day use. There are already a few examples of real implementations that have become a market success: Samsung DTV, Allwiner power controllers, and NASA for the control of TechEdStad.

2.1.2 OpenRISC1000 Repository Structure

OpenRISC 1000 project can be considered as a showcase of an open, modular standard specification geared towards real hardware implementations. The code for all projects in the OpenRISC IP Core family, such as OpenRISC1000, OpenRISC1200, and other builds, is combined into a single source tree and is hosted by the most popular web-based Git repository service GitHub (https://github.com/openrisc). The tree contains the complete source code for all the modules including project builds for each supported Operating System (OS) platforms. Linux is the main platform and the biggest parts of the project developments were done for it. In addition, the project has also been ported to Real Time Operating Systems like freeRTOS, RTEMS, and eCOS.

To fork all the OpenRISC repositories to a personal GitHub account and to download them to the local computer was the first mandatory step in our study. The complete project family, including compilers, tool chains, simulators and existing builds has a size on disk very close to 10 GB with 726,134 files and 47,543 folders. Most of the code was written in the Verilog, and C programming languages. Both languages have concepts of components, functions, methods and modules like any other standard programming languages.

The most important directories of the repository are the following:

- **Mor1kx** is a modular source base, the most recent version of the OR1K project. Mor1kx core is intended to replace the existing current OpenRISC1200 version. Mor1kx comes with 3 main configurations: Pronto-core is the configuration having three stages “delay-slot-free” and does not have a memory management unit (MMU); Espresso-core has a three-stage pipeline and does not have a MMU as well; Cappuccino-core has 6-stage pipeline and it can have MMUs and caches, which make it powerful enough to run Linux.
- **ORPSoc** is an OpenRISC reference platform (System on Chip) for further OS development. The trend is towards standard software distributions.
- **OpTiMSoc** is a tiled multi-core platform included in the OpenRISC project. The project can run on big FPGA boards.
- **Or1k-gcc** is the OR1KX Gnu collection compiler. Or1k-src it is a part of the tool chain containing the binutils and GDB libraries.
- **Or1ksim** is the official emulator for the Or1k project and is built as a single core.
- **Jor1k** is a JavaScript simulator written in Java.
- **UClibc-or1k** contains the uClibc embedded C libraries.
- **LLVM-Or1k (Low Level Virtual Machine for Or1k)** contains the preliminary support in LLVM and LINUX.

Each of the above-presented modules is hosted by GitHub as individual projects and each project can have up to 13 versions. The main branch is the one that accepts the current code commits. The testing is done using the mentioned simulators Orksim and Jor1k and Verilator (Verilog to System C) the dedicated testing tool made by the largest FPGA producer Xilinx. Using the existing results of the past testing activity and considering the limitations of resources in the software industry, it is very valuable to exactly predict the areas which are prone to failures in order distribute those resources properly to ensure the best results on launching software products.

By analyzing the existing data contained in the above systems, we can obtain valuable perspectives over their development processes [8, 18]. The large open software and hardware systems are increasingly important in the daily lives, and the software programming errors are affecting professionals at all levels. Hence, it is very important to have an exploratory study over a large project in order to demonstrate how the collected data and all the existing information on an open hardware project can help us to obtain even more knowledge and information.

2.2 Software Defect Prediction

A software defect, commonly referred as a “software bug” can be defined as an issue or deficiency in the software product which causes it to perform in an unexpected way [22]. Since defects in software can lead to malfunctioning of the system, which could in some cases affect the overall quality of the project, the general objective of software QA and validation activities is to release software with no known defects. Trying to achieve this goal, all the unexpected malfunctions are reported and documented in the defect database also known as online bug repository for the open source projects

[22]. All defects found during the software QA and validation activities become records of the repositories in a pre-defined format, often with the sole purpose of facilitating their resolution. Such repositories, which are usually accessible online, are generally called the bug repositories, or issue tracking repositories [4, 29].

The online databases usually provide the platform where worldwide programmers and testers can access the information about defects of their interest. They can add, edit, or update the information related to a given defect or comment, provide expertise or guidance to help resolve the defect, and track the progress of reported defects and monitor their statistics. To facilitate the documentation and exchange of information, various attributes are recorded for each reported defect. Some of these attributes are mandatory aimed at providing the basic information pertaining to given defect, while others are optional that provide additional details.

The most commonly used software QA and validation approach is software testing [26, 19]. Software testing verifies the behavior of the software system by executing the test scenarios and checking its runtime behavior against the specification. In addition to software testing, recently SDP is another complementary approach which can be used to reveal potential problems in a software system.

Software Defect Prediction (SDP) is an area of research that focuses on building prediction models, which use software metrics to predict defect-prone areas of a software system [28]. SDP uses various metrics extracted from the source code, the historical revisions, as well as the bug repositories [31].

There are many existing works in the area of SDP. For details please refer to [11]. We describe here the two pieces of works which are most relevant to this thesis.

1. The ways to practically identify and prioritize defects are addressed in multiple research papers in the software defect prediction field, including the one written by Zimmermann and Zeller, "*Predicting Defects for Eclipse*" [32]. This paper is the first work and proposes the idea of mining an open source, publicly available software repositories to predict potential defects.
2. Regarding mining hardware repositories, there is only one published work. It is written by Parizy, Takayama, and Kanazawa, called "Software Defect Prediction for LSI Designs" [21]. This is a study that was developed by Fujitsu Laboratories aiming to predict software defects by mining hardware repositories. This study was done in the industry using proprietary software defect database and proprietary hardware repositories focusing only on entropy studies.

However, there are no existing works which predict defects on open source hardware repositories. Hence, the current state of the research in the field gives us the opportunity to bring our contribution.

In this thesis, we have made a first empirical study on predicting bugs from open source hardware repositories. We have used OpenRISC1000 project and proceeded in our work using the existing state of the art methods and trying to see if there are better ways to leverage the software metrics. Although Parizy et al. [21] also studied the projects written in the Verilog language, they only applied the entropy as their prediction techniques. In this work, in addition to the entropy studies, we have used several other data mining techniques (e.g., regression and random forests) to predict the hardware defects.

Chapter 3
Case Study Overview and Setup

In this chapter, we describe our process of applying SDP techniques to predict potential defects in the hardware repositories. In particular, we explain the setup and the tools used in our case study.

3.1 Overview

The main objective of this thesis was to apply the SDP technique to a hardware repository. In this study, we picked OpenRISC1000 as our case study project. In order to achieve this objective, similar as described in Zimmermann and Zeller paper [31], our process consisted of five steps as illustrated in Figure 3: (1) data collection; (2) data cleaning; (3) complexity metrics calculation; (4) building prediction model; and (5) evaluation.

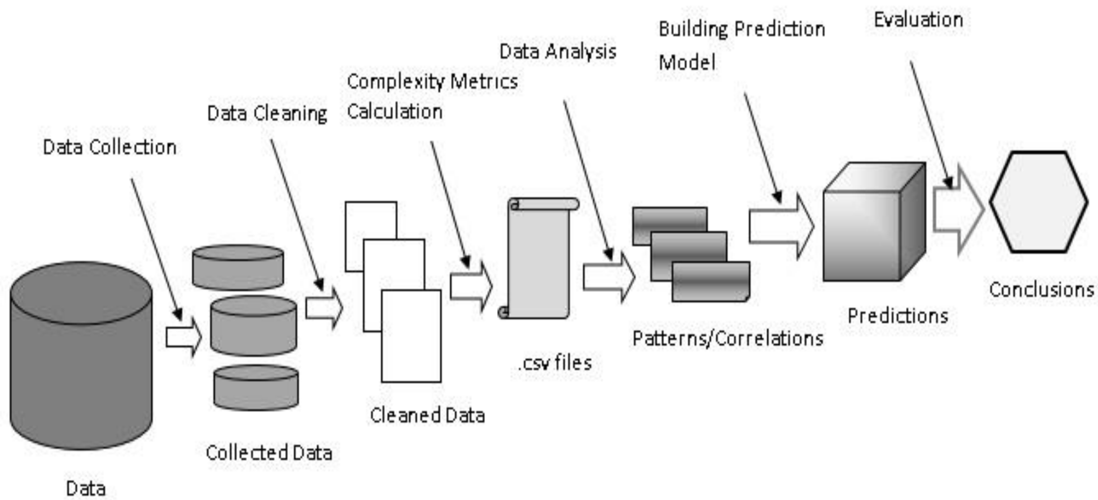

Figure 3: Our process to mine the OpenRISC1000 hardware repository

(1) During the *data collection* step, the historical data from the online OpenRISC1000 open source repository was extracted. There were two types of historical data in our study:

i. the GitHub repository, which contains the past revisions of the OpenRISC1000 source code; and

ii. the BugZilla database, which contains the reported defects for various releases of the OpenRISC1000 projects.

(2) During the *data cleaning* step, the unnecessary data was removed and the input data was converted into the desirable format which can be processed by the subsequent tools.

(3) During the *complexity metrics calculation* step, various complexity metrics were calculated for different releases of the software system. These complexity metrics were used in the later data mining models to predict the potential defects in the source code. Since there were two types of programming languages used in the OpenRISC1000

project, the Verilog and C. Hence, different types of complexity metrics were calculated for these two different programming languages.

(4) During the *data analysis* step, we studied the relationship between various code metrics with the bugginess of the systems. In particular, we explored the correlations between the complexity metrics and the past bug history.

(5) During the *building prediction model* step, various prediction models were built using the calculated complexity metrics as well as the past defect history. In particular, in this thesis, we studied two types of prediction models, classification, and ranking. For classification, we used two methods: logistic regression and random forest in order to determine which one provides better results. For ranking, we used the linear regression method.

(6) During the *Evaluation* step, the effectiveness of the prediction models was evaluated. In particular, we wanted to check whether the prediction models can be used to prediction bugs on the future releases of the systems.

3.2 Setup

There are three aspects related to the case study setup: (1) data download; and (2) the tools for complexity metrics calculation; and (3) the tool for building the prediction models.

3.2.1 Data Download

For the case study, we used two main sources of data available for the OpenRISC1000, the project repositories and the associated Bugzilla online database. Both were forked from the official location to a personal GitHub account and successfully downloaded to the local machine. For the hardware repository, the size of the download was approximately 9 GB. For Bugzilla online database, the size was around 70 MB.

3.2.2 Tools for Complexity Metrics Calculation

After collecting all the necessary data related to the study, the complexity metrics calculation phase naturally came next in the working flow. Software analysis generally extracts arbitrary properties of software source code. Software metrics provides various insights on various characteristics of the source code. Classic software metrics range in a large variety, from the very simple Source Lines of Code (SLOC) to more complex measures such as Cyclomatic Complexity measurements. Typical metrics report provides details on individual modules and summaries for subsystems. Such metrics are widely used to judge the quality or the complexity of source code.

The main advantages of using well-defined complexity metrics are their wide acceptance, unbiased assessment of source code quality, repeatability of measurements, ease of measurement, and the ability to judge progress in enhancing quality by comparing before and after assessments.

We built the bug prediction models using the complexity metrics guided by the thought that the most complex code would result in more bugs. However, as there were various complexity metrics proposed, our goal was to collect as many complexity metrics as possible when building the prediction models.

Since there were two types of programming languages in the OpenRISC1000 project, the Verilog and C, we used the HDL tool [16] to calculate the complexity metrics for the Verilog code and the UnderstandSCI [27] tool to calculate the complexity metrics for the C code.

The Hardware Description Language (HDL) Complexity Tool parses the Verilog code and calculates the code complexity metrics (Table 2).

Name	**Description**
Filename	the name of the file
Module	the name of the module
IO	input output elements
Net	design elements
McCabe	cyclomatic complexity
Sloc	lines of code
Comment lines	lines containing comments
Time	propagation time

Table 2: HDL-Complexity Metrics

The UnderstandSCI tool is a static analysis tool focused on source code comprehension and software metrics. The UnderstandSCI tool was widely used in various projects [16]. We used the UnderstandSCI tool to calculate the code complexity metrics for the C code. The following (Table 3) metrics were calculated:

Name	Description
Average Number of Blank Lines (Include Inactive)	the average number of lines that are not containing code
Average Number of Lines of Code (Include Inactive)	the average number of lines that are containing code
Average Number of Lines with Comments (Include Inactive)	the average number of lines that are containing comments
Blank Lines of Code (Include Inactive)	the number of lines that are not containing code
Lines of Code (Include Inactive)	the total number of lines of code
Lines with Comments (Include Inactive)	the total number of lines that are containing code
Average Cyclomatic Complexity	the number of linearly independent paths through a program's source code
Average Modified Cyclomatic Complexity	the average of the modified number of linearly independent paths through a program's source code
Average Number of Lines	the average number of lines
Average Number of Blank Lines	the average number of blanc lines
Average Number of Lines of Code	the average number of lines of code
Base Classes	the number of classes from which other classes are derived
Number of Children	the total number of direct subclasses
Classes	the total number of classes
Class Methods	the total number of methods
Class Variables	the total number of variables defined in a class
Number of Files	the total number of files
Function	the total number of functions
Instance Methods	the total number of subroutines
Instance Variables	the total number of variables defined in a class
Local Methods	the total number of local methods
Methods	the total number of methods
Modules	the total number of modules
Program Units	the total number of program units
Subprograms	the total number of subprograms

Table 3: C language – Complexity metrics

3.2.3 The Tool for Building the Prediction Models

We used the Waikato Environment for Knowledge Analysis (WEKA) as our tool to build the bug prediction model. WEKA is a machine learning workbench currently being developed at the University of Waikato. Its purpose is to allow users to access a variety of machine learning techniques for the purposes of experimentation and comparison using real world datasets. A workbench represents a set of tools bound together by the same user interface and each of these represents an individual program. The machine learning tools are written in a variety of programming languages (C, C++ and LISP). The last version of WEKA application includes multiple machine learning capabilities [10]. By providing the option to build a data mining model based on a training dataset, Weka has proven to be the ideal tool for our study.

Chapter 4
Experiments

In this chapter, we present our experiment on prediction bugs for the OpenRISC1000 project. The experiment was performed on a Dell XPS, I7-3770K single core desktop computer with 16 GB of RAM and 1 TB hard disk drive storage capacity.

4.1 Data Collection

4.1.1 The Hardware Repository

For this study, we used two main sources of available data for the OpenRISC1000, the project repositories and the associated Bugzilla online database. A total of 9.06 GB was downloaded to the local machine.

By looking through the downloaded data, we understood that this project started in 2001 and it is still ongoing at the present time. In total, there were 28 components/sub-projects inside the OpenRISC1000 project. Within these 28 sub-projects, there were more than 635,127 files overall. In total, there were more than 183,344,000 lines of code and 87,768,000 of statements for the entire project. Among them, 15,655 lines of code contained in 529 files were written in Verilog. The rest of 634,598 files containing 183,328,345 lines of code were written in C.

From all the OpenRISC1000 component sub-projects, we selected for our study only those that had multiple released versions in order to make possible future comparisons. As a result, only 2 sub-projects fulfilled this condition. To have a balanced study, we selected Mor1kx, the actual processor design written in Verilog, and Or1k-sim, the current simulator of the processor, written in C.

Verilog	**C**
Mor1kx	Or1k-sim
529 files	2,063 files
15,655 lines of code	704,430 lines of code

Table 4: Complexity metrics summary for Mor1kx and Or1k-sim

4.1.2 Bugs Collection

A copy of the project's post-release bugs was obtained from the online OpenRISC1000 Bugzilla Database. This database collects all the reported issues and modification requests that are submitted electronically by worldwide users. They are commonly referred to as "bug reports". This term is quite misleading as not all reported issues are defects, as some of the reported issues are just requests for optimization, or suggestion for different future improvements. Normally, every bug report should contain a variety of supporting meta-information such as a unique identification number, the software version, and the operating system it relates to, or the reporter's perceived importance. In addition, the entries should contain a short one-line summary of the issue at hand, followed by a more elaborate description. In our case, the additional information was missing for some of the reported bugs. Considering also the fact that the number of reported issues present in the database (106 instances in total) was too small to be used in the data mining study, we had to consider the second solution. Hence, we examined the commit logs of the online code repositories. Methodically, we analyzed all the existing logs of the code commits and we included the code revisions whose commit logs contained the words like “corrections” or “fixes”. Usually, when the developers submit their code, they usually include some short messages, called the commit logs, which contain the purpose of their submission. For example, if their code commit is for fixing some bugs, their commit log might contain phrases like “fixed 1067” or “bug #1025”. Figure 4 shows an example.

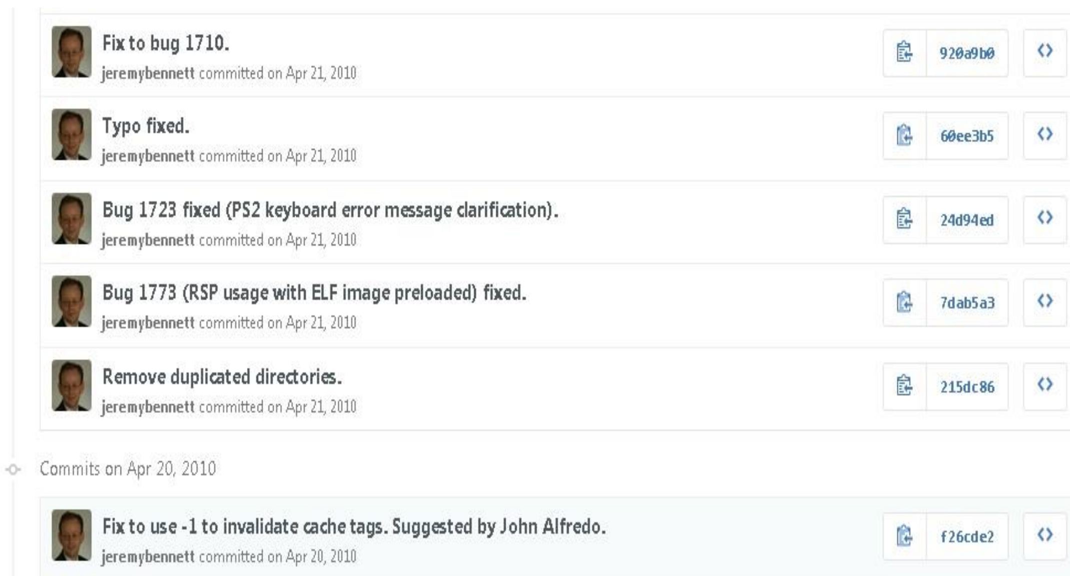

Figure 4: An example of code commit logs

We built a group of large spreadsheets for each sub-project containing the list of bugs, the patch for each modified file when fixing that bug and the date when the code commit was submitted.

Knowing the release date for each version and the date when each code commit was submitted, we checked and ensured that all the software defects were post-release bugs. Post-release bugs refer to bugs reported after that version of the system is released. We also took into consideration the existing comments inside the code commits. In this thesis, we focus on analyzing and predicting the post-release bugs is because these are the bugs which escape from software testing and can potentially impact the customer experience.

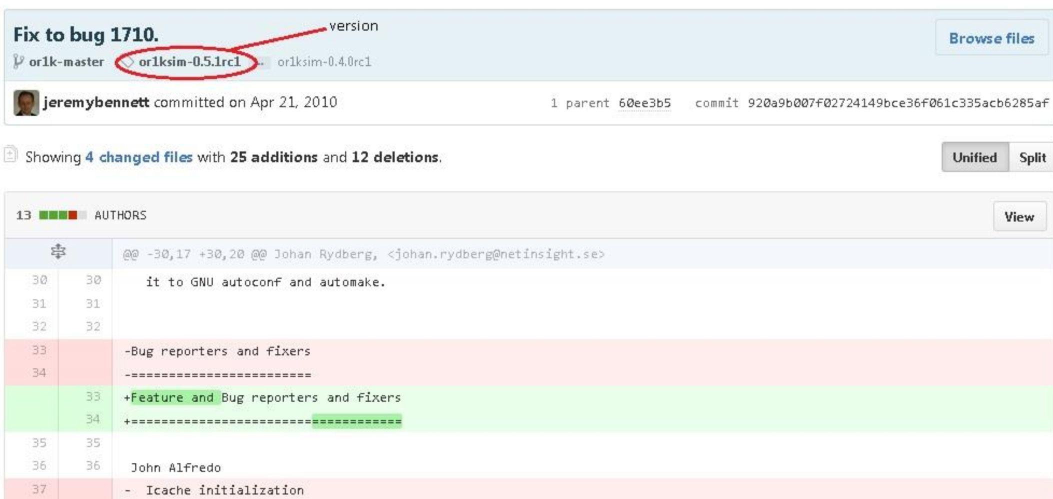

Figure 5: Code commits and according version

4.2 Data Cleaning

The second step in our data processing process is data cleaning. We verified if the data values were correct and conform to the existing dataset of rules. The non-useful information such as the "author" and "submission date" fields was removed. In this way, the existing data was prepared to be used for in depth analysis.

4.3 Complexity Metrics Calculation

Using the UnderstandSCI and HCT tools, we were able to extract from the analyzed repositories all the useful complexity metrics [6].

Verilog	C
Filename	Average Number of Blank Lines (Include Inactive)
Module	Average Number of Lines of Code (Include Inactive)
IO	Average Number of Lines with Comments (Include Inactive)
Net	Blank Lines of Code (Include Inactive)
Mccabe	Lines of Code (Include Inactive)
Sloc	Lines with Comments (Include Inactive)
Comment lines	Average Cyclomatic Complexity
Time	Average Modified Cyclomatic Complexity
AverageCyclomatic	Average Strict Cyclomatic Complexity
AvgCyclomaticModified	Average Essential Cyclomatic Complexity
Count Line	Average Essential Strict Modified Complexity
CountLineBlank	Average Number of Lines
CountLineCode	Average Number of Blank Lines
Count Line Comment	Average Number of Lines of Code
Count Stmt	Average Number of Lines with Comments
CountStmtDecl	Base Classes
CountStmtDecl	Number of Children
CountStmtExe	Classes
RatioComment	Class Methods
	Class Variables
	Number of Files
	Function
	Instance Methods
	Instance Variables
	Local Methods
	Methods
	Modules
	Inputs

Table 5: List of C and Verilog code complexity metrics used in our study

UnderstandSCI was used to analyze the C code and provided a generous set of metrics. The program proved to be very efficient in collecting them. Figure 6 shows a snippet of the collected metrics from the UnderstandSCI tool.

Name	AltAvgLin	AltAvgLin	AltAvgLin	AltCountL	AltCountL	AltCountL	AvgCyclor
\argtable2\arg_date.c	2	16	2	31	125	40	3
\argtable2\arg_dbl.c	2	15	3	27	113	41	3
\argtable2\arg_end.c	2	18	3	18	76	33	4
\argtable2\arg_file.c	1	14	2	36	139	41	3
\argtable2\arg_int.c	2	14	2	25	107	40	3
\argtable2\arg_lit.c	1	10	1	17	79	29	2
\argtable2\arg_rem.c	0	21	2	4	26	25	2
\argtable2\arg_rex.c	3	20	4	38	156	48	3
\argtable2\arg_str.c	2	13	2	24	102	37	2
\argtable2\argtable2.c	5	29	7	163	698	279	6
\argtable2\argtable2.h	0	0	0	41	222	85	0
\argtable2\getopt.c	10	88	20	119	593	311	17
\argtable2\getopt.h	0	0	0	19	49	71	0
\argtable2\getopt1.c	0	10	0	29	123	37	1
\bpb\branch-predict.c	2	13	1	52	179	87	3
\bpb\branch-predict.h	0	0	0	7	9	27	0
\build\config.h	0	0	0	91	75	117	0
\build\cpu\or32\execgen	258	5563	1021	261	5563	1046	899

Figure 6: Example of a .csv file

Most of the metrics in UnderstandSCI can be categorized as complexity metrics or volume metrics groups [30]. Cyclomatic complexity, Essential Complexity, Nesting level of Control Constructs are a few examples from the Complexity Metrics group. The Total number of Lines of Code, Total Number of Blank Lines or Total Number of Commented Lines, Number of Functions, Number of local Internal Methods are a few other examples belonging to the Volume Metrics group.

For Verilog language, the situation was a bit more complex. Using the HCT tool, a freely available program dedicated to Verilog code analysis, we were able to extract the most important code complexity metrics such us, Net, IO, McCabe, and Time.

Those metrics strictly describe the hardware design providing the Number of gates used (Net), the number of Input-Output elements (IO), the complexity of the design (McCabe) and the propagation time (Time) (Figure 7). They are essential in any hardware design analysis. We also understood that for a better study the set of metrics should be larger. Luckily, we found a way to enrich the Verilog set of metrics with code Volume Metrics.

FILENAME	MODULE	IO	NET	MCCABE	SLOC	COMMEN	TIME
mor1kx_wb~pre		11	0	1	38	17	0.1579
mor1kx.v		0	0	1	63	13	0.3978
mor1kx-sprs.v		0	0	1	253	33	0.033
mor1kx_rf~pres		10	11	1	65	25	0.3279
mor1kx_dmmu.		0	0	1	10	8	0.0037
mor1kx_cp~pre		2	0	1	55	12	0.3283
mor1kx_ct~ucci		0	0	1	18	16	0.0044
mor1kx_cp~ucci		2	0	1	59	12	0.3457
mor1kx_immu.		0	0	1	10	8	0.0038
mor1kx_tr~m_s		0	0	1	9	8	0.0036
mor1kx_dcache		0	0	1	10	8	0.0038
mor1kx_fe~ucci		0	0	1	14	12	0.0041

Figure 7: Mor1kx – HCT set of metrics

Going back to the UnderstandSCI tool, we made this powerful tool able to analyze Verilog Code. In this way, we extracted the Total number of Lines of Code, Total Number of Blank Lines or Total Number of Commented Lines and a few other useful metrics. The two complexity metrics tables, for each released version, were merged according to the commune column containing the path and the file name. An example of a merged table is showed in Figure 8.

Name	AvgCyclor	AvgCyclor	CountLine	CountLine	CountLine	CountLine	CountStm	CountStm	CountStm	RatioComn
C:\Users\	0	0	236	33	202	0	0	0	0	0
C:\Users\	0	0	283	29	253	0	0	0	0	0
C:\Users\	0	0	535	43	491	0	0	0	0	0
C:\Users\	0	0	53	11	41	0	0	0	0	0
C:\Users\	0	0	95	17	77	0	0	0	0	0
C:\Users\	0	0	192	33	158	0	0	0	0	0
C:\Users\	0	0	244	28	215	0	0	0	0	0
C:\Users\	0	0	567	44	522	0	0	0	0	0
C:\Users\	0	0	1297	44	1252	0	0	0	0	0
C:\Users\	0	0	751	39	711	0	0	0	0	0
C:\Users\	0	0	855	44	810	0	0	0	0	0
C:\Users\	0	0	1287	186	1100	0	0	0	0	0
C:\Users\	0	0	1435	219	1215	0	0	0	0	0
C:\Users\	0	0	1475	229	1245	0	0	0	0	0
C:\Users\	0	0	449	59	389	2	0	0	0	0.01

Figure 8: Mor1kx – UnderstandSCI set of metrics

In the next phase, we merged the total complexity metrics files with the bugs collection files resulted from harvesting the code commits for both projects Mor1kx and Or1k-sim.

The final resulting .csv files were enriched with 3 extra columns, the number of time when a file was modified after release, the number of bugs reported for that file, both being numeric fields and the field “buggy” a yes or no nominal field (Figure 9). Those last three fields were extremely important in our next steps.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
1	File Name	Lines	Statemen	Branch Sta	Lines with	Functions	Function S	Line Numl	Name of M	Complexit	Line Numl	Maximum	Average B	Average C	Statemen	Statemen	Statemen	Statemen	Statemen
2	or1k-gcc-gcc-4_3_1-	20	12	3	0	0	0	{undefined}		0	13	3	1.08	0	5	3	2	2	0
3	or1k-gcc-gcc-4_3_1-	838	426	92	175	4	116	393	GC_merge	13	713	9+	2.14	10	52	155	69	62	43
4	or1k-gcc-gcc-4_3_1-	1099	462	110	228	16	194	269	GC_maybe	12	439	6	1.66	3.94	62	196	105	51	35
5	or1k-gcc-gcc-4_3_1-	623	94	22	45	6	75	96	GC_regist	11	146	4	1.32	5.33	19	42	18	14	1
6	or1k-gcc-gcc-4_3_1-	469	219	45	105	16	173	211	add_edge	9	300	5	1.55	4.13	37	80	63	24	13
7	or1k-gcc-gcc-4_3_1-	300	121	27	65	5	45	86	GC_bl_init	6	189	4	1.4	3.2	25	45	32	15	4
8	or1k-gcc-gcc-4_3_1-	199	99	18	32	2	33	150	GC_check_	6	172	4	1.17	4	18	55	18	7	1
9	or1k-gcc-gcc-4_3_1-	919	580	141	115	32	484	361	CORD_sub	23	433	5	1.87	5.75	68	148	200	119	43
10	or1k-gcc-gcc-4_3_1-	396	287	125	45	7	270	56	extract_cc	69	215	8	3.78	11.29	17	45	21	55	31
11	or1k-gcc-gcc-4_3_1-	235	214	74	19	6	201	118	test_extra	37	33	4	1.39	13.33	13	116	75	8	2
12	or1k-gcc-gcc-4_3_1-	621	383	76	76	30	294	136	CORD_cm	16	150	4	1.37	3.67	62	186	81	41	13
13	or1k-gcc-gcc-4_3_1-	603	367	93	92	16	273	397	do_comm	44	506	6	1.89	6.63	54	140	63	55	24
14	or1k-gcc-gcc-4_3_1-	33	13	0	25	0	0	{undefined}		0	0	0	0	0	13	0	0	0	0
15	or1k-gcc-gcc-4_3_1-	366	208	53	42	6	187	47	WinMain(	11	253	6	2.52	4.2	21	41	46	50	24
16	or1k-gcc-gcc-4_3_1-	103	36	0	38	0	0	{undefined}		0	0	0	0	0	36	0	0	0	0
17	or1k-gcc-gcc-4_3_1-	510	173	38	59	7	154	304	GC_suspe	16	480	6	2.21	7.29	14	61	32	27	23
18	or1k-gcc-gcc-4_3_1-	1220	559	141	106	31	340	846	GC_debug	22	798	5	1.83	3.9	48	168	200	121	20
19	or1k-gcc-gcc-4_3_1-	1334	555	130	233	28	431	601	GC_regist	17	266	7	1.81	5.79	102	156	151	78	41
20	or1k-gcc-gcc-4_3_1-	959	488	101	127	17	309	570	GC_finaliz	38	685	6	1.95	5.53	47	186	101	77	53
21	or1k-gcc-gcc-4_3_1-	91	15	1	42	2	10	29	defined()	3	53	2	0.73	2	5	9	1	0	0
22	or1k-gcc-gcc-4_3_1-	516	190	26	180	1	1	468	_CleanUp_	1	281	4	1.03	1	53	89	38	9	1
23	or1k-gcc-gcc-4_3_1-	321	166	27	71	7	148	54	GC_init_g	9	159	4	1.46	5	18	73	57	16	2
24	or1k-gcc-gcc-4_3_1-	13	8	2	0	1	5	4	main()	3	7	2	0.88	3	3	3	2	0	0
25	or1k-gcc-gcc-4_3_1-	358	183	39	53	2	14	110	alloc_hdr(	3	277	5	1.62	2.5	24	78	41	27	10
26	or1k-gcc-gcc-4_3_1-	28	22	3	1	0	0	{undefined}		0	13	2	0.82	0	7	12	3	0	0
27	or1k-gcc-gcc-4_3_1-	41	26	3	4	0	0	{undefined}		0	20	2	0.88	0	8	13	5	0	0

Figure 9: Example of a csv file for a file written in C

The final resulting .csv files were converted into .arff type of file, which is the preferred dataset format by the Weka application (Figure 10).

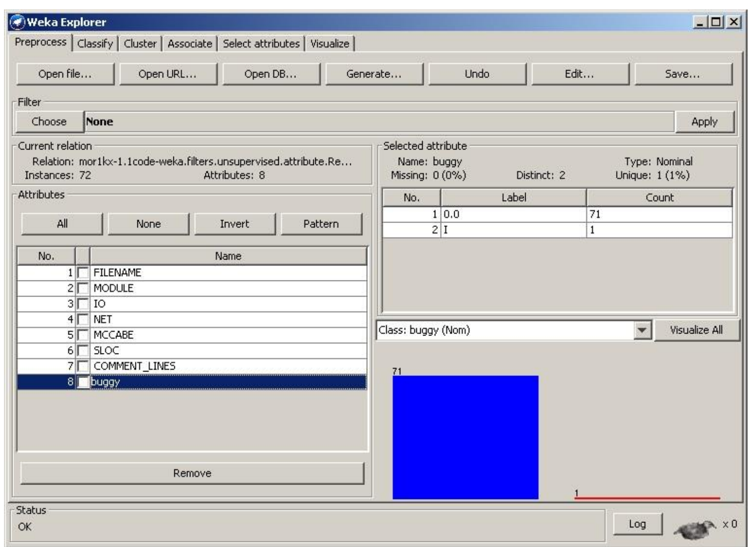

Figure 10: Weka – The Explore Interface

4.4 Data Analysis

4.4.1 The "Verilog" Subprojects Group

Mor1kx, the current design from the OpenRISC1000 processor family is the single package written in Verilog. Table 6 shows the existing released versions, the released date, the number of commits associated with each of them and the number of modified files before the next release.

Verilog					
Mor1kx					
version	**official name**	**Date**	**ID code**	**# of commits**	**# of touched files**
mor1kx.1.0	v1.0	9/1/2013	b5ca2ea	10	17
mor1kx.1.1	v1.1	9/9/2013	1eb23f2	1	1
mor1kx1.2	v.1.2	10/2/2103	143d9b8	34	65
mor1kx.2.1	v2.1	6/22/2014	7358c97	21	58
mor1kx.2.2	v2.2	8/8/2014	83d3415	57	73
mor1kx.2.3	v2.3	12/9/2014	91acc03	2	2
mor1kx.3.1	v3.1	12/13/2014	39c074a	15	20
mor1kx.4.1	v4.1	11/3/2015	b8c1a18	6	9
			Bug-fixing commits	146	245
			Total # of commits	594	

Table 5: Released versions for Mor1kx project

The statistics per entire Mor1kx project were resumed in the Table 7.

Project version	IO	NET	MCCABE	SLOC	COMM
Mork1x-1.0	84	10	40	2,268	692
Mork1x-1.1	158	10	30	2,268	692
Mork1x-1.2	158	32	38	2,268	692
Mor1kx-2.1	126	10	40	2,377	788
Mor1kx-2.2	94	24	42	2,377	788
Mor1kx-2.3	58	2	36	2,377	788
Mor1kx-3.1	22	0	34	1,611	715
Mor1kx-4.1	62	18	35	1,293	592

Table 6: Mor1kx HCT metrics summary

4.4.2 The "C" Subprojects Group

The second analyzed subproject ORK-Sim written in C had also eight released versions. They are presented in Table 8.

By using the UnderstandSCI tool, we were able to compute the complexity metrics for each package and each version.

An example of some complexity metrics statistics for each Or1k-sim released version is presented in the next table.

C					
Or1k-sim					
version	**official name**	**date**	**ID code**	**# of commits**	**# of touched files**
or1ksim-0.3.0	or1ksim-0.3.0	5/25/2009	0b63f32	3	148
or1ksim-0.4.0rc1	or1ksim-0.4.0rc1	6/3/2010	48c3d23	12	130
or1ksim-0.4.0rc2	or1ksim-0.4.0rc2	6/16/2010	c7a1d5e	2	14
or1ksim-0.4.0	or1ksim-0.4.0	6/22/2010	27806aa	15	208
or1ksim-0.5.0rc1	or1ksim-0.5.0rc1	9/7/2010	ef54033	2	22
or1ksim-0.5.0rc2	or1ksim-0.5.0rc2	10/2/2010	b55e843	21	131
or1ksim-0.5.0rc3	or1ksim-0.5.0rc3	2/11/2011	ae337dc	2	50
or1ksim-0.5.1rc1	or1ksim-0.5.1rc1	4/8/2011	7672c7e	45	179
			Bug-fixing commits	102	882
			Total # of commits	129	

Table 7: Released versions for Or1k-sim project

We wanted to study the following two research questions regarding the defect prediction studies on the OpenRISC1000 hardware repository:

1. **Classification**: to predict the files which contain software bugs (e.g., Mor1kx or Or1k-sim) and,
2. **Ranking**: to predict the files which contain the largest number of software defects.

4.4.3 Classification

The first analysis that had to be done was naturally the classification. Using this method, we were able to differentiate between buggy or not buggy files. In this work, we compared the results built using the random forest and logistic regression model to see which prediction model provides the best results.

- *Logistic regression* is used to estimate the probability of a binary response based on one or more predictor variables. Logistic regression measures the relationship between the categorical dependent variable, in our case "buggy or not buggy" and one or more independent variables. To classify files/packages as defect-prone or not based on code metrics we define the following based on the values outputted by the logistic regression model:

$$\text{Defect Classification} = \begin{cases} \text{defect-prone } (0.5 < \text{value} \leq 1) \\ \text{defect-free } (0 \leq \text{value} \leq 0.5) \end{cases}$$

- *Random forest* is another useful prediction method when predicting a binary outcome [15], in our case post-release bugs from a set of continuous or categorical variables. Compared to the logistic regression model, in which there are a few assumptions associated with the model (e.g., the normality of the data, the balance of the output data, etc.), the random forest model is less constrained.

Since there were more “bug-free” files than the “buggy” files, we had to re-sample the data before we can train them using the logistic regression model. The data was sampled automatically before proceeding with the logistic regression method to every dataset. By doing this, we ensured that for every test dataset the number of buggy records equals the number of the not buggy ones. Every version from both projects was used as a training dataset and all the remaining versions became one after another the test dataset.

Below we describe the performance metrics used to evaluate the effectiveness of the defect prediction models: precision, recall, and F-measure.

To properly explain the above three metrics, we need to consider the following prediction outcomes (Table 9):

- True positive (**TP**): buggy instances predicted as buggy
- False positive (**FP**): clean instances predicted as buggy
- True negative (**TN**): clean instances predicted as clean
- False negative (**FN**): buggy instances predicted as clean

		Actual Results	
		Yes	No
Predictive Results	Yes	TP (true positive)	FP (false positive)
	No	FN (false negative)	TN (true negative)

Table 8: Classification table

With these prediction outcomes, which are mostly used in the defect prediction literature, the following measures are defined:

- *Precision* – measures the ratio of the correctly classified positive modules to the set of positive modules.

$$precision = \frac{TP}{TP + FP}$$

- *Recall* – measures the ratio of correctly predicted positive modules in the whole modules with defects.

$$recall = \frac{TP}{TP + FN}$$

- *F-measure* – is the harmonic mean of precision and recall.

$$\mathrm{F} = \frac{2}{\left(\frac{1}{\mathrm{Precision}} + \frac{1}{\mathrm{Recall}}\right)}$$

Having this knowledge and based on the output values, the data can be interpreted immediately and it is easy to know if a file or a package contains software defects or not.

The Precision, Recall and F-measure values for the logistic regression models are listed below (Tables 10, 11, 12).

Precision									
testing\ training	mor1kx-1.0	mor1kx-1.1	mor1kx-1.2	mor1kx-2.1	mor1kx-2.2	mor1kx-2.3	mor1kx-3.1	mor1kx-4.1	average
mor1kx-1.0		0.25	0.974	0.563	0.612	0.25	0.573	0.813	0.576
mor1kx-1.1	1		0.974	0.594	0.612	0.833	0.573	0.917	0.786
mor1kx-1.2	0.875	1		0.563	0.584	0.833	0.537	0.917	0.758
mor1kx-2.1	0.837	1	0.751		0.612	0.25	0.837	0.292	0.654
mor1kx-2.2	0.612	0.25	0.612	0.657		0.5	0.786	0.4	0.545
mor1kx-2.3	1	1	1	0.612	0.612		0.643	0.708	0.796
mor1kx-3.1	0.569	1	0.431	0.597	0.563	1		0.5	0.665
mor1kx-4.1	0.661	1	0.633	0.5	0.359	0.833	0.407		0.627

Table 9: The prediction of the Mor1kx (Verilog) experiment built using the logistic regression prediction model - Precision

Recall									
testing\ training	mor1kx-1.0	mor1kx-1.1	mor1kx-1.2	mor1kx-2.1	mor1kx-2.2	mor1kx-2.3	mor1kx-3.1	mor1kx-4.1	average
mor1kx-1.0		0.5	0.972	0.563	0.611	0.5	0.571	0.7	0.631
mor1kx-1.1.	1		0.972	0.594	0.611	0.75	0.571	0.9	0.771
mor1kx-1.2	0.833	1		0.563	0.583	0.75	0.536	0.9	0.737
mor1kx-2.1	0.833	1	0.75		0.611	0.5	0.821	0.3	0.687
mor1kx-2.2	0.611	0.5	0.611	0.656		0.5	0.786	0.4	0.58
mor1kx-2.3	1	1	1	0.611	0.611		0.643	0.7	0.795
mor1kx-3.1	0.556	1	0.444	0.594	0.556	1		0.5	0.664
mor1kx-4.1	0.611	1	0.583	0.5	0.444	0.75	0.464		0.621

Table 10: The prediction of the Mor1kx (Verilog) experiment built using the logistic regression prediction model - Recall

F-Measure

testing\ training	mor1kx-1.0	mor1kx-1.1	mor1kx-1.2	mor1kx-2.1	mor1kx-2.2	mor1kx-2.3	mor1kx-3.1	mor1kx-4.1	average
mor1kx-1.0		0.333	0.972	0.563	0.61	0.333	0.569	0.67	0.578
mor1kx-1.1	1		0.972	0.593	0.61	0.733	0.569	0.899	0.768
mor1kx-1.2	0.829	1		0.561	0.583	0.733	0.53	0.899	0.733
mor1kx-2.1	0.833	1	0.75		0.61	0.333	0.819	0.293	0.662
mor1kx-2.2	0.61	0.333	0.61	0.656		0.5	0.786	0.4	0.556
mor1kx-2.3	1	1	1	0.61	0.61		0.643	0.697	0.794
mor1kx-3.1	0.532	1	0.416	0.59	0.543	1		0.495	0.653
mor1kx-4.1	0.579	1	0.54	0.382	0.345	0.733	0.367		0.536

Table 11: The prediction of the Mor1kx (Verilog) experiment built using the logistic regression prediction model - F-Measure

The above charts show that the vast majority of the values for precision, recall, and accuracy are in the [0.5:1] interval. It means we can use the logistic regression method to perform bug prediction on all mor1kx versions.

Similarity, we used one version of the data as training data for the random forest model and tested it against other versions. The tables below show the precision, recall and F-measure number for the Mor1kx project obtained by using this second method (Tables 13, 14, 15 for Mor1kx).

By analyzing the precision, recall, and accuracy results obtained through the random forest method, we can reach the same conclusion as for the logistic regression method: we can use the random forest method to perform bug prediction on all mor1kx versions. In addition, the prediction performance for the random forest method is better than the logistic regression method.

Precision									
testing\ training	mor1kx-1.0	mor1kx-1.1	mor1kx-1.2	mor1kx-2.1	mor1kx-2.2	mor1kx-2.3	mor1kx-3.1	mor1kx-4.1	average
mor1kx-1.0		1	1	0.65	0.594	0.974	0.644	0.946	0.829
mor1kx-1.1.	0.832		0.757	0.76	0.237	0.974	0.776	0.904	0.748
mor1kx-1.2	1	1		0.598	0.594	0.974	0.622	0.946	0.819
mor1kx-2.1	0.861	0.974	0.951		0.732	0.951	0.76	0.716	0.849
mor1kx-2.2	0.677	0.924	0.679	0.736		0.896	0.683	0.735	0.761
mor1kx-2.3	0.832	1	0.757	0.765	0.757		0.787	0.834	0.818
mor1kx-3.1	0.875	0.975	0.759	0.75	0.639	0.954		0.771	0.817
mor1kx-4.1	0.849	1	0.771	0.367	0.586	0.946	0.654		0.739

Table 12: The prediction of the Mor1kx (Verilog) experiment built using the random forest prediction model - Precision

Recall									
testing\ training	mor1kx-1.0	mor1kx-1.1	mor1kx-1.2	mor1kx-2.1	mor1kx-2.2	mor1kx-2.3	mor1kx-3.1	mor1kx-4.1	average
mor1kx-1.0		1	0.947	0.619	0.595	0.973	0.649	0.946	0.818
mor1kx-1.1	0.784		0.514	0.459	0.486	0.973	0.649	0.892	0.679
mor1kx-1.2	1	1		0.595	0.595	0.973	0.622	0.946	0.818
mor1kx-2.1	0.676	0.459	0.946		0.73	0.486	0.703	0.405	0.629
mor1kx-2.2	0.568	0.514	0.676	0.73		0.486	0.649	0.459	0.583
mor1kx-2.3	0.784	1	0.514	0.486	0.514		0.676	0.865	0.691
mor1kx-3.1	0.838	0.676	0.73	0.703	0.622	0.703		0.595	0.695
mor1kx-4.1	0.811	1	0.568	0.405	0.514	0.946	0.649		0.699

Table 13: The prediction of the Mor1kx (Verilog) experiment built using the random forest prediction model - Recall

F-Measure

testing\ training	mor1kx-1.0	mor1kx-1.1	mor1kx-1.2	mor1kx-2.1	mor1kx-2.2	mor1kx-2.3	mor1kx-3.1	mor1kx-4.1	average
mor1kx-1.0		1	0.973	0.634	0.594	0.969	0.646	0.946	0.823
mor1kx-1.1	0.711		0.376	0.318	0.318	0.969	0.535	0.859	0.583
mor1kx-1.2	1	1		0.596	0.594	0.969	0.622	0.946	0.818
mor1kx-2.1	0.696	0.601	0.946		0.729	0.603	0.706	0.489	0.681
mor1kx-2.2	0.598	0.66	0.675	0.731		0.612	0.654	0.541	0.638
mor1kx-2.3	0.711	1	0.376	0.37	0.376		0.588	0.839	0.608
mor1kx-3.1	0.846	0.782	0.724	0.7	0.613	0.784		0.657	0.729
mor1kx-4.1	0.761	1	0.477	0.289	0.411	0.946	0.569		0.636

Table 14: The prediction of the Mor1kx (Verilog) experiment built using the random forest prediction model - F-Measure

Based on the existing data, we have plotted 3 graphs where logistic regression and random forest can be accurately compared (Figures 11, 12, 13).

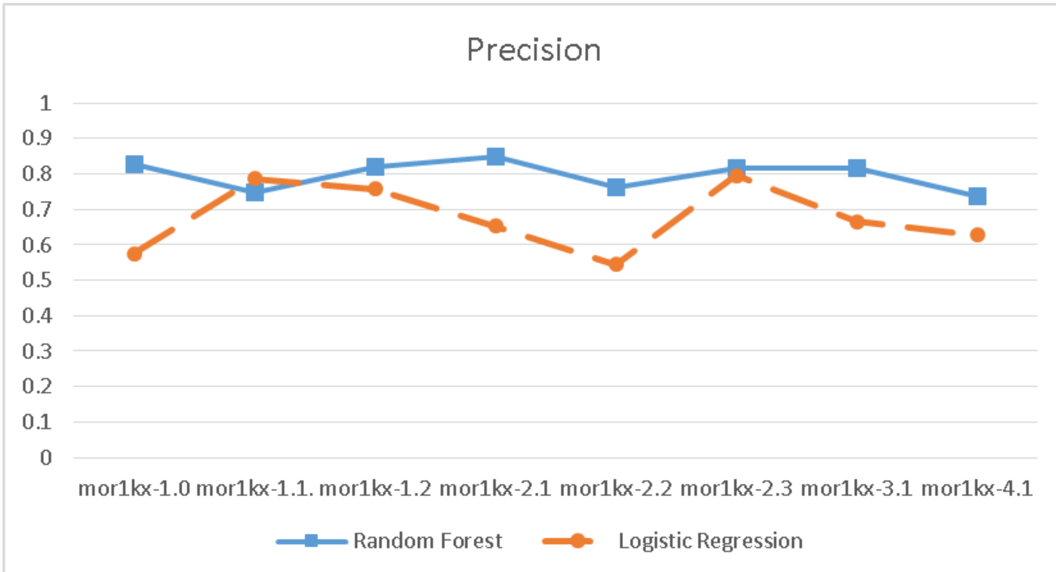

Figure 11: Mor1kx – Precision

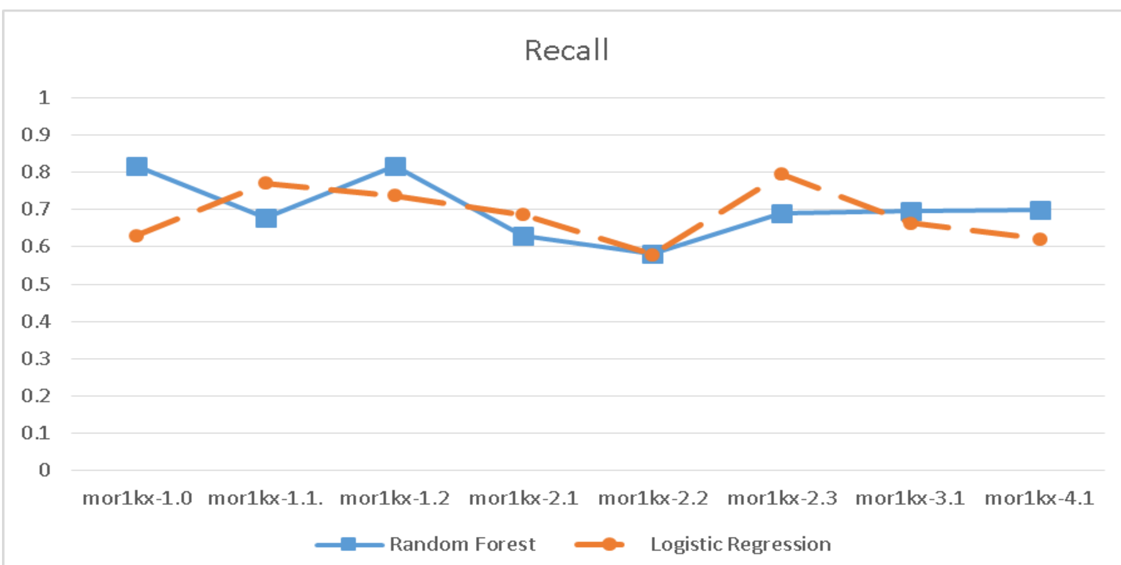

Figure 12: Mor1kx – Recall

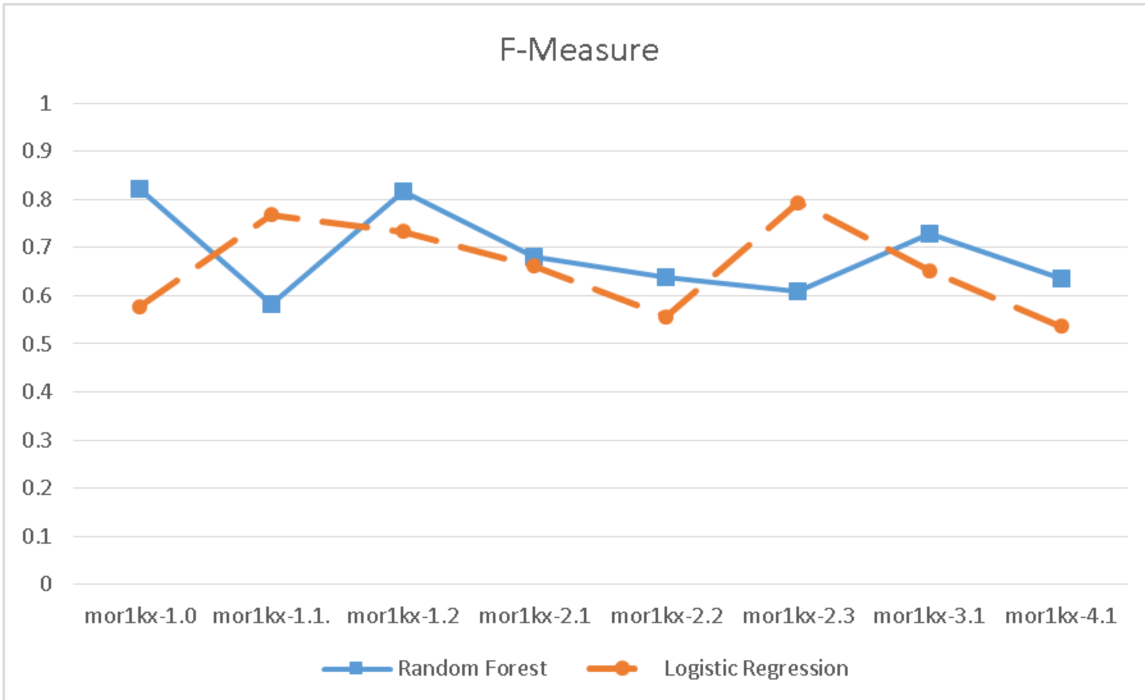

Figure 13: Mor1kx - F-Measure

We did a similar study for the Or1k-sim subproject. We applied both the logistic regression and the random forest methods. In this second study, the values for precision, recall, and accuracy were in the [0.92:0.99] interval. Similar as the mor1kx versions, the precision, recall, and F-measure values obtained through the random forest method were a little bit higher than the ones obtained through the logistic regression. The data is shown in the following tables.

Precision									
test\training	or1k-sim.0.3.0	or1k-sim.0.4.0	or1k-sim.0.4.0.rc1	or1k-sim.0.4.0.rc2	or1k-sim.0.5.0.rc1	or1k-sim.0.5.0.rc2	or1k-sim.0.5.0.rc3	or1k-sim.0.5.1.rc1	average
or1k-sim.0.3.0		0.97	0.978	1	0.978	0.957	0.987	0.953	0.974
or1k-sim.0.4.0	0.963		0.936	0.982	0.981	0.944	0.969	0.902	0.953
or1k-sim.0.4.0.rc1	0.948	0.922		0.979	0.96	0.906	0.94	0.922	0.939
or1k-sim.0.4.0.rc2	0.943	0.912	0.95		0.921	0.922	0.93	0.887	0.923
or1k-sim.0.5.0.rc1	0.959	0.942	0.958	0.965		0.937	0.974	0.93	0.952
or1k-sim.0.5.0.rc2	0.979	0.983	0.967	0.991	0.991		0.982	0.966	0.979
or1k-sim.0.5.0.rc3	0.979	0.959	0.962	0.996	0.987	0.974		0.953	0.972
or1k-sim.0.5.1.rc1	1	0.987	0.986	1	0.991	0.987	1		0.993

Table 15: The prediction of the Or1k-sim (C) experiment built using the logistic regression prediction model - Precision

Recall									
test\training	or1k-sim.0.3.0	or1k-sim.0.4.0	or1k-sim.0.4.0.rc1	or1k-sim.0.4.0.rc2	or1k-sim.0.5.0.rc1	or1k-sim.0.5.0.rc2	or1k-sim.0.5.0.rc3	or1k-sim.0.5.1.rc1	average

or1k-sim.0.3.0		0.969	0.978	1	0.978	0.955	0.987	0.951	0.974
or1k-sim.0.4.0	0.961		0.947	0.982	0.982	0.946	0.973	0.901	0.956
or1k-sim.0.4.0.rc1	0.952	0.929		0.973	0.964	0.919	0.951	0.915	0.943
or1k-sim.0.4.0.rc2	0.939	0.902	0.947		0.96	0.915	0.964	0.87	0.928
or1k-sim.0.5.0.rc1	0.957	0.938	0.956	0.982		0.933	0.973	0.924	0.951
or1k-sim.0.5.0.rc2	0.978	0.982	0.969	0.991	0.991		0.982	0.964	0.979
or1k-sim.0.5.0.rc3	0.978	0.96	0.964	0.996	0.987	0.973		0.951	0.972
or1k-sim.0.5.1.rc1	1	0.987	0.987	1	0.991	0.987	1		0.993

Table 16: The prediction of the Or1k-sim (C) experiment built using the logistic regression prediction model - Recall

F-Measure									
test\training	or1k-sim.0.3.0	or1k-sim.0.4.0	or1k-sim.0.4.0.rc1	or1k-sim.0.4.0.rc2	or1k-sim.0.5.0.rc1	or1k-sim.0.5.0.rc2	or1k-sim.0.5.0.rc3	or1k-sim.0.5.1.rc1	average
or1k-sim.0.3.0		0.966	0.975	1	0.973	0.949	0.985	0.946	0.97
or1k-	0.953		0.936	0.982	0.981	0.938	0.969	0.88	0.948

F-Measure									
test\training	or1k-sim.0.3.0	or1k-sim.0.4.0	or1k-sim.0.4.0.rc1	or1k-sim.0.4.0.rc2	or1k-sim.0.5.0.rc1	or1k-sim.0.5.0.rc2	or1k-sim.0.5.0.rc3	or1k-sim.0.5.1.rc1	average
sim.0.4.0									
or1k-sim.0.4.0.rc1	0.95	0.922		0.976	0.962	0.908	0.945	0.898	0.937
or1k-sim.0.4.0.rc2	0.914	0.863	0.927		0.94	0.882	0.947	0.817	0.898
or1k-sim.0.5.0.rc1	0.946	0.926	0.944	0.973		0.915	0.965	0.911	0.94
or1k-sim.0.5.0.rc2	0.976	0.982	0.966	0.991	0.991		0.982	0.962	0.978
or1k-sim.0.5.0.rc3	0.976	0.957	0.96	0.995	0.985	0.971		0.946	0.97
or1k-sim.0.5.1.rc1	1	0.987	0.986	1	0.99	0.986	1		0.992

Table 17: The prediction of the Or1k-sim (C) experiment built using the logistic regression prediction model - F-Measure

Random forest method

Precision

test\training	or1k-sim.0.3.0	or1k-sim.0.4.0	or1k-sim.0.4.0.rc1	or1k-sim.0.4.0.rc2	or1k-sim.0.5.0.rc1	or1k-sim.0.5.0.rc2	or1k-sim.0.5.0.rc3	or1k-sim.0.5.1.rc1	average
or1k-sim.0.3.0		0.97	0.978	1	0.978	0.957	0.987	0.953	0.974
or1k-sim.0.4.0	0.963		0.936	0.982	0.981	0.944	0.969	0.902	0.953
or1k-sim.0.4.0.rc1	0.948	0.922		0.979	0.96	0.906	0.94	0.922	0.939
or1k-sim.0.4.0.rc2	0.943	0.912	0.95		0.921	0.922	0.93	0.887	0.923
or1k-sim.0.5.0.rc1	0.959	0.942	0.958	0.965		0.937	0.974	0.93	0.952
or1k-sim.0.5.0.rc2	0.979	0.983	0.967	0.991	0.991		0.982	0.966	0.979
or1k-sim.0.5.0.rc3	0.979	0.959	0.962	0.996	0.987	0.974		0.953	0.972
or1k-sim.0.5.1.rc1	1	0.987	0.986	1	0.991	0.987	1		0.993

Table 18: The prediction of the Or1k-sim (C) experiment built using the random forest prediction model - Precision

Recall									
test\training	or1k-sim.0.3.0	or1k-sim.0.4.0	or1k-sim.0.4.0.rc1	or1k-sim.0.4.0.rc2	or1k-sim.0.5.0.rc1	or1k-sim.0.5.0.rc2	or1k-sim.0.5.0.rc3	or1k-sim.0.5.1.rc1	average
or1k-sim.0.3.0		0.969	0.978	1	0.978	0.955	0.987	0.951	0.974

Recall									
test\training	or1k-sim.0.3.0	or1k-sim.0.4.0	or1k-sim.0.4.0.rc1	or1k-sim.0.4.0.rc2	or1k-sim.0.5.0.rc1	or1k-sim.0.5.0.rc2	or1k-sim.0.5.0.rc3	or1k-sim.0.5.1.rc1	average
or1k-sim.0.4.0	0.961		0.947	0.982	0.982	0.946	0.973	0.901	0.956
or1k-sim.0.4.0.rc1	0.952	0.929		0.973	0.964	0.919	0.951	0.915	0.943
or1k-sim.0.4.0.rc2	0.939	0.902	0.947		0.96	0.915	0.964	0.87	0.928
or1k-sim.0.5.0.rc1	0.957	0.938	0.956	0.982		0.933	0.973	0.924	0.951
or1k-sim.0.5.0.rc2	0.978	0.982	0.969	0.991	0.991		0.982	0.964	0.979
or1k-sim.0.5.0.rc3	0.978	0.96	0.964	0.996	0.987	0.973		0.951	0.972
or1k-sim.0.5.1.rc1	1	0.987	0.987	1	0.991	0.987	1		0.993

Table 19: The prediction of the Or1k-sim (C) experiment built using the random forest prediction model - Recall

F-Measure									
test\training	or1k-sim.0.3.0	or1k-sim.0.4.0	or1k-sim.0.4.0.rc1	or1k-sim.0.4.0.rc2	or1k-sim.0.5.0.rc1	or1k-sim.0.5.0.rc2	or1k-sim.0.5.0.rc3	or1k-sim.0.5.1.rc1	average

or1k-sim.0.3.0		0.966	0.975	1	0.973	0.949	0.985	0.946	0.97
or1k-sim.0.4.0	0.953		0.936	0.982	0.981	0.938	0.969	0.88	0.948
or1k-sim.0.4.0.rc1	0.95	0.922		0.976	0.962	0.908	0.945	0.898	0.937
or1k-sim.0.4.0.rc2	0.914	0.863	0.927		0.94	0.882	0.947	0.817	0.898
or1k-sim.0.5.0.rc1	0.946	0.926	0.944	0.973		0.915	0.965	0.911	0.94
or1k-sim.0.5.0.rc2	0.976	0.982	0.966	0.991	0.991		0.982	0.962	0.978
or1k-sim.0.5.0.rc3	0.976	0.957	0.96	0.995	0.985	0.971		0.946	0.97
or1k-sim.0.5.1.rc1	1	0.987	0.986	1	0.99	0.986	1		0.992

Table 20: The prediction of the Or1k-sim (C) experiment built using the random forest prediction model - F-Measure

Similar graphs were plotted to compare the logistical regression and random forest methods, for Or1k-sim project in this case (Figures 14, 15, 16).

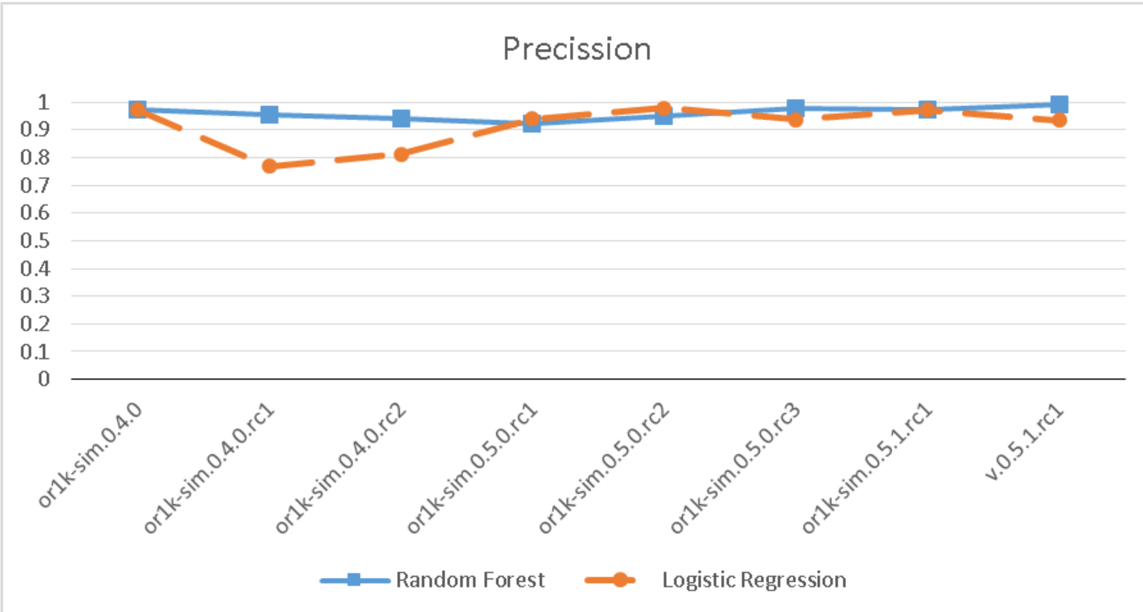

Figure 14: Or1k-sim- Precision

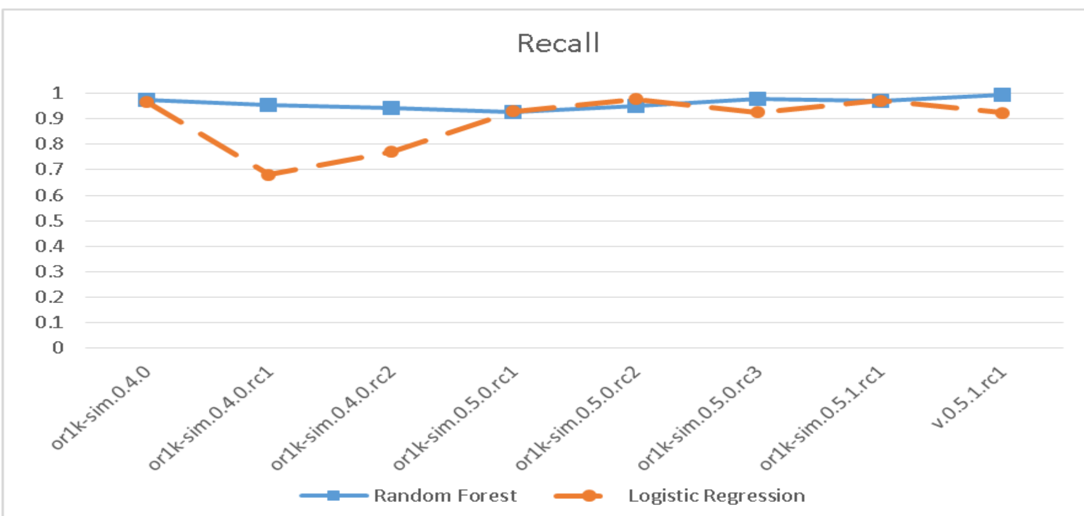

Figure 15: Or1k-sim Recall

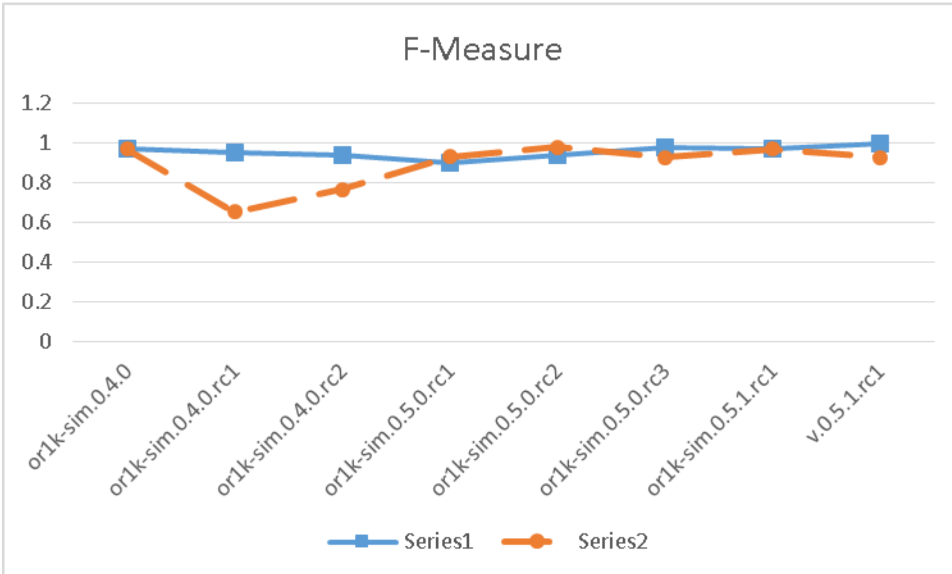

Figure 16: Or1k-sim F-Measure

Using classification through logistic regression, we were able to determine which released versions were prone to software defects. The data indicated that the Mor1kx.1.1 and Or1k-sim.0.4.0.rc2 were the most prone to bugs versions.

In general, the random forest models had higher precision than the logistic regression models for both Or1k and Mor1kx sub-projects. For the Or1k sub-project, the random forest model generally out-performed the logistic regression model in all three metrics (precision, recall, and F-measure).

4.4.4 Ranking

To answer the second question related to which files and released versions contain the most of the software defects, we applied the linear regression method and the entropy method to the “number of bugs field” for every remaining version of those 2 projects.

- Given a dataset $\{y_i,\, x_{i1}, \ldots, x_{ip}\}_{i=1}^{n}$ of n statistical units, a linear regression model assumes that the relationship between the dependent variable y_i and the p-vector of regressors x_i is linear.

Another idea to predict the number of bugs is based on the entropy of changes. Hassan et al proposed the use of Shannon Entropy defined as [2]: $H = -\sum_{i=1}^{M} P_i \log_2 P_i$. The idea consists in measuring over a time interval how the changes are distributed in a system. The more spread they are, the higher the complexity is.

To evaluate the effectiveness of the above two ranking approaches, we used the Spearman correlation coefficient as a statistical tool to measure the strength of the relationship between two sets of data. The values of Spearman's correlation coefficient can be between -1 and +1. A positive correlation coefficient indicates a positive relationship between the two variables (the higher the x values, also the higher the y values), while a negative correlation coefficient expresses a negative relationship (the lower the x values, the lower the y values). A correlation coefficient of 0 indicates that there is no relationship between the two studied variables.

In our case, the Spearman's correlation coefficient measures the correlation between the predicted bugs and the existing observed bugs. The results are shown in Tables 22 and 23.

Spearman Correlation Coefficient									
testing\training	mor1kx-1.0	mor1kx-1.1	mor1kx-1.2	mor1kx-2.1	mor1kx-2.2	mor1kx-2.3	mor1kx-3.1	mor1kx-4.1	average
mor1kx-1.0		0.2035	0.8824	0.584	0.6509	0.2164	0.4369	0.2301	0.457
mor1kx-1.1.	0.8356		0.9541	0.9724	0.9513	0.9227	0.8743	0.955	0.923
mor1kx-1.2	0.7179	0.0116		0.3585	0.0978	0.0111	0.1542	0.0362	0.198
mor1kx-2.1	0.0551	0.0202	0.4266		0.1241	0.0207	0.0724	0.1166	0.119
mor1kx-2.2	0.0107	0.1244	0.0906	0.0188		0.1238	0.5347	0.0311	0.133
mor1kx-2.3	0.4444	0.9964	0.6525	0.5799	0.6065		0.3685	0.5129	0.594
mor1kx-3.1	0.4852	0.1868	0.7065	0.5914	0.8507	0.1817		0.1883	0.455
mor1kx-4.1	0.4981	0.3755	0.8581	0.8544	0.8623	0.372	0.462		0.611

Table 21: Mor1kx – Spearman correlation coefficient

Spearman Correlation Coefficient									
test\ training	or1k-sim.0.3.0	or1k-sim.0.4.0	or1k-sim.0.4.0.rc1	or1k-sim.0.4.0.rc2	or1k-sim.0.5.0.rc1	or1k-sim.0.5.0.rc2	or1k-sim.0.5.0.rc3	or1k-sim.0.5.1.rc1	average

Spearman Correlation Coefficient									
test\ training	or1k-sim.0.3.0	or1k-sim.0.4.0	or1k-sim.0.4.0.rc1	or1k-sim.0.4.0.rc2	or1k-sim.0.5.0.rc1	or1k-sim.0.5.0.rc2	or1k-sim.0.5.0.rc3	or1k-sim.0.5.1.rc1	average
or1k-sim.0.3.0		0.5091	0.4742	0.1457	0.1992	0.3708	0.0592	0.3314	0.298
or1k-sim.0.4.0	0.2735		0.5004	0.3237	0.1721	0.3143	0.1133	0.3545	0.293
or1k-sim.0.4.0.rc1	0.3319	0.6121		0.5757	0.1023	0.3883	0.048	0.5802	0.376
or1k-sim.0.4.0.rc2	0.346	0.694	0.7974		0.2134	0.6562	0.1716	0.6666	0.506
or1k-sim.0.5.0.rc1	0.2888	0.4892	0.3908	0.0681		0.4863	0.0996	0.4665	0.327
or1k-sim.0.5.0.rc2	0.1778	0.4677	0.3842	0.1773	0.151		0.5416	0.4202	0.0331
or1k-sim.0.5.0.rc3	0.1336	0.3688	0.2776	0.0452	0.0976	0.7112		0.3857	0.288
or1k-sim.0.5.1.rc1	0.1002	0.3966	0.5174	0.3172	0.1502	0.3214	0.1168		0.274

Table 22: Or1k-sim – Spearman correlation coefficient

By studying the Spearman correlation coefficient obtained by applying the logistic regression method, we noticed that the complexity metrics and the number of software defects were positively correlated. The coefficient values were in the 0.119 and 0.923 interval. The strongest correlation coefficient (0.923) was observed when we used Mor1k-1.1 as the training dataset and all the other remaining versions as the testing dataset. The weakest correlation coefficient (0.119) was observed when Mor1kx-2.1 was used as the training dataset and all remaining versions as the testing dataset.

For Or1k-sim, the correlation coefficients values were even smaller, being in the interval [0.274:0.506]. The highest correlation coefficient was obtained when or1k-sim-0.4.0rc2 was used as the training dataset.

We listed as well the root relative squared error for both Mor1k and Or1k-sim groups, which measure the goodness of fit for the linear regression models (Tables 24 and 25).

Root relative squared error								
testing\ training	mor1kx-1.0	mor1kx-1.1	mor1kx-1.2	mor1kx-2.1	mor1kx-2.2	mor1kx-2.3	mor1kx-3.1	mor1kx-4.1
mor1kx-1.0		155.31%	65.26%	85.53%	82.84%	148.03%	91.73%	121.76%
mor1kx-1.1.	53.25%		42.71%	56%	50.56%	41.81%	57.68%	41.28%
mor1kx-1.2	132.33%	151.90%		104.59%	117.10%	150.40%	159.52%	157.81%
mor1kx-2.1	148.87%	138.15%	106.92%		118.76%	138.16%	154.04%	141.32%
mor1kx-2.2	151.34%	140.85%	135.82%	136.40%		141.34%	134.99%	146.82%
mor1kx-2.3	90.71%	8.55%	91.70%	93.99%	93.67%		93.80%	87.21%
mor1kx-3.1	108.15%	141.34%	70.87%	81.06%	65.12%	144.11%		129.98%
mor1kx-4.1	86.87%	135.11%	70.22%	76.28%	74.37%	136.22%	87.22%	

Table 23: Mor1kx – Linear regression - Root relative squared error

or1k-sim.0.3.0	or1k-sim.0.4.0	or1k-sim.0.4.0.rc1	or1k-sim.0.4.0.rc2	or1k-sim.0.5.0.rc1	or1k-sim.0.5.0.rc2	or1k-sim.0.5.0.rc3	or1k-sim.0.5.1.rc1
	86.79%	88.63%	201.52%	141.26%	92.91%	170.38%	98.11%
145.19%		94.64%	239.95%	192.25%	101.10%	221.43%	112.35%
137.98%	87.07%		252.19%	73.02%	74.86%	88.93%	60.27%
93.60%	81.14%	76.72%		102.26%	87.19%	108.63%	81.76%
102.46%	86.78%	91.93%	171.77%		88.86%	137.60%	88.09%
168.53%	114.54%	114.61%	269.32%	212.90%		212.54%	119.40%
107.86%	92.42%	96.18%	163.66%	118.33%	81.68%		91.96%
150.50%	105.48%	91.78%	206.06%	175.86%	99.38%	99.38%	

Table 24: Or1k-sim – Linear regression – Root relative squared error

Based on the tables above, we obtained the following charts (Figures 17 and 18):

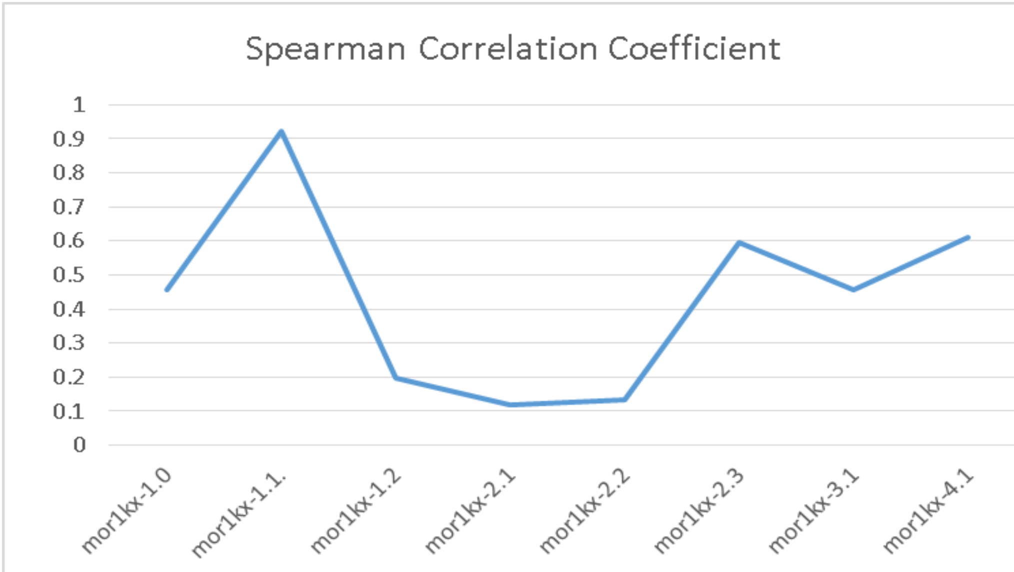

Figure 17: Mor1kx- Linear Regression - Spearman correlation coefficient

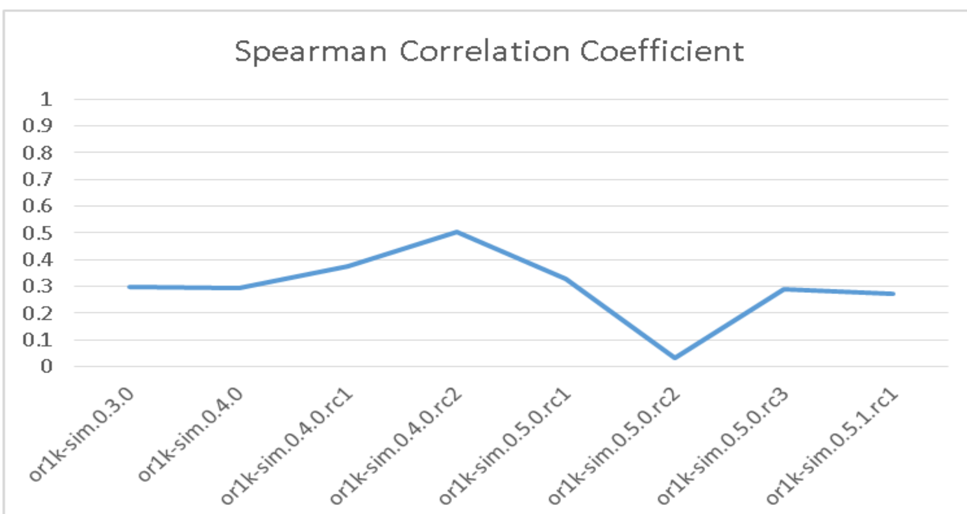

Figure 18: Or1k-sim – Linear Regression – Spearman correlation coefficient

All correlations with metrics were positive. It is obvious that an increase in the file complexity from one version to the other implies and increase the probability of containing defects but this relation is very strong only for mor1kx-1.1, mor1kx-2.3, and mor1kx-4.1.

Trough ranking and linear regression, we detected Mor1kx-1.1 and Or1k-sim.0.4.0.rc2 as being the released versions having the largest number of software defects.

The second approach was to predict the number of bugs based on the entropy of changes. We present below the calculated Shannon Entropy value for each analyzed version (Tables 26 and 27).

	Shannon Entropy
Mor1kx-1.0	3.175
Mor1kx-1.1	1
Mor1kx-1.2	4.115
Mor1kx-2.1	4.054
Mor1kx-2.2	4.120
Mor1kx-2.3	1
Mor1kx-3.1	3.64
Mor1kx-4.1	2.725

Table 25: Mor1kx – Entropy of modified files

	Shannon Entropy
Or1k-sim.0.3.0	6.02
Or1k-sim.0.4.0	6.7
Or1k-sim.0.4.0.rc1	5.04
Or1k-sim.0.4.0.rc2	3.80
Or1k-sim.0.5.0.rc1	4.45
Or1k-sim.0.5.0.rc2	5.586
Or1k-sim.0.5.0.rc3	4.508
Or1k-sim.0.5.1.rc1	5.88

Table 26: Or1k-sim - Entropy of modified files

To use the entropy of code changes, Hassan et al defined the weighted history of complexity metric (WHCM) where each modified file gets the entropy of the system, in our case the release version, weighted with the probability of the file being modified [2]

$$HCPF_i(j) = \begin{cases} c_{ij} * H_i, & j \in F_i \\ 0, & otherwise \end{cases}$$

We proceeded with the calculation of each file weight C_{ij} for each file of each version. As a final result, we obtained .csv files containing three fields, file name, bugs and WHCM. The figure below presents an example of those values.

Name	bugs	cijH
\mor1kx-defines.v	0	0
\mor1kx-sprs.v	2	0.112876712
\mor1kx.v	5	0.282191781
\mor1kx_branch_prediction.v	0	0
\mor1kx_bus_if_avalon.v	0	0
\mor1kx_bus_if_wb32.v	2	0.112876712
\mor1kx_cfgrs.v	0	0
\mor1kx_cpu.v	0	0
\mor1kx_cpu_cappuccino.v	3	0.282191781
\mor1kx_cpu_espresso.v	5	0.282191781
\mor1kx_cpu_prontoespresso.v	1	0.112876712
\mor1kx_ctrl_cappuccino.v	2	0.169315068
\mor1kx_ctrl_espresso.v	7	0.451506849
\mor1kx_ctrl_prontoespresso.v	1	0.056438356
\mor1kx_dcache.v	1	0.056438356
\mor1kx_decode.v	4	0.451506849
\mor1kx_decode_execute_cappu	0	0

Figure 19: Entropy .csv file example

As a final step of our entropy studies, we performed the linear regression method (LR) on the entropy field and we listed the following charts. We compared the Spearman coefficient obtained by applying the linear regression analysis to every released version with the Spearman coefficient obtained by applying the linear regression analysis to the entropy field and we plotted the graphs below (Figures 20 and 21).

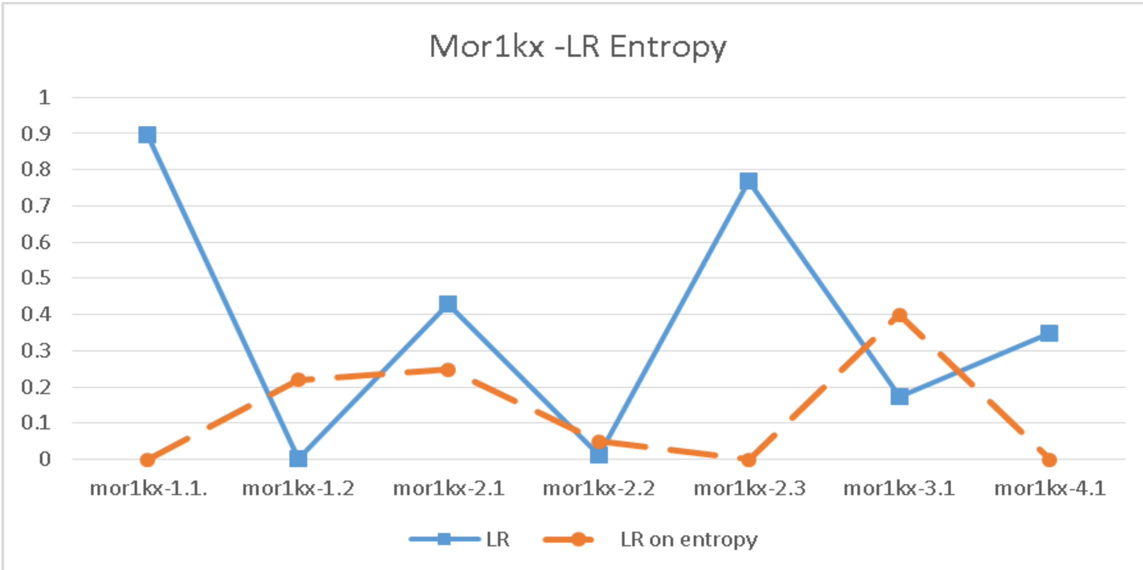

Figure 20: Mor1kx- Comparison between linear regression (LR) and linear regression on entropy

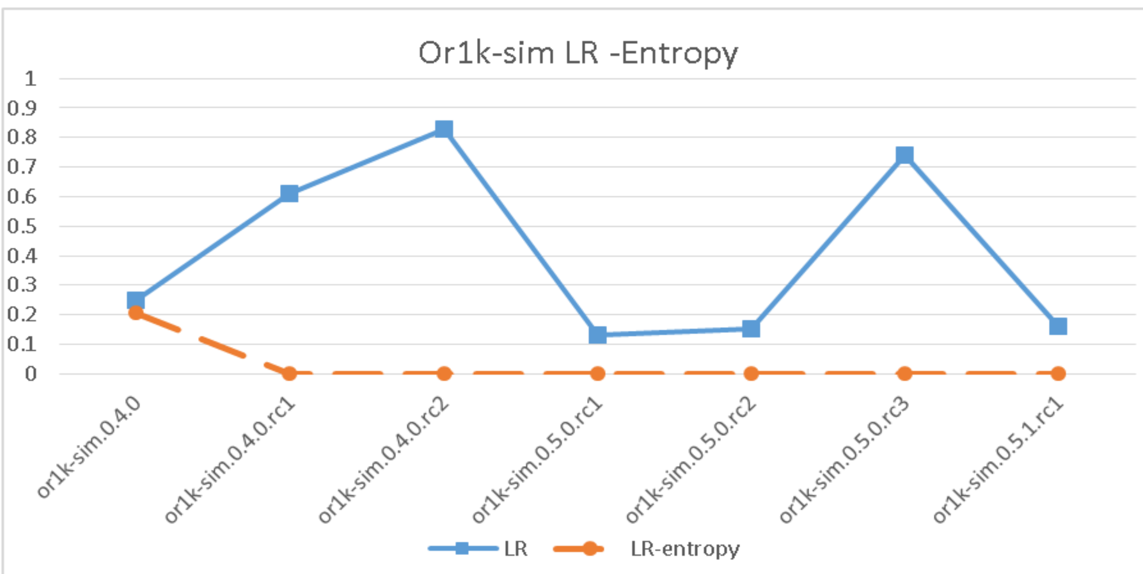

Figure 21: Or1k-sim - Comparison between linear regression and linear regression on entropy

We identified the Mor1kx-1.1 and Mor1kx-2.3 as having the highest probability to have bugs in the future. However, the ranking study was not as good as the classification study due to the very small number of files that had changed between the versions and the frequently changed files were different between each two adjacent versions. Hence, the historical data cannot help too much in terms of predicting the buggiest files in the future.

Chapter 5
Conclusions and Future Work

The importance of mining software repositories field increased over the years due to the availability of a large number of free and available software and hardware repositories [9]. While mining software repositories is an area that has been booming over the last 12 years, hardware repository mining has just started. As a result, we had the chance to do a first empirical defect prediction study on an open source hardware project. Based on the similarities between hardware description languages and the software programming languages, we successfully adapted the existing SDP work to the open source hardware design making a step ahead in the industry and opening the door to other similar studies.

The project was a big challenge due to the size of the project. During our study, by exploring the relationship between the code complexity metrics and the hardware design bugs, we were able to answer which of the released version for the Verilog processor design and the C simulator had the largest number of defects.

We were able to demonstrate that the random forest model provided slightly more accurate results than logistic regression when predicting which files were more bug-prone (classification). This was probably because of the random forest, which is an ensemble method and is more robust towards noisy data.

In terms of predicting which files have more bugs (ranking), we found that the linear regression model built using the complexity metrics was more effective than the entropy model built using the historical data. We think this is likely due to the lack of historical change data (a.k.a., less number of revisions in the hardware repositories than the software repositories).

Our work provides a good starting point for mining hardware repositories considering that the programming languages for hardware systems are very different from the software systems and the complexity metrics are also different.

The future work will include the study of additional complexity metrics for the hardware-based programming languages in order to increase the precision for predicting bugs in hardware repositories. In addition, we also plan to contact the open source hardware development communities to gather feedback on our research. Finally, we will look into new approaches to improve the performance of our ranking studies.

Bibliography

1. Baxter, J. "*Open Source Hardware Development and the OpenRISC Project*", IMIT, 2012
2. Baxter, J., Lampret, D. "*OpenRISC 1200 IP Core Specification*" (Preliminary Draft), January 2011.
3. Bennett, J. "*Integrating the GNU Debugger with Cycle Accurate Models: A Case Study using a Verilator SystemC Model of the OpenRISC 1000*", Embecosm Application Note 7, Issue 1, March 2009.
4. Betenburg, N., Hassan, A.E. "*Studying the impact of Social Structures on Software Quality*", The 18th IEEE International Conference on Program Comprehension, Portugal, 2010.
5. Clarke, P. "*Free 32-bit processor hits the Net*", EE Times, February 2008.
6. Dinari, F. "*Halstead Complexity Metrics in Software Engineering*", 2015.
7. Hassan, A.E. "*Minning Software Repositories to Assist Developers and Support Managers*", 2006.
8. Hassan, A.E. "*The Road Ahead for Mining Software Repositories*", Software Analysis and Intelligence Lab, School of Computing, Queen's University, Canada, 2006.
9. Hassan, A.E., Holt, R.C., Mockus, A. "*Mining Software repositories*", MSR, Edinburgh, Scotland, 2004.
10. Holmes, G., Donkin, A., Witten, I.H. "*WEKA: A machine Learning Workbench*", University of Waikato, Hamilton, NewZealand, 2001.
11. Kamei, Y., Shihab, E. *"Defect Prediction: Accomplishments and Future Challenges"*. In proceedings of 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016.
12. Lammers, D. "*VSIA's new leader has revitalization plan*", EE-design, 2003.
13. Lampret, D. "*OpenRISC 1000 architecture*". Retrieved from http://opencores.org/openrisc,architecture, October 2016.
14. Lampret, D. "*OpenRISC 1200 IP core specification*". Retrieved from http://opencores.org/openrisc,or1200, October 2016.
15. Malik, H., Hassan, A.E. *"Supporting software evolution* using adaptive change *propagation heuristics*". In proceedings of the International Conference in Software Maintenance (ICSM), 2008.

16. Maurer, S. "*The HDL Complexity Tool*", 2009. Retrieved from http://hct.sourceforge.net/, October 2016.
17. Moertti, G. "*Your Core, my design, our problem*", October 11, 2001.
18. Nagappan, N., Williams, L., Vouk, M., Osborne, J. "*Using In-Process Testing Metrics to Estimate Post Release Field Quality of Java Programs*", Trollhattan, Sweden, 2007.
19. Ohlsson, N., Alberg, H. "*Predicting fault-prone software modules in telephone switches*", Linkoping Univ., Sweden, 1996.
20. OpenCores Project website http://www.opencores.com. Retrieved from http://hct.sourceforge.net/, October 2016.
21. Parizy, M., Takayama, K., Kanazawa, Y. "*Software Defect Prediction for LSI Designs*", IEEE 2014.
22. Rana, R. "*Software Defect Prediction Techniques in Automotive Domain*", University of Gothenburg, Sweden, 2015.
23. Salem, M.A., Khatib, J. "*An Introduction to Open-source Hardware Development*", EEDesign.com, 2008.
24. Sanchez, P., Villar, E. "*Using OpenSource Cores in Real Applications*", Universidad de Calabria, Conference Paper, January, 2003.
25. Schaller, R.R. "*Technological innovation in the Semiconductor Industry*", 2004.
26. Schroter, A., Zimmermann, T. "*Predicting Component Failures at Design Time*". Saarland University, Germany, 2006.
27. Scientific Toolworks Inc, "Understand SCI Tools" 2015. Retrieved from https://scitools.com, October 2016.
28. Shihab, E. "*An Exploration of Challenges Limiting Pragmatic Software Defect Prediction*", 2012.
29. Thomas, S.W., Hassan, A.E., Blostein, D. "*Mining Unstructured Software Repositories*", 2014.
30. Wanner, J.F. "Source Monitor: Expose Your Code", 2000.
31. Zimmermann, T., Nagappan, N., Zeller, A. "*Predicting Bugs from History*". Software Evolution, Chapter 4, Pages 69-88, Springer, March 2008.
32. Zimmermann, T., Premaj, R., Zeller, A. "*Predicting Defects for Eclipse*". In proceedings of the Third International Workshop on Predictor Models in Software Engineering (PROMISE), 2007.