Characterization of Maintenance Activities Using Linux

by

Ayelet Israeli

Supervised by

Prof. Dror Feitelson

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTERS OF SCIENCE

The Rachel and Selim Benin School of Computer Science and Engineering

The Hebrew University of Jerusalem

November 2008

Abstract

This thesis aims at characterizing software maintenance activities using Linux kernel, based on calculation of different software metrics. We perform our analysis and calculation for over 800 versions of the Linux kernel. We perform our analysis using several dissections — comparing production and development versions, as well as the new releases branch (2.6), and also comparing the core of the kernel to the *arch* and *drivers* directories. We present the results and the calculated metrics and then try to tie them to the different maintenance activities and make significant observations about them. We also aim at examining whether some of Lehman's Laws exist in the development of the Linux kernel.

Acknowledgments

First and foremost, I would like to express my gratitude to Prof. Dror Feitelson, for his invaluable support, encouragement, supervision and useful suggestions throughout this research work. Without his guidance and persistent help this thesis would not have been possible.

I would like to thank Trinity Software And Beyond Ltd, the Israeli representative of the Klocwork software, for providing the Klocwork CASE tool, its documentation and the technical support.

I am grateful to the Levin Edgar Fund for the generous grant that funded this research.

Finally, I would like to thank my family and friends for their unconditional love and support throughout my studies.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Software Evolution and Software Maintenance

Software Evolution is the software growth process: first the initial software product is developed and later on there are stages of updating it thus creating more releases of the initial product. The process of updating the product is the maintenance phase. Usually, as the program is more successful we see more releases and also a longer maintenance phase.

One of the first computer scientists to research this field was Prof. M. M. Lehman. He established a list of observations of what takes place in software evolutions, known as "Lehman's Laws of Software Evolution" [10, 11]. These laws can be validated and verified using different software metrics and other types of data (for example lines of codes metric or amount of hours each developer contributed).

One of the interesting observations is that the software must continuously change. In other words, the maintenance process is not only making sure the initial program is working as required but also developing it and updating the requirements [22]. There is a classification of software maintenance into 4 types: Corrective, adaptive, perfective, and preventative (preventative is sometimes bundled into perfective) [21, 32, 27].

The process of software evolution is important and significant in software engineering and therefore it is interesting to try to characterize and understand it. Specifically, we would like to observe the process in a successful software product.

One of the challenging parts of such research is to find data; Obtaining data from commercial software is hard and might suffer from confidentiality issues. As Kemerer and Slaughter [8] note, one of the most critical factor of an empirical study of software evolution is finding a good commercial (closed source) partner that has the necessary data (including past versions) and who is willing to cooperate with the researchers and give them the data, staff time and other resources. However, today we can get data from open source products. Luckily, this is good enough, since it was shown that the growth and structure characteristics are similar in open and close source products [20].

In this study we will use the Linux Kernel. Linux is undoubtedly a successful open source product. Linux has been used before to provide data to various studies [1, 28, 27, 5, 20, 33]. Specifically, we will analyze the progress made in hundreds of versions of Linux, released between March 1994 and August 2008, each containing hundreds of source files (thousands in the later versions) and millions of lines of code. There were over 800 versions released in this time span, a larger number of versions, in depth and in width, than used in any other study. Until 2005, the Linux kernel had two major versions maintained in parallel — production and development. We will analyze both in this study, and also the later versions, without this separation.

1.2 Objectives and Contributions

There are two main objectives in this study:

The first is to try to identify and characterize different types of maintenance activities by analyzing the different versions of Linux. This will open a new opportunity for research on one of the fundamental aspects of software development and help in the understanding of the maintenance process and its role.

The second is to examine the existence of some of Lehman's Laws in the development of the Linux kernel. A restricted version of this was done in the past by Godfrey and Tu [5], who examined the growth of the Linux kernel over its first six years (1994-2000), and found that at the system level its growth was super-linear. Validation of the laws was done before, especially in closed-source products [14, 13]. This was done in much smaller scale with fewer versions and over a shorter span of time.

The growth of the Linux kernel is in both maintaining the existing product and its features and in adding improvements and new features. We will try to characterize and quantify the maintenance/development activity and its affect on the software.

1.3 Organization

The organization of the reminder of this document is as follows:

In chapter 2 we will provide some background regarding the Linux kernel, software maintenance, Lehman's laws, and software metrics.

In chapter 3 we will describe and discuss our ideas regarding the analysis of the maintenance activities and Lehman laws.

In chapter 4 we will describe and discuss the research process — the methodology, the data, the challenges, and our final process.

In chapter 5 we will describe and discuss our results, and will also try to analyze their implications for maintenance activities and Lehman's laws.

In chapter 6 we will provide a summary of our conclusions and discuss future work.

2 Background

2.1 Linux Kernel

The Linux kernel is an operating system kernel used by several distributions [35]. Linus Torvalds, the father and creator of the Linux kernel, released a first version to the internet in September 1991. There followed $2\frac{1}{2}$ years of development by a growing Internet community, and then only in March 1994 the first production version was released.

In the next 10 years, hundreds of additional kernel versions were released. These were all numbered using a 3-digit system. The first digit is the generation; this was 1 in 1994 and changed to 2 in 1996. The second digit is the major kernel version number, and the third digit the minor kernel version number. Importantly, a distinction was made between even and odd major kernel numbers: the even digits correspond to stable production versions (1.0.*, 1.2.*, 2.0.*, 2.2.*, and 2.4.*), whereas versions with odd major numbers are development versions (1.1.*, 1.3.*, 2.1.*, 2.3,*, and 2.5.*), used to test new features and drivers leading up to the next stable version. Accordingly, new releases (with new minor numbers) of production versions included only bug fixes and security patches, whereas new releases of development versions included new (but not fully tested) functionality.

The problem with the above scheme was the long lag time until new functionality (and new drivers) found their way into production kernels, because this happened only on the next major version release. The scheme was therefore changed with the 2.6 kernel in December 2003. With this kernel, a fourth number was added. The third number now indicates new releases with added functionality, whereas the fourth number indicates bug fixes and security patches. Kernel 2.6 therefore acts as production and development rolled into one. However, it is actually more like a development version, because new functionality is released with relatively little testing. It has therefore been decided that version 2.6.16 will continue to be updated with bug fixes even beyond the release of subsequent versions, and act as a more stable production version.

Although the numbering scheme has grown somewhat messy with the 2.6 kernel version, the Linux release archives still provide a huge dataset for studying the evolution of a large-scale system over a long period of time. The Linux kernel is highly available — all the versions can be downloaded from www.kernel.org. Also, the Hebrew University holds a local mirror of the entire code.

The Linux kernel sources are arranged in twelve major subdirectories [24]:

arch – contains all the kernel code which is specific for different architectures (for example i386 for the matching processor).

include – contains most of the include (.h files) needed to build the kernel code.

init – contains the initialization code for the kernel.

mm – contains the architecture independent memory management code for the kernel.

drivers – all the system device drivers.

ipc – contains the inter-process communication code for the kernel.

modules – contains the built modules of the kernel.

fs – contains the file system code for the kernel.

kernel – contains the architecture independent main kernel code.

net – contains the networking code for the kernel.

lib – contains the architecture independent library code for the kernel.

scripts – contains the scripts used for kernel configuration.

An interesting note is that both the */arch* and */drivers* directories are practically external to the core of the kernel, and each specific kernel configuration uses only a small portion of these directories. However, these are a major part of the code, which grows not only due to improvements and developments in the Linux kernel code itself, but also (and mainly) due to changes in the environment (for example additional drivers and processors). Fig. 1 demonstrates the growth in these directories. As seen in the figure, they consist of around 50% of the total source files throughout the Linux kernel versions (i.e. the “core” of the kernel is only about 50% of it).

Fig. 2 shows similar ratios and growth, this time comparing number of lines of code (LOC). In this case we see that the ratio is over 60%. For each of the figures, and throughout this document, the X-axis of the graphs is the year and the indication near the lines shows the appropriate Linux kernel version. The minor versions of the major version 2.6 are named on the graph by their third digit (e.g. version 2.6.16 is named 16).

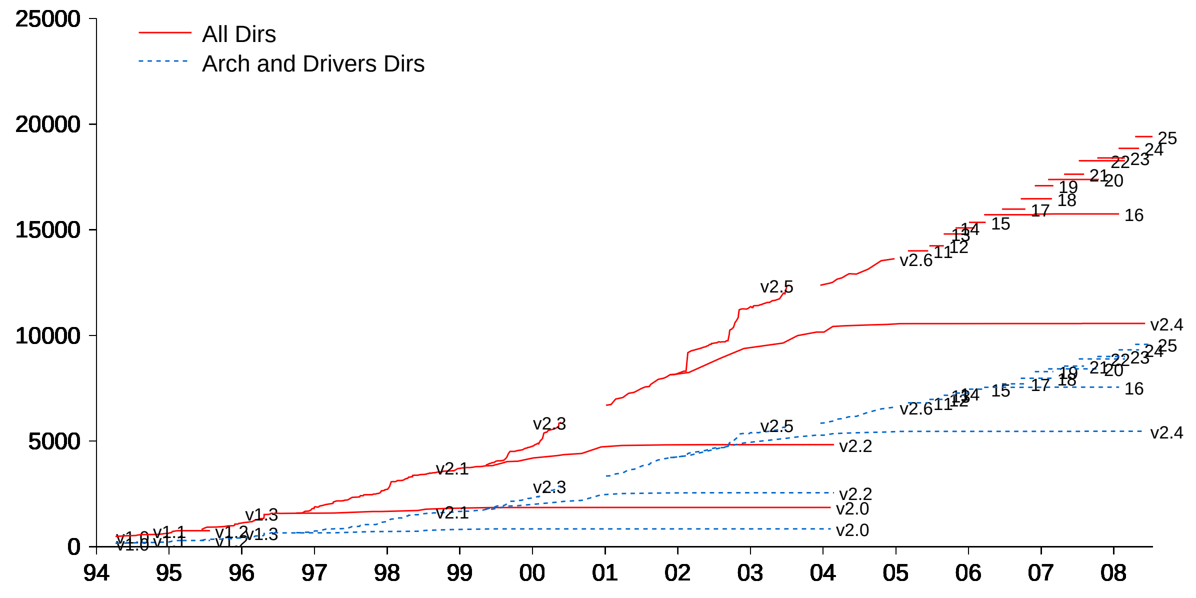


Figure 1:
The growth of source files in Linux, for all subdirectories vs arch and drivers only.

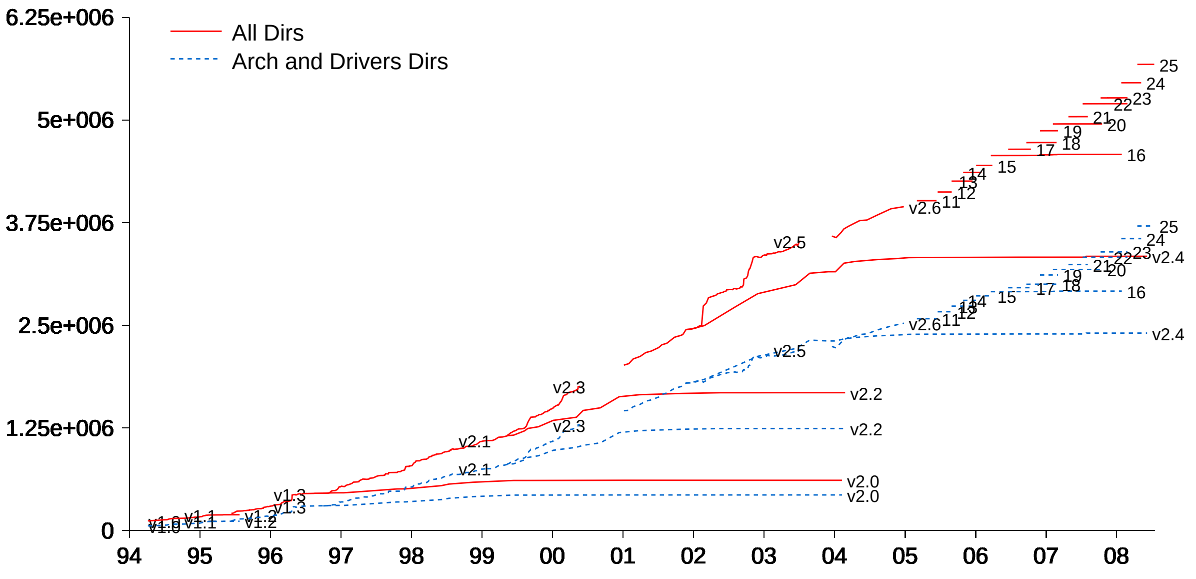


Figure 2:
The growth of LOC in Linux, for all subdirectories vs arch and drivers only.

2.2 Maintenance

As described in the introduction, the maintenance process of the software occurs after the first release, when the product is being updated repeatedly. According to the literature (for example [8, 10], at least 50% of the software effort is dedicated to maintenance. Generally, there are two types of maintenance — correction of bugs and further development of the product. In the literature, there is a classification of software maintenance into four types [21, 32, 26]:

1. Corrective – Correction of discovered problems (fixing bugs, etc).

2. Adaptive – Adapting the software product to the changing environment (keeping the software product usable upon changes).

3. Perfective – Improving performance or maintainability (including adding new features).

4. Preventative (sometimes bundled into perfective) – Correction and detection of latent faults in the software product before they become effective faults.

An important issue is the relative effort invested in the different types of maintenance activities [29]. We will attempt to estimate the effort involved with a new kernel version using a variety of different metrics. One group of metrics consists of measures of code changes, including the number of files handled (added, deleted, or modified) [12]. Other metrics are the time spent, and the number of people involved and the rate of new versions. These measures will be partly based on the Linux change logs, but with careful regard to their potential shortcomings [4]. Yet another type of metric is how many developers participate in each type of activity.

Our aim is to be able to determine whether maintenance is corrective, perfective, adaptive, or preventative by examining changes in appropriate code metrics. In particular, we will compare successive production versions of Linux and successive development versions to establish whether code metrics can be used to distinguish between corrective and perfective maintenance. Finally, we will examine a broad spectrum of metrics before and after restructuring events, and determine which, if any, of those metrics decrease. This will provide insights into the effects of restructuring on the metrics concerned.

2.3 Lehman's Laws of Software Evolution

Lehman studied several large scale systems and identified the following laws of software evolution [10, 11]. These have been extended and modified over the years from 1974-1996:

1. Continuing Change (1974): A program that is used must be continually adapted else it becomes progressively less satisfactory.

2. Increasing Complexity(1974): As a program is evolved its complexity increases unless work is done to maintain or reduce it.

3. Self Regulation (1974): The program evolution process is self regulating with close to normal distribution of measures of product and process attributes.

4. Conservation of Organizational Stability (Invariant Work Rate) (1980): The average effective global activity rate on an evolving system in invariant over the product life time.

5. Conservation of Familiarity (1980): During the active life of an evolving program, the content of successive releases is statistically invariant.

6. Continuing Growth (1980): Functional content of a program must be continually increased to maintain user satisfaction over its lifetime.

7. Declining Quality (1996): Programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment.

8. Feedback System (1974, updated in 1996): Programming Processes contain multi-level feedback loops and must be treated as such to be successfully modified or improved.

Lehman's laws were validated in different studies [10, 11, 14, 12, 13], some of which used different metrics in order to validate the laws. Godfrey and Tu [5] examined the growth of the Linux kernel over six years (1994-2000) and found that at the system level its growth was super-linear, and not slowing down with its growth, as expected by other studies, including Lehman's [12].

We will extend the work of Godfrey and Tu [5] to other laws, and using a lot more Linux versions, over a longer span of time.

2.4 Software Metrics

The software crisis has led to the invention many different quantitative measures of software products named "software metrics" [7, 16]. They claim to provide a tool for development and validation of models in the software process. Most of the studies on software evolution use different metrics to demonstrate their point. Generally, there are product metrics which measure the product at any stage of its development (size of the program, number of documented lines) and process metrics which measure the development process (such as development time, methodology, experience of the programming staff). We will focus on product metrics for which we have more available data. Within product metrics there are different classes of metrics [16]:

1. Size metrics — attempt to quantify the "size" of the software. These include LOC (lines of code), number of comments, number of functions, number of modules, etc.

2. Complexity metrics — attempt to measure the complexity of the software product. McCabe's cyclomatic complexity which measures the number of independent paths in the product's control graph and its extensions are an example of these metrics.

3. Halstead Product metrics — known as Halstead's software science proposes a set of metrics that check different aspects of a program: the vocabulary, the length, and the volume. These also derive the difficulty, the effort, and the estimated faults in the software.

4. Quality Metrics — attempt to measure aspects like reliability, correctness, maintainability by the number of bugs, number of changes in each release, mean time to failure (MTTF), etc. Here metrics such as cyclomatic complexity might also be used, since the more complex the program is, the harder it is to maintain it. The same applies to common coupling that was measured in Linux in several studies and can indicate maintainability of the code [2, 28, 37].

There are two reasons why it is important to perform code-based measurements. The first is accuracy; surveys and logs can be highly inaccurate. For example, a survey of maintenance managers yielded the result that 17.4 percent of maintenance is corrective in nature; the actual figure, obtained by analyzing changes to source code, is more than three times larger [29]. Similarly, a comparison between change logs for three software products and the corresponding changed source code itself showed that up to 80 percent of changes made to the source code were omitted from change logs [4].

The second reason why code-based metrics are important is that certain phenomena can be measured only by examining the code itself. For example, common coupling has been validated as a measure of maintainability [2], and the only way to measure the common coupling within a software product is to examine the code itself. In the case of Linux, the number of files and directories of production kernels appears to be quite stable. However, common coupling increases exponentially when considering both production and development versions combined [28]. This may be an indication of degraded quality as a result of corrective maintenance.

Various code metrics have been challenged on both theoretical and experimental grounds. For example, it has been shown that cyclomatic complexity is strongly correlated with lines of code; furthermore, cyclomatic complexity measures control flow complexity but not data flow complexity [30]. Similar critiques have been leveled against other code metrics [31]. Accordingly, it is not much use to be told that product A has cyclomatic complexity 123 whereas product B has cyclomatic complexity 456; comparing the numbers does not tell us much.

However, the objections of the previous paragraph do not apply when the code metrics in question are applied to successive versions of the same product or module. Accordingly, we will measure changes in a variety of metrics for successive versions of Linux. The metrics we will examine are LOC, McCabe cyclomatic complexity, Halstead's software metrics and Oman's Maintainability Index. We will also examine some size metrics such as number of files and directories, intervals between releases, and files and directories handled (added, deleted, or modified) in order to understand the broad picture. Each of these metrics has its advantages and its disadvantages, as reflected in previous studies [30, 31, 7, 20], but these measures have all been used in empirical studies by other researchers. By measuring a set of different metrics we hope to overcome the disadvantages and to be able to see the full picture.

2.4.1 Size Metrics

Lines of code (LOC) is one of the most used metric in the literature, perhaps since once defined it is very easy to calculate. The problem is the definition — should we aggregate also the commented lines or the empty lines in these measures? On one hand, we want to investigate the code itself, so it seems that only statements should interest us. On the other hand, the number of comments affects the understandability of the code and thus its maintainability, and the empty lines affect the readability of the code.

Lehman et al. [10, 11] examine the growth of software with regard to the number of software modules. As discussed in Godfrey and Tu [5], these can be considered as LOC or number of files. Since there is a variation in the size of files, they decided LOC gives a better picture of the source code. Capiluppi et. al. [3] examines different structural properties of software evolution for the ARLA open source system. They find that all the size related metrics (LOC, files, folders, etc) grew over releases.

Intervals between releases give a picture of the software health — highly frequent releases indicate high maintenance activity which might mean many errors and bugs (mainly in the production versions, as development versions are governed by Torvalds's principle of "release early, release often" [23]). Too rare releases may indicate software that is not changing and developing. In a report for the Linux Foundation regarding version 2.6.* [9], they show that in the 2.6.* release the release frequency is about 2.7 month between releases (between the major 3-digit releases). This follows the decision of Linux to releases every 2-3 month in order not to back up new developments. They also show that the number of developers and number of companies investing in Linux development is growing over time. In fact, according to their statistics, the number of individual developers has doubled in the last three years. Still, the majority of the work is done by a small percentage of the developers (the top 10 individual developers have done 15% of the work, and the top 30 have contributed 30%).

The number of files and directories handled are related to the software maintenance. As shown in [3], reorganization of the code can be identified by the changes in the number of files and directories.

2.4.2 McCabe's Cyclomatic Complexity

McCabe's Cyclomatic Complexity (MCC) [15] measures the complexity of a program. The idea is to calculate the number of linearly independent paths in the program's control flow graph.

The graph G is the control flow graph of a program when its nodes correspond to the basic blocks and a directed edge connects blocks v and u if the block u can be executed immediately after v. The general formula to compute the cyclomatic complexity is

$$M = V(G) = E - N + 2P$$

where:

$V(G)$ – Cyclomatic number of G

E – Number of edges

N – Number of nodes

P – Number of connected components in the graph

McCabe also show equivalence to a simpler calculation — the number of predicates (conditional branches) in the program plus 1. These include if-then-else, for, while, do, and case statements. McCabe also introduced what is referred as the "extended cyclomatic complexity" which is counting the actual conditions, and not the conditional statements. The reason for that is the connectives "and" and "or" add complexity to the code: for example "if(c1 and c2) then" can be written (without connectives) as "if(c1) then if (c2) then". Thus the contribution of the statement to V(G) should be 2 and not 1. Myers [17] suggested that the cyclomatic complexity should be the range between the number of predicates plus 1 and the extended complexity.

The cyclomatic complexity of a program is the sum of the cyclomatic complexity of all the functions in the program.

According to McCabe a function with cyclomatic value of over 10 should be rewritten or remodeled. The SEI (Software Engineering Institute) suggested the following table:

$V(G)$	Complexity and Risk
$1-10$	a simple program, without much risk
$11-20$	more complex, moderate risk
$21-50$	complex, high risk problem
>50	untestable program (very high risk)

As mentioned before, although validated by many studies, it has been shown that cyclomatic complexity is strongly correlated with lines of code; furthermore, cyclomatic complexity measures control flow complexity but not data flow complexity [30, 31]. However, since we are interested in comparing this value for different versions of the same product we believe that it is reasonable.

2.4.3 Halstead's Software Science

Halstead [6] suggested a few metrics that are all based on software science in which a computer program is a collection of tokens that can be identified as operators or operands. In C, operators are the function calls, the different types of parentheses, the unary, binary and ternary operators and reserved words of C which are function-like. Operands are all the identifiers and constants in the C program.

The variables are:

n_1 — Number of distinct operators in a program

n_2 — Number of distinct operands in a program

N_1 — Total number of operators

N_2 — Total number of operands

Based on these, the following metrics (Halstead's Software Science) are calculated:

Vocabulary $n = n_1 + n_2$

Length $N = N_1 + N_2$

Volume $V = N \lg_2 n$

Difficulty $D = \frac{n_1}{2} * \frac{N_2}{n_2}$

Effort $E = V * D$

These all are defined at the function level. For a program, we will aggregate the Halstead metrics of all its functions.

The Halstead Volume (HV) actually measures the total number of bits needed to write the program. This is the product of the total number of tokens and the number of bits needed to identify all the unique tokens. Halstead Difficulty (HD) is the number of unique operators times the average usage of the operands divided by two. This measure indicates how difficult the program is to develop, maintain and test. The more operators and the more usage of single operands, it will be more difficult. Lastly, Halstead Effort is simply HV times HD.

It should be mentioned that if n equals zero HV cannot be calculated. Similarly, if n_2 alone is zero, HD cannot be calculated. This can occur in empty functions or in functions which hold only the "return" statement.

2.4.4 Oman's Maintainability Index

Oman developed the Maintainability Index (MI) as a composite metric that attempts to fit data from several software projects [18, 34]. MI is defined as:

$$MI = 171 - 5.2\ln(AvgHV) - 0.23AvgV(G) - 16.2\ln(AvgLOC) + 50\sin(\sqrt{2.46perCM})$$

where:

$AvgHV$ — average Halstead Volume per module.

$AvgV(G)$ — average cyclomatic complexity per module.

$AvgLOC$ — average lines of code per module.

$perCM$ — average percent of comments per module.

The MI value is 100-point scale from 25 to 125, matching the questioners Oman used in order to construct the MI formula, where low values correspond with lower maintainability and high values with higher maintainability.

Thomas [33] notices that the term perCM should refer to the fraction of comments in the file and not the percentage. This is because if we refer to percentage perCM is between 0 and 100, thus 2.46 perCM is between 0 and 246 and the square root value is between 0 and 15.68. If we use a fraction (i.e. perCM is a value between 0 and 1), the range of $\sqrt{2.46perCM}$ is 0 to approximately $\frac{\pi}{2}$. Therefore, perCM will be calculated as the ratio of comments to the total number of lines. Also, like Thomas [33] we will consider perCM as the ratio between comments and comments plus lines of code with no comments or blanks. This way a line which contains both code and comment appears as a comment but also as a line of code.

The LOC value will be the average lines of codes with no comments or blank lines. The other values are straight forward (when we consider module as a function).

Also, the value of MI per series will be the average value of MI per files in the series.

3 Our Ideas

3.1 Different Maintenance Activities

In regards to the Linux kernel code we associate the different types of maintenance activities with different parts of the kernel, as follows:

1. Corrective maintenance is reflected in successive versions of production kernels. This is a reasonable partition due to the structure of the Linux versions. We can safely assume that successive versions in the production kernel are usually corrective.

2. Adaptive maintenance is reflected in the *arch* and *drivers* directories (especially in the development kernels). Since the *arch* and *drivers* directories encapsulate most of the interactions of the system with its environment, a change in them will indicate adaptation to a change in the environment.

3. Perfective maintenance is reflected in successive versions of development kernels. Again, this matches the structure of the Linux versions — we can assume that successive versions in the development kernels will be perfective.

4. Preventative maintenance is reflected in isolated events in which many files are partitioned, removed, or moved (mainly in development kernels). Here we assume that preventative maintenance is related to code reorganization and can be identified, for example, by changes in the number of directories [3].

Collecting data and calculating different software metrics will allow us to further characterize the maintenance process in Linux.

3.2 Lehman's Laws

Most of our analysis of Lehman's laws stems from previous literature regarding verification of these laws [14, 13]. We will examine the Laws of *Continuing Change*, *Continuing Growth* and *Self Regulation* according to the number of LOC in each version (similarly to Godfrey's test of the *Continuing Growth* law in Godfrey and Tu [5]). Continuing Change will be examined through the *arch* and *drivers* directories, as these have to do with changes in the environment. The other two can be examined looking at all directories or only at the core kernel. Continuing Growth (which deals with added features) will be examined mainly through releases of development kernels, where we expect functionality to be added.

The law of *Increasing Complexity* will be analyzed using the different software metrics (McCabe's Cyclomatic Complexity, Halstead Metrics).

The law of *Invariant Work Rate* will be analyzed using information on code changes in files and directories. It is common to try and examine this law using "work hours" or number of developers, but as we have stated above, it is hard to find such information, and even if it will be found, it is of doubious accuracy. Another idea that we will check is the number of versions released per time unit, as an indication of constant work rate. This will be examined especially in development versions, where version releases are more frequent and versions are more easily released (less stable releases can be released, in comparison to production versions).

The law of *Conservation of Familiarity* will be examined by incremental growth, as suggested in [13]. For this analysis we will use successive versions of release 2.6, in which the versions are released in a relatively stable rate and are more managed.

The laws of *Declining Quality* and the law of *Feedback Systems* are not in the scope of this work, but we will try to make observations about them. According to Lehman et al. [13], the first follows the principle of uncertainty and is supported due to a theoretical analysis. The second is supported by the other 7 laws, since it can be used to rationalize them. Moreover, for the feedback systems, there was much work on supporting this law (Lehman's FEAST project) including different measures and regressions and system dynamics.

For each analysis, we will examine the data in two different directories divisions – all directories and the core kernel (i.e. without only *arch* and *drivers*).

4 Methodology

We have measured the different properties and metrics of the Linux Kernel in a manner that we will now describe.

Major Version	Number of Versions	Last Version Included
v1.0	1	1.0
v1.1	36	1.1.95
v1.2	14	1.2.13
v1.3	114	1.3.100 (pre2.0.14)
v2.0	41	2.0.40
v2.1	142	2.1.132 (2.2.0-pre9)
v2.2	27	2.2.26
v2.3	61	2.3.51 (2.3.99-pre9)
v2.4	61	2.4.36.6
v2.5	76	2.5.75
v2.6	12	2.6.10
v2.6.11	13	2.6.11.12
v.2.6.12	7	2.6.12.6
v2.6.13	6	2.6.13.5
v2.6.14	8	2.6.14.7
v2.6.15	8	2.6.15.7
v2.5.16	63	2.6.16.62
v2.6.17	15	2.6.17.14
v2.6.18	9	2.6.18.8
v2.6.19	8	2.6.19.7
v2.6.20	22	2.6.20.21
v2.6.21	8	2.6.21.7
v2.6.22	20	2.6.22.19
v2.6.23	18	2.6.23.17
v2.6.24	8	2.6.24.7
v2.6.25	12	2.6.25.11

Table 1:
Analyzed Versions Information - the names and number of versions analyzed for each major version.

4.1 Research Data

We examined all the version from March 1994 to August 2008, these are 810 versions in total. This includes 144 production versions (1.0, 1.2.*, 2.0.*, 2.2.* and 2.4.*), 429 development versions (1.1.*, 1.3.*, 2.1.*, 2.3.* and 2.5.*), and 237 versions of 2.6.*. The detailed information on these versions appears in Table 1. Notice that for the development versions 1.3, 2.1, and 2.3, the last versions were called pre-2.0, 2.2.0-pre, and 2.3.99-pre, respectively, and were designed to ease the move between versions. In these three, the name of the last "pre" version is in parentheses.

We aimed at examining the entire source code of Linux Kernel. A typical C program consists of files with two different suffix: .h or .c. The .c files consist of the executable statements and the .h files of the decelerations. Both should be looked at in order to get the full picture of the Linux code. Some studies included only the .c files (for example [28]), but their results are problematic because they do not include the decelerations in the header files. Godfrey and

Tu [5] examined all the source files which are both .c and .h files. Thomas [33] also examined both .h and .c files. However, since he used a CASE tool which requires preprocessing, only files in the preprocessed configuration were examined.

We decided to examine both the development and the production versions in order to identify different patterns in each of their metrics. Thomas[33] examined only the second generation production series (2.0.*, 2.2.* and 2.4.*), and Godfrey and Tu [5] examined both development and production version until January 2000 (the last production version was 2.2.14 and the last development version was 2.3.39). They examined only some of each group of kernels, when they decided to examine more production kernels, which were less frequently released.

4.2 Klocwork — The case of the CASE tool

Our initial research plan was to use Klocwork, the CASE (Computer Aided Software Engineering) tool used by Thomas [33]. Klocwork (see www.klocwork.com) is a static code analyzer. It can detect a wide variety of defects and vulnerabilities, as well as architecture and header file anomalies. In addition, it can compute nearly 100 different metrics: at the file level, at the function level, and (for object-oriented languages) at the class level. These metrics vary from the familiar (e.g., lines of code, cyclomatic complexity, Halstead volume) to the less frequently encountered (e.g., average level of control nesting).

Klocwork is powerful; it can analyze large software products, consisting of 5 million or even 10 million lines of code. As mentioned above, Klocwork works with pre-compiled code. In theory, pre-compiling Linux is easy; all one has to do is to invoke the gcc compiler with the appropriate options. The resulting code can then be submitted to Klocwork. In practice, there are serious problems, because the C language has evolved in time. Some features have been dropped ("deprecated"), and others have been added. Accordingly, each version of Linux has to be pre-compiled using an appropriate version of the gcc compiler. The source code of each version of gcc is available on the Web, together with its libraries. The problem is that, to compile a given version of gcc, one needs a compiled version of the appropriate earlier version of the gcc compiler for the appropriate operating system. This also requires choosing a specific configuration for each compilation. The chosen configuration will affect the structure and the function of the pre-compiled Linux kernel, and the files that will be analyzed.

Another problem we find with the analysis of pre-compiled code and not the code as is, is that we are interested in complexity from the developer's point of view, and not complexity of the compiled binary. For example, macros are tools to simplify the readability of code: the macro "$\#define\ MAX(X,Y)\ (X > Y)?X : Y$" should not increase the complexity of the code each time a developer uses it. True, in run time the code will be more complex, but the whole purpose if such macro is to "hide" the condition and to effectively reduce the independent paths in the code and the MCC. Thus, we believe that these should be counted as seen by the developer and not by the computer (i.e. as the code is before pre-processing).

To avoid compilation, Klocwork allows to manually choose directories of files and header files which will be directly pre-compiled with the Klocwork pre-processor (avoiding the use of gcc). This option allows analyzing the entire code, with no specific configuration limitations, but does not include all the linking and the definitions. We felt that using this method, we will truly analyze the code as a developer or a software engineer views it — the whole source code, including all possible definitions. We decided to go on and continue with this method.

In order to make sure that we do not suffer from choosing this method, we extensively compared the results of the gcc method and the Klocwork pre-processing method. Although the second method allowed analysis of much more files (over 6 times more files analyzed and over 6 times data size than the gcc method), our comparison revealed two major problems with the Klocwork pre-processing method:

a) Definition Problems: Without compilation there were no definitions of certain values (for example Linux configuration macros). This resulted in inconsistent code being created. We came up with three different ways to handle this:

   1. Manually choose which definitions should be used and to use them. This option limits us to a chosen configuration, and therefore was not chosen.
   2. Edit all the files before the Klocwork analysis, and remove all the #if, #else pre-processor commands. This led to an inconsistent code since some of the time the #if scope defines a function in one way, and the #else scope in another. We had to impose our own logic in choosing which scope is better.
   3. Leaving the situation as is. Klocwork handles lack of definitions by choosing the "else" branch in the pre-processing. This means that if certain parts of the code are in a scope of #ifdef pre-processor commands we do not analyze them. We will analyze only the #else scope or the #ifndef scope. Basically, this is equivalent to our configuration being the "else" configuration. We analyzed the cases and occurrences of such pre-processor commands (As described in the appendix), and found out that about 30% of the lines out of all the lines in the code are in the #if scope in the early version and about 20% in the later ones. This means that the upper bound of code not analyzed will be 30% (many #if have #else in their scope, and there are also #ifndef scopes, so these are analyzed). We came to a conclusion that when we choose the no-compilation method we still analyze a large amount of the code, which is our main interest. This results in definitions that are not consistent or do not define a "normal" configuration of Linux, but working in this method still allows us to analyze more code and understand better the trends in Linux kernel code.

b) Errors in Metrics: Some of the metrics were miscalculated without proper gcc pre-compilation.

While we felt that we can overcome the problem in (a), as describes above, we couldn't ignore the fact that sometimes some of the metrics are miscalculated. One option was to go back and use the standard way of operating Klocwork — using the appropriate gcc and choosing a (maybe few) specific configuration. Another option was to try to do our analysis using a different tool.

Eventually, we decided to write our own tool. This had, of course, its own problems and limitations, but we felt that this way, although we limit ourselves to a few specific metrics, we know exactly how they are calculated.

4.3 Our Program

We coded a perl program that, given a c-code file, separates it into its different tokens, and generates an output file that calculates the following metrics in the following manner:

The following are calculated at the file level:

lines — simply counting the number of lines in the file.

empty — the lines that have no character written in them.

comments — the lines that contain any type of comment (/*...*/ and // comments which were allowed later in the C compiler)

combined comments — the lines that contain both comments and other statements.

define — number of #define commands.

include — number of #include commands.

includes (local only) — number of local includes.

funcNum — number of functions in this file.

ifElse — number of #if/#elif/#ifdef/#ifndef directives.

MCC — McCabe's Cyclomatic Complexity. It is the aggregation of all the cyclomatic complexity values of all the functions in the file.

EMCC — Extended McCabe's Cyclomatic Complexity. It is the aggregation of all the extended cyclomatic complexity values of all the functions in the file.

The following are calculated at the function level:

funcName — the function name

valid operators — all operators to be counted in the function (described below).

valid operands — all operands to be counted in the function (described below).

unique operators — distinct operators from the valid operators list.

valid operands — distinct operands from the valid operands list.

num lines — lines in this function.

num empty — empty lines in this function.

num comments — commented lines in this function.

num combined comments — combined commented lines in this function.

num ifElse — #if/#elif/#ifdef/#ifndef in this function.

MCC — McCabe's Cyclomatic Complexity. It is calculated as the number of if/case/for/while commands in the code and the number of "? :" ternary operator.

EMCC – Extended McCabe's Cyclomatic Complexity. It is calculated as MCC plus the number of "ands" and "ors" in the conditions of the above predicates.

Operators and Operands:

The distinction between operators and operands is as follows (based on [25] for pascal, [19] for c):

Operands: Identifiers, constants.

Operators:

Function calls,

The following reserved words: break, case, continue, default, do, else, for, goto, if, return, sizeof, switch, while.

The following symbols: {} () [] ! % & | ∗ + − / , . : ; < > = ? ∼ ˆ # \ ′ ‘ %= && &= || ∗= ++ += −− −= −> ... /= :: << <<= <= >= >> >>= == ˆ= |= =& ##

With the following rules:

1. In parentheses only the opening parenthesis counts.
2. In if(), for(), while(), and switch() the parentheses are a part of the construct. Also in “case:” the colon is part of the construct.
3. The ternary operator ?: is counted as one operator.
4. The comments are not considered for any of this.
5. The string between quotation mark is counted as a single operand. Same with a string between <> in #include.

As for the #if/#elif/#ifdef/#ifndef issues we found in Klocwork, we decided to analyze the entire code, including the scope of these pre-processor commands. This way we analyze the code as the programmer sees it — a whole file, with the different possibilities of definitions. When analyzing these lines created inconsistent code (for example both the #if and the #else declared the same function with different input parameters or both had an opening parenthesis without a closing one), we identified this file and counted it a problematic file. Later on, we didn’t use it in our analysis.

Fig. 3 demonstrates the analysis of the Linux/kernel/panic.c file in version 1.0. Each analysis file has a different length — depending on the number of functions in the file.

We ran this program for all the .c and .h files of all the versions, creating an output file with the calculated metrics for each one. Then, for each version we aggregated all the data from these output files and generated one output file per version with the following information: total number of files, total number of functions, number of empty functions, number of empty files, number of problematic files, number of defines, number of includes, number of if/else commands, total lines of code, lines of code without comments or blanks, McCabe Cyclomatic Complexity value, Halstead’s Software Science metrics, Oman’s Maintainability Index.

As mentioned above, the size of each of these output files is proportional to the number of functions each source files contain. We separated the results of each kernel major version (v1.0, v1.1, v1.2, v1.3, v2.0, v2.1, v2.2, v2.3, v2.4, v2.5 and v2.6) to its own directory which contains all the files of the appropriate minor versions. The sizes of these directories ranges from 2 MB

```
lines: 35
empty: 6
comments: 10
combinedcomments: 1
define: 0
include: 3
includes(local only): 0
funcNum: 1
ifElse: 0
MCC: 3
EMCC: 3
funcName: panic
valid operators: 37
valid operands: 20
unique operators: 14
unique operands: 13
num lines: 12
num empty: 1
num comments: 0
num combined comments: 0
num ifElse: 0
MCC: 3
EMCC: 3
```

Figure 3: *An example of the output file for a single source file.*

in v1.0 and hundreds of MB in v1.* series, to a few GB in the v2.* series and around 20GB for the v2.6 series, depending on the number of files in each version and the number of functions in each. These files contains large amounts of data, which are hard to handle and analyze, thus a single file for each version was needed.

Aggregating metrics at the kernel level

We used the same approach used in other studies (such as [33]) and as explained in the original metrics definitions.

LOC can be calculated at all levels — function, file or entire kernel. Here, the A-LOC value for the entire kernel is all the lines (including comments and blanks) in all the files in the kernel. LOC are all the lines not including comments or blanks.

McCabe is at the function level, and for a specific file McCabe cyclomatic complexity is simply the sum of all the cyclomatic complexity values of its functions (meaning, files with no function, such as some header files, will have complexity of 0). The McCabe cyclomatic complexity for the kernel will be the sum of all the files' complexities. The same applies for the extended complexity. In order to compare the different versions, despite the addition of new files, we will sometimes look at the average cyclomatic complexity of the files of the kernel.

Halstead's software metrics are calculated per function and it is customary to aggregate them for the file level. Again, for a specific kernel we will aggregate the values of all its files. In order to compare the different versions, despite the addition of new files, we will sometimes look at the average Halstead metrics of the files of the kernel.

Oman’s MI is defined at the file level. Since it is a 100-point scale metric from 25 to 125, we cannot aggregate the values. Instead, we will average the value of all files in the kernel.

We calculated the different metrics for each of the kernels, with and without the *arch* and *drivers* directories.

```
Linux/fs/stat.c

NAME LOC EMPTY COM COMST DEFINE INC INC-LOCAL FUNCNUM IFS MCC EMCC HV HD HE
file 202 21 19 0 2 6 0 9 0 29 34 3975.36 153.02 76119.92
cp\_old\_stat 16 1 0 0 0 0 0 0 0 1 1 605.40 9.72 5882.50
cp\_new\_stat 51 3 14 0 0 0 0 0 0 5 5 1397.19 26.25 36676.22
sys\_stat 12 1 0 0 0 0 0 0 0 3 3 232.79 16.25 3782.88
sys\_newstat 12 1 0 0 0 0 0 0 0 3 3 232.79 16.25 3782.88
sys\_lstat 12 1 0 0 0 0 0 0 0 3 3 232.79 16.25 3782.88
sys\_newlstat 12 1 0 0 0 0 0 0 0 3 3 232.79 16.25 3782.88
sys\_fstat 11 1 0 0 0 0 0 0 0 3 5 326.98 14.4 471.47
sys\_newfstat 11 1 0 0 0 0 0 0 0 3 5 326.98 14.4 4708.47
sys\_readlink 16 1 0 0 0 0 0 0 0 5 6 387.64 23.25 9012.73

Linux/fs/block\_dev.c

NAME LOC EMPTY COM COMST DEFINE INC INC-LOCAL FUNCNUM IFS MCC EMCC HV HD HE
file 216 20 20 2 1 6 0 3 0 38 44 5149.45 171.09 472 486.89
block\_write 55 4 0 0 0 0 0 0 0 10 11 1698.17 61.2 103927.99
block\_read 130 10 15 2 0 0 0 0 0 27 32 3431.63 107.39 368509.77
block\_fsync 1 0 0 0 0 0 0 0 0 1 1 19.65 2.5 49.13
...
Linux/kernel/panic.c

NAME LOC EMPTY COM COMST DEFINE INC INC-LOCAL FUNCNUM IFS MCC EMCC HV HD HE
file 35 6 10 1 0 3 0 1 0 3 3 271.03 10.77 2918.77
panic 12 1 0 0 0 0 0 0 0 3 3 271.03 10.77 2918.77
....
=========================================================================================
Fi Fu eFi EFu Probs fiLOC fiNCNB fiAMCC fiAEMCC AMcc AEMcc fiAHV fiAHD fiAHE AHV AHD AHE MI
487 3170 0 19 2 165652 102069 40.34 46.43 6.17 7.10 5735.61 154.51 333163.13 881.15 23.74 51183.11 106.67
166 1611 0 15 1 84574 50461 59.87 67.68 6.13 6.93 8418.19 206.55 505441.72 867.42 21.28 52081.52 109.25
321 1559 0 4 1 81078 51608 30.27 35.48 6.21 7.28 4348.35 127.59 244072.02 895.33 26.27 50254.73 103.26
```

Figure 4: *An example of the output file for a specific kernel.*

Fig. 4 is a part of the summarization file of version 1.0. For each of the files in this kernel, the data is summarized in a table. In addition to the values that appear in the analysis file, here we calculate HV HD and HE which are the Halstead Volume, Difficulty, and Effort respectively. The last three lines in the file include the aggregated data for the entire kernel. The first line is the data for all the source files, the second includes only source files from the *arch* and *drivers* directories and the last line includes data for all files in other directories.

The values are in the following order: total number of files, total number of functions, number of empty files, number of empty functions, number of problematic files, total lines in the kernel, total lines not including blanks and comments in the kernel, the average McCabe Cyclomatic Complexity in the file level, the average Extended McCabe Cyclomatic Complexity in the file level, the average McCabe Cyclomatic Complexity in the function level, the average Extended McCabe Cyclomatic Complexity in the function level, Average Halstead volume in the file level, Average Halstead difficulty in the file level, Average Halstead effort in the file level, Average Halstead volume in the function level, Average Halstead difficulty in the function level, Average Halstead effort in the function level and the Maintainability Index.

The size of the summarization file for each version range from hundreds of KB in the smaller versions, to around 1 MB at the beginning of the 2.* series, to between 10 and 18 MB for the 2.6 series.

5 Results

We examine the results of our calculations in three different aspects - looking at all the kernel's directories, looking at the directories without *arch* and *drivers*, and looking only at those two separately.

5.1 Data Used in Analysis

One of the big challenges that our experience with Klocwork raised was handling the preprocessor commands. We believe that the correct way to perform our analysis is as the programmer views it — the macros are planted there for the ease of use, the "ifdef" sections are there in order to handle different definitions using similar code and the same file structures. Both should be measured appropriately with the different metrics — the macros should be calculated with all their properties upon their definition and as a function call in other cases. The "ifdef" sections should be calculated fully and not ignored because a programmer coding such a file must be aware of all different possibilities of the flow of the code.

In most cases, we were able to analyze files which had "ifdef" sections. In other cases, we saw that when analyzing both paths of the "if" a malformed code is created and thus we were not able to analyze it with our automated tools. Trying to do this manually is also a challenge — how does one decide which path to analyze? Therefore, when we encountered such files we removed them from the analysis. Other malformed files (very few) were removed as well and are not a part of our calculations.

Empty function and files were not considered problematic and are calculated in the metrics. They are a part of the code and might be a part of a special design (for example implementing polymorphism or the null object oriented design or just leaving modularity and expansion options for different versions).

Overall, less than 1.5% of the source files were not analyzed at all. Among the *arch* and *drivers* subdirectories between 0.3%–3% of the files were not analyzed, whereas in the other parts of the kernel the worst case of un-analyzed files was less than 0.7%. Thus the vast majority (almost 99%) of the files of the Linux kernel were analyzed and their data is aggregated in the different metrics.

5.2 Calculated Metrics

In this section, we will display and discuss our results. We will also compare them to those of Godfrey and Tu [5] and Thomas [33], which are the two papers with the most similarity to our work.

As a reminder, Godfrey and Tu [5] examined the growth of Linux using mainly LOC. They examined all the source files in selected versions from both development and production versions (mostly production versions) until January 2000 (the last production version was 2.2.14 and the last development version was 2.3.39). Thomas [33] analyzed different software metrics in Linux in order to learn about quality and maintainability, using Klocwork to examine all the source files of the second generation production series (2.0.*, 2.2.* and 2.4.*).

For each of the graphs that show a different metric for all versions, the development versions (v1.1, v1.3, v2.1, v2.3, v2.5) are in a dashed blue line, the production versions (v1.0, v1.2, v2.0, v2.2, v2.4) are in a solid red line and version 2.6 is in a dashed green line. As stated above, the label at the end of each line indicates the major version number, and for the 2.6 series it will indicated the third digit of the minor version.

5.2.1 Number of Files

Fig. 5 shows the growth in number of files in Linux. We see here that most of the growth occurs in the development versions, whereas the production versions are usually stable (except maybe an increase at the beginning, they are constant later on). In versions 2.0 and 2.2 these increases are slow and small, but for version 2.4 we see an increase from its initial version until the release of version 2.6. This behavior can be explained by the many software requirements for that version and also by the delay of its release (notice the 6 month gap between 2.3 and 2.4 which indicates no releases at that time). This strange behavior in the first years of version 2.4 will be discussed in detail below.

The 2.6 series demonstrates differences between the development and production versions, when we see that for each minor version (differentiated by the third digit — i.e. the 2.6.X view) the number of files is constant (as in production versions), but it grows between the different versions (as in development versions).

When trying to characterize the shape of the growth in the production versions, is it seen to be superlinear up to version 2.5, maybe indicating that the growth is proportional to the current size. But in 2.6 the growth seems to be linear.

Another issue that can be seen in the graph is the differences between *arch* and *drivers* and the rest of the kernel: the division of number of files across the kernel, as mentioned in the background, is almost 50% for the *arch* and *drivers* directories and 50% for the rest. We can also clearly see that the growth for each of them is qualitatively similar, although there are a few points which have different shape. For example, version 2.5 grows more and with higher "jumps" each time for all other directories than the *arch* and *drivers* separately. Another example is 2.2 which has a jump in the beginning for *arch* and *drivers* and is pretty steady for other directories.

The first jump in 2.5 (from 4017 to 4783 files in the core of the kernel, versus a little more than 100 in the *arch* and *drivers*) is explained by the insertion of the sound subdirectory to the kernel. In previous versions there was a sound directory in the drivers, but generally the sound project was developed separately until then and introduced to Linux in version 2.5.5. This created an increase both in the sound directory and also in the include directory, which was updated with header files. Although sound related files were removed from the drivers directory, the effect on *arch* and *drivers* was not negative due to changes there (enhancements and improvements of the ppc arch directory) and due to the fact that the original sound related files were much fewer (only 200 versus over 500). An important note here, is that a major increase in arch or drivers will usually also derive an increase in the other directories, due to the addition of header files to the include directory.

The next jump is actually a combination of two effects. One is in the drivers and arch directories, when improvements and developments of the arch um directory were made (version 2.5.35). In the consecutive version (2.5.36) a new file system support was added (xfs), generating an increase in the core kernel number of files.

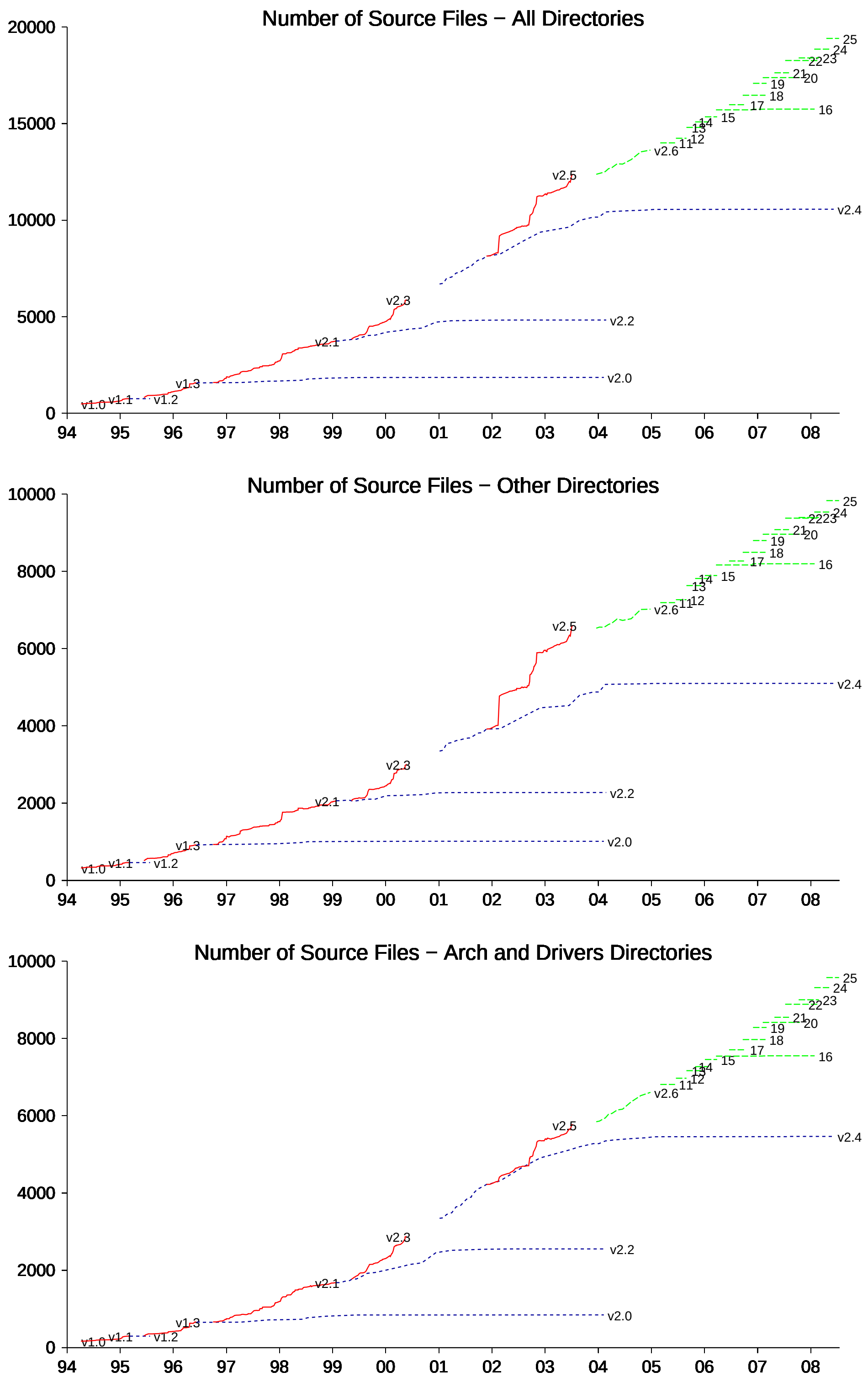


Figure 5:
The growth of number of files in Linux

After that, the pattern of the increase is pretty “normal” without major jumps and spikes.

5.2.2 Number of Functions

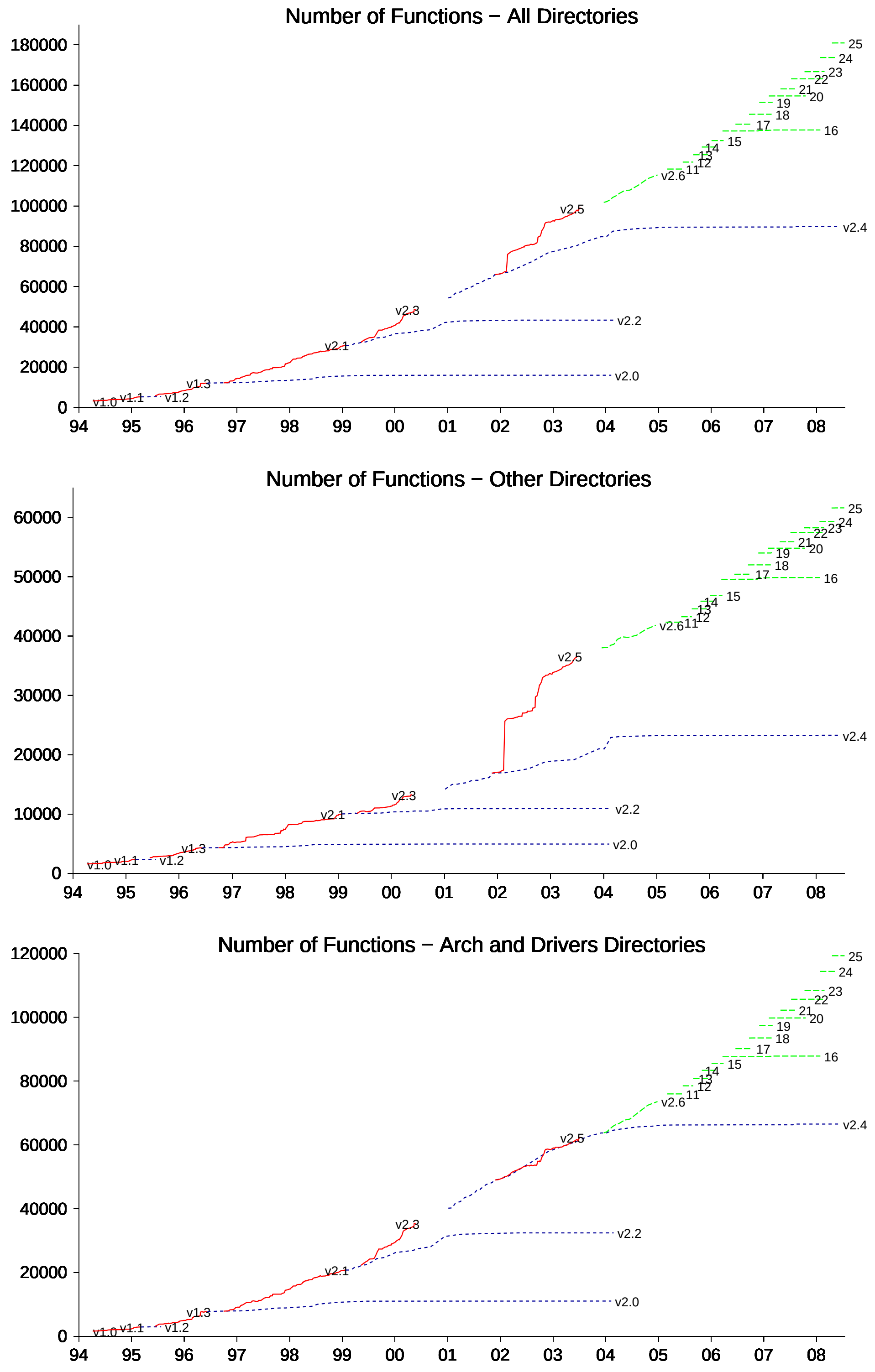


Figure 6:
The growth of number of functions in Linux

Fig. 6 shows the growth in number of functions in Linux. Overall, we see a similar picture to that of number of files — they are steadily increasing over time, where they are usually more constant in the production versions. We also see the same behavior regarding the 2.6 series.

We can see here significant differences between *arch* and *drivers* directories and all others. First, we see that in terms of sheer number of functions, about $\frac{2}{3}$ to $\frac{3}{4}$ of them are in *arch* and *drivers* and only $\frac{1}{4}$ to $\frac{1}{3}$ are in all other kernel directories. Another issue is the growth which is different for each of the subsystems: notice the differences in v2.2 and v2.5. In v2.2 the *arch* and *drivers* grow rapidly whereas they were pretty steady for all other directories. In v2.5 on the other hand we see big "jumps" for the rest of the kernel — meaning more development and enhancements were added there (the *arch* and *drivers* are growing slowly in this time, merging with the growth of v2.4 — the only merge in the graph). This matches the patterns we have seen in the file number level and shows a similar picture from the functions point of view.

Notice that comparing with the number of files, the jump in the number of functions for version 2.2 in the *arch* and *drivers* directories is much higher (over 3000 functions added in one version — 2.2.18, at the same time about 200 files were added as well). These increases are due to many changes and additions preformed in the version — improved USB support, addition of network drivers and more. Note that 2.2 is a production version, but still it was decided to add features to it.

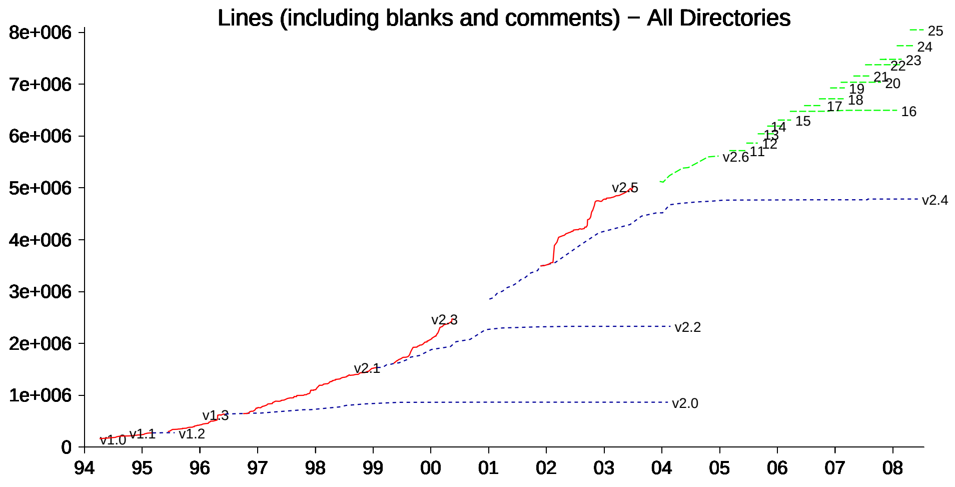
5.2.3 Lines of Code

As mentioned above, we calculated the LOC both as total lines in the file and as lines of actual code sans empty lines and comments (to be denoted "lines" and "LOC"). When examining the number of lines in a file and number of lines of code (LOC), we see a very similar picture (Fig. 7). This picture resembles also the patterns we have seen above, for files and functions increases. The number of lines grows with time, and the growth rate is much higher for development versions, whereas in production versions there is usually slow growth or constant value. We also see the same behavior of version 2.6 (each minor version growth is constant, but all together there is a linear growth rate). These results match those of previous literature, Godfrey and Tu [5] measured lines of code in 2 different manners: the actual lines in the file (using wc -l Linux command) which is the lines including blanks and comments, the lines without comments and blanks (using an "awk" script). Their findings reveal that the percentage of comments and blank lines in the files are almost constant at between 28–30%. Eventually, they used uncommented lines of code as a measure of the size of the software system and used this value in the growth analysis. They found that the growth of the system, in its first six years, was super linear.

When looking at the graphs for the different directories, we can easily spot the jumps we have seen before for 2.2 and 2.5. See for example Fig. 8 which shows the growth of LOC for the *arch* and *drivers* directories and for all others. In relative terms, the jumps in LOC in 2.5 are larger than the jumps in files and even in function, indicating that the added files and functions were larger than average.

Next, we calculated these metrics of general number of lines and LOC at the file and the function level (averaging the values over the number of files/functions accordingly). We will show the results for the LOC value, as the general number of lines graphs are qualitatively similar, and highly correlated.

When comparing the averages per file (Fig. 9) we see the following picture: when examining the average for all the directories, generally there is a slow growth between major versions,



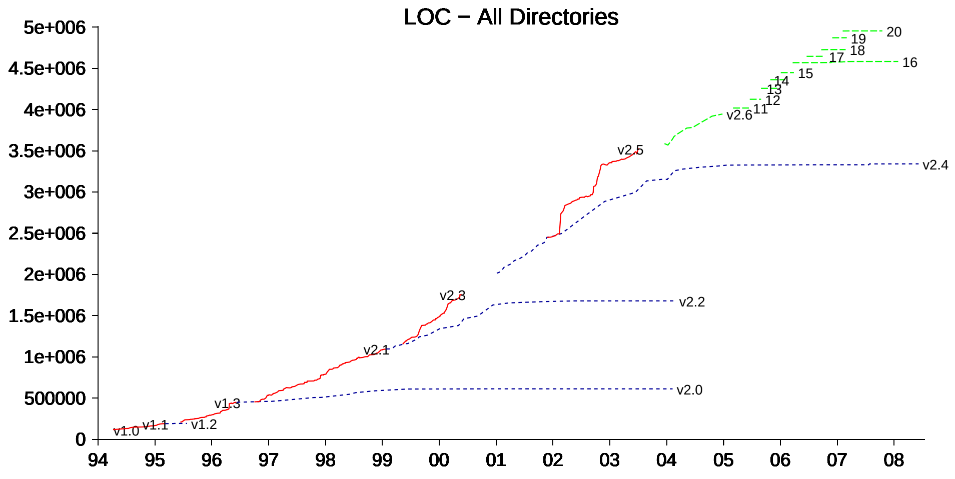
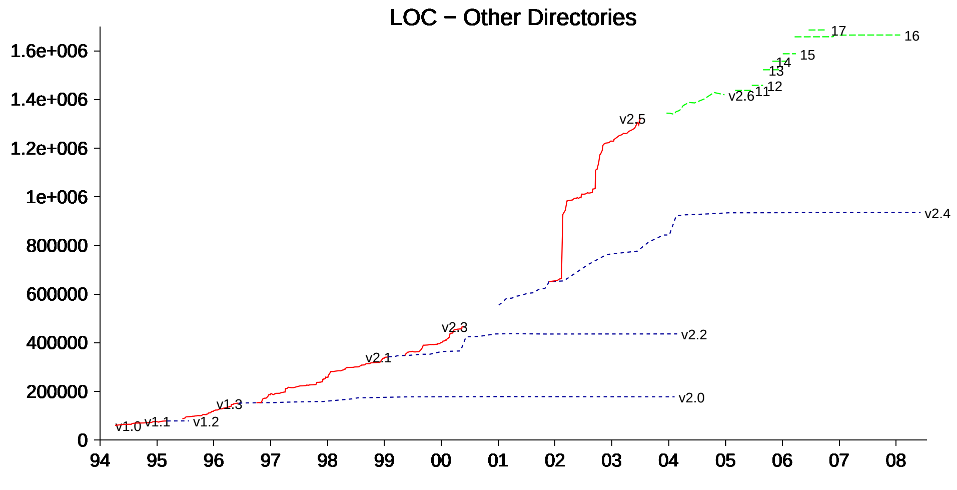


Figure 7:
The growth of number of lines in Linux, with and without comments and blank lines

until about version 2.3, and then a very slight decrease. Another interesting finding is that for the production versions series 2.0, 2.2, 2.4 the initial growth is much higher than of consecutive development versions, and they end up having the highest LOC/file. Another issue is that whereas the other versions have much more volatility in the LOC per File value, for these production versions (2.0, 2.2, 2.4) we see that after a more "smooth" increase (less volatile) comes a constant value. Interestingly, most of the volatility comes from the *arch* and *drivers* directories, where the files are also bigger on average and span the range of 370–500 LOC. Interestingly, in these directories we see a "hump" shape, which started in an increase and then turned into a decline. In the other kernel directories, on the other hand, the LOC per file is almost always in the relatively narrow range of 160–200. The only structure in the other directories is a drop in 1.1, and jumps in 2.2 and 2.5 (only one, corresponding to the first jump identified previously).

Combining this information with the general LOC (Fig. 7) and the growth in number of files (Fig. 5), we find the following: the LOC (and number of files) was increasing the whole time, and increasing between versions, and here we find also decrease, a much more volatile trend and a higher increase for production versions than others. Since in each version we add



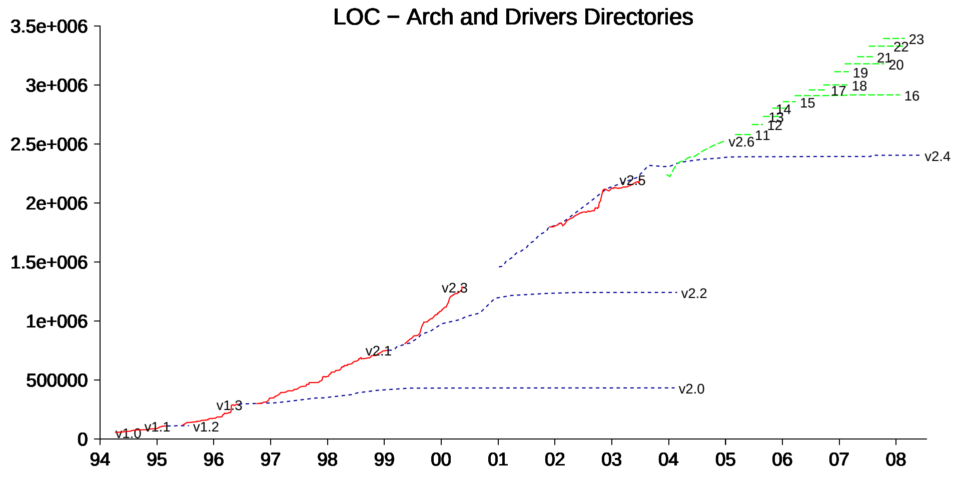


Figure 8:
The growth of LOC in Linux, for different directories

different number of files with different LOC values and also update existing code and increasing the LOC, the growth is not smooth and the changes between two consecutive versions can be large (and in both directions). For the production versions we can see that the growth in the number of files was smaller than that of the LOC, meaning each file, in average, has “gained” more LOC. In other words, looking at each of the graphs LOC and number of files separately reveals only a small part of the picture — we just see an increase and we can say Linux is constantly growing, both in amount of files and in LOC, but when we look at the averages we see the “marginal change” per file, only when we will look at the real data we can understand what exactly happened. An example of the incomplete information is in version 2.2.16 — in this case we see a small increase in LOC and in number of files, but a very large increase in the average. The cause of this is addition of only 4 files to the fs directory with total of over 54000 LOC (the jump is especially visible when looking at the data for the core kernel files).

Looking at the data per function (Fig. 10) we find a slightly different picture: initially the LOC per function is relatively stable, and then in version 2.6 it is decreasing. Again we can spot a different behavior of versions 2.0, 2.2 and 2.4 which are more constant over time (with the exception of 2.2 explained above) — matching the development scheme of production versions.

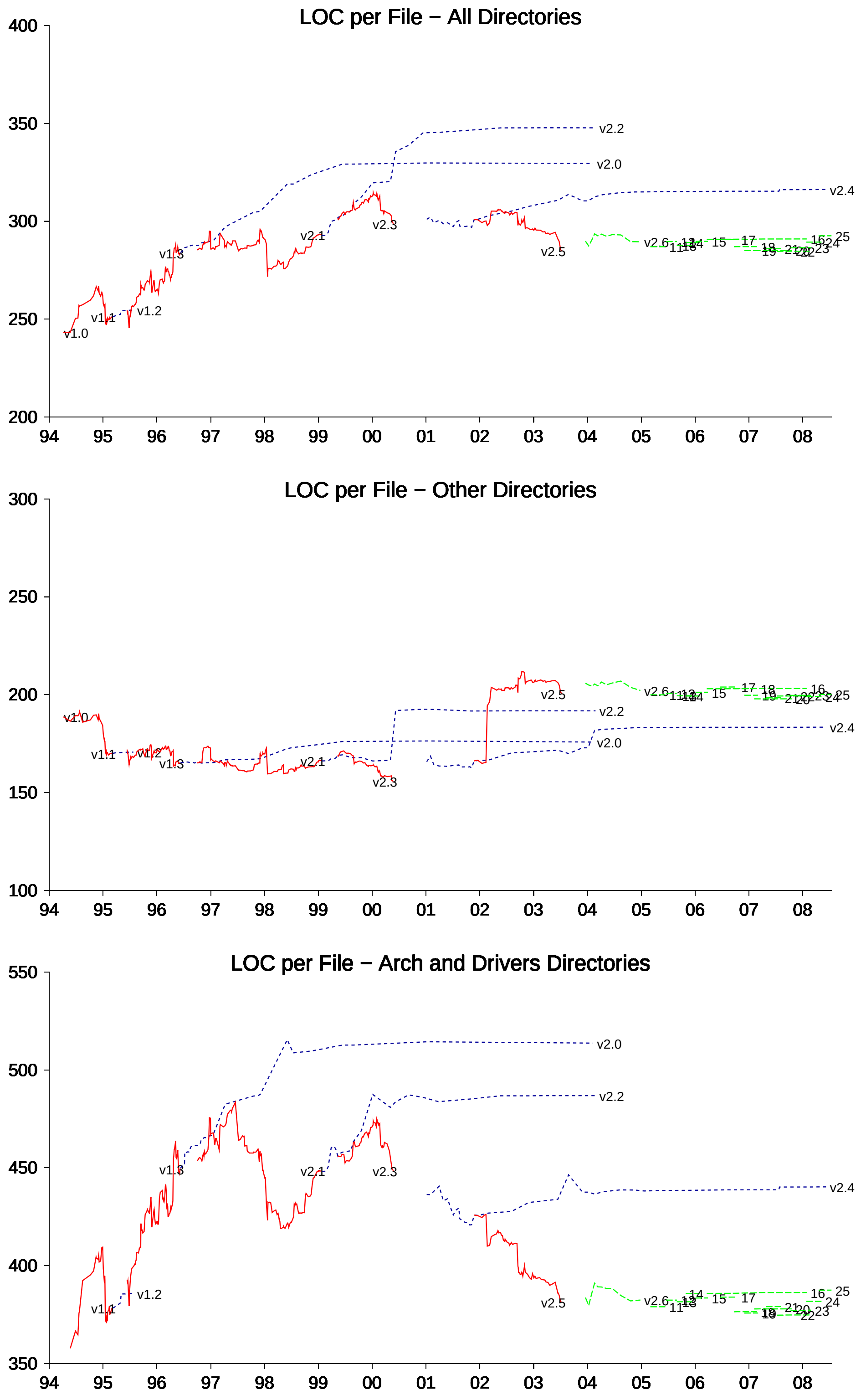


Figure 9:
The Average LOC per file

The explanation for the decreasing LOC per function while the general LOC is increasing will

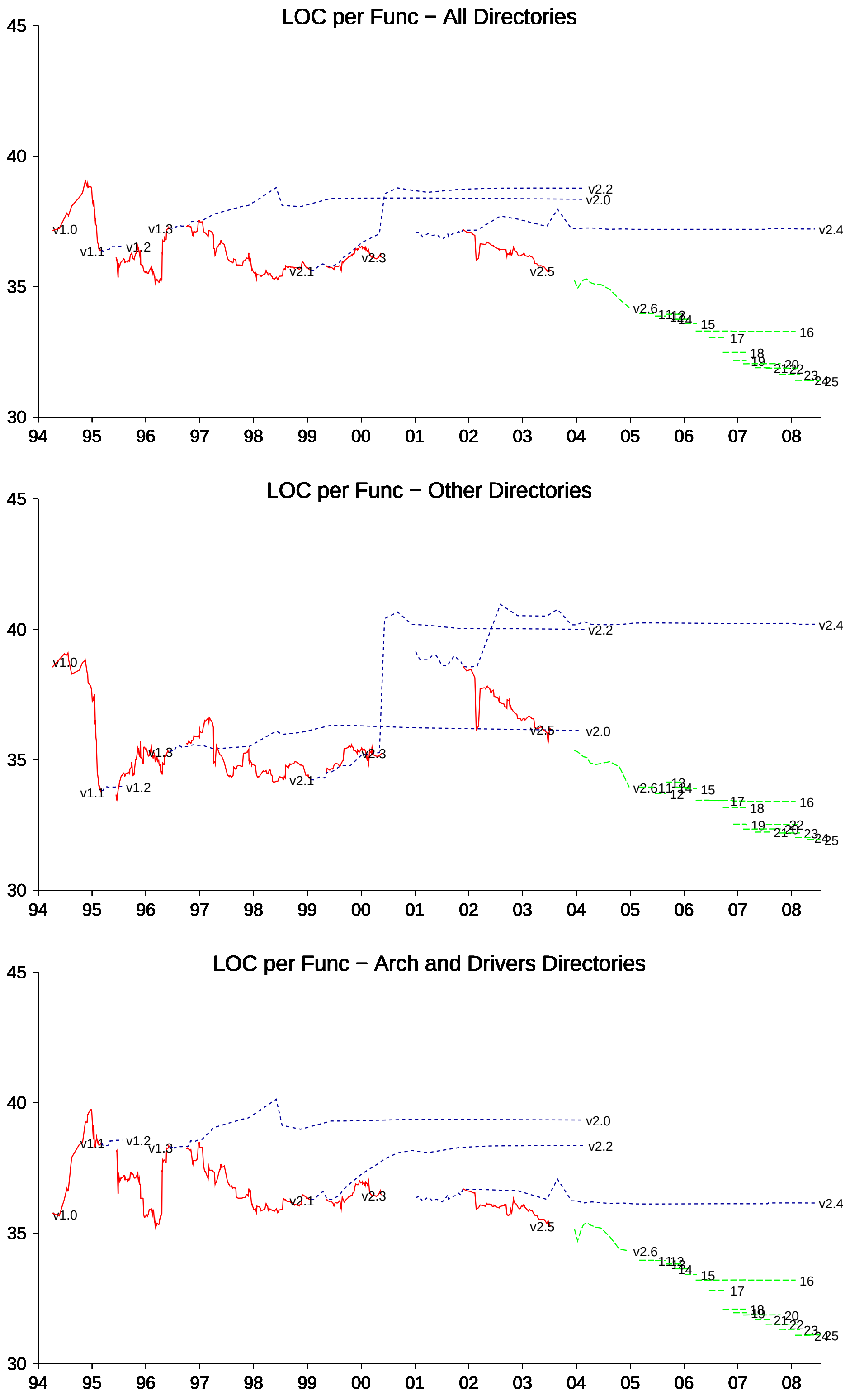


Figure 10:
The Average LOC per function

naturally be an increase in the number of functions (as we have seen in Fig. 6).

This result is similar to that of Thomas [33] which examined the software quality and maintainability metrics in Linux using a CASE tool. He used the production version 2.0.*, 2.2.* and 2.4.*. The tool produced two different LOC metrics: one which was the actual lines of code and the other NCNB-LOC which is the non-comment non-blank LOC. In all his measures he used the latter metric as the lines of code. Thomas found two phenomenas regarding LOC. At the kernel level, the size of the kernel grows linearly over time, with much less volatility than between series. On the other hand, the average lines of code in a function consistently decreased from series to series. These two finding lead to a conclusion that the number of functions must be increasing over time.

Focusing on the core kernel directories, we see a somewhat strange behavior. First, there is a big decrease in LOC per function in version 1.1, while there is in increase in the *arch* and *drivers* directories of the same version. Then, there is a big jump in version 2.2, which seems to also lead to a high initial value in the 2.5 series — but that decreases gradually throughout the series.

Another related metric is the percent of comments (more commented code is expected to be more maintainable), which gives another perspective to the relation between the two terms LOC and NCNB.

Fig. 11 displays the ratio of the comments to the total non-empty lines (comments and LOC) in files in each kernel. We see that generally, the comments comprise around 25% (Similar to the results of Godfrey and Tu [5]), and this percentage is more stable for production versions. In the scope of all directories, there is a general decreasing trend, to about 22%, while the production kernels (2.0, 2.2 and 2.4) have somewhat higher values, and are more stable. These general trends are a combination of different behaviors for the *arch* and *drivers* directories and the rest. In *arch* and *drivers*, there is a big increase in comments in version 1.1, and then a general decreasing trend (to around 20%), where again the production version's values are more constant and higher. The trend in the core kernel directories is also a big increase in 1.1, but then it stays relatively flat, except for big jumps in 2.2 (upwards) and 2.5 (downwards).

5.2.4 McCabe's Cyclomatic Complexity

First, we examined the total McCabe Cyclomatic Complexity (MCC in Fig. 12) and the Extended McCabe Cyclomatic Complexity (EMCC in Fig. 13) for all versions. As can be seen by the graphs, they are very much qualitatively similar (of course, EMCC is higher in magnitude), and for the rest of the analysis, we will discuss only MCC (the more commonly used metric), as it is practically the same. The pattern of the graphs, for all different divisions into directories, resembles that of the number of LOC, not surprising due to the correlation that was found between the two (as discussed above).

These findings also match those of Thomas [33] who used the cyclomatic complexity calculated by the CASE tool which is the "simple" complexity — number of predicates plus one (MCC above). His findings show that there is a linear growth in the cyclomatic complexity, which is more volatile between series than within the series. He also notes that since series 2.2.* and 2.4.* have larger code bases, there are more branches in the code and thus the complexity is higher.

Next, we looked at the average MCC per function, which is interesting since the original metric is used for module complexity and the modules here are functions. Looking at this

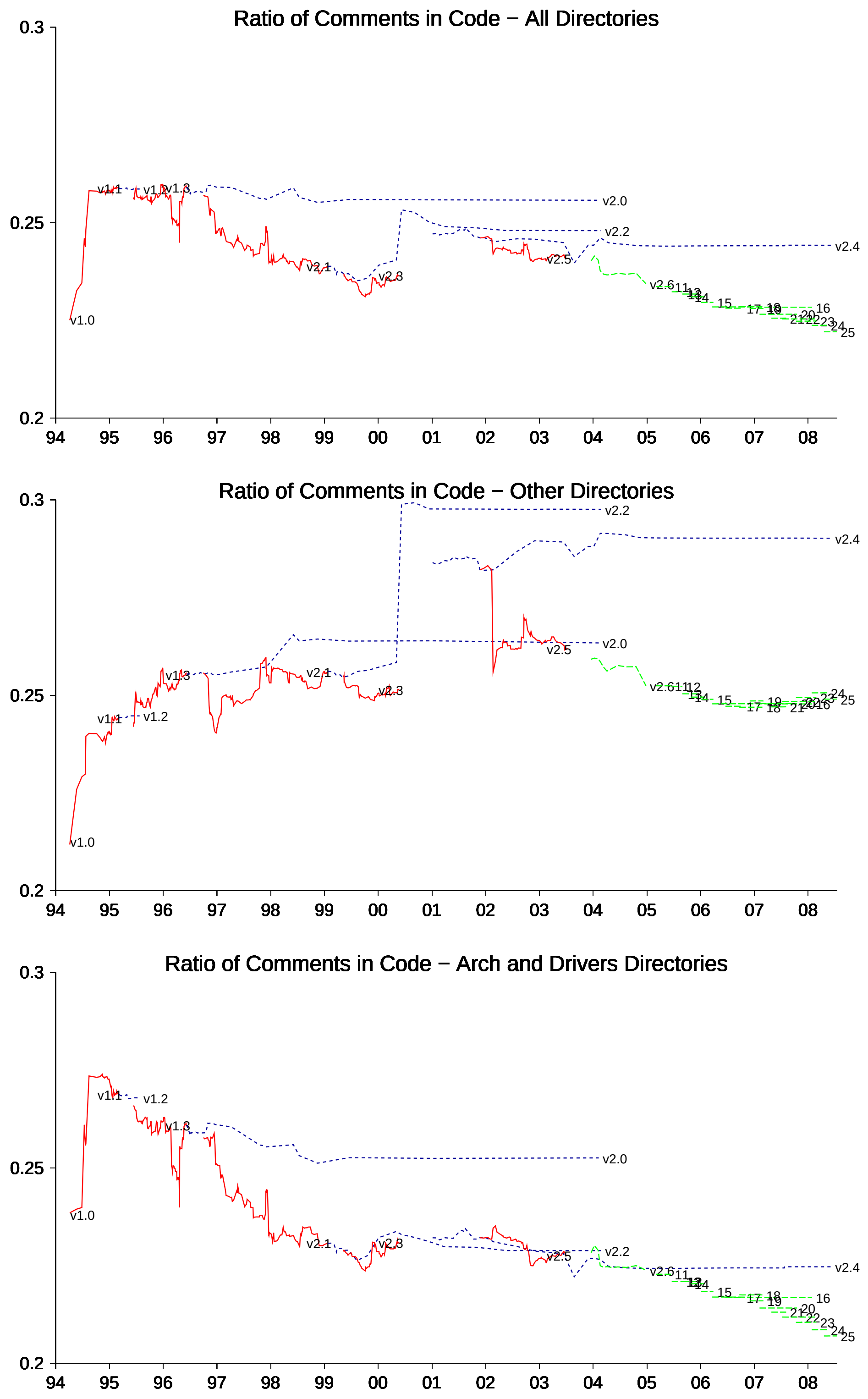


Figure 11:
Ratio of Comments in the Code, out of the sum of LOC and Comments

metric (Fig. 14) we find a decrease over time. The same pattern was found by Thomas [33]. In his results the values are slightly different and also the 2.2 series values are much lower than

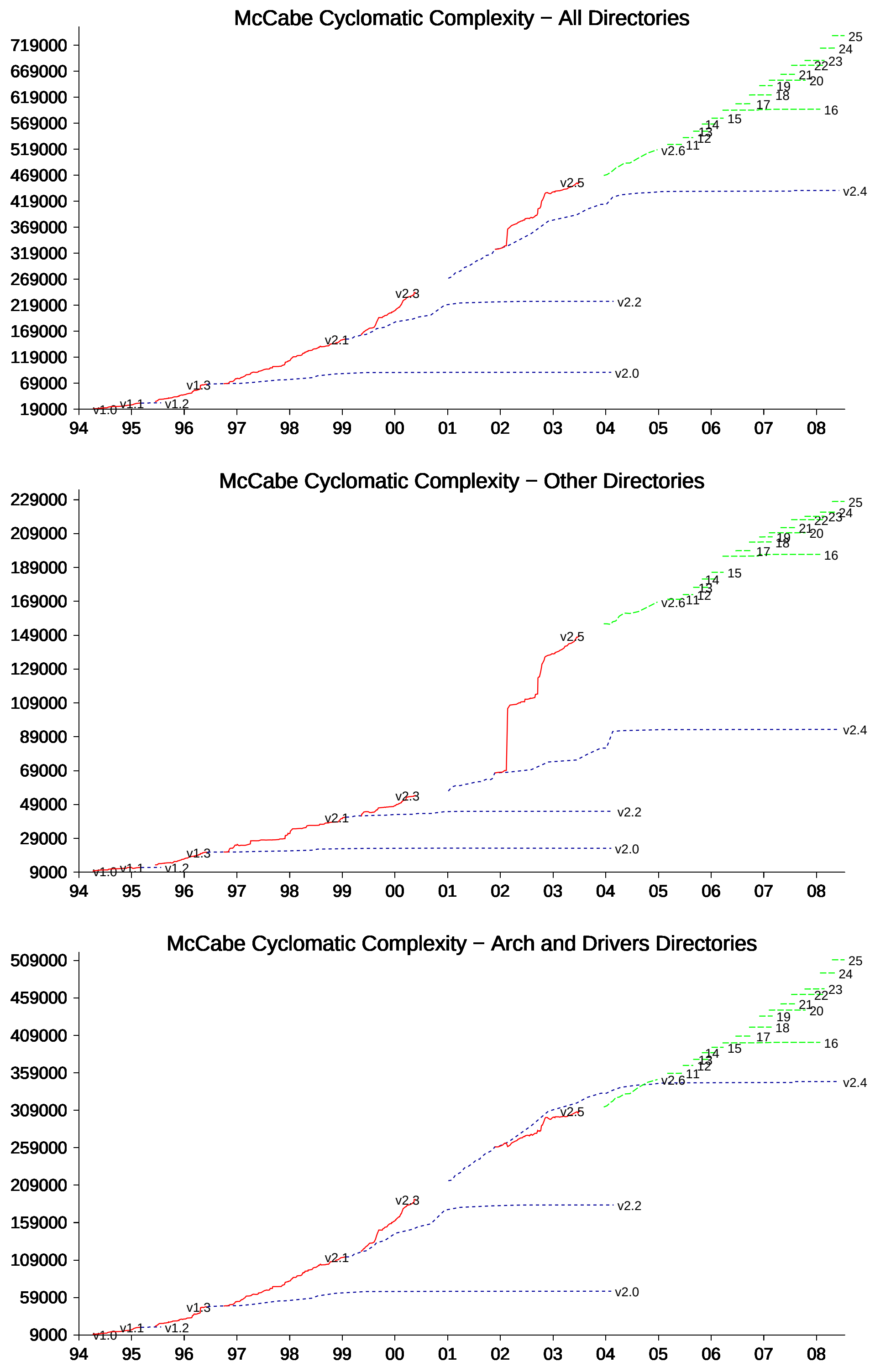


Figure 12:
McCabe's Cyclomatic Complexity for all Kernels

those of 2.0 and of 2.4, without apparent explanation. Since he analyzed different files and used a different tool, we expect slight changes, but the general picture remains the same.

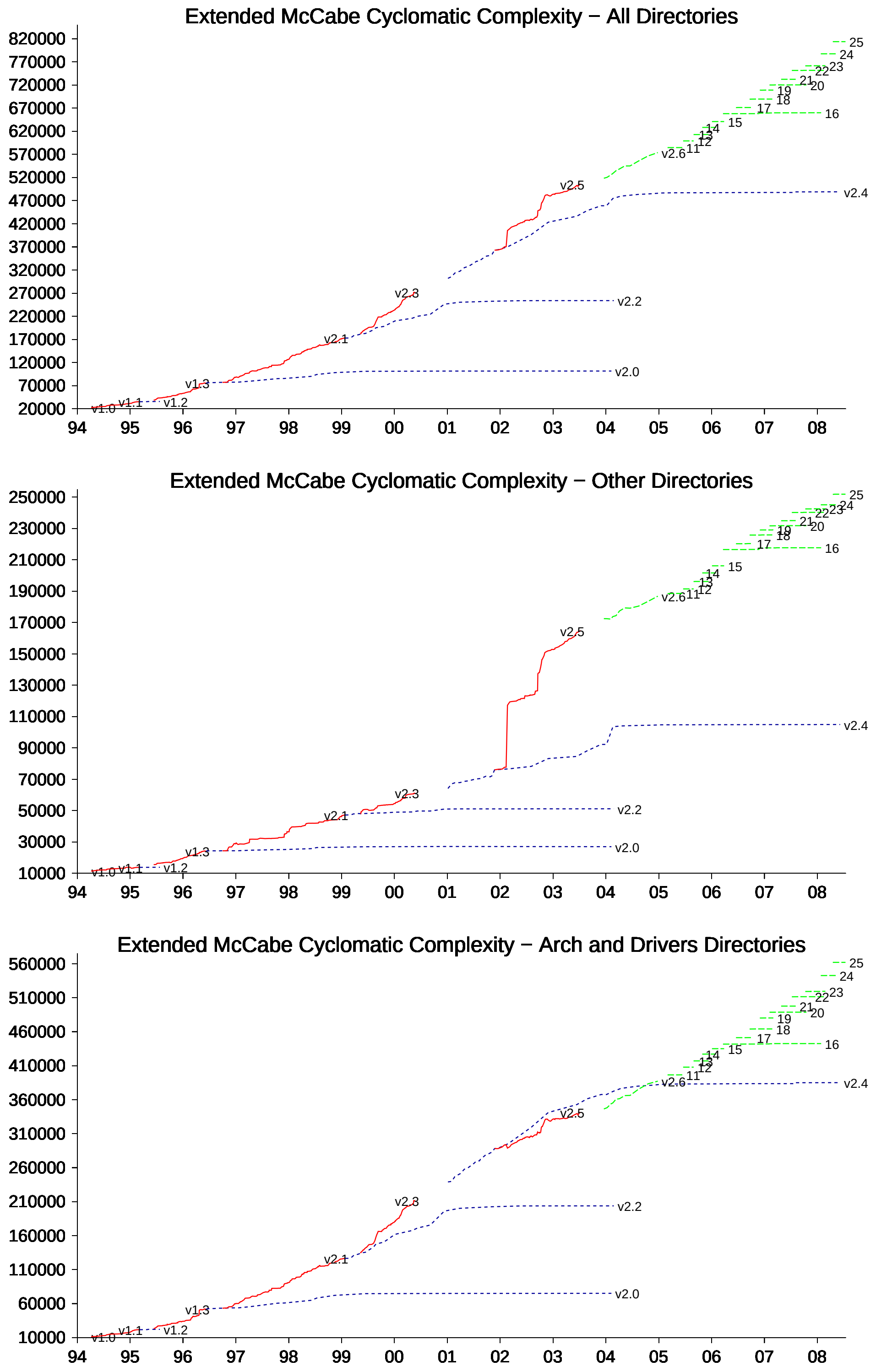


Figure 13:
Extended McCabe's Cyclomatic Complexity for all Kernels

Looking at Fig. 14 might give the impression that with time the average complexity of the functions in decreasing, indicating that maybe the quality of the kernel is improving with time!

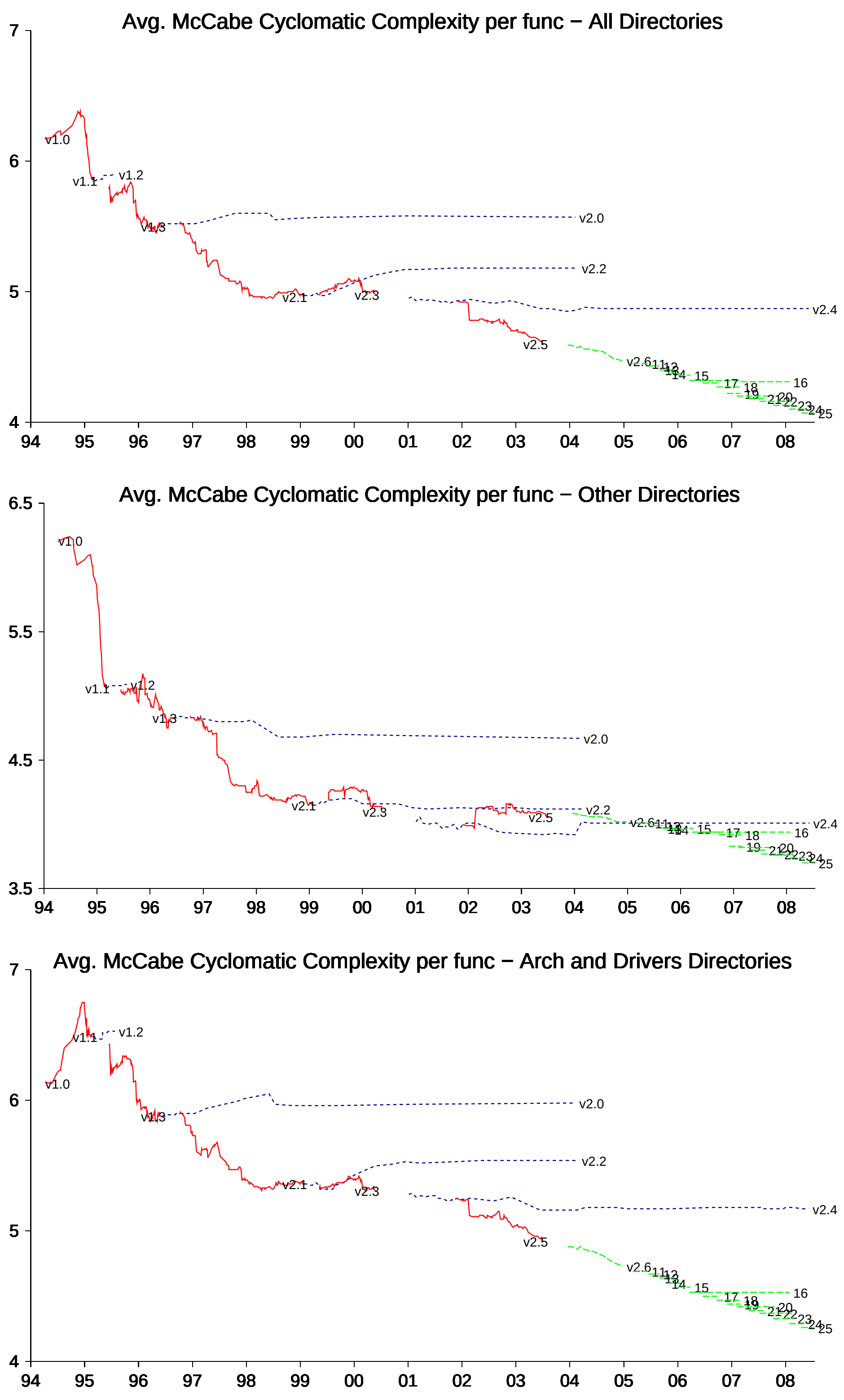


Figure 14:
Average McCabe's Cyclomatic Complexity per Function for all Kernels

However, one should not get too excited prematurely; we must remember that the number of functions and number of LOC also increases with time. It might just be that we have more functions with low complexity or just more functions which will give this false picture. In order to check whether this is the case we looked at the distribution of the MCC per function over the different versions. Fig. 15 displays the cumulative distribution functions of the MCC of functions for most of the major kernels. For each kernel displayed the data is taken from its first version (The chosen kernels are those where we saw changes in MCC per function values between them and their preceding version. That explains why the development versions starting version 2.1 are not shown).

The lines are qualitatively similar, and are hard to distinguish at first, but notice that the order of the lines corresponds to that of the kernels in the following way — the most inner line is kernel v1.1, the next is v1.2, etc., and the topmost one is v2.6.16. Notice also that v1.2 and v1.3 kernels almost merge and so do v2.2 and v2.4. The fact that the order of lines corresponds with the order of kernels in such way, immediately says the following: the cumulative distribution function of kernels over time, although qualitatively similar, is more concave. In other words, over time we have more functions with lower MCC value.

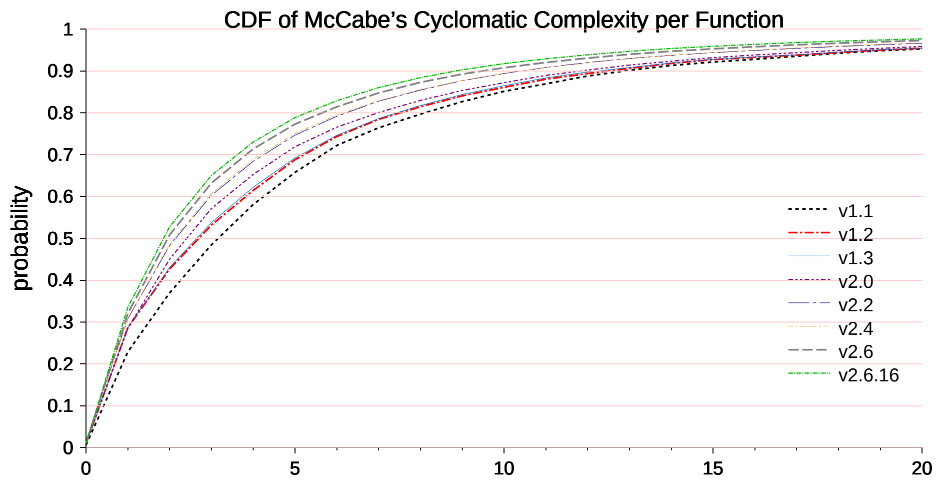


Figure 15:
Cumulative Distribution Function of McCabe's Cyclomatic Complexity per Function for selected kernels

Notice first, that until about MCC value of 12, the line corresponding with v1.1 is much lower, i.e. there are much less functions in 1.1 with low (0–12) MCC value in comparison to the later versions. This can imply that work was done in order to improve the complexity of the initial version code. The other versions are more similar to each other, especially for MCC values of 1 and under, or 12 and above. Looking at the figure we can easily see that starting with version 1.2 around 28% of the functions have MCC value of 1 or 0. This increased to 30% and over in version 2.2. The same behavior occurs with the value of 2 and less (starting at around 42% in the older versions and approaching 50% in the newer versions). The percentages of the rest of the values are very similar and slightly going down; for example, in earlier versions 70% of the code was under MCC value of 5, and in newer versions it is close to 80%. For the range of MCC values of over 1 and under 10, we can see that the big changes are in versions v2.0, v2.2 and v2.6. In other words, the changes (improvements) are in production versions except for the initial version 2.4 for which the MCC values are similar to 2.2. This can be explained due to the problems with the release of version 2.4, which was long-waited for and had many issues and complications in development and testing (as we describe below).

We can also see that 90% of the code has MCC value of 10 or less (86% in older versions), which, as stated above, indicates of “a simple program, without much risk” according to the SEI. This implies that 90% of the functions in Linux are easy to maintain. Also, there are very few functions with high-risk or which are unstable according to this metric. Regarding MCC of zero — in each of the kernels there is about 1% of the function with MCC of the value 0. This happens when the function is simply empty.

In other words, there are two major reasons for the decrease in the average: (1) the number of functions in increasing faster than the MCC; and (2) to some extent, there are changes in the “types” of functions inserted to the kernel over time (more low complexity functions are inserted). It is hard to conclude that the complexity is becoming smaller (especially since the vast majority of the functions are “not at risk”), but we might conclude and say that it is not increasing.

It should also be mentioned that out of the boundaries of the graphs there are cases of single functions (one function each time) with very high MCC. Their contribution to the distribution is close to zero (for example, in version 2.6.16 we have a single function with MCC of 255; in version 2.2 there is a function with 470). To study the high MCC functions, we look at the tails of the distributions. This is done by plotting the survival function of the MCC values per function (i.e. for each MCC value, what is the fraction of functions that has higher value than that) in a log-log scale.

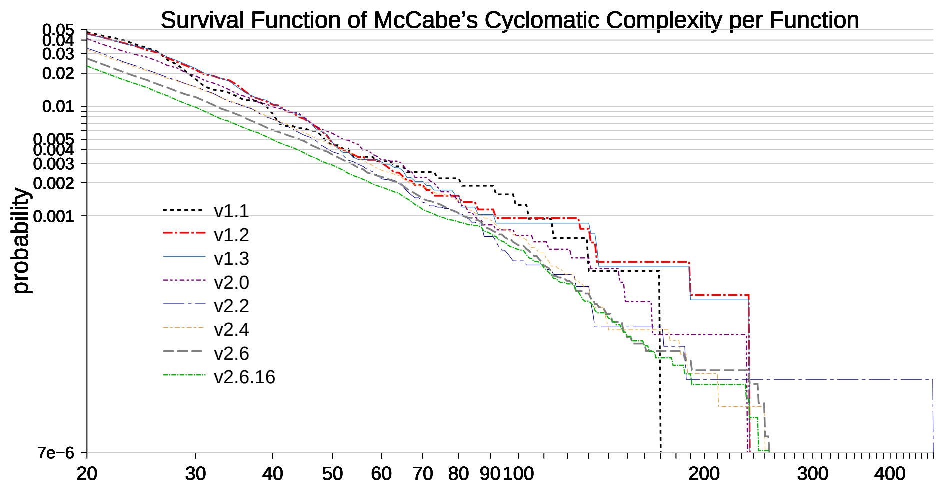


Figure 16:
Log-log survival functions of McCabe’s Cyclomatic Complexity per Function for selected kernels

Fig. 16 displays the results. First, we see that the fraction of complex function (MCC over 20) is less than 5%, and the fraction of those with high risk (MCC over 50) is less than 0.6% for each of the kernels. The plots are close to being straight lines in the log-log axes, indicating a power-law relationship and that the distribution has a heavy tail.

Looking at the 20-50 MCC range, the 4 newest kernels (2.2, 2.4, 2.6, and 2.6.16) keep the same order as in the CDF function above: the newer a version is, it drops off faster. In other words, it has lower percentage of functions with high MCC values. The other 4 kernels behave a bit differently than in the lower MCC values: version 1.1 has the highest percentage of function with MCC values between 20 to 26, but then it crosses the other lines, until its values are lower than those of version 2.0, indicating that until about MCC value of 60, 1.1 version has a lower percentage of function with high MCC in comparison to versions 1.2, 1.3, and 2.0. Notice that for the value of 60 and up the other lines cross version 1.1 again, and have lower percentage of functions with high complexity. A reasoning behind this behavior, that version 1.1, although initial and with a larger fraction of complex functions for lower MCC values, has a lower percentage than later versions for higher MCC values, could be that the software at 1.1 was more simple and initial, and that later improvements, bug fixes, and environment and architecture changes added complexity to it.

For versions 1.2 and 1.3 we see that they are merging for some of the values and interlacing for others, but generally are very close to one another. We see that for some values (about 26 to 42) they are "worse" than 1.1 and have the largest percentage of complex functions, as described above. Version 2.0 is "worse" than the rest starting about MCC value of 42.

Starting with MCC values of 70 and higher, we see a "step" behavior of the functions, which is due to the fact that we are actually observing individual function. This causes the plots to cross each other, and they do not keep a certain order. in version 1.1 the highest MCC value is 170, in version 2.2 it is 470, and the other versions have values between 230 and 260. We can see that the newest version has most of the time the lowest values, but we cannot conclude much about the rest. However, since the probability of functions to be in such values is so low, this is not the significant effect on the complexity of each kernel.

Looking at the tails raises more questions — what are the functions with the high MCC values and what happens to them over time? Analyzing the high MCC value functions revealed a few interesting observations. First, the high value MCC functions are usually from the *arch* and *drivers* directories (they are such for all the top values of the examined kernels). Second, these functions are many times *interrupt handlers* or *ioctl* functions, which receive a request/command as an integer, interprets it and behaves accordingly. The implementation of this is many times using long switch statement with tens of different cases, or using multiple "if" commands. This, of course, results in a high value of MCC. We also found that many of the high value functions are the same (i.e. in the "top MCC values") for different kernels.

Specifically, we display here two examples of high MCC value functions, and analyze them over time. The first (Fig. 17) is a representative example of functions which have high values throughout the different kernels. In this case, it is the *vt_ioctl* function in the *"linux/drivers/char/vt.c"* file. In the figure, we see the following graphs: on the left side, the two top graphs are the LOC and MCC of this specific function, next to each of them is the LOC and MCC in the file level. The bottom left is the MCC value of the function after its migration (see details below) and on the bottom right we can find the number of functions in the original file.

Starting with the early version of 1.1.0, the MCC value for this function was 100, and it jumped to about 200 in version 1.2.0, since then it had about the same value. At the same time there were no drastic changes in the file or function (LOC, MCC and number of functions changes moderately). In mid version 2.5 we see a change. First, we see a small change in the function's MCC value from about 180 to about 160, and later the function is completely removed from the file and the number of function in the file increases dramatically (from less than 20 to over 100), as a result the LOC and MCC in the file level increase as well. Interestingly, the *vt_ioctl* function was not replaced by the new functions, the structure of the directory

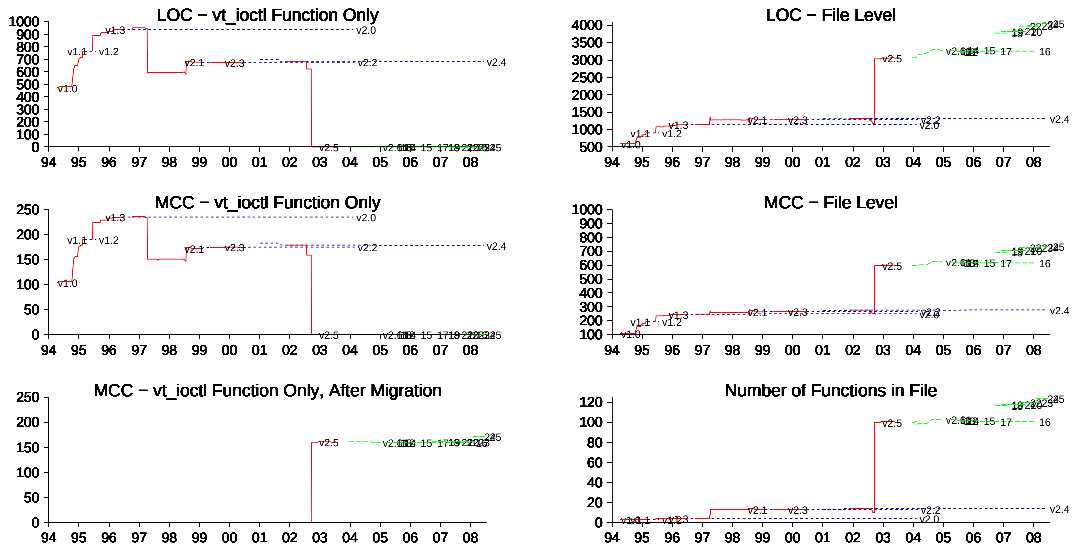


Figure 17:
Analysis of constantly high MCC value function, over time

was changed —– a new file (*"linux/drivers/char/vt_ioctl.c"*) was added, and the function migrated there. In its new home, the MCC value stayed at the same level of 160, and grew up to 170 (the MCC values remained as if it never moved). Notice that the migration occurred in a development version, while the production versions remained with the old structure.

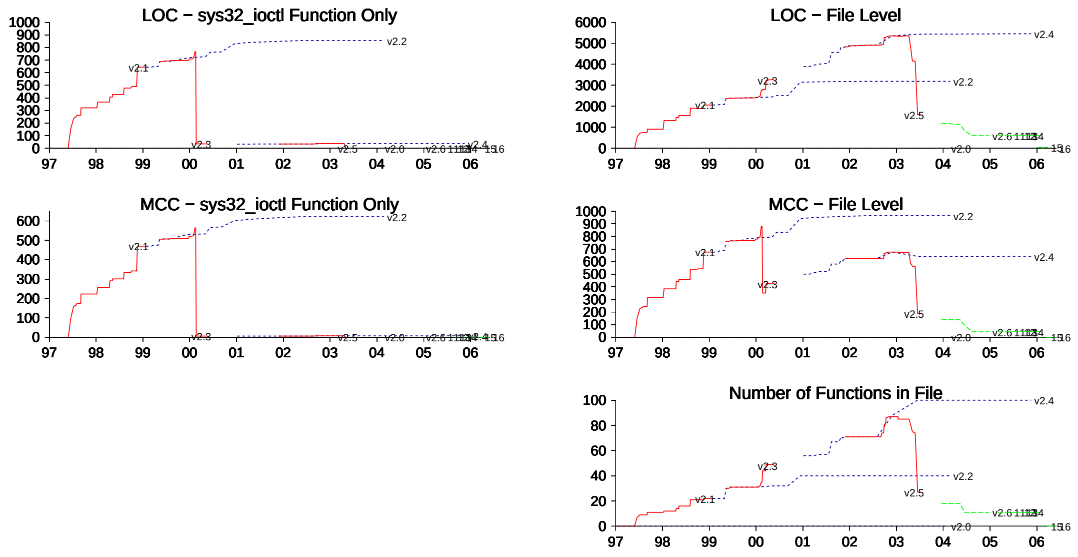


Figure 18:
Analysis of high MCC value function, changing over time

The second example (Fig. 18) is of the highest MCC function in the whole dataset, which is the *sys32_ioctl* function in the *"linux/arch/sparc64/kernel/ioctl32.c"* file, with an MCC value of 470 in release 2.2.0 (by the way, the second highest value of version 2.2 is 187, which is close to the highest value of the other functions. It belongs to a function with high value for other kernel versions as well). This case is slightly different from the above. First, the file and function were introduced to the Linux kernel only in the middle of version 2.1 (2.1.42 in mid 1997). It grew both in LOC and in MCC value throughout versions 2.1 and 2.2 (reaching an MCC value of almost 600 at the beginning of 2001). However, in the middle of version 2.3 (2.3.47 in the beginning of 2000), the MCC value and the LOC of this function dropped drastically to the MCC value of 6. It remained in that level throughout 2.4 and in most of 2.5 (until version 2.5.69 in mid 2003). In that 2.5 version, the function was completely removed (and did not migrate in the kernel), and also many other functions were removed from the file. It remained that way also at the beginning of version 2.6, until the file was completely removed in version 2.6.16. The functionality of this file moved to other places in the kernel, using other files and function.

The drop from MCC value of around 600 to the value of 6 in version 2.3, was due to a major change in the file. At that time, the number of functions in the file and the LOC grew. A new method of handling the request was inserted, and instead of using the switch statement, a new *struct* with a handler was introduced. Now, instead of using tens of “case” statements, a table with all the different cases was used. This caused the MCC value to decrease so much.

The case of the *sys32_ioctl* function is an example of a function which had a very high MCC value (a very complex function, way off the SEI chart) that dropped to a very low value (that corresponds with “a simple program”). The same drop occurred in another maintainability measure — LOC, and, by definition of the other measures, in them too. This is a real case of perfective maintenance, in which the readability, the performance, and the maintainability of the file was immensely improved. Notice that this change occurred in a development version, while the production versions remained with the old function.

To conclude, we learnt that the high MCC values are usually related to *arch* and *drivers* directories, and to functions with long *switch* sections. Cases of high MCC are usually of functions that have high MCC values throughout the different kernels. However, sometimes these functions are rewritten and improved. In such cases, the kernel’s complexity is really improving with time.

We also saw that while there are changes in structure and functions’ maintainability in development versions, the production versions they emerge from remain with the old structure/functions. This gives an assurance to our assumption that perfective and preventative maintenance usually happens only in development versions.

5.2.5 Halstead’s Software Metrics

As stated above, there are some cases when Halstead’s Metrics can not be calculated (specifically, if the vocabulary n equals 0, then Halstead Volume (HV) can not be calculated, and if n_2, the number of operands in the function, equals 0, then Halstead Difficulty (HD) can not be calculated). These cases generally occur when the function is empty or when it includes only operators (for example, when the function is only return or a function call, etc). In these cases instead of the undefined value of HV and HD (due to attempt to calculate $lg_2 0$ or to divide by zero), we treated the function as if its volume and difficulty are 0; when we aggregate the volume and difficulty we ignored this specific function and when we calculated averages, we used it. In the all directories level, the amount of such cases is around 0.9% out of all functions where HD could not be calculated and around 1.5% out of all functions where HV could not be calculated.

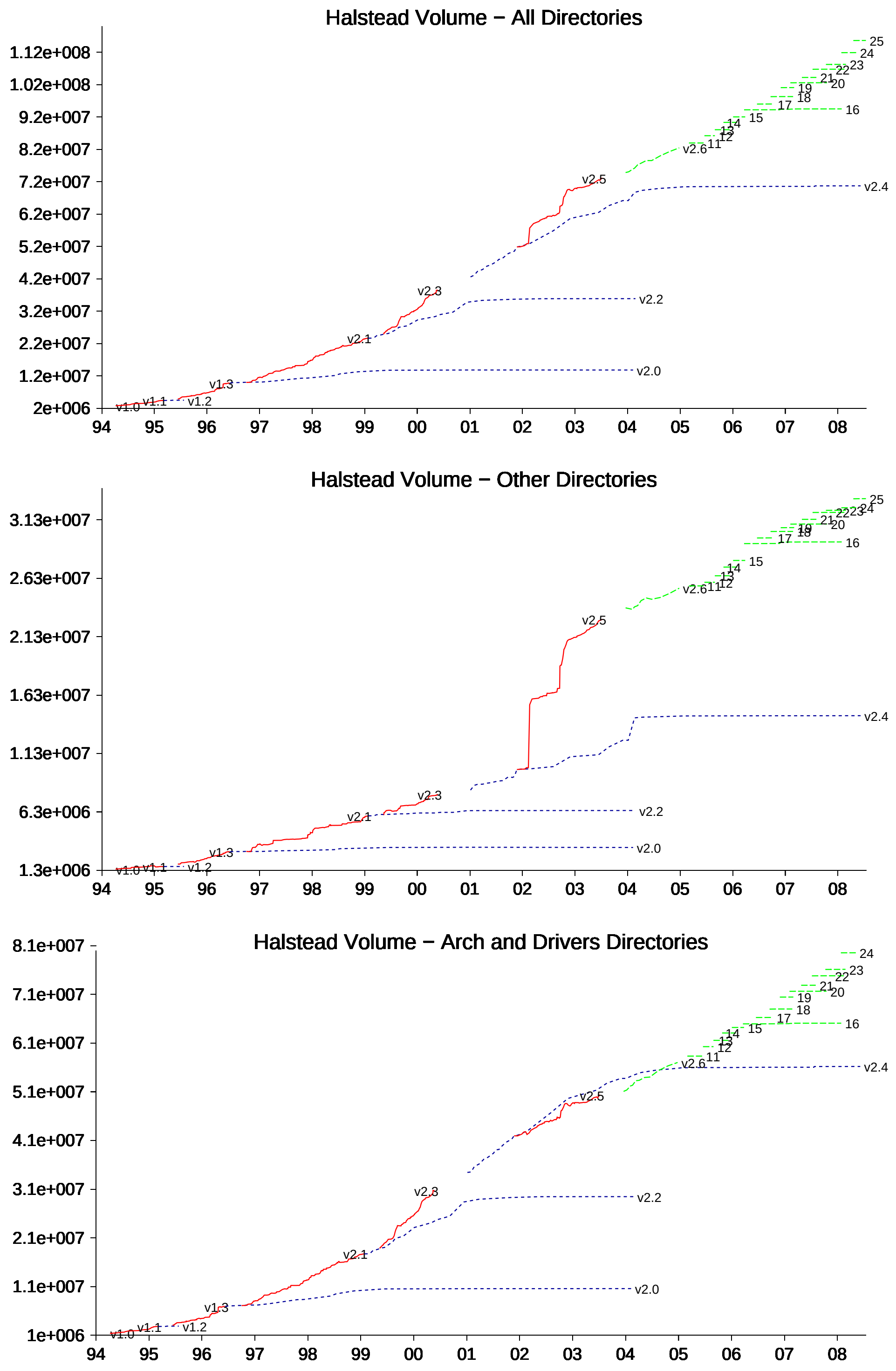


Figure 19:
Halstead Volume for each Kernel

Again, the pattern when looking at the kernel level is a growing one. All the different graphs (HV in Fig. 19, HD in Fig. 20, and HE in Fig. 21) show almost exactly the same pattern which resembles the one we have seen so far for all other metrics, including the jumps

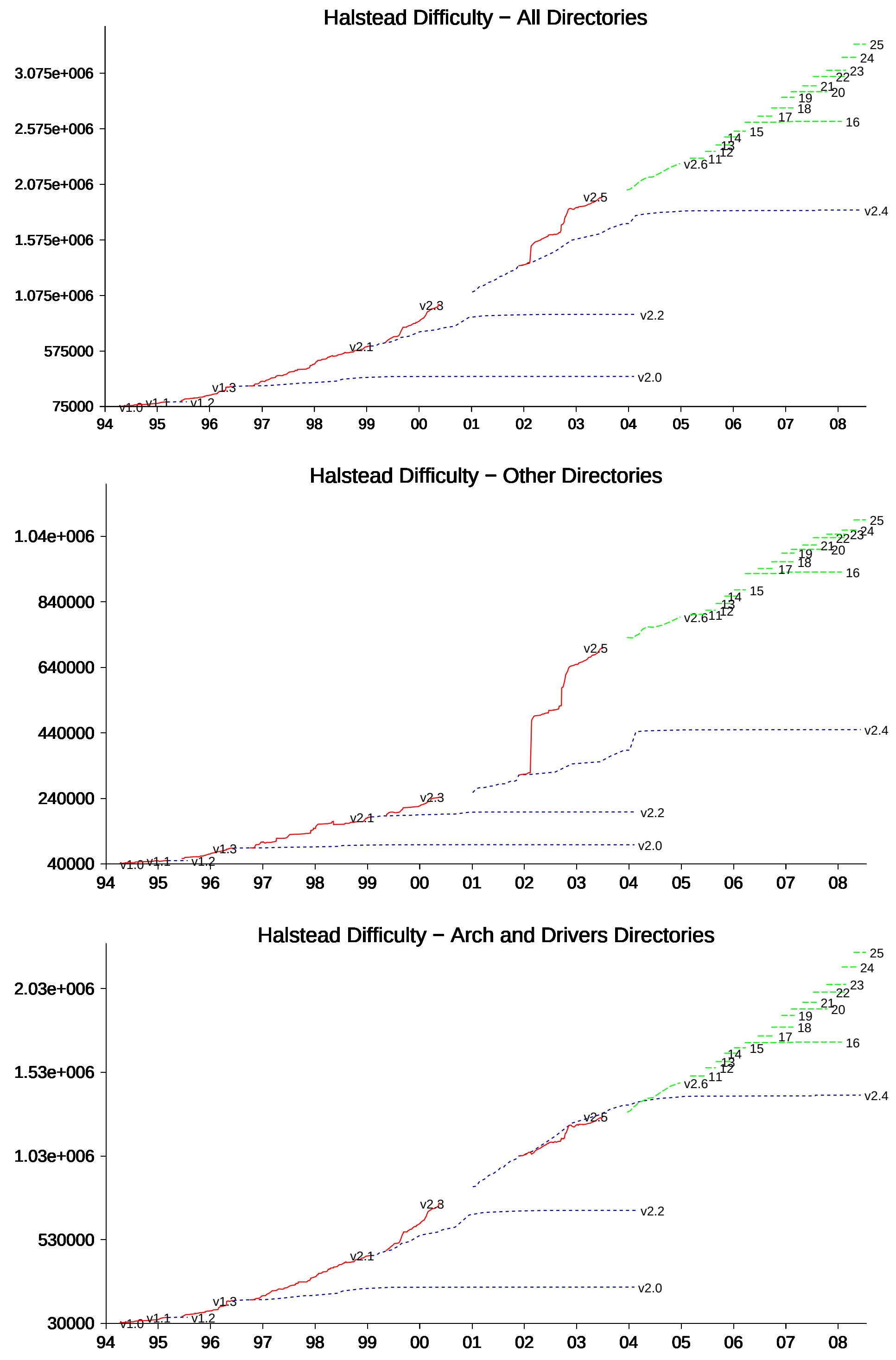
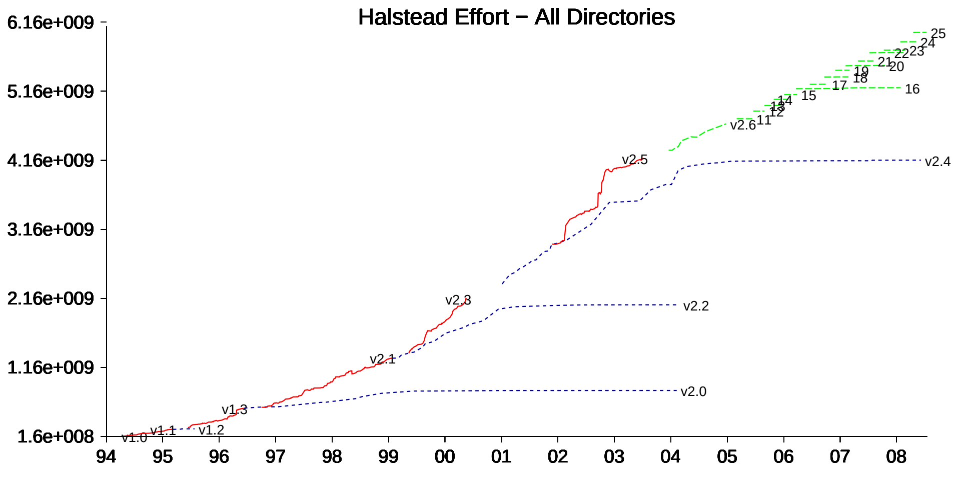
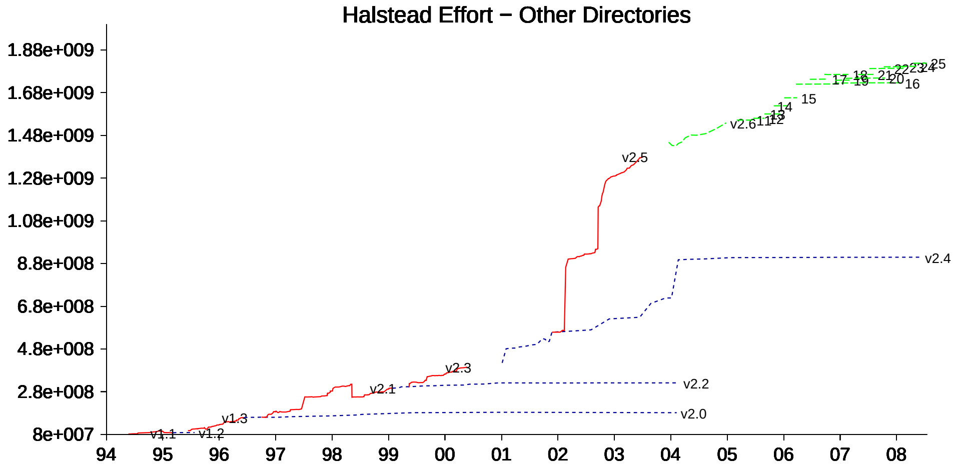


Figure 20:
Halstead Difficulty for each Kernel

and the stable periods. These results resemble those of Thomas [33] who measured Halstead's Volume and Halstead's Effort (using the CASE tool) and found a similar trend — an overall increasing trend, with much more variability between the different series than within-series.





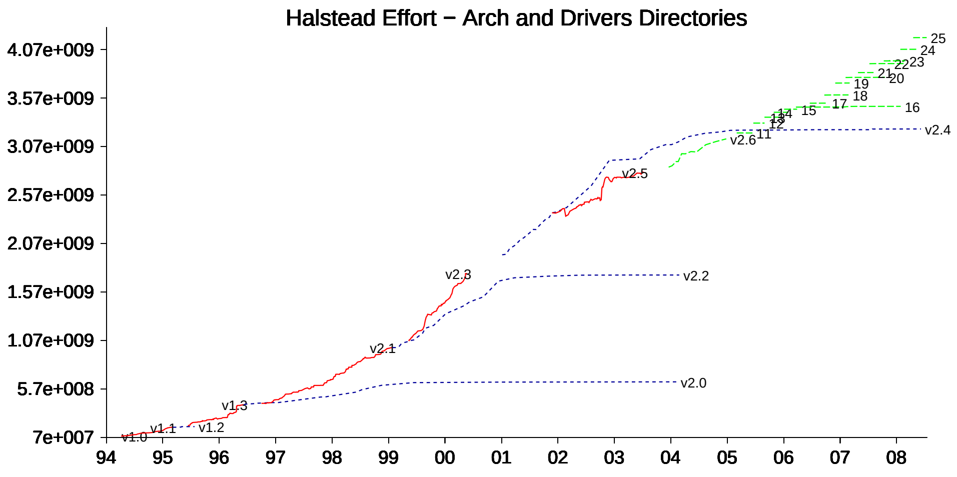


Figure 21:
Halstead Effort for each Kernel

He also noticed that the values of the volume and effort are much closer between 2.0.* and 2.2.* than to the 2.4.* series. His explanation to this is the number of extern inline functions, which was reduced from around 60% in 2.0.* 2.2.* series to 40% in 2.4.*. Since Thomas uses a

Category	Avg. HV	Avg. HD	Avg. HE
All 2.4.25	627.15	19.37	39224.57
xfs 2.4.25	1063.94	30.65	107102.2
other 2.4.25	590.18	18.41	33478.82
All 2.4.24	592.43	18.45	34252.18

Table 2: *Halstead Metrics for core Kernel directories in v2.4.25, for all functions, only xfs functions, and all other functions. For comparison, value for all functions of v2.4.24 are included.*

CASE tool that performs preprocessing of the code before calculating the metrics, the effect of the code in the extern inline functions appears in the functions from which they are called rather than the extern inline function itself and thus affects large parts of the code. Nevertheless, we also see that 2.0.* and 2.2.* are closer to each other than to 2.4.*.

Do these results imply that the software is becoming more and more complex? Again, looking only at these figures we can not tell, since we can not eliminate the growth in code and in files just looking at them. Furthermore, the shape of the growth resembles that of the growth of files (and of LOC). This is not surprising, since we expect Halstead metrics to grow with the growth in the LOC, by their definitions.

When looking at the values per function (all three metrics for all directories in Fig. 22) we see that generally, the average values per function are decreasing. Again, we see the same phenomena that for the production versions the changes are milder and they are much more stable than those of the development versions. We also see that while the general trend is a decrease, versions 1.2, 2.0, 2.2 and 2.4 are actually increasing and then stable. Notice that each one of them behaves that way, but when compared to each other, the general trend is downward.

When we look further into this data by dividing into the different directory groups (HV in Fig. 23, HD in Fig. 24 and HE in Fig. 25), we see that in the *arch* and *drivers* directories, the values for 2.0 and 2.2 are increasing (whereas they are constant in the core kernel group) and pretty stable for 2.4 (but increasing with a very sharp jump in the core kernel). Notice that generally, although each version is stable/not decreasing, between the versions the trend is a decreasing one. However, for 2.4 in the core kernel we see a different pattern — for all Halstead metrics (HV, HD and HV) the trend of version 2.4 is increasing. Specifically, the large “jump” happens in 2.4.25 (which was released with 2.6.3, and has close Halstead metrics values). Notice also that there are two jumps in 2.5 too.

When investigating the changes and differences between 2.4.24 and 24.25 we found that a new file system *xfs* was added to 2.4.25 (this file system was introduced first at 2.5.36 and was a part of the 2.6 initial production version). A deeper investigation of the *xfs* files revealed the following: at total 170 new source files were added to 2.4.25 (and just a bit below that to 2.5.36). These files include a total of 1786 functions. We calculated the average Halstead metrics values three times: only for these new functions, for all other function in the core kernel, and for the two function groups altogether. The results are shown in Table 2, including a comparison to the values of the previous version. It shows that the *xfs* functions raise the average of all Halstead Metrics and they are the reason for the big jump. Moreover, when comparing to previous versions we see that without this change, there would be a decrease for all average Halstead metrics values.

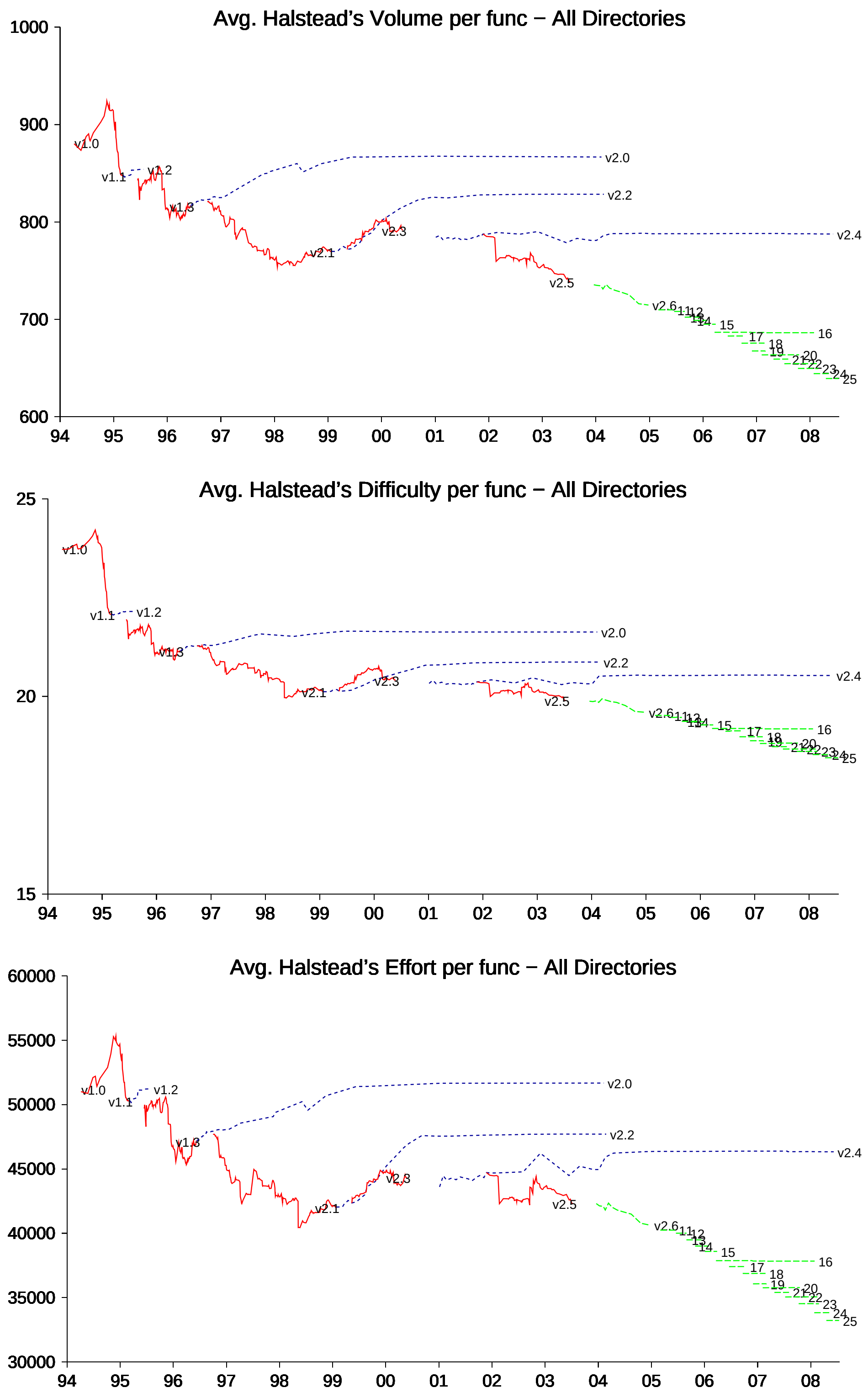
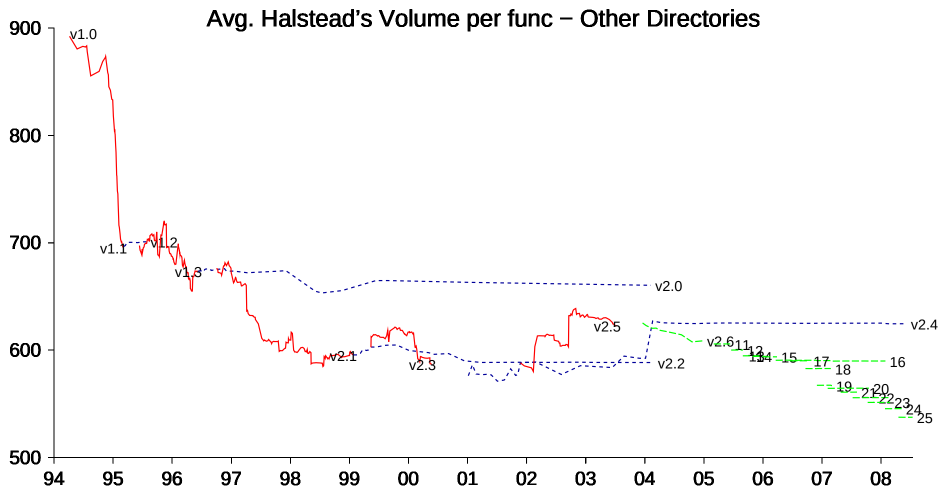


Figure 22:
Average Halstead Metrics per Function, for all directories

As mentioned above, these results can also explain the second jump in v2.5 (the release of v2.5.36).



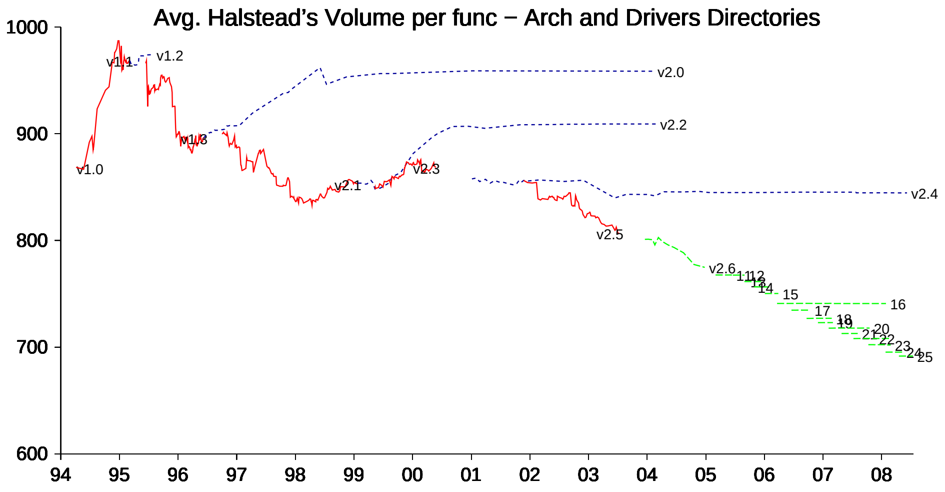


Figure 23:
Average Halstead Volume per Function, for different directories groups

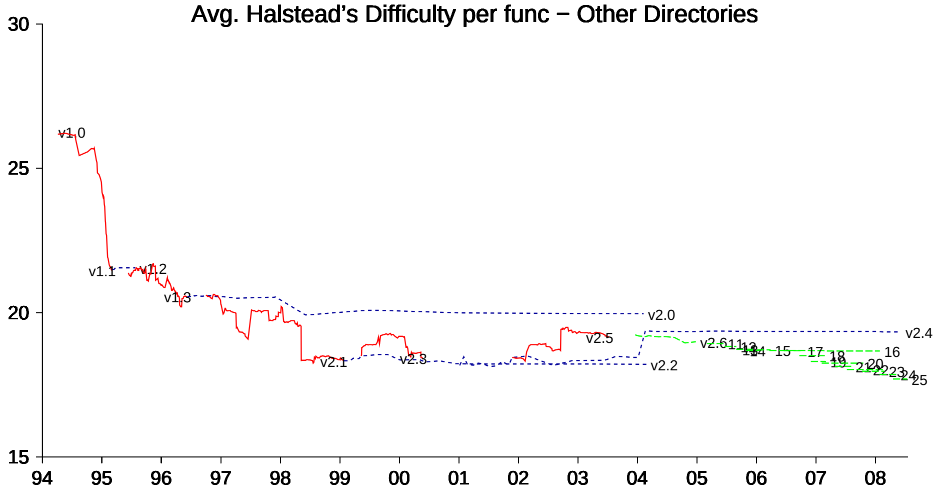
The fact that we see that the average Halstead metrics per function are decreasing, fits the picture we have seen earlier: there are less LOC on average per function, thus metrics correlated (or related) to LOC will be affected in the same way; we expect average MCC to decrease over time, and also Halstead metrics.

5.2.6 Oman's Maintainability Index

As described above, Oman's Maintainability Index (MI) is a linear combination of different metrics. As a reminder, the MI is:

$$MI = 171 - 5.2\ln(AvgHV) - 0.23AvgMCC - 16.2\ln(AvgLOC) + 50\sin(\sqrt{2.46perCM})$$

We have seen that per function ("module" in MI), the average values of Halstead Volume, McCabe's Cyclomatic Complexity and Lines of Code are decreasing. We have also seen that the perCM value (which, as identified by Thomas [33], is the ratio of comments in the file) is generally decreasing with time.



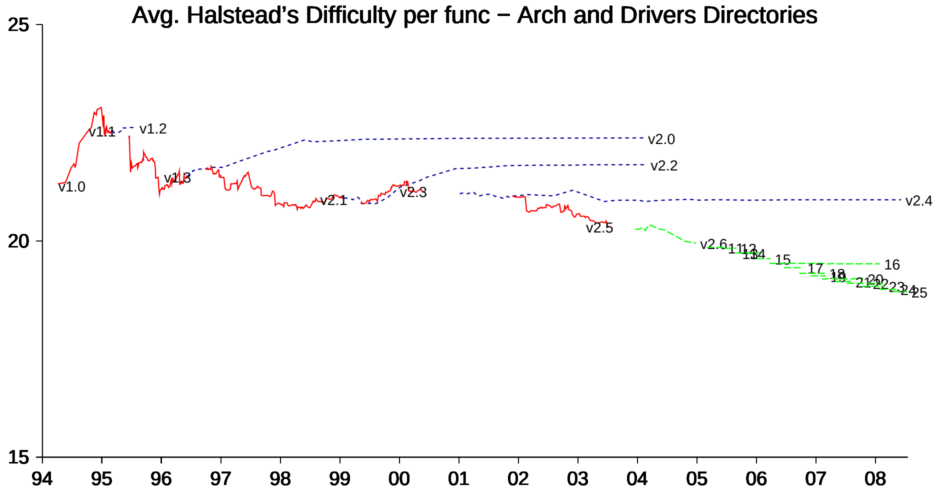
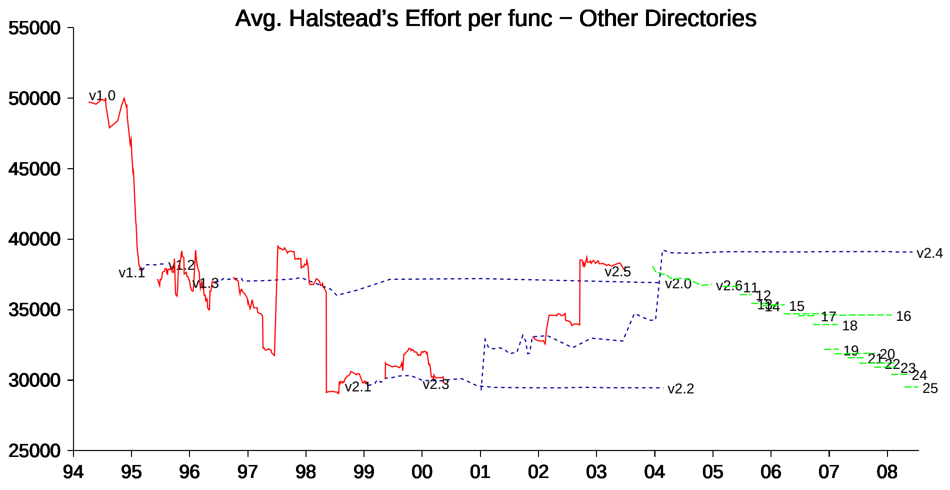


Figure 24:
Average Halstead Difficulty per Function, for different directories groups

Since $AvgHV, AvgMCC$ and $AvgLOC$ are all decreasing, and they are negative here, they will cause the MI value to increase with time. The only positive component is the one of perCM. Since $\sqrt{2.46perCM}$ ranges from 0 to approximately $\frac{\pi}{2}$, the additional contribution of this aggregate is between 0 and 50. As we saw, perCM is decreasing with time, thus its contribution to the MI will be in the opposite direction as the rest, i.e. the addition to MI will be smaller with time. However, since the changes in perCM and very small, and much smaller than the changes in the other components, this effect will be negligible. We expect MI to increase with time (indicating better maintainability), and so it does, as can be seen in Fig. 26.

Thomas [33] also found that the MI was increasing in all series, again with much less variability within series than between series. He was first surprised by that finding, since all other metrics increased, he expected the maintainability to decreasing, as the MI value runs in opposite direction than the other metrics. However, since the MI uses averages, which, as shown here, decrease with time, the MI increased over time (and not decreased, as first expected by Thomas).



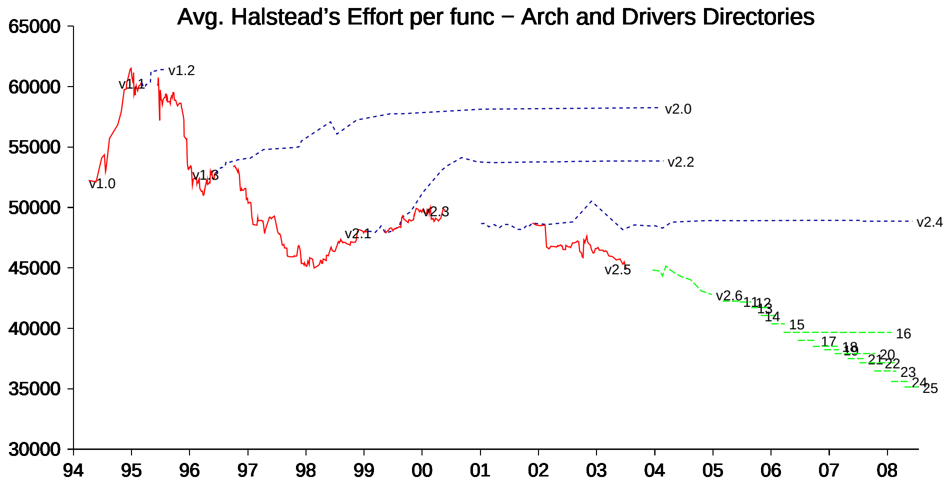


Figure 25:
Average Halstead Effort per Function, for different directories groups

Another interesting point is that since the quality values for the core kernel directories versus those of the *arch* and *drivers* were always better (i.e. less LOC, lower values for Halstead Volume, McCabe Cyclomatic Complexity and slightly more commented), we also see that the MI for these directories is higher — meaning it is "better quality".

5.2.7 Correlation Between Metrics

The literature is full of arguments about software metrics and their relative merits. In particular, it has been claimed that MCC is correlated with LOC and thus does not provide new information, and MI is by definition related to both MCC and Halstead's metrics.

From our observations of these metrics as they change in different versions of the Linux kernel, we can say that they are indeed often correlated to each other. However, there are specific instances where the metrics behave differently.

When comparing the LOC per function and the MCC per function for all directories, (using Fig. 10 and Fig. 14) for example, we see some differences: the most noticeable is in the behavior of release 2.2, especially in mid 2000. In LOC we see a huge jump, which causes the value of

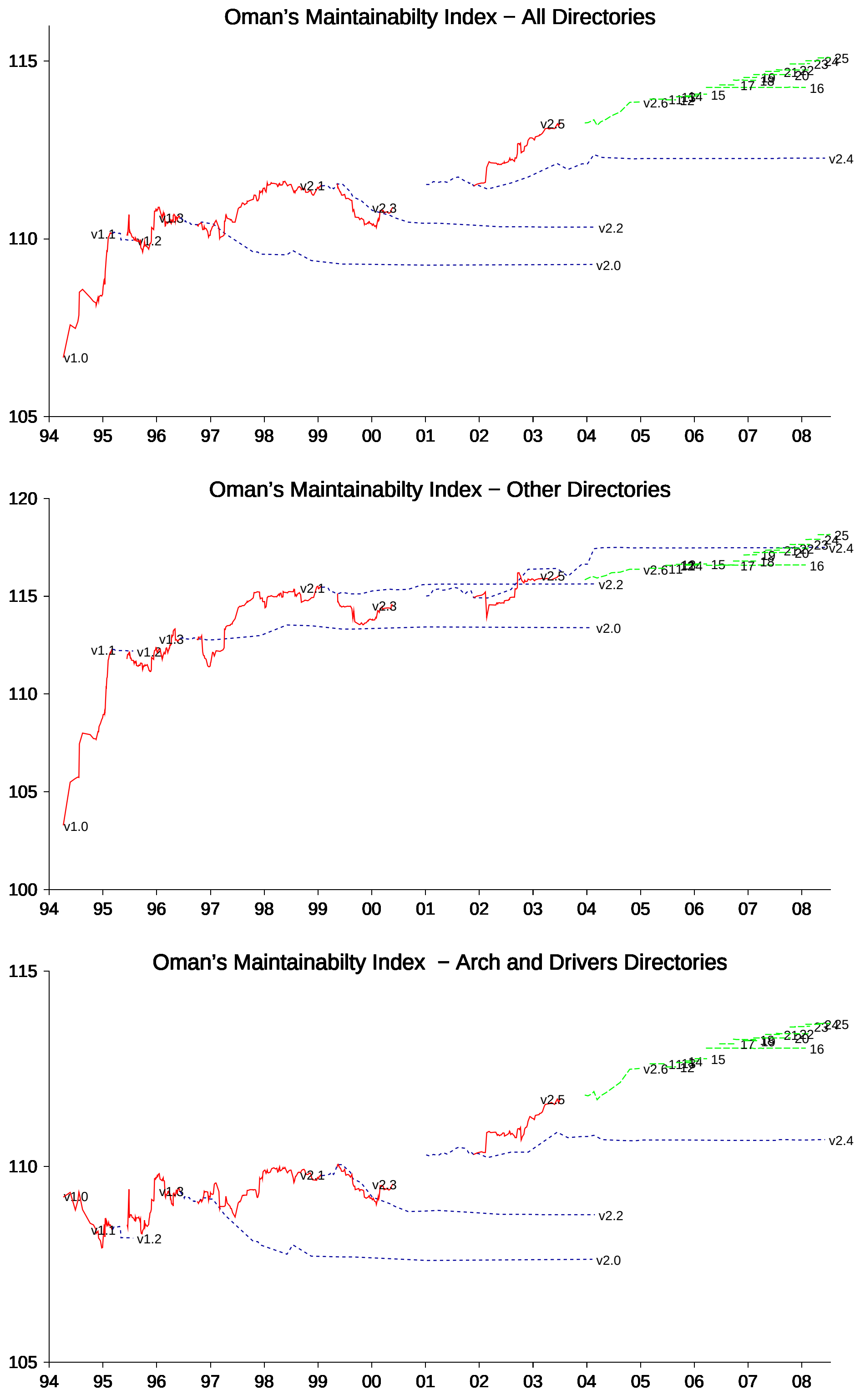


Figure 26:
Oman's Maintainability Index for each Kernel, divided by directories

LOC for version 2.2 to be even higher of the value for version 2.0, although the general trend is decreasing. However, when we look at MCC values, we don't see anything special about the behavior of version 2.2 it is smoothly changing as the trend of the general graph (same for the jump in LOC for version 2.4 in mid 2003). Another example in this graph is in version 1.3 which has a large increase at the end of those releases in matters of LOC, and a general decrease (with a very tiny increase) in MCC.

5.3 Maintenance Activities

We will now analyze the results with respect to maintenance activities, in order to try and characterize the maintenance process in Linux.

5.3.1 Development and Production Versions

As noted above, we expect development versions of the Linux kernel to reflect perfective maintenance, whereas production versions reflect corrective maintenance. This is based on the definitions of these two types of branches of the code, which are maintained in parallel to each other. However, it seems that in practice this distinction was not always followed.

The most extreme example of "mixing" the roles of the versions occurs at the beginning of v2.4. The last version of v2.3 was released on May 24th, 2000. The first version of v2.4 was only released on January 5th, 2001. and the first version of v2.5 only on November 23rd, 2001. Thus there is a gap of some *18 months* between successive development versions. However, it seems that the initial part of v2.4 served for development (or at least reflected development activity that was being done without officially being released in a development version), as all our graphs indicate that v2.5 branched out of v2.4. The remaining gap between v2.3 and v2.4 seems to have been filled at least partially by v2.2. Specifically, v2.2 exhibits strong growth in this period, especially in the *arch* and *drivers* directories, which matches the difference in size between the end of v2.3 and the beginning of v2.4.

5.3.2 Intervals Between Releases

First, it is interesting to see the Linux releases pattern (until version 2.6), as reflected in the graphs above: it seems that after a number of releases of a new production version, a new development version is released, which originates in that production version. After the development version is done (i.e. no more release in that major version), a new production version is released. During the development and releases of the new production version, there are still releases of the previous production version, probably in order to support users who didn't upgrade to the new production version (2.0 had continuous releases until the end of version 2.2, and version 2.4 is still updated, with new releases of version 2.6).

We also examined the intervals of time between successive releases (in days) within the same major version. Fig. 27 displays the raw data in the upper graphs and the statistics - medians, 5th, 25th, 75th and 95th percentiles of the intervals for each version, in the lower graph. The "N" under each of the bars in the figure indicates the number of releases the statistics were calculated for, notice that this number is the number of releases in the specific version minus one (since the first release in each version is counted as "time 0"). Looking at the upper graph, we see an interesting picture — the development versions have very low values, and so do the 2.6 versions, while the production versions have much higher values. The timeline adds

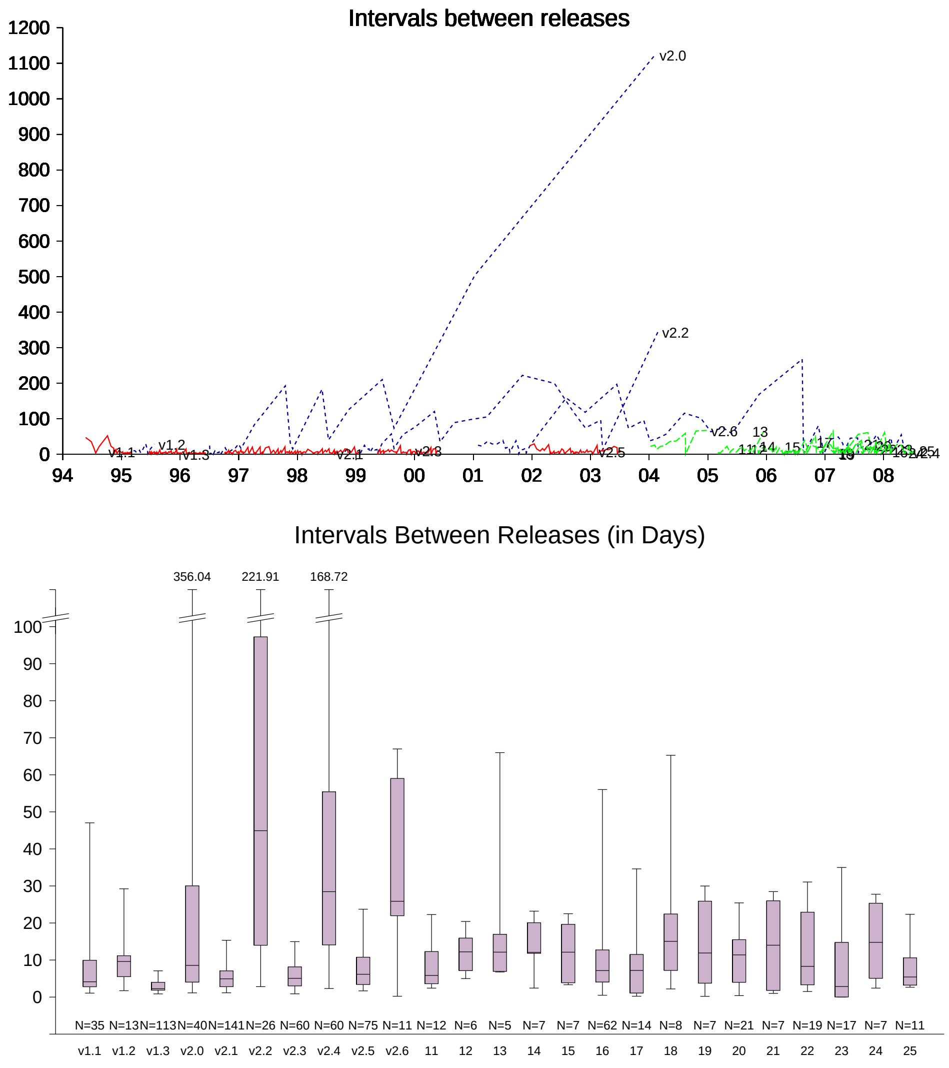


Figure 27:
Data and Statistics of Intervals between Releases, within Major Version, in days (5^{th}, 25^{th}, median, 75^{th} and 95^{th} percentile

another value — we notice that an ending of development version is almost simultaneous with the beginning of a production version (i.e. at each end of a red line, a blue line starts). However, we do see a significant gap between 2.3 and 2.4 and between 2.5 and 2.6. According to 2.4 logs and different Linux forums, the first was a result of the version "not being ready" for release (according to the agenda of Linus Torvalds, to release only when versions are stable) and due to complications with development and testing and many new requirements for the 2.4 production versions. The second gap is a result of the structural change in the Linux release scheme.

Looking at the first 10 versions (v1.1 to initial part of v2.6) in the lower figure, we immediately notice the following: the bodies of the distribution of the development versions are usually lower than those of the production versions. Moreover, the variance and the high values are usually much higher in production versions than in development versions (notice that versions 2.0, 2.2, and 2.4, the 95th percentile values exceed the top of the graph and are their true values appear on the labels above their box). In other words, we see that while development versions are released quite often (all medians and 75th percentiles are lower than 10 days) the production version are released much less frequently. This happens more in the more recent versions (v2.2 and on), where the median values are closer to a month and the 75th percentile is two month and more.

When we look at the minor versions of 2.6 (2.6.11 and on) we see a combination of the above. On one hand the distributions are generally much lower than in most production versions, but on the other hand they are higher than in the development versions. This indicates the success of the 2.6 scheme, which combines new functionality with faster releases: each of the 3-digit versions includes additional functionality, while the releases within them are "4^{th} digit releases", i.e. bug fixed and security patches.

This observation leads to the next point of view on release dates: looking at intervals between major "production releases" (meaning 1.2, 2.0, 2.2, 2.4, 2.6, 2.6.1, 2.6.2, ..., 2.6.25).

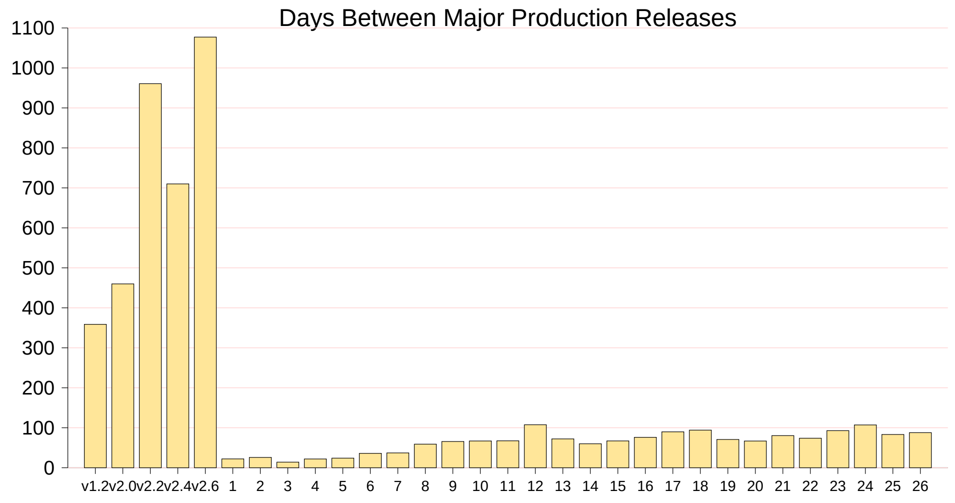


Figure 28:
Intervals Between Production Releases, in days

Here (Fig. 28) we see that while the wait for the first production version took almost a year, and the next major versions were waited for much longer than that (a year, two years, and the "record" of almost 3 years for 2.6), the 2.6 minor versions are released in a quite measured ratio of around 2.5 month. One should not forget that the long periods of time between the major production versions were not times without releases; there were many releases during that time, but their main purpose was bug fixes and security patches and (usually) not adding more functionality.

To conclude, releases of development versions are very frequent, and are on a daily to weekly basis. Production versions releases, however, are less frequent and usually on a weekly to monthly basis. The wait between major production versions was a year and more, and was not decided in advance, but dependent on progress and "readiness" of the versions. With version 2.6 (which symbolizes the structural change in the kernel releases), we find a much faster and more stable release rate. However, this too seems to be growing slowly, from about one month for the first 7 versions to 2.5–3 months now.

5.3.3 Corrective Maintenance

As indicated previously, we will analyze corrective maintenance as reflected in successive versions of production kernels, since we assume that successive versions in the production kernel are usually corrective, due to the structure of the releases in Linux.

As we have seen in the results, for each of the different metrics calculated — number of files, number of functions, LOC, MCC, Halstead's metrics and Oman maintainability index, usually the values of these metrics for the production versions are essentially constant. This is reflected in v1.2, v2.0, v2.2, v2.4 and in each of the minor versions of v2.6. In v2.6, the lines of the minor versions, which are created by the releases of corrective maintenance efforts (i.e. the dots created by 4^{th} digit changes), are parallel and rarely change within the same minor version.

However, the metric values in production kernels do change in two cases. One is the large "jumps" seen in versions 2.2 and 2.4. These are explained by changes in functionality, where significant new functionality was propagated into a production version, but without calling this a new major version. The main examples are the improved USB support and additional drivers that were added to 2.2.18 kernel, and the introduction of the *xfs* file system to the 2.4.25 kernel. Thus the assumption that production versions represent only corrective maintenance is not always correct.

The other metric change observed in production versions is that the average size metrics, such as LOC per file and LOC per function, tend to grow initially and only then become constant. The average complexity metrics such as McCabe and Halstead also tend to grow initially. This may indicate that corrective maintenance tends to add code and complexity to the existing structure, without restructuring and refactoring. However, it may also be just another case of creeping functionality updates. Resolving this will require a detailed analysis of the actual modifications done to the initial part of production versions.

One can thus tentatively conclude that corrective maintenance does not have a strong effect on the different kinds of metrics. If at all, the changes are mild and usually the values of the metrics are constant.

We can also state that corrective maintenance is much less frequent that others, as the releases of production versions are much less frequent and seem more planned.

5.3.4 Perfective Maintenance

As indicated previously, we will analyze perfective maintenance as reflected in successive versions of development kernels, since according to the structure of the Linux versions, the main motivation for successive versions in the development kernels is to improve and add functionality.

As indicated above, the different metrics for successive versions in development kernels are usually more volatile and more changing than for production kernels. The detailed behavior observed depends on both the metric being studied, and on whether one considers the whole kernel or average values per file or per function.

On one hand, it seems like less work is being done to "maintain" the code in development kernels; for example, the ratio of comments is lower for development versions than for production versions (i.e. they can be considered less readable). On the other hand, when comparing the other metrics we see a different picture: looking at either MCC or the Halstead metrics per function, it seems that the values for production kernels are higher. As a consequence, the MI is lower for the production kernels (i.e. less maintainable). In other words, we find here that development kernels are less complex and more maintainable than the production kernels. Of course, if we compare at the general kernel level, the picture will seem the reversed, since there are more files and more LOC in the development kernel.

It should be noted that the above pertains not only to a single snapshot of production and development kernels at a given time, but also to how the metrics change with time. Specifically, for production kernels the metrics initially tend to worsen a bit and then they stay constant; for development kernels, on the other hand, they tend to improve. Two interpretations of these dynamics are possible: either the code in development versions is indeed being improved, or else there are simply many small files and functions are being added at a higher rate than the complexity grows, so the averages shift in the desirable direction.

Initially, this result is surprising, since we would expect the production kernels — which are less frequent and have to be considered and tested much more before each release — to be more maintainable and less complex. On the other hand, the extensive testing and verifications might lead to fixes that cause additional complexity. In other words, files and functions that are added to the production kernels after their initial release may be more complex since they solve critical bugs and security issues.

Another explanation that is related to perfective maintenance is the following: it is easy to notice on the graphs (for example Fig. 14 and Fig. 22) that for most metrics each production version and the next first development version (i.e. production version number $X.0$ and development version number $X + 1.0$) have the same initial metric value. However, while the production version metric value tends to be constant (no major changes were preformed), there is work in the development branch in order to improve the code and to perform perfective maintenance activities, which include not only better performance of the code, but also improving the code itself. Doing this, the metric value is improving in the successive development versions, while for production versions where critical bug fixes are released, there is less importance for activities of perfecting the code.

We have seen specific examples (in the analysis of high MCC value functions) for perfective maintenance in development version, which improved the code complexity and structure, while the production versions remained unchanged.

5.3.5 Adaptive Maintenance

As indicated previously, we will analyze adaptive maintenance as reflected in the *arch* and *drivers* directories (especially in the development kernels), since they best reflect the adaptation to changes in the environment — the addition of new architectures and new devices that need to be supported.

As we have seen above, the *arch* and *drivers* directories usually have the same trends as the rest of the kernel, although many times with higher magnitude (for example, for all the kernel the number of LOC is growing, but the numbers for *arch* and *drivers* directories are higher than the core kernel). Another issue we have seen above is that development versions usually have a high volatility between successive versions.

We have also noticed that although the core kernel and the *arch* and *drivers* directories do not always have the same "amount" of change (for example the LOC and number of files changes in v2.5 are more noticeable in the core kernel, whereas in v2.2 they are more noticeable in the *arch* and *drivers* directories), there is an effect of *arch* and *drivers* on the core kernel.

This effect is due to the *include* directory which is usually updated with new header files once a major change or update in *arch* and *drivers* takes place.

We also saw that some metrics change in different directions when comparing *arch* and *drivers* to the core kernel. One case is the ratio of comments in files: as we saw above, this value grows with time for the core kernel, but decreases in *arch* and *drivers*. Not only that, but to begin with, the ratio of comments for these directories is lower. This might indicate the adding of more code than comments, which might indicate a degradation of maintainability, due to declining readability of the code.

The other, and perhaps more significant difference (although it can be claimed to follow from the first), is that for all the different complexity and maintainability metrics (MCC, Halstead Metrics, and Oman's MI) we found that theses metrics for *arch* and *drivers* are worse than for the rest of the kernel (higher for MCC and Halstead Metrics and lower for MI). In other words, the complexity and maintainability of these directories is worse than the core kernel. Saying this, we must not forget that all of these metrics are correlated with and related to LOC, and as the *arch* and *drivers* are larger in LOC and in number of files and functions, we would expect them to be "worse". However, since we have seen that in many of these cases the "worseness" holds also when we compare the metrics per file and per function, we can conclude that the complexity and maintainability of *arch* and *drivers* isn't as good as that of the core kernel. If so, we can say that adaptive maintenance, as reflected by the *arch* and *drivers* directories, leads to lower maintainability and higher complexity values, and is more volatile than other types of maintenance.

5.3.6 Preventative maintenance

As indicated previously, we will analyze preventative maintenance as reflected in isolated events in which many files are partitioned, removed, or moved.

We examined the code changes between successive development versions (we decided also to add version 2.4 to our analysis, since as we have seen above, it served, at least in the first years, for development); we tried to quantify them by looking at the number of files which were changed (i.e. added, deleted, grew, or shrunk) in Fig. 29 and the same for directories in Fig. 30. The green solid lines are the values for the development versions and the red dashed lines are those of version 2.4. The "deleted" group are files/directories that were removed, and "added" is the difference between the versions plus the number of files deleted (this way we count all the files that were added to this version). The names "grew" and "shrunk" refer to the size of the files/directories (and not the number of files in directories, for example).

In both figures, on the left column we see the actual number of files/directories per each metric, and on the right column we see the percentage. The comparison for each is continuous, i.e. we compared minor versions within the same major version and also did a cross-version comparison for the first release of a major version and the release of the previous major version that was the base of this release. The first release of version 1.1 was compared to the only release of 1.0. Version 1.3.0 was compared to version 1.2.10, version 2.1.0 was compared to 2.0.21. Version 2.3.0 was compared to version 2.2.8 and version 2.5.0 was compared to 2.4.15. The last two comparisons revealed that the initial versions of 2.3 and 2.5 are completely identical to the production versions they origin in (i.e. all values in the graphs for the initial versions of 2.3 and 2.5 are zero). This complies with our findings above and our observations regarding the release patterns. As we have seen, these development versions have exactly the

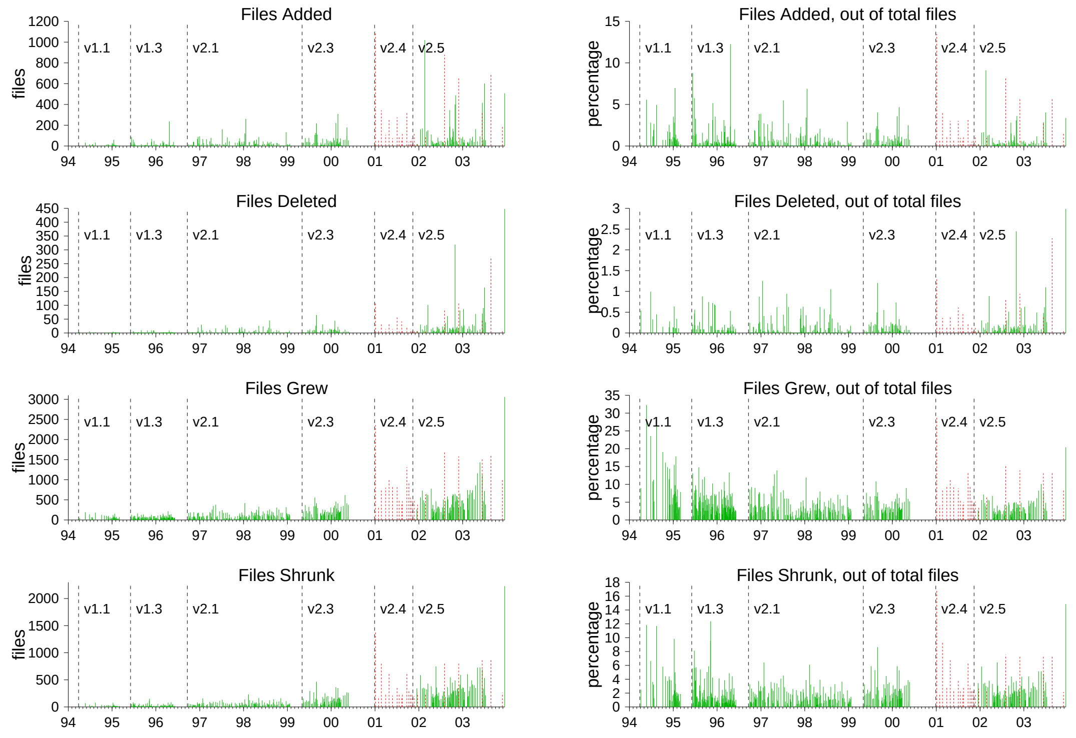


Figure 29:
Files added, deleted, grew or shrunk among development versions

same values as the production versions for the different metrics. Versions 1.3 and 2.1 were not identical to the production versions they originated from, which were slightly modified. Version 2.4.0 was compared with the last 2.3 version (2.3.99-pre9), as it emerged from that version. The last comparison in each graph is between the last version in 2.5 (2.5.75) and the first in 2.6 (2.6.0). The spaces between the bars are times with no development versions.

When looking at the two figures, we see immediately (and not surprisingly) that they are similar. Looking at the left column, especially for “grew” and “shrunk” data, we see that the *number* of files/directories changed grows with time. This is also true for “deleted” and “added”.

When looking at the right column of the figures, the picture changes slightly; we see that for the most case (except for a few “spikes” —— for example the one which indicates the change between 2.3 and 2.4, which is expected, given the information above regarding those releases), the percentage of changes in the files and in the directories is constant or maybe even slightly decreasing. We see a downwards trend especially in the percentage of directories growing, which might be explained by the stabilization of the versions and releases. With time, we expect the directories to have fewer changes in them, at least in size and LOC. The trend of changes indicates that the rate of change is pretty constant between consecutive development versions.

For version 2.4 we see that the growth in number of files and directories matches the changes that we have seen above, both in growth of files and also in number of functions and LOC. We also see that many files were edited at this time. In 2002 and in 2003 there were 3 releases each. In those there are relatively large changes in version 2.4, especially within the

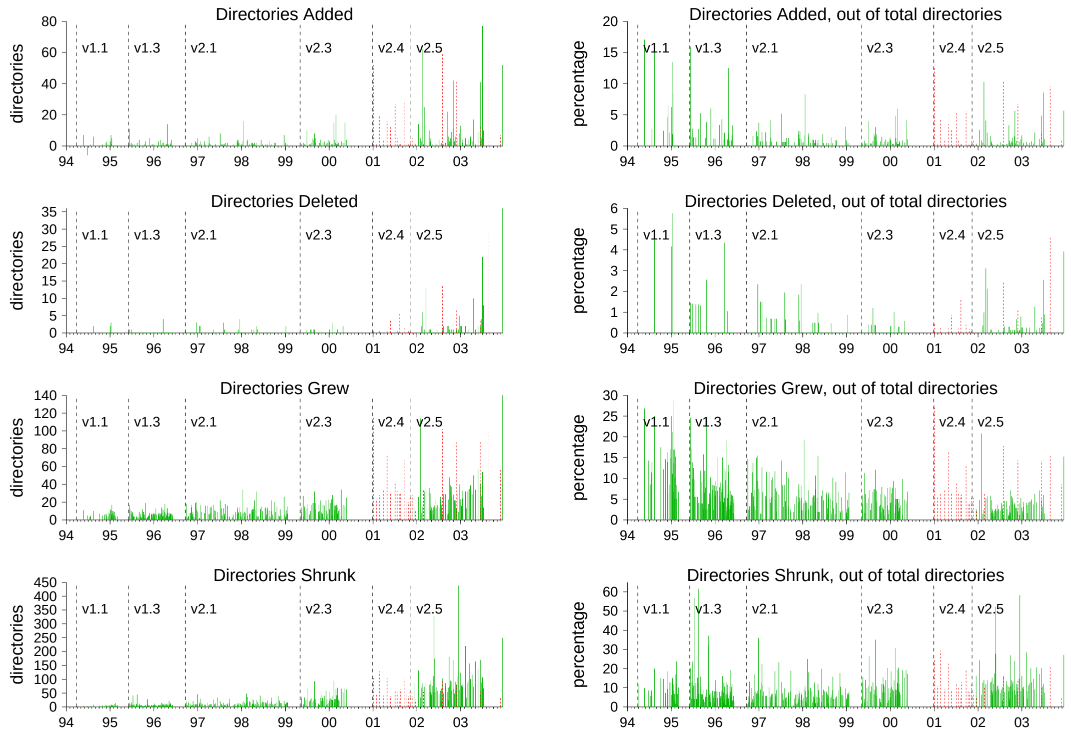


Figure 30:
Directories added, deleted, grew or shrunk among development versions

files themselves (much more than deletions/additions). They might be related to the changes in 2.5 at that time. Also, since versions are so rare at this period, we expect that each change would be larger than usual.

There are a few extreme values on the graphs, which we ought to explain:

In version 2.5 we see a large amount of files deleted, around November of 2002. We do not see many files added, or a noticeable causal effect on directories at that time. The reason for this is that in that version (2.5.45) there was a replacement of the Linux configuration system. Therefore the old configuration files (*Config.help* and *Config.in*) were deleted in many directories and were replaced by a single *Kconfig* file.

In version 2.5 we see a relatively large amount of directories deleted, around March of 2002, at the same time we see also a large amount of directories added. This is due to changes in the structure in the *arch* and *drivers* directories at that time and also the changes we have seen above in version 2.5.5 (the addition of "sound" subdirectory to the kernel). We also see growth in the size of the directories corresponding with these changes.

In version 2.5 we see a relatively large amount of directories shrunk, around mid 2002, at the same time we see also a large amount of files shrunk, the corresponds to that. In the matching version there was a decrease in size of many files, which belong to many different directories. Another huge amount of directories shrunk is at about the end of 2002. Again, there is a corresponding increase in the number of files shrunk at that time, although not very significant. This could be explained by the changes being spread among a very large amount of different directories. By the way, 75% of the shrinking directories are in *arch* and *drivers* directories.

The initial releases of the two production versions, 2.4 and 2.6, have large changes in comparison to the development versions they originated in. This is not surprising. First, since they are production version, we expect them to be different than development versions, which are less tested and might contain more functionality. Second, both of these versions were released after a long wait of around 6 month after the development version. At this time, a lot of work was done and we do not expect the new version to stay very similar to the development ones, especially when both versions had many new requirements and functionalities in them.

As we have seen above, in cases where there are changes in the structure of the code, there are changes in the metrics value which reflect them. This is more easy to see in production kernels where such changes are less frequent. Inferring this observation on development kernels, we can assume that the high volatility identified in all the different metrics has to do with the high volume of change in development kernels. We could not find a clear relation between the large changes spotted in the graphs and preventative maintenance.

Another observation is that v1.1 is special — there seem to be especially significant improvement in the code in that version: the fraction of comments grew, and all the complexity metrics (McCabe and all of Halstead's metrics) decreased considerably. One may therefore conjecture that this version saw much preventative maintenance, being the first version after $2\frac{1}{2}$ years of development from the initial announcement in August 1991 to the first release in March 1994. However, verifying this conjecture will require detailed scrutiny of the code to assess the reasons for the improvement in the metric values.

5.4 Lehman's Laws

We will now analyze the results with respect to Lehman's Laws, in order to try and see whether they are supported by our Linux data. Most of our analysis of Lehman's laws stems from previous literature regarding verification of these laws [14, 13].

Table 3 summarizes Linux's support of Lehman's laws, according to our analysis below.

5.4.1 Continuing Change

According to this law, a program that is used must be continually adapted else it becomes progressively less satisfactory. The intention here was adaptation to the environment. Usually, it is hard to distinguish between adaptation to environment and general growth (as reflected in the Contentious Growth law), however, due to the unique structure of Linux, we have an ability to separate them: we can examine the continuing Change law with respect to the *arch* and *drivers* directories which have to do with the changes in the environment.

As we have seen before, the values of LOC, files, and functions all grow in these directories. This growth can be identified as changes and adaptation to the environment. In fact, when looking at change logs, Linux forums, and explanations about the content of different versions, one of the things that returns in each versions is new drivers and adaptation to new architectures. Moreover, as we have seen above, the *arch* and *drivers* directories hold over 60% of the Linux LOC at all times. We can conclude that we see support in Linux for this law.

No	Name	Support	comment
1	Continuing Change	$\surd$	Growth of *arch* and *drivers* directories was seen throughout the results and reflects continuing adaptation.
2	Increasing Complexity unless Prevented	$\surd\!\!\!\times$	No final conclusion on Linux complexity. However, there are indications that average complexity is actually decreasing, which may either support or contradict the law.
3	Self Regulation	?	Possibly supported by steady overall growth rates, but more information regarding feedback mechanism is required in order to establish this.
4	Conservation of Organizational Stability (Invariant Work Rate)	$\surd$	Rate of releases has been relatively stable since 1997. The 2.6 method of timed releases also creates an invariant amount of releases per time unit.
5	Conservation of Familiarity	$\surd\!\!\!\times$	Long-lived production versions reflect this law — successive minor releases have little functionality changes. But there are big changes between successive production versions.
6	Continuing Growth	$\surd$	Growth in functionality is obvious. Growth of LOC, files and function was seen throughout the results. Here, the growth is super-linear up to version 2.5 and then it is closer to linear.
7	Declining Quality	?	Anecdotal contradiction (increasing usefullness) and anecdotal support (due to adaptation to the operational environment)
8	Feedback System	?	Anecdotal support

Table 3:
Support for Lehman's Laws in Linux

5.4.2 Increasing Complexity

According to this law, as a program is evolved its complexity increases unless work is done to maintain or reduce it. This is very hard to prove or disprove, as it allows both trends: if complexity increases it fits the law, and if it is reduced then maybe work was done to reduce it.

As we have seen in Linux, the complexity of the system is increasing at the full kernel level (as LOC, files, and functions are increasing) and deceasing at the single function level (and hence MI is improving with time). We did see, when we looked at the CDF of MCC values per function in the different versions, that this distribution is improving somewhat with time, since more lower complexity functions are used. In particular, when comparing v1.1 to recent versions there is much improvement over time, so maybe we can say that there was work to reduce the complexity of the kernel. However, it is possible that a large part of this apparent improvement is due to the increasing number of functions. We have also seen above that it can be implied that perfective maintenance is decreasing the complexity.

The bottom line is that we were not able to conclude whether the software is really becoming better. On the other hand, either conclusion may be said to support or to refute the law, bacause the law allows either increasing complexity or reduced complexity (depending on whether steps to reduce complexity are taken).

5.4.3 Self Regulation

According to this law, the program evolution process is self regulating with close to normal distribution of measures of product and process attributes. In [13] they identified ripples in the graph of size (measured in modules) as a function of release serial number as indicating this phenomenon. The ripple is repeated pattern of deviations from the averate growth rate, alternating between periods of faster growth and correction of the trend by periods of slower growth. They claim that this ripple indicates the existance of a feedback system which checks and balances the system in order to drive it to its goals.

The ripples in the growth of LOC as well as the number of files and the number of functions might suggest self regulation, as does the generally smooth growth of these metrics. However, in order to support the law, identification of the underlying mechanism is required (as in [14]).

5.4.4 Conservation of Organizational Stability (Invariant Work Rate)

According to this law, the average effective global activity rate on an evolving system is invariant over the product life time. This measurement is problematic, since we are trying to look at “work” on the project. As mentioned before, data about man hours or number of developers is hard to get in closed-source, and much harder in open-source projects. We will try to examine this law looking at code changes in files and directories and also at the amount of versions per time unit.

We have seen before, that over time, on average the rate of change of code for files and directories is constant. This evidence supports this law.

In Fig. 31 we can see the number of releases per month, for all development versions (v1.1, v1.3, v2.1, v2.3, v2.5)in purple and for the beginning of version 2.4 (which has development version characteristics) in lavender. We only examine the development versions since production versions are more sparse (Linus’s method of releasing only when a version is stable and ready, and less frequently). Each stub represents a year, and each bar represents a month. The vertical lines with the version label represent the beginning of that new major release. Months in which no version were released are months were no development version at all was released (only production versions were released, and the work completely stopped on the last development version). Since mid 1997, the rates seem stable and are around 3–6 releases per month. We can see also that altough it has many features of development versions, version 2.4 was released less frequently than the development versions. Usually there was one release per month, and the maximum is three. By the way, once 2.5 was released, and until 2.6 was released, there were only 7 releases of 2.4 (in a period of over 2 years).

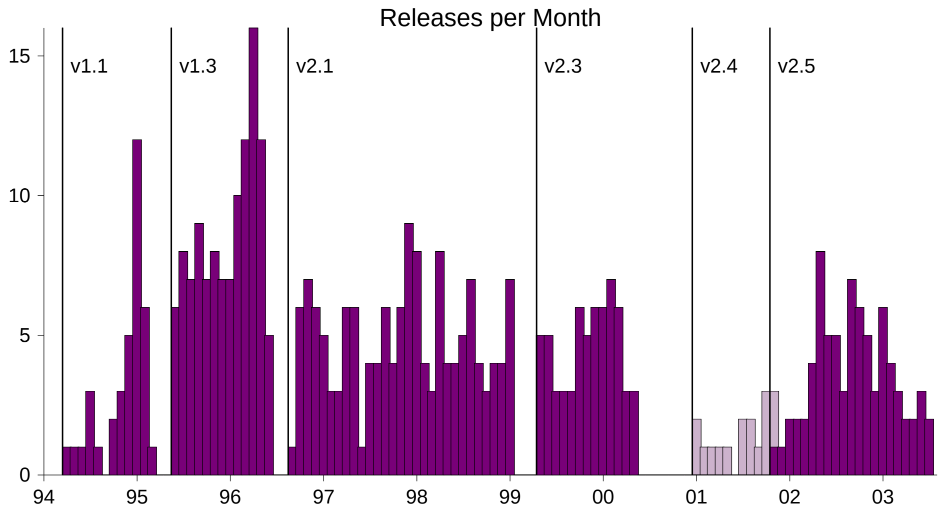


Figure 31:
Number of Releases per Month for Development Versions only

As mentioned above, starting with version 2.6 the versions are timed to be released once every 2.5–3 months. Again, development versions are released according to need. The method of timed release supports this law.

5.4.5 Conservation of Familiarity

According to this law, during the active life of an evolving program, the content of successive releases is statistically invariant. This is relevant of course when looking at successive production versions, which are those that are operational and released to users.

Lehman et al. [14] suggests looking at the incremental growth — and if it is constant or declining in average, it indicates familiarity. Moreover, they suggest a threshold (the average increment), for which if two or more consecutive points exceeds it, the next points should be close to zero or negative. The kernel releases are such that in successive releases of the same major production version (or minor version in 2.6) the changes are very small (most of the time zero, but sometimes increasing/decreasing) One can thus say that Linux growth follows the trends described in [14] and thus supports the law.

However, the changes between successive major versions are significant. In fact, they are so significant that users may opt to continue using an out-of-date previous release. This is witnessed by continued support for production versions long after the next one is released, and by the extension of the recent 2.6.16 version. Thus we have both support for the law (as witnessed by the longevity of production versions) and contradiction of the law (because successive production versions with significant changes are nevertheless released).

5.4.6 Continuing Growth

According to this law, functional content of a program must be continually increased to maintain user satisfaction over its lifetime. The functionality additions can be seen in differences between successive versions of development kernel releases or as major versions (minor in 2.6) of production kernel releases. Obviously there are significant functional enhancements and additions to the Linux kernel all this time.

In addition, size measures are also growing. In fact, they are growing in a super-linear rate up to version 2.5 and in a linear rate in version 2.6. This resembles the results of Godfrey and Tu [5] who identified Linux growth as super linear. Previous studies on Lehman's Laws show close to linear long term growth with a superimposed ripple [13]. Perhaps the results we are seeing in v2.6 can support such findings.

5.4.7 Declining Quality

According to this law, programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment. This case is similar to the "Increasing Complexity" law, in the sense that it is very hard to prove or disprove, as it allows both trends (if quality declines, the law is supported, and if it is not, it might be due to the maintenance and adaptation efforts). Moreover, it is hard to measure "quality".

While it is hard to obtain precise quantified metrics for this, Linux has been in growing use for 14 years, and there are no indications that its adoption rate is abating. It is widely used on large-scale servers, clusters, and supercomputers. This may be taken as anecdotal evidence that its quality is not declining, but rather that its usefulness is increasing.

We have also seen that there are functional changes over time and those may indicate of adapting to the operational environment, which, according to the law, might result in non-declining quality. These changes might explain the increase in usefulness.

5.4.8 Feedback System

The issue of feedback was mentioned already as an element of the self regulation law. More generally, Linux is the archetypical open-source system in which continued development is guided by feedback from the user community [23]. Specific evidence for feedback is the switch to the v2.6 release scheme, in response to user discomfort with the long delays in releasing enhanced production versions. However, this is hard to quantify and express as a law.

6 Conclusions and Future Work

6.1 Conclusions

The Linux kernel is one of the most successful open-source software projects in the world. Over the last 15 years it has grown and improved continuously, introducing more features and enhancements, as well as better reliability and experience.

In this study, we have calculated different metrics for the different releases of Linux, and analyzed them in order to better understand the maintenance process in Linux.

Due to the structure of Linux, and the releases paradigm, we analyzed our results in several different dissections — we compared between the development and the production versions and version 2.6, and also distinguished between the *arch* and *drivers* directories and the "core" kernel.

Looking at the different size metrics, we found, as expected, that Linux is growing. In fact, until version 2.5 it was growing in a super-linear rate, but starting with version 2.6 the growth is closer to linear. When looking at the LOC metrics per function (i.e. averaging it over functions) we find that it is decreasing. This is due to the growth in number of functions, which has a higher rate than that of LOC (i.e. the new functions are smaller on average).

We also found that the growth of the size metrics in the production versions is much more smooth than in the development versions, and the metrics usually stabilize at a constant level after a while.

For MCC and Halstead metrics, we found that when aggregating them over the entire Kernel we see a growth (i.e. it becomes more complex, or more hard to maintain), but when we analyze them at the function level (averaging over the number of functions) we see a different picture. In that case, the general trend is a decreasing one, where the values for production kernels are usually higher than of consecutive development kernels (meaning they are “more complex”), and the values for development kernels are more volatile. The reasons for this can be real improvements and perfective maintenance in the development kernel, or just addition of more files/function in those kernels, or maybe just correcting production versions without caring about the complexity at that time.

As a result of the findings above, Oman’s maintainability index was generally increasing (i.e. the Linux kernel seems to become more maintainable). Similarly to the described above, for the production versions we found a slight decrease at the beginning and then a constant value of Maintainability Index.

Specific cases where the production versions or the development versions did not follow the general trend were analyzed and explained in detail. These were usually instances of adding a large module that was developed elsewhere into the kernel.

As for the distinction between the *arch* and *drivers* directories and the “core” kernel, we found that *arch* and *drivers* usually (but not always) displayed the same trends as the “core” kernel, but with higher (or “worse”) values. For example, there are more LOC, files, functions, and the complexity and maintainability indexes imply lower quality software for those directories. The same behavior holds when looking at the function level.

Generally, we have seen correlation between the different metrics, with a few exceptions, but they all showed the same picture.

The Linux releases paradigm had two phases: before 2.6 and starting with 2.6. Before 2.6 there were two branches — development and production. The production kernels were released less often and usually corresponded to corrective maintenance, while the development versions were released quite often, and had perfective, preventative and adaptive maintenance as well. Starting with 2.6, the production minor (3^{rd}-digit) versions are released in an almost constant rate (every 2.5–3 month), with all types of maintenance included, and corrective maintenance versions are released when required (as a 4^{th} digit release).

We tried to characterize the different maintenance activities according to our results. We found that, unlike our presumption, production versions sometimes include more than just corrective maintenance. However, if we try to analyze only the corrective maintenance, where small and specific changes were inserted into the code, we see that it hardly effects the different metrics, and usually keeps them constant.

As for perfective maintenance, since it seems that development versions improve their complexity/maintainability over time and for production versions it is at first worsens and then stays constant, we might conclude that it improves the code. The same can be said about preventative maintenance. A special case of this is version 1.1, which had improved significantly over time, perhaps since there was much effort on preventative and perfective maintenance.

For adaptive maintenance, as reflected by the *arch* and *drivers* directories, it seems that it leads to lower maintainability and higher complexity values, and is more volatile than other types of maintenance.

We have also tried to identify whether some of Lehman's Laws are reflected in the evolution of Linux. We found support for Continuing Change, Invariant Work Rate and Continuing Growth. For Increasing complexity and Conservation of Familiarity we found both supporting and contradicting evidence. For self regulation, we were not able to conclude, but did not find evidence that is contradicting. Declining Quality and Feedback System laws were not in the scope of our study, but we found both anecdotal contradiction and anecdotal support for the first and ancdotal support for the latter.

6.2 Future Work

Many of the observations above were verified by the code, the metrics, different Linux forums, etc. However, for some of them, we suggested different explanations without giving a final conclusion. A further investigation of the code and the changelogs is required in order to resolve those issues and give sufficient explanations to the phenomena. One of those issues is, of course, whether the code is becoming less complex; another is the role of corrective maintenance in affecting software metrics. More work and analysis could be done for version 2.6, in order to better understand the effect of the merger of the two branches — production and development. More analysis could also be done in order to verify the rest of Lehman's Laws in Linux.

This study can also be extended by providing additional metrics, such as the indirect metrics developed by Yu [36] and more size metrics such the time spent, and the number of people involved, and how many developers participate in each type of activity. These might allow us to characterize further the different maintenance activities.

Another suggestion is to replicate the study for other operating system kernels and for other large software, and to compare the trends and the qualitative results of each, in order to try to understand whether our results are unique to Linux, or could be extended to other operating systems or maybe even to any large software. It might also be relevant to perform this comparison for open and closed source software in order to understand the differences in the development. The problem, of course, is to obtain suitable data for such a study.

Further analysis could be done to try and predict the growth and different metrics, comparing it with our results. This can allow software engineers and project managers to understand the different product stages and allocation of time, effort, and workers to each of them.

Appendix: Analysis of the effect of having no definitions when pre-compiling Linux

The different pre-compiler commands that might be effected be former macro definitions are: #ifdef, #ifndef,#else,#if defined, #if !defined, and #elif. As mentioned before, when we choose to perform analysis that requires per-processing without compilation we are in the "else" scope. This means that code that is surrounded by #ifdef and #endif or #ifdef and #else will not be analyzed. The same goes for code that is in a #if defined scope or #elif defined. Code in #ifndef scope or #if !defined or #else which matches #ifdef or #if defined will be analyzed.

In order to assess what is the loss of choosing to perform analysis without compilation, we examined the amount of code surrounded by any type of #if and #endif. This is an upper bound of the amount of code that is not analyzed, since we do not distinguish between the different types of "#if"s.

We checked the amount of code not analyzed by choosing this method for all files in the kernel, for .c files and for .h files. We notice here that about 30% of the lines out of all the lines in the code are affected in the early version and about 20% in the later ones. About 20% of the lines of the .c files are effected in the earlier versions to about 10% in the late ones. In .h files we see a phenomenon in which about 80% of the code is affected. This is due to the method of putting the entire code of an .h file in a #ifndef scope. In these cases, the analysis will be performed on the .h files since it is a #ifndef scope.

When we examined the "core" kernel, i.e. without the *arch* and *drivers* subdirectories, we see similar results. Here about 20%-30% of the lines out of all the lines in the code are affected in the early version and about 30%-40% in the later ones. About 5-10% of the lines of the .c files are effected in the earlier versions, then about 20% and since 2001 about 10% are affected. In .h files we see 80%-90% of the code affected.

In other words, the upper bound of the code not examined is 23% in some versions to 10% in most versions.

References

[1] I. T. Bowman, R. C. Holt, and N. V. Brewster, "*Linux as a case study: its extracted software architecture*". In 21st *Intl. Conf. Softw. Eng.*, pp. 555–563, May 1999.

[2] L. C. Briand, J. Wust, and H. Lounis, "*Using coupling measurement for impact analysis in object-oriented systems*". In *Intl. Conf. Softw. Maintenance*, pp. 475–482, Aug 1999.

[3] A. Capiluppi, M. Morisio, and J. F. Ramil, "*Structural evolution of an open source system: a case study*". In 12th *IEEE Intl. Workshop Program Comprehension*, pp. 172–182, Jun 2004.

[4] K. Chen, S. R. Schach, L. Yu, J. Offutt, and G. Z. Heller, "*Open-source change logs*". *Empirical Softw. Eng.* **9**, pp. 197–210, 2004.

[5] M. W. Godfrey and Q. Tu, "*Evolution in open source software: a case study*". In 16th *Intl. Conf. Softw. Maintenance*, pp. 131–142, Oct 2000.

[6] M. Halstead, *Elements of Software Science*. Operating and Programming Systems Series, Elsevier Science Inc., 1977.

[7] S. H. Kan, *Metrics and Models in Software Quality Engineering*. Addison Wesley, 2nd ed., 2004.

[8] C. F. Kemerer and S. Slaughter, "*An empirical approach to studying software evolution*". *IEEE Trans. Softw. Eng.* **25(4)**, pp. 493–509, Jul/Aug 1999.

[9] G. Kroah-Hartman, J. Corbet, and A. McPherson, *Linux Kernel Development — How Fast is it Going, Who is Going it, What are they Doing, and Who is Sponsoring it*. Technical Report, the Linux Foundation, Apr 2004.

[10] M. M. Lehman, "*Programs, life cycles, and laws of software evolution*". *Proc. IEEE* **68(9)**, pp. 1060–1076, Sep 1980.

[11] M. M. Lehman, "*Laws of software evolution revisited*". In *European Workshop on Software Process Technology*, pp. 108–124, Oct 1996.

[12] M. M. Lehman, D. E. Perry, and J. F. Ramil, "*Implications of evolution metrics on software maintenance*". In 14th *Intl. Conf. Softw. Maintenance*, pp. 208–217, Nov 1998.

[13] M. M. Lehman, D. E. Perry, and J. F. Ramil, "*On evidence supporting the FEAST hypothesis and the laws of software evolution*". In *Software Metrics Symposium*, pp. 84–88, Nov 1998.

[14] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perrry, and W. M. Turski, "*Metrics and laws of software evolution – the nineties view*". In 4th *Intl. Software Metrics Symp.*, pp. 20–32, Nov 1997.

[15] T. McCabe, "*A complexity measure*". *IEEE Transactions on Software Engineering* **2(4)**, pp. 308–320, 1976.

[16] E. Mills, *Software Metrics*. Technical Report Curriculum Module SEI-CM-12-1.1, Software Engineering Institute, December 1988.

[17] G. Myers, "*An extension to the cyclomatic measure of program complexity*". *SIGPLAN Not.* **12(10)**, pp. 61–64, Oct 1977.

[18] P. Oman and J. Hagemeister, "*Construction and testing of polynomials predicting software maintainability*". *J. Syst. & Softw.* **24(3)**, pp. 251–266, Mar 1994.

[19] T. Oy, "*Halstead metrics*". URL http://www.verifysoft.com/en_halstead_metrics.html, 2007.

[20] J. W. Paulson, G. Succi, and A. Eberlein, "*An empirical study of open-source and closed-source software products*". *IEEE Trans. Softw. Eng.* **30(4)**, pp. 246–256, Apr 2004.

[21] R. S. Pressman, *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2nd ed., 1987.

[22] V. T. Rajlich and K. H. Bennett, "*A staged model for the software life cycle*". *Computer* **33(7)**, pp. 66–71, Jul 2000.

[23] E. S. Raymond, "*The cathedral and the bazaar*". URL http://www.catb.org/˜esr/writings/cathedral-bazaar/cathedral-bazaar, 2000.

[24] D. A. Rusling, "*The linux kernel*". URL http://tldp.org/LDP/tlk/.

[25] N. Salt, "*Defining software science counting strategies*". *SIGPLAN Not.* **17(3)**, pp. 58–67, Mar 1982.

[26] S. R. Schach, *Object-Oriented and Classical Software Engineering*. McGraw-Hill, 7th ed., 2007.

[27] S. R. Schach, T. O. S. Adeshiyan, D. Balasubramanian, G. Madl, E. P. Osses, S. Singh, K. Suwanmongkol, M. Xie, and D. G. Feitelson, "*Common coupling and pointer variables, with application to a Linux case study*". *Software Quality J.* **15(1)**, pp. 99–113, March 2007.

[28] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and A. J. Offutt, "*Maintainability of the Linux kernel*". *IEE Proc.-Softw.* **149(2)**, pp. 18–23, 2002.

[29] S. R. Schach, B. Jin, L. Yu, G. Z. Heller, and J. Offutt, "*Determining the distribution of maintenance categories: survey versus measurement*". *Empirical Softw. Eng.* **8**, pp. 351–365, 2003.

[30] M. Shepperd, "*A critique of cyclomatic complexity as a software metric*". *Software Engineering J.* **3**, pp. 30–36, Mar 1988.

[31] M. Shepperd and D. C. Ince, "*A critique of three metrics*". *J. Syst. & Softw.* **26**, pp. 197–210, Sep 1994.

[32] I. Sommerville, *Software Engineering*. Addison-Wesley, 8th ed., 2007.

[33] L. Thomas, *An Analysis of Software Quality and Maintainability Metrics with an Application to a Longitudinal Study of the Linux Kernel*. PhD thesis, Vanderbilt University, 2008.

[34] E. VanDoren, *Maintainability Index Technique for Measuring Program Maintainability*. Technical Report, Software Engineering Institute, Mar 2002.

[35] wikipedia, "*Linux kernel*". URL http://en.wikipedia.org/wiki/Linux_kernel.

[36] L. Yu, "*Indirectly predicting the maintenance effort of open-source software*". *J. Softw. Maintenance & Evolution: Res. & Pract.* **18(5)**, pp. 311–332, Sep/Oct 2006.

[37] L. Yu, S. R. Schach, K. Chen, and J. Offutt, "*Categorization of common coupling and its application to the maintainability of the Linux kernel*". *IEEE Trans. Softw. Eng.* **30(10)**, pp. 694–706, Oct 2004.