An Analysis of Work Rhythms in Open Source

PHILIPP RIEMER

Master Thesis

Department of Computer Science
Open Source Research Group
Friedrich-Alexander-Universität Erlangen-Nürnberg

SUPERVISOR: Prof. Dr. Dirk Riehle

STUDENT ID: 21516617
E-MAIL: thesis@philippriemer.de

TIME FRAME: 1.4.–15.9.2012

PLAGIARISM DECLARATION

I hereby declare that, to the best of my knowledge and belief, this Master Thesis titled "An Analysis of Work Rhythms in Open Source" is my own work. I confirm that each significant contribution to, and quotation in this thesis from the work, or works of other people is indicated through the proper use of citations and references.

Ich versichere, dass ich meine Masterarbeit mit dem Titel "An Analysis of Work Rhythms in Open Source" ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Nürnberg, 15.9.2012

Philipp Riemer

ABSTRACT

Is the development of Free/Libre/Open Source Software (FLOSS) still like roughly 20 years ago, when a Finnish student wrote the basis of a nowadays successful operating systems in his spare time at home? Or is FLOSS becoming mainstream in the perception of companies? How much of the work on FLOSS is done during working hours?

To answer these questions, this thesis examines the history of over 9 000 projects. Based on the commit times, it is evaluated if open source software development has become commercial and if this fact can be shown using empirical data. As a prominent example, a special focus is put on the Linux kernel.

By analyzing the work rhythms in Open Source Development based on the time when each contributed change was performed, it can be shown that most of the development work is done during the working week on company time. In addition, trends spanning over a period of seven and eight years, respectively, are compared, providing insights about how the ratio of working to spare time developers changed over time.

By measuring how much work is professionally performed based on a large variety of distinct projects, this thesis helps understanding the amount and impact of commercial entities on Free/Libre/Open Source Software development.

ZUSAMMENFASSUNG

Ist die Entwicklung von Open-Source-Software (FLOSS) immer noch wie vor rund 20 Jahren, als ein finnischer Student die Grundlage für ein heutzutage sehr erfolgreiches Betriebssystem in seiner Freizeit am heimischen PC schuf? Oder ist FLOSS inzwischen etwas Selbstverständliches für Unternehmen geworden? Wie viel der Entwicklungsarbeit an FLOSS erfolgt hauptberuflich, also während der bezahlten Arbeitszeiten?

Um diese Fragen zu beantworten, untersucht die vorliegende Masterarbeit die Historie von über 9000 Projekten. Sie untersucht, ob FLOSS-Entwicklung von Unternehmen betrieben wird und ob jene Tatsache anhand empirischer Daten nachgewiesen werden kann. Als prominentes Beispiel wird hierzu das Linux-Kernel-Projekt gesondert betrachet.

Durch die Analyse der Arbeitsrhythmen von Softwareentwicklern, basierend auf dem Zeitpunkt, wann eine beigetragene Veränderung in einem Projekt akzeptiert wurde, kann gezeigt werden, dass die meiste Entwicklungsarbeit an offener Software unter der Woche während der regulären Arbeitszeiten durchgeführt wird. Zusätzlich werden die Trends über Zeiträume von sieben bzw. acht Jahren verglichen, was Aufschlüsse darüber gibt, wie sich das Verhältnis von Arbeits- zu Freizeitentwicklung über die Zeit verändert hat.

Basierend auf der Analyse einer großen Anzahl unterschiedlicher Projekte hilft diese Arbeit, den Einfluss von Firmen auf die Entwicklung von FLOSS besser zu verstehen.

CONTENTS

FIGURES

TABLES

ACRONYMS

ACF	Autocorrelation Function
API	Application Programming Interface
ARIMA	Autoregressive Integrated Moving Average
CDF	Cumulative Distribution Function
CEST	Central European Summer Time
CET	Central European Time
CVS	Concurrent Versions System
DST	Daylight Saving Time
FAU	Friedrich-Alexander University Erlangen-Nuremberg
FLOSS	Free/Libre/Open Source Software
HPC	High-Performance Computing
IANA	Internet Assigned Numbers Authority
ICANN	Internet Corporation for Assigned Names and Numbers
IDE	Integrated Development Environment
ISO	International Organization for Standardization
LOESS	Locally Weighted Scatterplot Smoothing
LRZ	Leibniz-Rechenzentrum
OECD	Organisation for Economic Co-operation and Development
POSIX	Portable Operating System Interface
RHEL	Red Hat Enterprise Linux
RRZE	Regionales Rechenzentrum Erlangen
SQL	Structured Query Language
SVN	Subversion
TUM	University of Technology in Munich
UTC	Coordinated Universal Time
VCS	Version Control System

1 INTRODUCTION

While traditional software companies try to keep the source code of their applications proprietary, so-called Free/Libre/Open Source Software (FLOSS) tries to do the opposite [Wei12]. In general, the latter describes software that is licensed under an open source license and, thus, complies to the requirements constituted in the open source definition [OSI].

FLOSS has grown from a negligible phenomenon to an omnipresent part of today's life [Rie11]. Products such as Firefox, Open-/LibreOffice, Linux, WordPress, or VLC are popular and well known. Several studies indicate that FLOSS has also become a substantial part of today's business world and is used in nearly all areas of industry and government [EY11; Rie11; Rie07].

In 2009, an analyst of the Gartner Group estimated that at least 80% of all software product companies will use FLOSS by 2012 [Dri09]. Moreover, also in 2009, 46% of all responding enterprises in *Forrester's Enterprise and SMB Software Survey, North America and Europe* were using or implementing free software products [HGS09]. It can be assumed that this number has grown since then.

FLOSS products are the market leaders since years in some business areas, such as web-servers or High-Performance Computing (HPC) [Net12; Top12]. Another recent example can be found when considering the development of mobile phone market. Here, the Android operating system, initially announced in 2007, already had a market share of more than 59 percent on all smart phones in the first quarter of 2012 [Ben12]. In addition, the commercial value of FLOSS can be demonstrated by the fact that the open source company MySQL AB was acquired in 2008 for a price of one billion US-dollars [MyS08] by Sun Microsystems, that had already multiple open sourced products in its portfolio. Shortly afterwards, in 2009, the buyer was taken over as well in a deal valued at more than 7 billion US-dollars by Oracle [Fin09].

In the light of these activities, this thesis examines if FLOSS has become mainstream in the perception of companies like e.g. Forrester predicted[HGS09]. Assuming that not only the usage but also the development has become common, evidences are searched as proof that FLOSS is not solely developed by hobbyists as spare-time projects.

The document at hand evaluates if the influence of commercial enterprises can be identified in FLOSS projects. It inspects how much of the work is performed during general working hours and, hence, on a paid basis, backed by companies. Therefore, the commit history of over 9 000 projects is analyzed. Based on the individual commit times, this thesis evaluates if open source software development has become commercial and if this fact can be shown using empirical data.

Furthermore, general patterns in the behavior of committers are highlighted that become visible when inspecting the evolution of one or multiple projects, respectively, over the time frame of several years. By aggregating the commits into different time spans, multiple patterns regarding development activity are examined.

The insights presented here might be used to better understanding work rhythms in FLOSS development, e.g. for more effective coordination of meetings (micro level) or releases (macro level). In addition, by highlighting patterns in the behavior of contributors of thousands of projects, general trends can be made visible, such as the close to stable ratio of working time to volunteer work in FLOSS presented in Section 4.3.

This thesis is outlined as follows:

In Chapter 2 the reader is introduced into the foundations for later discussions, such as background information and definitions. Moreover, a short overview of the work needed for creating this document and the performed evaluations in this context is given. The chapter is concluded with a review of existing literature in the field of interest.

After that, Chapter 3 discusses the Linux kernel project as a single but prominent example of FLOSS work in detail. Based on its comprehensive publicly available repository data, the behavior of close to eight thousand contributors who submitted patches over the last seven years is analyzed. Furthermore, having the possibility of accessing the precise release dates, an evaluation is performed if the regular cadence of this project has an influence on the integration of patches. In addition, the ratio of committers' work with authors over time is inspected, as the Version Control System (VCS) of the Linux kernel project allows the extraction of such information.

Then, a large variety of several thousand distinct FLOSS projects is analyzed in Chapter 4 and it is evaluated if the patterns, found for the Linux kernel, can also be observed in general in FLOSS development. The collection of a data source with such a large number of projects from multiple origins cannot be performed without trade-offs. Consequently, the compromises as well as the needed data preparation are outlined before starting the analysis.

Finally, Chapter 5 provides a summary and concludes with an outlook of how the findings can support a better understanding of Free/Libre/Open Source Software and the impact of commercial entities to its development. Aspects and ideas that could not be worked on due to time or scope restrictions are listed here as well. The following appendix contains code extracts of scripts and queries used for the creation of this work.

2

BASICS

2.1 RESEARCH DEFINITIONS

2.1.1 *Author vs. Committer*

There are several options for classifying members of a programming project. The following explains one by introducing the concept of authors and committers/maintainers:

Independent of their size, FLOSS projects can be considered as projects with a large number of involved parties, since they are – due to their open nature – in theory globally accessible and changeable. Thus, the number of potential developers of a single FLOSS project can be seen as equal to the total number of people alive on earth. During the development of larger projects with many participants, one can distinguish between the role of an AUTHOR, meaning people who contribute new code or change existing functionality, and the role of a COMMITTER or maintainer, representing people who validate the contributed pieces and eventually integrate them into the official repository [Han+04; Rie06].

In this context, a maintainer is responsible for a distinct part of the project's codebase, or so-called subsystem. He or she also acts as an (additional) quality assurance instance to ensure, as well as possible, that no malicious behavior is introduced into the code and that guidelines are respected. In large projects, like the Linux kernel, domain experts manage the code e.g. for specific file systems, drivers, or hardware architectures. As a result, one needs the acceptance of the respective maintainer(s) to introduce a new functionality or a change to a subsystem. [Tor+12; Cor09]

In most FLOSS projects only a small part of all developers has commit rights to the official repository. If someone wants to change a part of the software, he or she has to download (or "checkout") the source code and write an initial version of the change locally. Once that is done, a *patch* needs to be created with the `diff` program and then sent to a mailing list or uploaded to an issue tracking application. The proposed changes are then evaluated by a maintainer or a person who has been contributing for a longer period and, as a result of his or her work, is trusted by the maintainer. If one or multiple programmers approve the change after their review, it will be integrated into the project's codebase [Rie06]. This means, that it will be merged with the existing code in the project's repository by a committer and might get additional information such as a pointer to the solved issue. Otherwise the author gets a negative response over the preferred communication channel, most of the time with a

request for further improvements, some hints, and the motivation to re-submit the updated patch.

To find out who was responsible for the introduction of potentially faulty code in case of an error, already early, centralistic version control systems – e.g. the Concurrent Versions System (CVS) or its successor Subversion (SVN) – store information about a committer at check-in, such as user name and commit time [CVS12; ASF12].

In contrast to that, the modern generation of today's distributed version control systems follows a different networking principle. Instead of having the repository solely on a central server, every user has a full local copy on his or her computer. This has the advantage of availability, allows working and committing while being disconnected (e.g. on a plane) and simplifies handling backups and availability problems, due to the fact that every copy has all data [Toro7a][1].

In such distributed systems multiple developers can change parts of the code before merging them back into one collaborative codebase, using commands such as `push`, `pull` and `merge` [ASM12]. This allows new models of collaboration, such as a lieutenant system built around a "chain of trust" or *integration-manager model*, like e.g. used for the development of the Linux kernel, or a *Benevolent dictator model*, e.g. used for the development of Python [Coro8; RA12; Rup10]. Since not only the person who integrated the code from different repositories, but also the original developer, would be of interest in case of later problems, distributed version control systems distinguish explicitly between author and committer and store information like a person's name and the *local* time of a commit (preserving a person's time zone) for both contributors individually. [Cha09; Bir+09]

For trend analysis, in projects with centralized Version Control Systems the creation time of a patch can be only – if at all – found by searching the mailing list history or old reports in issue trackers. In contrast to that, modern distributed solutions provide direct access to this information by storing both commit times for each change. This fact allows more profound analyses of contributors' behavior by also considering the author's time stamp (cf. Sections 3.2 and 3.3).

Using the information provided by the Linux kernel repository, Section 3.4.3 shows how this ratio of authors and committers changed over time. While this is specific to a single project, it might provide an indication of how such a ratio evolved for centralized VCSs as well.

2.1.2 *Commit Size*

A software project is typically developed in multiple iterations, in a series of changes to its artifacts [KSR12]. A commit is an individual code contribution of an author that has been integrated into the code repository by a commit-

1 Quoting Linus Torvalds (00:17:18–00:17:35): *I have a theory of backups, which is: I don't do them. I put stuff up on one site and everyone else mirrors it. And if I crash my own machine, I don't really care because I can just download my own work right back – and it works beautifully well.*

ter [KRS10]. For programming projects, LIND and VAIRAVAN [LV89] show that the changed lines of code are an appropriate proxy for work spent on code. Consequently, a commit represents the basic unit of work performed by a contributor.

The lines changed by a commit are counted per file. The computation is typically performed by a utility with the name `diff`. As a result, a single commit consists of one diff per changed file. Each diff captures the changes made between consecutive versions of the same file. [HM76; Hec78]

Several different implementations are available today that differ in the algorithm for change determination. However, a `diff` tool in its essence can only reliably measure the number of lines added and removed. This becomes problematic when having an equal number of added and removed lines for a single file. In that case it might be possible that (a) only a slight change occurred or (b) a file was renamed or (c) a part of the file was deleted and new code was introduced that coincidentally resulted in the same number of changed lines. Unfortunately, the `diff` utility cannot determine this with certainty, which makes it in hindsight impossible to determine the correct total size of a commit. [HR09]

As KOLASSA, RIEHLE, and SALIM [KRS10] state, a changed line is always counted as one line removed and one line added, but should count as one line of work, while an added and a separately removed line of code should count as two lines of work. For getting more precise results, CANFORA, CERULO, and DI PENTA developed a sophisticated algorithm using the *Levenshtein* distance algorithm to measure the distance between two strings [CCDP09; CCDP07]. However, it is computationally expensive and does not scale to the large amounts of source code analyzed here. Instead, a simpler algorithm developed by HOFMANN and RIEHLE [HR09] is used that calculates the size of a commit as the mean of the minimum and maximum number of lines possibly touched in a given diff-block:

$$\begin{aligned} \text{min\_size} &= \max(\text{lines\_added}, \text{lines\_removed}) \\ \text{max\_size} &= \text{lines\_added} + \text{lines\_removed} \\ \text{commit\_size} &= (\text{min\_size} + \text{max\_size})/2 \end{aligned}$$

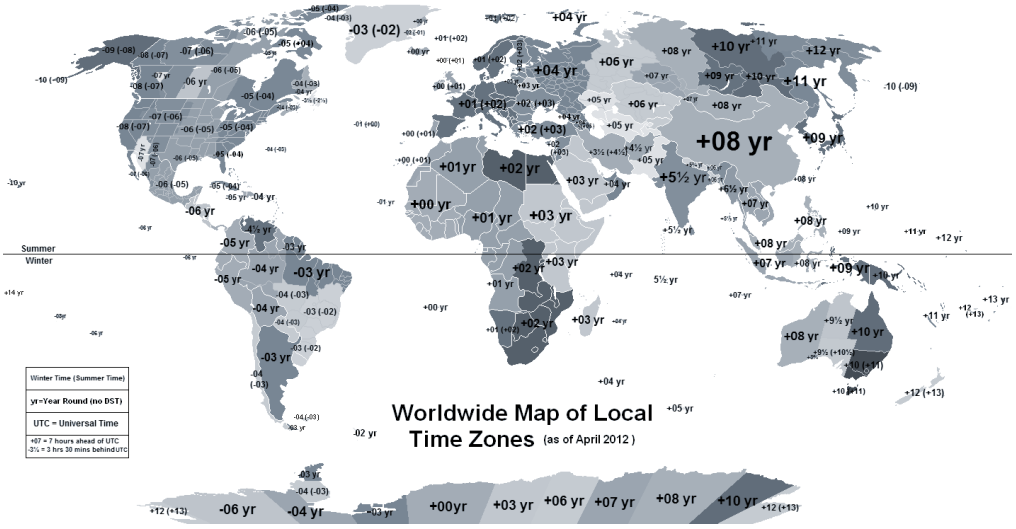
2.1.3 *Time*

The concept of local time is an essential part of the calculations that serve as the basis for this thesis. Consequently, this section provides an introduction into the topics of *time zones*, *daylight saving time*, as well as the internal representation of time in computers, the so-called *POSIX time*. Based on that, the foundation of this thesis, namely *working time* and *spare time*, are defined.

2.1.3.1 *Time Zones*

A time zone is a region on earth that has a uniform standard time for legal, commercial, and social purposes. These time zones tend to follow the boundaries of countries and their subdivisions, as depicted in Figure 2.1.

Figure 2.1: World Map with Time Zones



Source: http://en.wikipedia.org/wiki/File:Worldwide_Time_Zones_(including_DST).png (under CC-SA3.0)

ISO 8601 is an international standard covering the exchange of date and time-related data [ISO04]. It was first published by the International Organization for Standardization (ISO) in 1988 and is available in version 3 since 2004. The specifications in it provide a standardized method of communicating time-based information across time zones by attaching an offset to Coordinated Universal Time (UTC). Most of the 40 time zones are offset from UTC by a whole number of hours (from UTC-12 to UTC+13), but a few are offset by 30 or 45 minutes. Figure 2.1 depicts an overview of this concept.

The standard does not cover dates and times with words in their representation. This includes previous representations of time zones, e.g. CET for Central European Time, which is now defined as UTC+1. As a result, ISO 8601 is applicable whenever representation of dates in the Gregorian calendar, times in the 24-hour timekeeping system, time intervals and recurring time intervals are included in information interchange. [ISO04]

In this thesis the more familiar term UTC OFFSET is used rather than the term *zone designator* used by the standard.

2.1.3.2 *Daylight Saving Time*

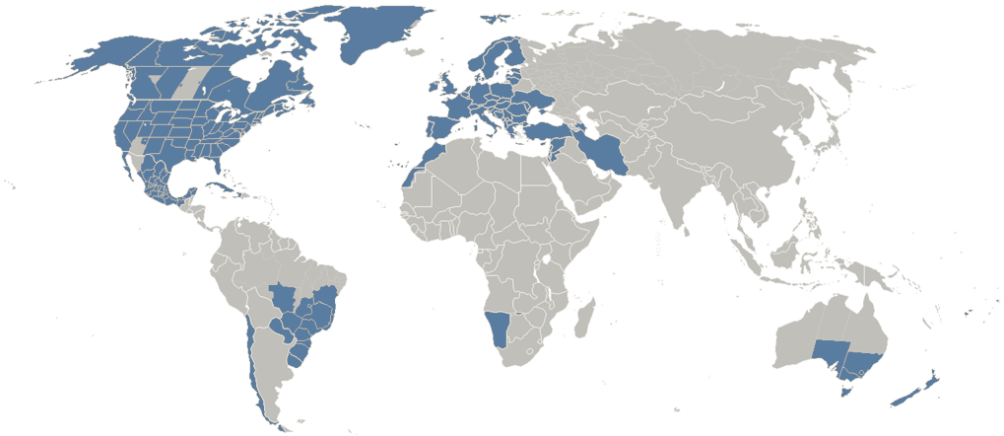
Daylight Saving Time (DST) – also summer time in several countries, e.g. German (*Sommerzeit*), Spanish (*Horario de verano*), and many others, as well as in British English, and European official terminology – is the practice of advancing clocks so that evenings have more daylight and mornings have less.

Typically clocks are adjusted forward one hour near the start of spring and are adjusted backward in autumn. [Pre05]

Though mentioned by Benjamin Franklin in 1784[2], the modern idea of daylight saving was first proposed in 1895 by George Vernon Hudson (1867–1946). It was first implemented during World War I as a wartime measure aimed at conserving coal [Gib10]. Most countries around the equator do not observe DST, since there the seasonal difference in sunlight is minimal.

The name of local time typically changes when DST is observed. American English replaces standard with daylight: for example, Pacific Standard Time (PST) becomes Pacific Daylight Time (PDT). British English typically inserts summer into other time zones, e.g. Central European Time becomes Central European Summer Time. In contrast to that, the ISO standard solely adds an hour without further indication of DST [ISO04].

Figure 2.2: Countries with Daylight Saving Time



Source: http://en.wikipedia.org/wiki/File:DST_Countries_Map.png (under CC-SA3.0)

Despite controversy, many countries have used the concept off and on. Figure 2.2 depicts the worldwide usage of DST by highlighting all countries that currently change clocks (January 2012).

The time zone database[3] of the Internet Assigned Numbers Authority (IANA) maps zone information to the named location's historical and predicted clock shifts. This database is used by many computer software systems, including most Unix-like operating systems – like Linux or Mac OS X – Java, and Oracle databases [EO07]. Internal time is stored in time zone independent epoch time (see next chapter). This allows to independently localize time display for each of potentially many simultaneous users and processes interacting on a single computer. Since October 2011 the Internet Corporation for Assigned Names and Numbers (ICANN) took responsibility for the maintenance of the database.[4]

2 http://www.webexhibits.org/daylightsaving/franklin.html (visited on 07/16/2012).
3 http://www.iana.org/time-zones (visited on 07/18/2012).
4 The mailing list announcement can be found under http://mm.icann.org/pipermail/tz/2011-October/008090.html (visited on 07/18/2012).

2.1.3.3 *POSIX Time*

The Unix time stamp, sometimes also known as Epoch time[5] or POSIX time, is a way to represent time by a single number. It is defined as the number of seconds that have elapsed since midnight of January 1st, 1970 (UTC). [ISO96] The name originates from the acronym for *Portable Operating System Interface*, a family of standards (IEEE 1003.x) for maintaining compatibility between operating systems. However, in its original form, it is neither a linear representation of time nor a true representation of UTC since it does not consider leap seconds[6].

As an example, the date and time "Sat, 31 Dec 2011 23:59:59 (UTC)", is represented as a Portable Operating System Interface (POSIX) time stamp by the single number 1325375999.

2.1.3.4 *Working Time*

The EU Council Directive 93/104/EC states that "working time shall mean any period during which the worker is working, at the employer's disposal and carrying out his activity or duties, in accordance with national laws and/or practice; rest period shall mean any period which is not working time" [EU93]. Thus, the workweek and weekend are those complementary parts of the seven-day week devoted to labor and rest, respectively.

In most Western countries the legal workweek ranges from Monday morning until Friday noon or evening. The weekend in Western countries comprises Saturday and Sunday, when most employees do not have to work. In cultures with a seven-day week, the day of rest derives from the main religious tradition: Sunday (Christian), Saturday (Jewish), or Friday (Muslim). Since most of the inspected FLOSS projects were developed in the western world, as depicted in Figure 2.3, a WORKWEEK is considered to be the weekdays Monday to Friday [Cen10].

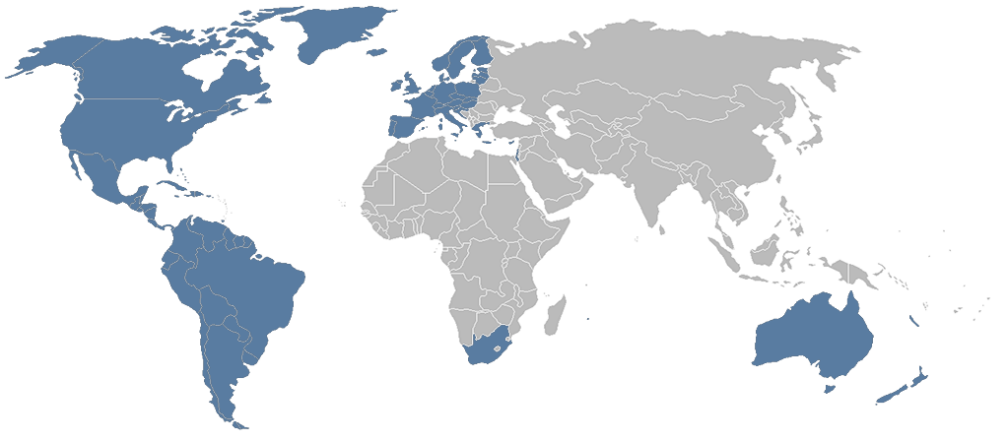
Historically, Henry Ford was the first prominent employer in the United States who standardized his factories on a five-day workweek, instead of the prevalent six days, without reducing employees' pay in 1926. Twelve years later President Franklin Roosevelt signed the Fair Labor Standards Act of 1938, which established a five-day, 40-hour workweek as a standard for every worker. Since then the phrase *9 to 5* is an expression in the United States originating from the traditional American business hours of 09:00 to 17:00, Monday through Friday, representing a workweek of five days with eight hours of work on each, comprising 40 hours in total. [Lom10]

In Germany, as another example, an 8-hour-workday was constituted by law in 1918, while the 40-hour-week, consisting of five days with each 8 hours of work on average, was established in 1965 (West) and 1967 (East) [HBS10; DKW09]. Here, the Working Hours Act (in German: Arbeitszeitgesetz, ArbZG)

5 See e.g. http://www.postgresql.org/docs/8.4/static/datatype-datetime.html#DATATYPE-DATETIME-SPECIAL-TABLE.

6 A more detailed discussion of this topic can be found under http://www.ucolick.org/~sla/leapsecs/onlinebib.html#POSIX (visited on 07/18/2012).

Figure 2.3: Western World Countries



Source: en.wikipedia.org/wiki/File:Westerncultures_map.png (under CC-SA3.0)

constitutes working time as the time span between start and end of daily labor without rest periods (§2.1 ArbZG). Furthermore, it specifies that an employee must not work more than eight hours per day on average over the period of six months, with a maximum of ten hours per day (§3 ArbZG). [BdJ12]

According to the Organisation for Economic Co-operation and Development (OECD), several countries have adopted a 40-hour workweek [OEC12].

While *9 to 5* is an established expression in the United States, the time period was originally defined with physical labor and manufacturing work in mind which might not apply to today's work anymore, especially in the information technology sector [Lom10; Falo7]. In addition, the definition does not consider rest periods, in contrast to e.g. Germany where they are defined by law (§4 ArbZG).

Since working time differs between countries with regard to start and end time, as well as resting breaks, this thesis adds a buffer of one hour to each edge of *9 to 5* and defines WORKING TIME as the period between 08:00 and 18:00 local time, while SPARE TIME is all time outside of the defined working time. Consequently, these time definitions are dependent on a person's time zone (cf. Section 2.1.3.1).

2.1.4 *Types of Developers*

After introducing the concepts of time in the previous sections and defining the related terms *working time* and *spare time*, contributors to FLOSS projects will be classified as follows in this thesis:

- PROFESSIONALS contribute mostly during working time.
- VOLUNTEERS work mostly during spare-time and weekends.

When analyzing working behavior, a hobby or spare time developer might have some commits that are not done during his or her spare time. This might happen due to events such as vacation or sickness, or due to a working period

slightly outside of the defined working time. Consequently, an assumption that hobbyists work solely during evening and weekend hours might be too strong. The same applies for the other extreme: professionals who are work full-time on an open source product might start their work before 08:00, end later or even finish work on weekends. As a result, a THRESHOLD of 80 % is assumed in the following, meaning that if 80% of a person's commits are done in his or her spare time, then this individual is counted as a hobby FLOSS developer, while if 80% of the commits are executed during working hours he or she is assumed to be a professional.

2.2 PROCEDURE OF WORK

2.2.1 *Sequence of Work Performed*

Due to the fact that the author was not the first person at the professorship working on the data inspected in Chapter 4, the work and R-scripts of predecessors were first analyzed. In addition, those scripts were extended to make them more generally applicable, while also fixing some erroneous behavior. Moreover, access was requested to the database that contains information about the various open-source projects provided by Ohloh (for more details see Chapter 4).

At the same time, an evaluation of multiple Integrated Development Environment (IDE) solutions for R took place. Here, open-source as well as proprietary software was taken into consideration.

One of those commercial products was `Revolution R Enterprise`[7], which offers free academic subscriptions and claims to be optimized for working with large amounts of input data (also known as *BigData*). However, the IDE is based on `Microsoft Visual Studio 2008`[8] and hence only works under Microsoft Windows, while the customized R runtime is also available for Red Hat Enterprise Linux (RHEL). Revolution's IDE does not provide an option to cancel running calculations and, thus, does not allow blocking constructs, like infinite loops, to be stopped without a complete shutdown of the software. As a minor fact, the runtime was based at that time on version 2.13.1 of R, even though the upstream open-source version was already 2.15.1 and provided most notably a highly improved stack for parallel computation. Consequently, by offering no platform independence and only minor advantages due to proprietary extensions, this solution was not included in further evaluation.

Since it should be possible to perform all calculations on all major operating systems (Windows, Mac OS X, Linux), different IDEs running on the Java[9] platform were inspected. One of them, the `Java Gui for R`[10], could not handle the massive amount of input and crashed each time after loading the research data.

7 http://www.revolutionanalytics.com/products/revolution-enterprise.php
8 http://www.microsoft.com/visualstudio/en-us/products/2008-editions
9 http://java.com/
10 http://rforge.net/JGR/

Another, StatET for R[11], that is based on the well-known Eclipse[12] platform, provides the advantage of familiar components and, hence, a short training period. In the end, a majority of coding for this thesis was done using RStudio[13], a new cross-platform open source IDE started in 2010 that is under active development and has good community support.

At the same time, the author organized the existing R-code into multiple packages that encapsulate the procedures for database access, data aggregation, and the different analytic methods.

As soon as access was provided to the database, the table structure of it was analyzed using SchemaSpy[14]. The software works similar to e.g. Doxygen[15] and outputs an HTML documentation of all tables, columns and relationships in a database. When providing the dump of their database, Ohloh did not supply further documentation about the various tables or their content. The only source for further information are archived e-mail conversations from 2007 that do not help much for a better understanding of the structures and relationships. After performing a few calculations, it became obvious that the data needed a lot of cleanup upfront, since around 60 percent of its entries were duplicated[16] and most tables did not have properly defined relationships, i.e. foreign keys. More information regarding this topic can be found in Section 4.1.3. In addition, the large amount of data led to long running calculations and consequently some errors in the existing scripts only appeared after several hours of processing.

After a while of working on and cleaning the Ohloh data, the idea arose to also use checkouts of the current repositories of the open source projects and to analyze them in addition. However, this approach was discarded, although an exception was made for the Linux kernel project (see Chapter 3). The reason for this step was not the storage requirement of several hundreds of Gigabytes, it was rather due to the fact that after 2008 (the time at which the database dump was provided) many projects migrated from a centralistic Version Control System (VCS), such as CVS or SVN, to a distributed VCS, e.g. Git. Having over 9 000 distinct projects, this would have meant quite some manual effort in finding the new sources. Furthermore, some of the projects were abandoned after 2008 and, thus, the repository is nowadays not available anymore.

In addition to that, the process of extracting a complete history from centralistic repositories took more time than anticipated. As an example, reading and parsing the data of the Apache SVN project[17] – with at that time 1 245 446 revisions – took more than 76 computational hours. Other repositories, as e.g. that of the Boost C++ library[18], suffered serious performance problems due to the analysis with git-svn[19] (which downloads every single revision

11 http://www.walware.de/goto/statet
12 http://www.eclipse.org/
13 http://rstudio.org/
14 http://schemaspy.sourceforge.net/
15 http://doxygen.org/
16 A more detailed discussion of this can be found in [Hof12].
17 http://svn.apache.org/repos/asf/subversion/trunk/
18 http://svn.boost.org/svn/boost/trunk
19 The manual page can be found at http://git-scm.com/docs/git-svn (visited on 06/21/2011).

individually for analysis) and as a result, the repositories completely blocked anonymous read-only access for several days.

However, not only the download itself provided some hurdles. Since these types of VCSs do not store time zone information, all time-related data would have had to be calculated afterwards. In addition, the usernames would have been needed to be mapped to the real names and email addresses of the contributors to be able to analyze development activities over multiple projects. This can be handled using `svn2git`[20], which not only maps internal repository structures, such as tags and branches, better than `git-svn` but also provides ways to map user data. Nevertheless, researching user data of roughly 40 000 contributors exceeded the scope of this thesis and, consequently, this idea was scrapped as infeasible.

In contrast to the centralistic VCSs, the checkout of projects using Git repositories worked seamlessly and also contained all time information. However, a majority of projects was either too small to be relevant here or not active anymore. Consequently, this path was also not followed further. The data collected were solely used to verify the validity of the algorithms described in Section 4.1.5.

An analysis with `GitDM`[21] (git data miner), initiated by Jonathan Corbet for generating statistics to describe the development of the Linux kernel (e.g. [CKM12]), was evaluated as well. The software provides the option of utilizing Git's user mapping capabilities, as discussed in Section 3.1.1, and calculating various key figures based on the repository data. In the end, GitDM was not used since

- a majority of the projects were missing information, as described above,
- the heterogeneity and sheer number of repositories complicated the taks and most significantly,
- a great deal of effort would have been needed to program the required metrics into the program as they are currently unsupported.

As a result, the decision was made to work solely with the statistics software `R` [R12], which already provided implementations for most of the needed functionality.

Having a wide spectrum of potential options of investigation with this ample data set at hand, various ideas were discussed, evaluated, calculated, and eventually discarded over time. Due to the large amount of data under inspection, each of these steps takes days and, thus, most of the time, multiple paths were followed in parallel. One of those was the investigation of a variety of cluster algorithms (hierarchical clustering, k-means/k-medoids, etc.) and approaches to classify contributors and projects. However, due to the high heterogeneity and large size, non of the tried methods provided additional insight to the metrics already known or to those presented in Section 4.4.2.

20 https://github.com/nirvdrum/svn2git/
21 Repository at git://git.lwn.net/gitdm.git

Furthermore, additional features, such as the import of `Git`'s data into `R`, were programmed and calculations were performed using a variety of computers, as described in the next section. The results of these calculations are presented in this thesis, which itself has been typeset using LaTeX.

For the high-quality presentation of the resulting plots, several options and plotting solutions, such as `gnuplot`[22], were evaluated. Most of them provided only pixel-based output (e.g. PNG or JPEG format) or generated plots that were not pleasing to the eye as they looked if they were made 20 years ago. If scalable vector graphics, such as EPS or SVG, were supported, the programs only provided a small selection of fonts, which made the graphics look not proper. Eventually, GGPlot2 [Wic09], a free visualization package for `R`, was selected due to its quality and tight integration with the statistical analysis platform. The plots were converted into Ti*k*Z[23] graphics, which stands out due to the fact of being native LaTeX files with support for transparencies and vector-based plotting [MS07]. Consequently, the graphics do not show compression or scaling artifacts or other issues related with pixel-based images when zooming. In addition, they have the correct font styles also used for the rest of the text and not just a font of the few provided by the plotting program. Thanks to Ti*k*Z, also the texts of the illustrations can be selected, searched and copied in the PDF version of this thesis.

2.2.2 *Computers Used*

Most of the presented analyses were computed using the author's personal workstation equipped with 4+4 CPU cores (each 2.4 GHz) and 24 GB of memory. In addition, the professorship of open source research has a virtual machine with 4 CPU cores (each 2.8 GHz) and 16 GB memory. It hosts the database instance and other projects so that it could not be used for calculation, as other services would have suffered performance issues, but was solely considered as additional storage.

With this large amount of data used here, several calculations would have taken weeks with the hardware on hand. Consequently, the author requested access to the HPC infrastructure of the Regionales Rechenzentrum Erlangen (RRZE), the service provider and supercomputing department of the Friedrich-Alexander University Erlangen-Nuremberg (FAU). Later, the allowance was given to use the high-memory servers there, each with 16 CPU cores (each 2.3 GHz) and 128 GB of memory. However, `R` could only be used in a single-threaded manner running on only one core per calculation. Since this did not provide calculation speedup [MW11], another request was sent to the Leibniz-Rechenzentrum (LRZ), the supercomputing facility of the University of Technology in Munich (TUM), where an installation for `R` exists that can utilize multiple CPU cores.

22 http://www.gnuplot.info/
23 http://sourceforge.net/projects/pgf/

The cluster at TUM with the name "UltraViolet3" consists of 2 240 CPU cores (each 2.4 GHz) and 3.5 TB of main memory. It was used for all large-scale calculations. At peak times up to 1 500 cores were computing in parallel for the presented work here, utilizing approximately two terabyte of memory at that time. Even then, some calculations had to be largely customized to finish in the given time limit of 48 hours for a single job.

The option of working together with the *Distributed Data Mining* project at BOINC[24] was quickly rejected since too much programming effort would have been needed upfront and the results would not have been available in time. A request to Amazon, where `EC2` instances can be used for free by selected academic research projects, was sent but the next approval and selection process was scheduled on a date at which this analysis should already be finished. Consequently, a potential successor of the author's work might have the option of working on Amazon's infrastructure, while for the work presented here this option would have been too expensive.

2.3 LITERATURE REVIEW

A few studies exists that examine distributed development of FLOSS projects. Spinellis [Spi06] investigates the impact of geographical location on code quality in FreeBSD. He concludes that global development allows working around-the-clock on a project, but finds significant differences between the types of work performed at the various inspected locations. However, impacts on the quality of code and on productivity are negligible. In contrast to that, Cataldo and Nambiar [CN09] find that there is a high impact, and that the more distributed a project is the less impact does process maturity have.

Tang, Hassan, and Zou [THZ09] identifies the locations of contributors of PostgreSQL and GTK+ using e-mail metadata, such as top level domains and IP addresses. They find that a majority of contributions were performed in the United States and Germany. This finding is in line with Robles and Gonzales-Barahona [RG09], who analyze projects at SourceForge with the help of the time zone and e-mail data stored at the website, and conclude with the same two countries being the leaders in FLOSS development. Moreover, another study analyzing projects at GitHub[25] finds a majority of developers being located in the United States and Europe [TH10]

In a paper published this year, Bird and Nagappan [BN12] identify the top contributors in the CVS repositories of the Firefox and Eclipse projects until 2008. For finding out their location and employer information, they perform a completely manual look-up using a large number of different online sources. Similar to the previous mentioned publications, they conclude that the primary regions involved are the United States and Europe. Furthermore, the authors point out that solely approximately 22 % of Firefox and 3 % of Eclipse were

24 http://www.distributeddatamining.org/
25 https://github.com/

contributed by volunteers, while the rest was coded by professionals employed by various companies.

Di Penta and German [DPG09] describe the difficulties to identify copyright owners for parts of a system for legal reasons. In addition to former work, they not only consider committer information but also parse the source code for further information to find e.g. pointers to organizations. They also point out that matching CVS committer-ids and contributors is not trivial as individuals might have different credentials or multiple accounts.

The general identification of developers and the tracking of them across multiple projects is a common task of researchers since the beginning of software repository mining. Centralized VCS rely on usernames, while auxiliary systems, e.g. issue tracking, uses e-mail addresses for identifying users. The first paper describing simple heuristicts to match developers between Subversion repositories and mailing lists was published in 2005 [RG05]. Bird et al. [Bir+06] use similar methods to semi-automatically align multiple e-mail addresses to unique users with the help of social network analysis. Based on that, Gousios and Spinellis [GS09] add string matching heuristics to this approach, which slightly improves the results. Moreover, Poncin [Pon10] addresses this topic in detail in his master thesis, by combining similar string and partial matching heuristics with the ability to weight them to compensate different conventions for username generation across various projects [Pon11]. Also only recently, Cadenas [Cad12] proposes an approach for cleaning low quality data sets in various text formats and for replacing missing elements using interpolation techniques for the research of software repository databases.

Using modern VCSs, such as Git, the identification is alleviated, as those use e-mail addresses to identify contributors. Consequently, matching these e-mails with the ones used in bug descriptions and other auxiliary systems becomes a close to trivial task. Some issues in this regard are addressed in Section 3.1.1. By combining these information with public hosting providers, such as GitHub, this allows mapping users with metadata as well as the activity of a single person across multiple projects with higher precision [GS12; TH10].

The topic of distributed VCSs is also not widely discussed in publications, yet. Two papers as well as several blog posts of developers[26] point out that distributed VCS have advantages compared to centralized VCS by allowing non-core members to easier contribute and keep authorship of their submissions, in addition to the advantage of being able to work offline [Bir+09; AS09]. Rodriguez-Bustos and Aponte [RA12] discuss the impact of distributed VCSs on FLOSS projects by comparing the history of the Mozilla project before 2007 (CVS) and since then (Mercurial). They find that while the average number of changes per month is similar for both types of repositories, the distributed system lowers the entry barrier for new contributors. In addition to that, Bird et al. [Bir+09] find that the size of commits differs by only two lines on average when comparing the history of a project before and after switching to a distributed

26 E.g. http://lwn.net/Articles/246381/, http://www.python.org/dev/peps/pep-0374/, http://planet.mysql.com/entry/?id=13353, or http://use.perl.org/use.perl.org/articles/08/12/22/0830205.shtml (all visited on 08/20/2012)

VCS. In their work, the authors point out several advantages of the latter for repository mining, such as easier access to the data, author information, and smaller repository sizes. In 2012, the *MSR mining challenge* [SKB12] is the first of theses challenges that is performed using data from a Git repository. Previous ones were based on either CVS or SVN data. As one of the participants, SINHA, MANI, and GUPTA [SMG12] profile the developer community and industrial contributions to Android, and find a high influence of large companies, such as Google, Intel, IBM, or RedHat. Already since 2008, the Linux Foundation publishes an annual report of the Linux kernel development based on the metadata extracted from the Git repository of the project [CKM12]. The latter is also considered by CORBET [Cor09] who examines the way patches take until they appear in the main line repository of Linus Torvalds.

To overcome the problems with respect to centralized repositories, GOUSIOS and SPINELLIS [GS12] suggest using data from the GitHub hosting site, which is based on Git and, consequently, does not have most of the listed shortcomings. They propose a combination of MongoDB[27] and the peer-to-peer BitTorrent protocol [Coh08] to gather and distribute event streams and data of the hosted projects, such as commits, bug tracker entries, wiki pages etc. In combining a NoSQL database for flexibility against changes of the GitHub API and a high distribution to overcome the hourly access limitations and to provide scalable storage space, the mirroring of data from the almost four million repositories seems feasible in their eyes. In their paper, the authors also analyzed the commit sizes based on the programming language used as well as the time when commits were submitted, similar to the work in this thesis. However, in contrast to Sections 3.2.2 and 4.2.2, GOUSIOS and SPINELLIS [GS12] transformed all commit time stamps into UTC to solely show the round-the-clock development. The authors also find increased productivity during the working week days, which is in line with the findings presented here.

In a recent paper published this year, RODRIGUEZ, HERRAIZ, and HARRISON [RHH12] classify twelve common sources for repository mining research based on the considered sources, research fields, access type, and type of content. Furthermore, the authors address quality issues and list eight problem classes with such preprocessed data. Moreover, GOUSIOS and SPINELLIS [GS12] discusses some sources, however not in such detail as the previously mentioned publication.

Using social network analysis, YU and RAMASWAMY [YR07] inspect the interaction frequency between developers and classify them into the groups of core members and associate members. CROWNSTON and HOWISON [CH05] examine the consistency of social structure in FLOSS projects by evaluating the network centrality of communication during bug-fixing processes. They find that no pattern of communication centralization, or decentralization, respectively, is characteristic across multiple projects for such task and that each project has individual characteristics. In line with this, SCACCHI [Sca05] outlines the socio-technical activities in different communities.

27 http://www.mongodb.org/

Already in 2002, Mockus, Fielding, and Herbsleb [MFH02] discuss how developers and users are positioned in communities. In their onion model, it is possible to distinguish between core and co-developers, as well as active and passive users of a software. Similar that model, Yuan et al. [Yua+10] provide a similar onion-like classification for developers into 26 different roles, after clustering the projects and contributors registered at SourceForge by calculating the social network centralization between each of them. Unsurprisingly, they find that the most important roles here are developers and project managers. Robles, Gonzales-Barahona, and Herraiz [RGH09] investigate the evolution of the most active group of developers in two large projects. In their eyes, stability and permanence of this group is of great importance for the evolution and sustainability of a project. In addition, they discuss to which extent non-coding activities, e.g. writing of documentation, should be considered as well when identifying the core team of developers. Unfortunately, covering this information for such a large group of project exceeds the focus of this thesis.

To the best knowledge of the author of this thesis none of the mentioned publications nor others address the issue of working time versus volunteer work discussed here. No research could be found that analyzes the commercialization of FLOSS projects by investigating when what work was done. A reason might be that modern version control systems, such as Git and Mercurial, have allowed accessing commit history data in detail, including time zone information, only recently. Older systems, such as CVS or SVN only store a single UTC time stamp per commit, leading to the aforementioned issues in data quality, as well as difficulties in developer identification and time zone prediction.

3 LINUX KERNEL

3.1 RESEARCH APPROACH

In this chapter the data in the Linux kernel repository is analyzed. The project was chosen since it might be the most popular FLOSS project worldwide. The Linux kernel is also the largest open and collaboratively developed software project in the history of computing [CKM12].

The development of the Linux kernel was started by Linus Torvalds at the end of 1991 [Tor91]. Due to the worldwide expansion of the Internet as well as the switch to the distributed Version Control System *Git* [Git12], which was also initiated by Torvalds, the project's awareness and collaboration was raised further. All data of the Linux kernel project are managed in a Git repository since April 16th 2005 [And05; Tor07b]. Since then approximately 8000 different developers from nearly 800 distinct companies have participated in the kernel's development [CKM12].

An advantage stemming from the usage of Git is that all commit times are stored in the contributor's local time, including time zone information as UTC offset (cf. Section 2.1.3.1). This is in contrast to older solutions, such as CVS or SVN, where the committer time was internally stored in UTC and was always transformed and displayed in the user's time zone (cf. Section 2.1.1). As a result, the time zone for a contributor's local time is directly available for analysis and does not have be calculated upfront (cf. Section 4.1.5).

In addition, thanks to the usage of Git, a clear distinction between authors and committers (cf. Section 2.1.1) is possible, since commit information is stored for both [Git12].

The primary source is a clone of the current official Git repository of the Linux kernel, which can be downloaded with the command "`git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git`". The project's commit history is then examined with the help of `R` [R12].

In this thesis all commits between April 16th 2005 and December 31st 2011 are analyzed. The lower bound was chosen due to the fact that Git was introduced on this date to provide better manageability to the Linux kernel project. All commits in the repository before that date are most likely due to misconfiguration of the local time settings of individual developers (32 commits in total, 0.01%). While the development of the Linux kernel did not stop at the chosen upper bound, the latter was chosen in order to analyze data for entire calendar years. Given these time constraints, a total of 283 905 individual commits were selected for the analysis.

3.1.1 *The `.mailmap` File in Git*

Due to the nature of distributed Version Control Systems (VCSs), it might happen that a single person uses different credentials (name, email) when contributing. That was not possible in a central Version Control System having only one point of authorization. The `.mailmap` file in Git provides the option to map various e-mail addresses and names with different spellings to a single user. The easiest way for archiving this is to create a text file with the name `.mailmap` in the top level directory of a Git repository. Due to the fact that the file name starts with a dot it is usually hidden, like the rest of Git's configuration, and hence does not disturb the normal work with one's project files.

The `.mailmap` file allows the reduction of author and committer names, as well as multiple e-mail addresses, onto a unique combination of name and e-mail. Consequently, it is possible to pool and analyze contributors with slightly different spelling. In addition, the `.mailmap` file helps to reduce the computational requirements by allowing the correction of spelling mistakes etc. in advance, so that e.g. nearly similar e-mail addresses that would be classified as different by computers can be marked as identifying the same person.

The following list shows examples of how the `.mailmap` file might be used (a similar form can be also found in the man-page of Git-Shortlog[1]):

- *Mapping of different name spellings*
  Joe has added multiple commits to a project. On most of his computers he configured the system so that his name contains a middle name initial (sometimes with a dot afterwards, sometimes without), and on one computer he just used the combination of his first name and last name. With an appropriate rule it is possible to map *Joe Developer*, *Joe R Developer*, and *Joe R. Developer* to the latter version when showing commit information or creating statistics.
- *Normalization of e-mail addresses*
  Joe's colleague Jane received a new development machine and started programming on a new feature without first adjusting the configuration of her local Git installation. By default, Git uses the user account information of the operating system if no other settings are configured. As a result, Jane's name will probably be correct in the commit information, whereas the e-mail address defaults to the PC's name. This leads to entries like `<jane@laptop.(none)>` or `<jane@desktop.example.com>`, in addition to her official one `<jane@example.com>`. With a rule in the `.mailmap` file it is possible to map all pseudo addresses automatically to the correct version when listing them e.g. for finding out who introduced a change.
- *Reduction of spelling mistakes*
  When Joe received a new computer he first configured Git before starting

1 If Git is installed on a unix-like operating system (Linux, Mac OS X, etc.) it can be found via the console command `man git-shortlog`. In addition, the manual page is online at http://git-scm.com/docs/git-shortlog (visited on 06/21/2011).

development. During configuration he got distracted and accidentally misspelled his name *Jeo*. Unfortunately, he discovered this fact after he merged all code of his repository with the official one and thus was not able to easily change his name entries anymore. However, with an additional line in the `.mailmap` file, he can still hide the spelling mistake from the eyes of his co-workers.

- *Separation of Commits with nearly similar credentials*
  When fixing problems on-site for customers, Joe and Jane used the same e-mail address `<bugs@example.com>`, so that the customer can lookup the correct address for reporting additional features or change requests. However, for internal analyses it should be possible to distinguish between the two committers to find out who introduced what change. Since both developers used their individual name when committing, it is possible to create rules in the `.mailmap` file that split the commits with the generic e-mail address and map them to the specific ones.

For the Git repository of the Linux kernel project such a `.mailmap` file exists. It contains only 109 entries. Since there were around 7800 developers contributing to the project in the time span between 2005 and 2011, this small number of rules covers just approximately 1% of all existing shortcomings.

As a consequence the author extended the file after analyzing the author and committer data in the repository. It grew up to 2500 entries and now handles aliases and spelling mistakes in names and e-mail addresses for most of the developers. The updated file was also sent as a change request to the Linux kernel Mailing List[2], where it is now under review as of this writing. The additional work of updating the existing `.mailmap` file allows more detailed insights of the work of single contributors without necessitating a manual data clean before each analysis. Additionally, other researchers would also profit from this work as soon as the changes are accepted to the official repository.

3.1.2 *Linux Kernel Limitations*

Even if the Linux kernel project has existed for over 20 years now, only the data of the official Git repository (see above) over the timespan of seven years is analyzed.

An attempt has been made to also collect previous activities in a holistic Git repository spanning the whole project life time [Lan11]. The reasons for not using it in this thesis are twofold: On the one hand, the Linux kernel project was using the Version Control System BitKeeper[3] between 1998 [McV98] and 2005 [And05], which only stores committer but no author data (see e.g. [Ste12]). On the other hand, all commits before 1998 were only manually managed by Linus Torvalds without any Version Control System [McV98]. Consequently, there exist no precise information about the author's creation time of a patch (which, at the latest, is the point in time when he or she sends the mail to the

2 https://lkml.org/lkml/2012/5/16/247 (visited on 05/16/2012).
3 http://www.bitkeeper.com/

mailing list), and there also exists no committer time at which the patch was integrated into the existing codebase.

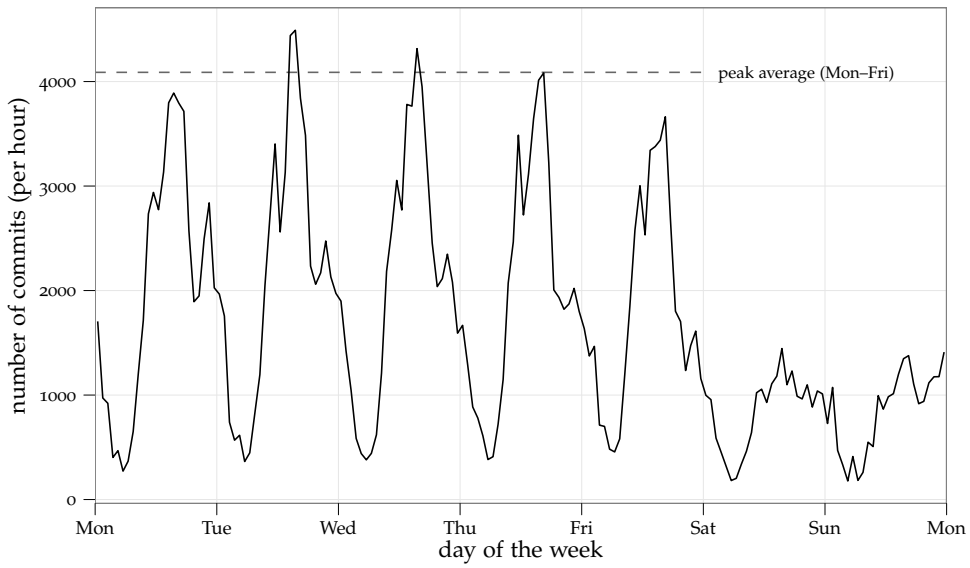
Another aspect to consider when working with the Linux kernel is the fact that – besides being the largest world-wide FLOSS project [CKM12] – the community also introduced many specialized rules, guidelines, and workflows to be able to manage the project [Coro8]. Thus, it is often analyzed as a prime example of a popular and large-size FLOSS project. However, one can assume that especially smaller projects are showing different behavior. Therefore, a large number of FLOSS projects (which also includes the Linux kernel for the inspected time frame) is later analyzed in more detail in Chapter 4, to evaluate if the results of the current chapter can also be applied to a wider range of projects.

3.2 WEEKLY WORK PATTERN

3.2.1 *Hourly Commits Condensed into One Week*

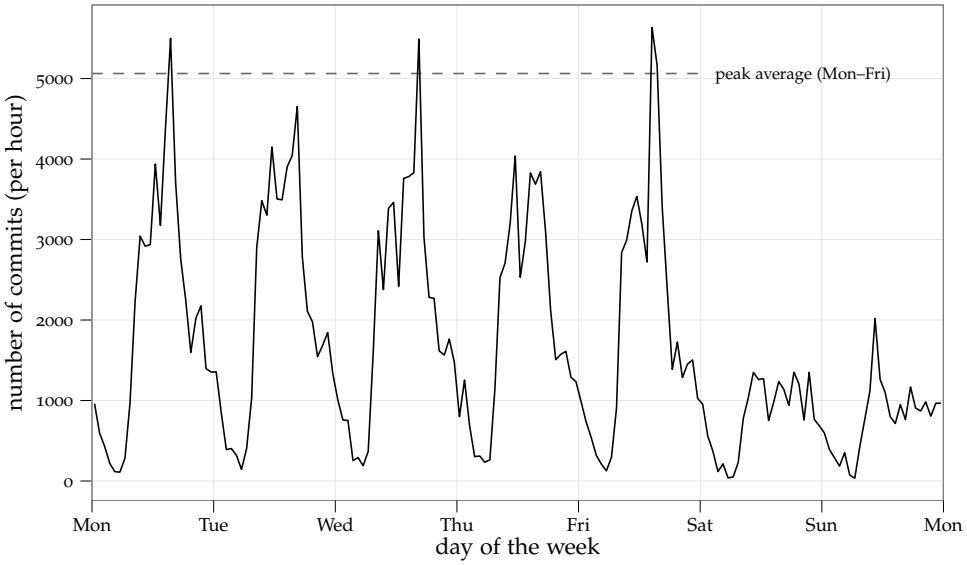
This section examines in a holistic approach on which day of the week and at which time of day developers primarily contribute to the Linux kernel project. To accomplish this, all activities over the whole inspected time frame are compressed into a single week.

Figure 3.1: Work Distribution of *Authors*



Figures 3.1 and 3.2 show the combined number of all commits for the timeframe from Monday 00:00 (or 12 am) until Sunday 23:59 (or 11:59 pm). Intuitively one can see that a majority of all commits is contributed between Monday and Friday. The amount of contribution per hour at the weekend adds up to only approximately a quarter of the peaks during the working week.

Figure 3.2: Work Distribution of *Committers*



Also intuitively visible is a pattern of the curve in both plots that repeats for every day from Monday to Friday and consists of a local maximum followed by a minor drop-off around noon before reaching a day's peak afterwards. Then the curve drops and has another local maximum in the evening before a new day follows and the pattern repeats. That characteristic will be analyzed in more detail below in Section 3.2.2.

Table 3.1: Absolute Number of Commits per Day of Week

day of week	authors		committers	
Monday	49 223	(17.3 %)	49 058	(17.3 %)
Tuesday	51 633	(18.2 %)	51 601	(18.2 %)
Wednesday	50 269	(17.7 %)	50 845	(17.9 %)
Thursday	48 175	(17.0 %)	46 753	(16.5 %)
Friday	44 065	(15.5 %)	47 727	(16.8 %)
Saturday	20 195	(7.1 %)	19 372	(6.8 %)
Sunday	20 313	(7.2 %)	18 517	(6.5 %)
Total	283 873	(100.0 %)	283 873	(100.0 %)

Looking at the numbers representing the total amount of commits per day as shown in Table 3.1, one can see that authors produce approximately twice as much during workweek days compared to weekend days. In contrast to that, contributors with commit rights are only pushing approximately a third of the number of commits on weekends compared to the workweek. The weekend represents around 28.6% of a 7-day-week; during that time authors submitted 14.3% of their commits while committers integrated 13.3% of all commits.

When comparing the ranges Monday to Friday and Saturday to Sunday separately, the total amount of commits stays nearly equal and no day significantly outperforms the others. However, when examining the differences between the daily *peaks* of the workweek and their average, which is depicted as a dashed line, it becomes evident that the values for Tuesday and Wednesday are slightly higher for authors, while committers have higher peaks on Monday, Wednesday, and Friday that are higher than average.

Especially the latter is interesting, as it seems as if the observed high commit numbers were submitted at the same hours. Furthermore, a potential explanation for the author's behavior is that people catch up on things they could not finish during the previous week. In addition, the break from work over the weekend allows developers to rethink solutions or think out new requirements that are then directly discussed and worked-on on Mondays. On Tuesdays administrative necessities are also performed and potential questions are resolved, so that the productivity and thus the amount of commits increases for authors.

In contrast to that, the activities of authors on Friday and of committers on Thursday and Friday are clearly below average when looking at the total numbers of commits. The reason for that might be twofold: In some countries, such as e.g. Germany, most companies try to finish earlier on Fridays while still conforming to a 40-hour-week. As a result, people might be distracted by the upcoming weekend planning. It would also explain why the drop on Fridays is much steeper compared to the days before. Another reason might be that a problem might be too big to be solved in a single day and – assuming a professional developer contributing to open source on a payed employment – is then postponed to the following week and only small changes are made on that day. In addition, the peak for authors and committers in the evening on Fridays is much lower than on the days before and only reaches the level of the weekend activity.

A high-level examination of commits on weekends shows that they do not contain such an obvious pattern. The number of commits per hour stays – even if lower than on the days before – on a nearly constant level without a large difference.

Moreover, the plots for authors (Figure 3.1) and committers (Figure 3.2) differ only slightly. One detail that catches one's eye is the fact that committers have a much steeper curve at the end of a day and seem to have no real peaks in the evening. A deduction might be that a majority of the Linux kernel maintainers spend their time on the project as a professional paid FLOSS developer with a lower number of hobby developers than in the group of authors (see also [CKM12]). Furthermore, the committer's curves are not as smooth as those of the authors and have a stronger maximum in the afternoon.

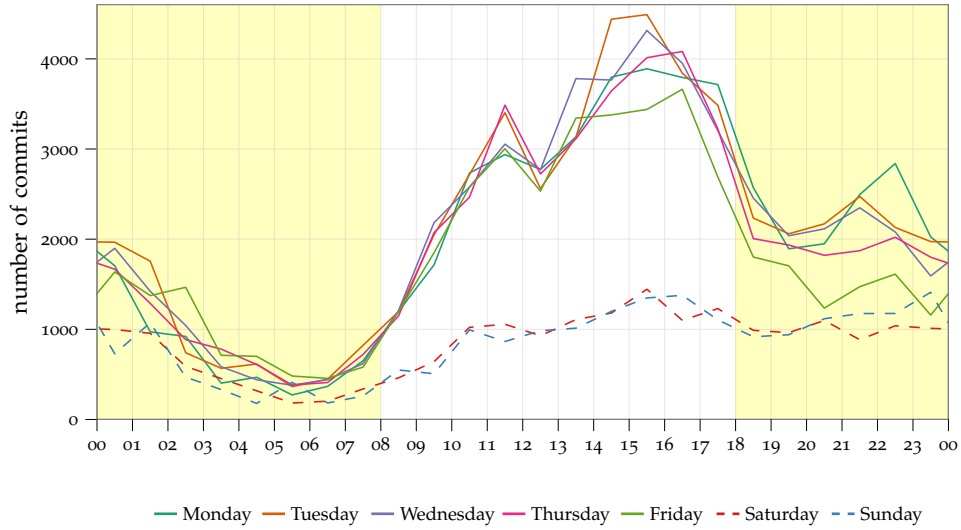
Interestingly, one can observe that the peak on Thursday afternoon is somehow missing in Figure 3.2. An idea that comes to mind is that there might be one or more companies – that are heavily involved in the development of the Linux kernel – that have a scheduled meeting at that time. By participating in the meeting, committers are hindered from integrating new changes during this period of approximately one hour. However, research in this direction,

based on the repository data at hand, revealed no clear pattern indicating a single commercial entity, when using the domain names of committer's email addresses. Applying a more sophisticated company mapping turned out to exceed the scope of this thesis, as e.g. companies are acquired by others and, as a result, the email domain changes for employees. Consequently, this would have required an individual manual look-up for each contributor of the inspected 30 releases (cf. Section 3.4.2) over time to assign him or her to the right employer. There exists cleaned-up lists of email addresses that are available for free online[4], however, these did not provide enough information in this context. Nevertheless, having a variety of members using other email addresses, the Linux Foundation seems to be a major factor in the observed drop of commits on Thursday afternoon when combining multiple data sources[5].

3.2.2 *Hourly Commits per Day*

After the examination of the time span of a whole week in the previous section, the following pages will delve deeper into the analysis of a single day.

Figure 3.3: Commits of *Authors* per Hour



In Figures 3.3 and 3.4 the same approach as described in Section 3.2.1 is used by aggregating all commits over the whole inspected project's time into one week. The plots show each a 24-hour timespan on the x-axis with the number of commits on the y-axis. The different days of a week are depicted by separate curves with individual colors, while the lines of those days on the weekend are additionally dashed to ease visual separation.

4 Repository at `https://github.com/gregkh/kernel-history` (visited on 06/28/2012).

5 Not only did the Foundation rename itself from "Open Source Development Labs" into "Linux Foundation" and, thus, changed their domain from `osdl.org` to `linux-foundation.org`, but it also has well-known members, such as Greg Kroah-Hartman, who are pushing commits using their individual email domains (e.g. `@kroah.org`)

Like already assumed when describing the overall week trends in Section 3.2.1, both plots for authors (Figure 3.3) and committers (Figure 3.4) are mostly similar and only differ in details with the individual curves lying nearly on top of each other. The majority of developers seems to end their work at the latest around 02:00 in the morning, independently of their role on the project.

For authors, the productivity – measured here by the amount of commits – steadily increases starting at 07:00 in the morning and eventually reaches a peak at the hour between 11:00 and 12:00. Then the curves drop slightly during the next hour, which might be due to a lunch break common at that time. After that, the number of commits remains continuously high between 13:00 and 18:00 before it significantly drops. The productivity is then only roughly a half compared to the hours before. It rises again at around 20:00 (except for Fridays), peaks between 21:00 and 23:00 depending on the day of the week, and decreases then towards the overall minimum between 03:00 and 07:00 in the morning.

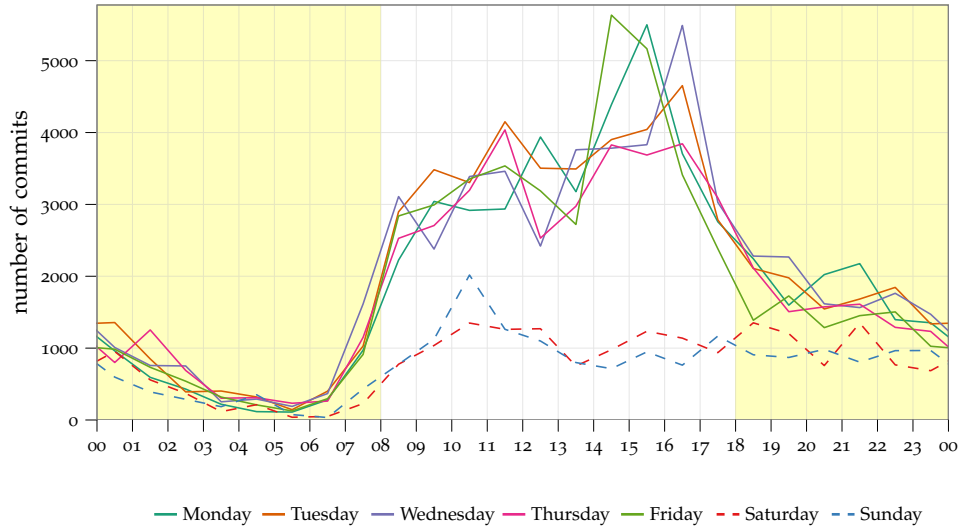
This pattern might be due to the fact that most developers commute home in the evening, which lowers the ability to commit new code during this period. Having finished dinner, some but not all authors work throughout the evening as well. Since the number of commits after 19:00 compared to the number during working hours is significantly smaller, it can be assumed that the number of hobby developers working on the Linux kernel during their spare time compared to the amount of professionals is lower as well. Interestingly, the number of evening commits between Monday and Friday decreases with each day of the week.

When eyeballing the weekend statistics for authors, nearly all hourly values are below the ones during the working week. Exceptions are at the transition between Saturday and Sunday at around 01:00 to 02:00 and at the transition between Sunday and Monday between 23:00 and 24:00. Both curves representing the weekend activity are close to each other and do not differ much over the day, showing nearly consistent activity between 09:00 and 24:00 with only slight dips during lunch- and suppertime.

Looking at the committer data, a much lower activity during evening and morning hours can be observed. However, contributors with commit rights seem to start their day earlier than authors during the workweek by having a much larger number of commits integrated into the codebase between 07:00 and 09:00. The behavior pattern between 08:00 and 14:00 is not as cleanly shaped as for authors and shows a high fluctuation. The evening activity is as high as for authors, but shows a more steady decrease with less late-night commits. As shown in Table 3.1, the number of commits per day of the week does not differ much between authors and committers. Consequently, having less activity in the morning as well as in the evening, results in higher peaks in the afternoon. However, also during this time frame, the curves differ quite a lot from each other as well as the points in time for the peaks.

The values of the weekend curves of committers are mostly similar to the ones of authors. However, a lunch break between 13:00 and 14:00 is more obvious

Figure 3.4: Commits of *Committers* per Hour



here. As an exception, a peak in committer productivity for the period on Sunday between 10:00 and 11:00 catches the eye.

The observed behavior during the workweek implies that maintainers of the Linux kernel are mainly paid professionals who work on that project during their working hours. The fact that committers and authors have nearly similar shaped curves when inspecting the daily number of commits might be the result of not every committer being a paid developer, but there are some of them who also work during their spare time.

Furthermore, one can conclude that authors mainly commit their changes during the workweek between 09:00 and 18:00, while committers mostly integrate those new code pieces from Monday to Friday between 08:00 and 17:00. This supports the time frame assumed in the working hour definition made in Section 2.1.3.4 covering the period between 08:00 and 18:00.

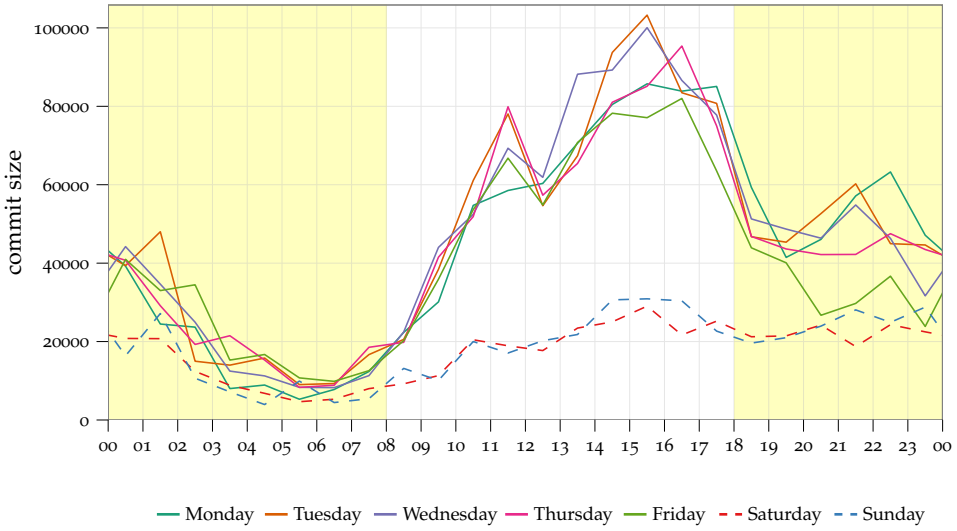
3.2.3 *Commit Size Changes per Hour*

The previous sections inspected developer behavior based on the number of commits per hour. In the following, the number of commit size changes is analyzed as well to examine if there are visible differences between commits and committed code.

For the understanding of the following, please recall that commit size is defined as a combination of added and deleted lines of code (cf. Section 2.1.2).

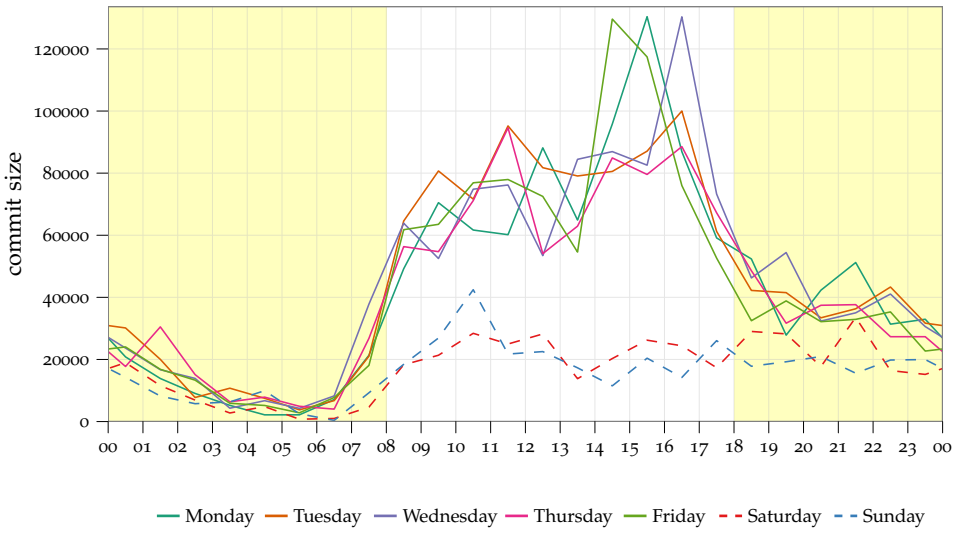
The differences are marginal when comparing Figure 3.5 with Figure 3.3. The number of commit size changes on Monday and Tuesday between 12:00 and 16:00 is slightly more pronounced while it is in general lower during the "sleeping phase" (02:00–07:00). The latter indicates that during these hours the

Figure 3.5: Commit Size Changes by *Authors* per Hour



patch size is lower and each patch changes only a few lines of code, while the former shows that big changes are most likely introduced in the afternoon. A reason for that might be the available capacity for concentration, which is typically much lower at night for most people.

Figure 3.6: Commit Size Changes by *Committers* per Hour



A comparison of Figure 3.6 with Figure 3.4 also shows no real differences. Noteworthy is only that the number of code changes on Monday between 19:00 and 20:00 is lower compared to the number of integrated commits at that time. Here the changes per patch are also smaller. This seems reasonable in combination with the assumptions mentioned in Section 3.2.2 that most

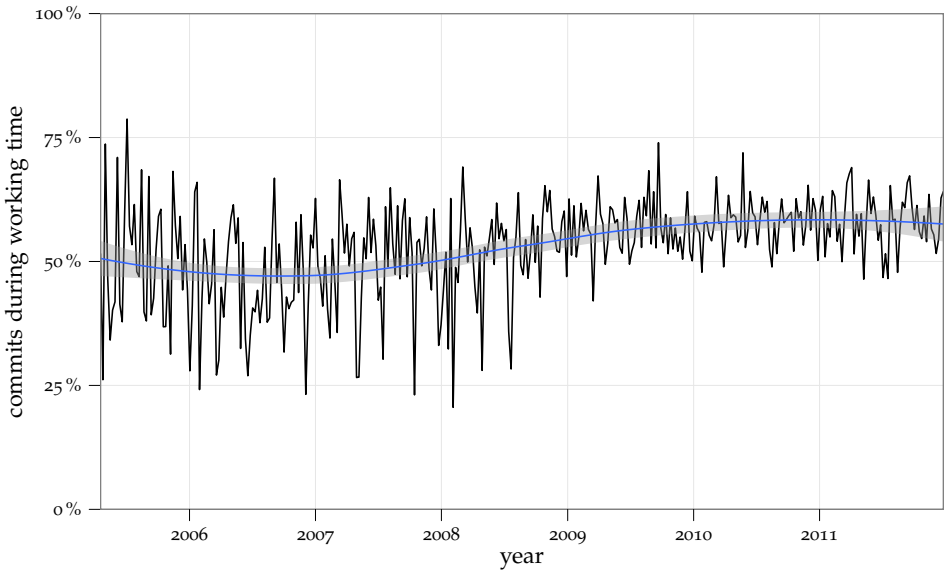
maintainers of the Linux kernel are paid professionals and commute home at that time.

3.3 WEEKLY WORK PATTERN TRENDS

In this section a trend over the overall lifetime of the Linux kernel project under Git is analyzed. The following plots contain a single value per week that represents the ratio between commits during worktime and the total amount of commits during that week. As a result, a year is depicted by 52 points that are connected by a line to visualize the changes. In addition, a Locally Weighted Scatterplot Smoothing (LOESS) curve is plotted on top of the data to show the overall trend of the time series.

3.3.1 *Trends for Authors*

Figure 3.7: Percentage of Commits during Working Time for *Authors* on a Weekly Basis

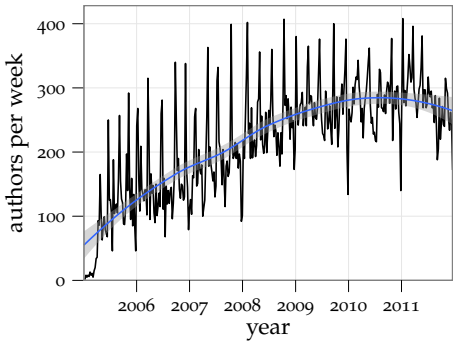


When looking at Figure 3.7, a decrease of fluctuation in the time series for authors can be observed. The percentage values of distinct weeks representing the commits in working time at the beginning (ignoring the first few weeks of Git's adoption) vary between approximately 20% and 80%. At around the middle of 2008 the trend stabilizes and the weekly percentages are consistently above 40%, close to 50%, and mostly below 70%. Also the "running average", represented by the light-blue LOESS curve, shows this trend by first rising only slightly until mid of 2007 before showing a clear upwards slope. Since 2010 it plateaus shortly below 60% and neither increases nor decreases further over the timespan of two years.

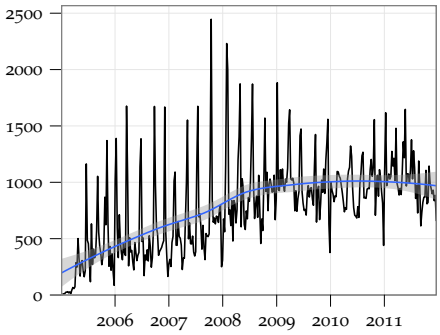
Comparing these facts with the clear upwards trend of the number of involved authors per week, as depicted in Figure 3.8a, a reduction of the fluctuation

Figure 3.8: Additional Time Series Plots for *Authors*

(a) Number of *Authors* per Week



(b) Commits by *Authors* per Week



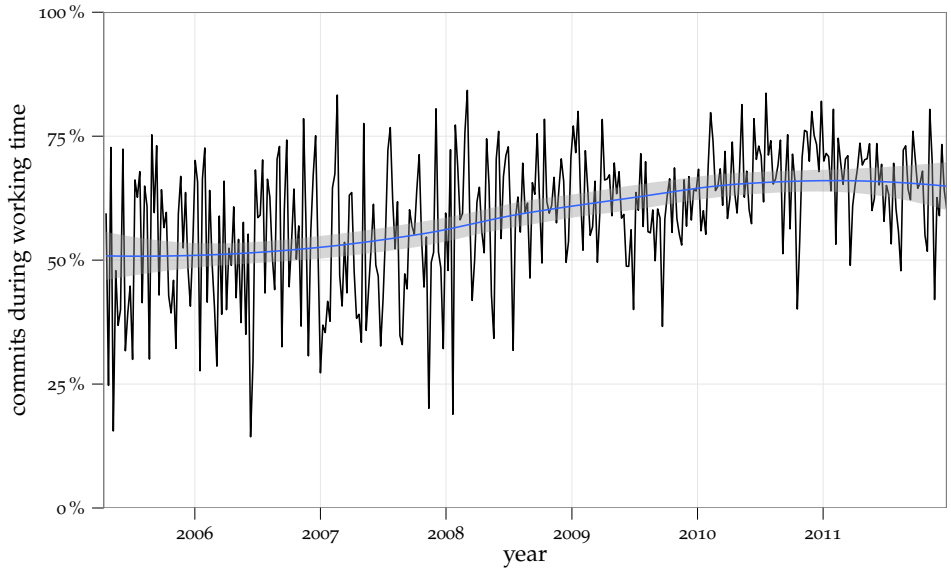
means a higher involvement of developers working on a professional basis during the defined working hours.

In contrast, the LOESS curve depicted in Figure 3.8b shows a similar stagnation in growth for the number of submitted patches. Although one can assume a high correlation between the number of patches and the number of contributors, the trend does not rise linearly with the latter. Thus, not only the work rhythm was adapted, but also the cadence of commits became more stable.

3.3.2 *Trends for Committers*

In Figure 3.9, which depicts the percentage of commits during worktime for committers, a similar upwards trend is visible as described previously for authors.

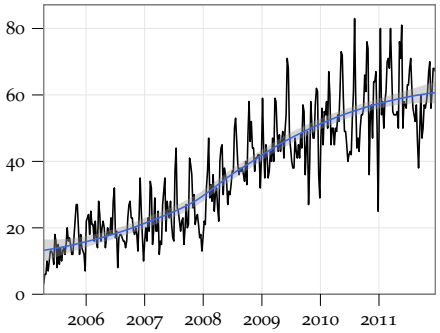
Figure 3.9: Percentage of Commits during Working Time for *Committers* on a Weekly Basis



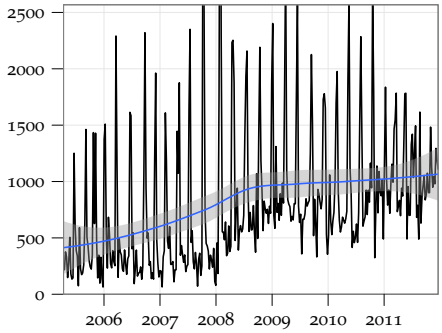
It increases even higher and plateaus at approximately 65% of all commits being integrated in worktime. The fluctuation between weeks does not decrease as much over time as in Figure 3.7 and varies since 2009 between 35% and 85%, with a clear drop in each October, which might be due to vacations.

Figure 3.10: Additional Time Series Plots for *Committers*

(a) Number of *Committers* per Week



(b) Commits by *Committers* per Week



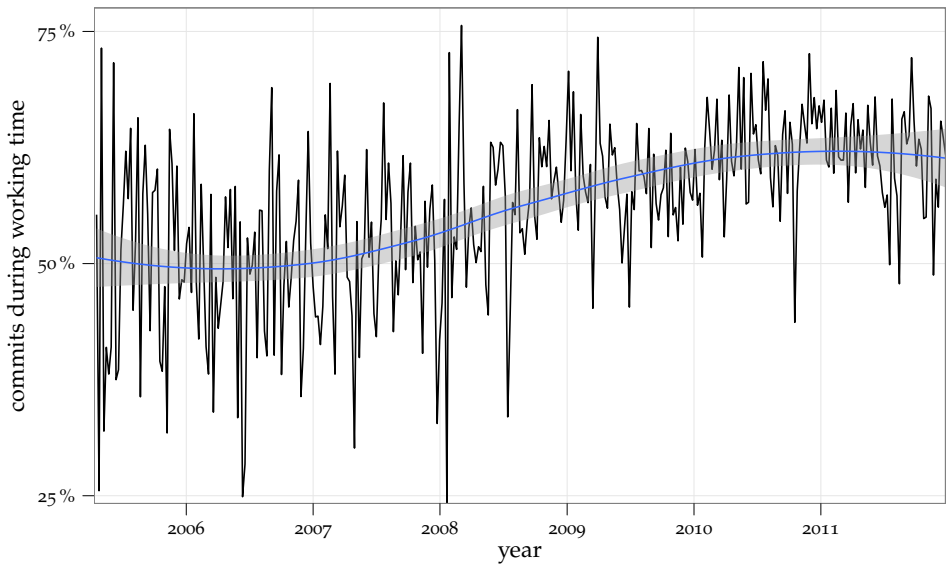
Even if the total number of developers with commit rights per week is – somewhat intuitively – lower compared to the total number of authors, a steep upwards trend is visible in Figure 3.10a. It shows an increase from under 20 committers in the first year of the usage of Git up to about 60 per week on average nowadays. A reason might be the partition of the Linux codebase in a hierarchy of various subsystems and even smaller administrative areas. This not only allows a better concentration of domain knowledge but also provides more options to gain commit rights for a part of the project.

Figure 3.10b is in line with Figure 3.8b shown earlier, since committers can only integrate as much changes as authors provide. However, the fluctuation between consecutive weeks is much higher than the author's one as the gray ribbon, representing the standard error, around the LOESS curve indicates. One has to keep in mind that code changes adding new features are only allowed during a short period after the announcement of a new kernel release, while over the rest of the period until the next release, only bug fixing changes are integrated. Consequently, the graph shows a low activity in general with regularly occurring high peaks. This will be discussed in more detail in Section 3.4.2.

3.3.3 *Trends for Both*

For inspection of an overall trend of the ratio between commits during working time and during spare time, Figure 3.11 depicts the combination of both previously distinct examined time series. Each of the fifty two values per year represents the average percentage of all commits performed during working time over the period of a single week.

Figure 3.11: Percentage of Commits during Working Time for *Authors* and *Committers* Combined on a Weekly Basis



Inspecting the blue LOESS curve, that represents a robust moving average in this context, one can observe that the overall number of commits during working time increases. The latter tops the value of 50% at the end of 2008. Since then, it remains above this value, with few exceptions, and, hence, higher than the percentage of commits during spare time. Consequently, one can deduce that the number of professional developers increased for both groups – authors and developers – and, thus, that the Linux kernel can be seen as highly backed by companies.

This is consistent with Corbet, Kroah-Hartman, and McPherson [CKM12], who state that only approximately 25% of all work on the Linux kernel is done by the groups "volunteer", "academia" or "unknown"; all other commits are done by commercial entities.

3.4 OVERALL TRENDS

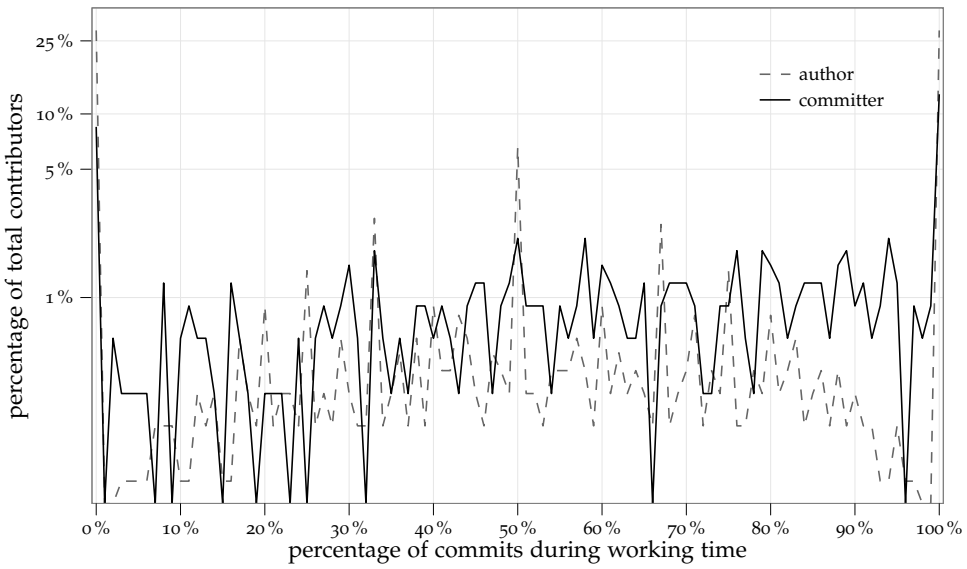
3.4.1 *Distributions*

In the following, trends over the total inspection period of the full seven years are analyzed.[6]

Figure 3.12 depicts the distribution of commits during working time. Here, authors are represented by a dashed line and committers by a solid one. The values on the vertical axis show the number of contributors having a specific

6 As a technical explanatory note, the presented numbers are based on all commits that changed at least one line and, hence, also include merge commits that integrate other repositories.

Figure 3.12: Distribution of Contributors per Percentage of Commits during Working Time



percentage of all commits at that level on the horizontal axis. Please note that the vertical axis is logarithmically scaled.

Intuitively one can see the peaks on both ends, indicating a large group of contributors working exclusively during spare time as well as a nearly similar sized group of developers contributing solely during working hours. For the group of authors, peaks for multiples of 25% as well as for multiples of 33% are visible; the curve for the group of committers does not show such clear peaks.

A reason for this might be that a large amount of the Linux kernel patch authors are just committing a rather small number of patches to the project, in contrast to committers taking care of the work of multiple authors as shown in Section 3.4.3. For example, assume an author contributed three changes over time: two were done during spare time and one during working hours (e.g. on vacation). In that case, the overall percentage of that person will be 33%, which explains the peaks at exactly these characteristic values and their multiples.

Table 3.2 provides a list of the number of contributors with respect to their commit behavior and, as such, shows how many of them worked mostly during spare time or during working hours. Please note that developers having exactly fifty percent of their work done during working hours are included in both groups here. Consequently, both rows in Table 3.2 add up to more than a hundred percent.

As one can observe, nearly thirty percent of all authors worked exclusively during their spare time. Including also all developers up to ninety percent spare time contributions does only enlarge this group slightly. The same is true for the other extreme, when looking for professional authors who only contribute during working hours (with a maximum deviation of ten percent). Approximately a third of all authors contribute with more than seventy five

Table 3.2: Cumulative Number of Contributors

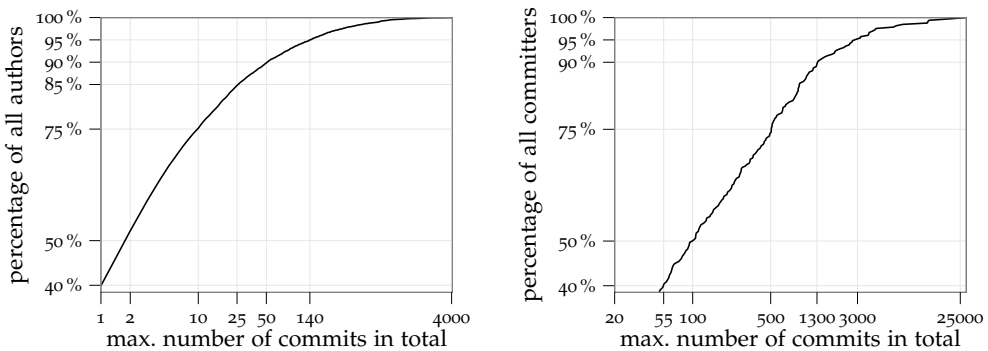
	% worktime	authors		committers	
Volunteers	0 %	2268	28.5 %	28	8.5 %
	⩽ 5 %	2293	28.8 %	33	10.0 %
	⩽ 10 %	2356	29.6 %	40	12.2 %
	⩽ 25 %	2767	34.7 %	60	18.2 %
	⩽ 50 %	4192	52.6 %	129	39.2 %
Professionals	⩾ 50 %	4303	53.99 %	207	62.9 %
	⩾ 75 %	2890	36.3 %	130	39.5 %
	⩾ 90 %	2388	30.0 %	73	22.2 %
	⩾ 95 %	2308	29.0 %	54	16.4 %
	100 %	2275	28.5 %	42	12.8 %

percent of their work done during spare time or during working time. Dividing the overall group in the middle, "spare timers" and "professionals" are close to equal in size.

In contrast to that, the group of committers performing at least seventy five percent of their work on the Linux kernel during their spare time reaches barely a sixth of the total number. On the other hand, nearly forty percent integrate changes in at least seventy five percent during working hours. A professionalization is confirmed by the shift towards it, visible when dividing all committers into two equal groups. Here, over sixty percent of all committers did their part on the Linux kernel evolution using at least fifty percent of their (here assumed) paid time. Interestingly, exactly forty two of them performed all their contributions exclusively during working hours.

Figure 3.13: CDF of Total Number of Commits per Contributor

(a) *Authors* (b) *Committers*



For getting a better feeling about the analyzed groups, Figure 3.13 depicts the Cumulative Distribution Functions (CDFs) of contributors by their total contribution to the Linux kernel. Please note that both axes are logarithmically

scaled for a better representation of the exponential behavior of the data at hand. Eventually each curve reaches a value of a hundred percent vertically and a lower percentage indicates how many contributors of the total group of authors or committers, respectively, have a less or equal amount of commits registered in the codebase.

The domain of the horizontal axis for authors, as depicted in Figure 3.13a, ranges from one commit up to a maximum of approximately 4 000 commits per person over the period of seven years. As one can observe, fifty percent of all authors made only one or two commits, with a majority of those having solely a single contribution (39.89 %, n=3179). Three quarters of all authors contributed ten or less commits to the success of the project.

The remaining quarter, however, contains a wide value range: Adding ten more percent to the already discussed group, adds all authors with twenty five or less commits; five percent more everyone with fifty or less. Thus, the top ten percent of all authors – still a group of 797 developers – are responsible for quite a few changes per person. However, half of them (now we are at 95 %) contributed less then one hundred and forty changes each. Eventually, there are eight authors with 2 000 to 3 000 commits in total, two have between 3 000 and 4 000 and two, David Miller and Linus Torvalds, contributed more than 4 000 patches per person.

When inspecting the same percentage ranges, as described above, for the group of committers, as shown in Figure 3.13b, there are interesting differences observable. Although the total number of developers with commit rights is smaller compared to the one of authors, the number of code integrations per person is much higher. Only 2.74 percent, or nine developers, have solely one commit assigned to them. Half of all committers have 100 or more contributions. Moreover, ninety five percent of all authors have less registered commits than forty five percent of all committers.

The upper quarter of all committers has made at least 500 changes per person during the investigated period of time. As one can observe, the topmost five percent, or seventeen committers, have over 3 000 contributions assigned to them. Of this group, five members are in the range between 3 000 and 4 000, six are between 4 000 and 10 000, and six committers supervised even more than 10 000 changes between 2005 and 2011.

However, when analyzing the amount of code changed by contributors, sorting them in the described groups by the percentage of work during working time, the values at the edges of Figure 3.14 are only marginal. Here, authors developing fifty four percent of their changes during working time are most influencing with 17.5 % of all changes made with that percentage are done during working hours. In contrast to that, committers have their maximum commit size contribution (23.1 %) at sixty three percent and furthermore a peak (15.7 %) at eighty five percent can be observed.

Figure 3.15 depicts the CDF of commit size changes. A single clear jump in authors' code changes located at 53–54 percent can be observed, while the parts in front of it and also afterwards are close to linear shape.

Figure 3.14: Commit Size Changes during Working Time per Contributor

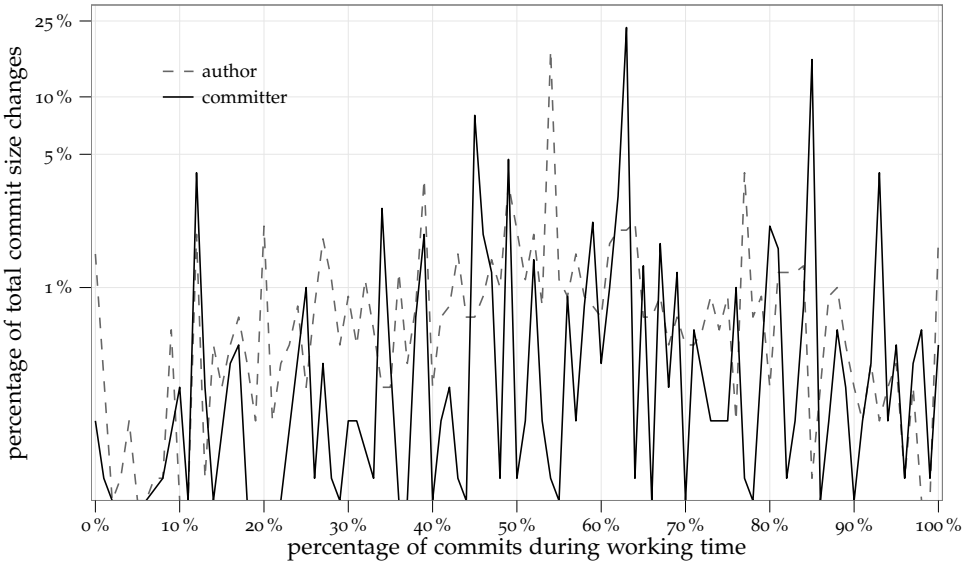
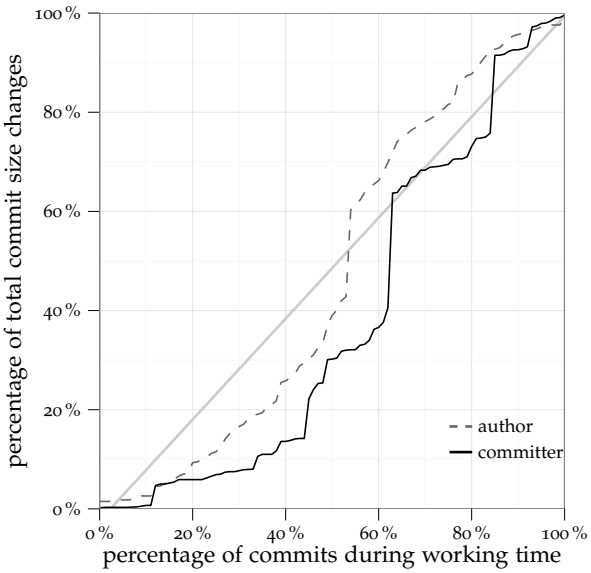


Figure 3.15: CDF of Commit Size Changes during Working Time per Contributor



In contrast to that, the latter fact is not true for the curve representing the committers' changes. Here, multiple increases are observable at the percentage values already described above. The activity remains low with only approximately 30 percent of all changes integrated from developers working primarily in their spare time on the project. There are two remarkable groups, integrating large parts of changed code, at 62–63 and 84–85 percent working time.

The overall percentage of changes introduced to the "official" Linux kernel repository of Linus Torvalds during working hours over the period of seven years is depicted in Table 3.3. As one can see, the numbers for committers are

Table 3.3: Total Percentage of Changes during Working Hours

	authors	committers
commits	52.1 %	60.8 %
code changes	52.1 %	61.3 %

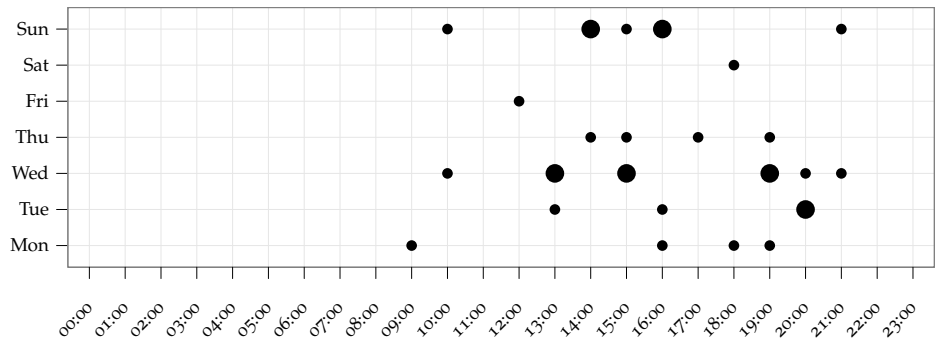
higher, indicating a larger professionalization in the sense of commercially paid development here as well.

3.4.2 *The Influence of Releases*

In line with the work of HINDLE, GODFREY, and HOLT [HGH07], the influence of releases with respect to commit behavior is analyzed in this section. The inspected period of development starts shortly before version 2.6.12 (announced 05/17/2005) and ends close to the end of version 3.1 (announced 10/24/2011, version 3.2 01/04/2012). The data at hand contains 30 releases in total, which is aligned to the Linux kernel's cadence of a new stable release every 81 days[7] on average [CKM12].

Figure 3.16 depicts the day and hour at which each of them was introduced to the codebase, using the concept of a punched or Hollerith card [Rou08]. Similar to the plots at GitHub[8], a dot represents a day and time when a new version was released, and the larger ones indicate that more than one release was announced at that time over the whole inspected period. As one can observe, the Linux kernel has no preferred time for its releases since they are spread over the whole week.

Figure 3.16: Time at which Releases were Introduced



The following figures consist of a gray area indicating the median of all committers' contributions in these releases over the time of 60 day before and after a release. The period of 60 days was chosen due to the fact that it is the

7 For the inspected time frame between release 2.6.11 (03/02/2005) and release 3.1 (10/24/2011) the development time per release averages 80.94 days.

8 See https://github.com/blog/1093-introducing-the-new-github-graphs (visited on 08/27/2012).

minimum of all release windows during the inspected time frame (cf. [CKM12]). In addition, a solid line shows the number of commits, again as a median over all releases, that were contributed by authors. The maximum of each group is visualized by a light blue horizontal line for better visibility that is solid for committers and dashed for authors. The release date is centered and depicted by a dashed, red vertical line. In addition, a dotted, red vertical line indicates the end of the two week merge window in which new code can be integrated into the code base [Coro8].

Figure 3.17: Average Commits (daily)

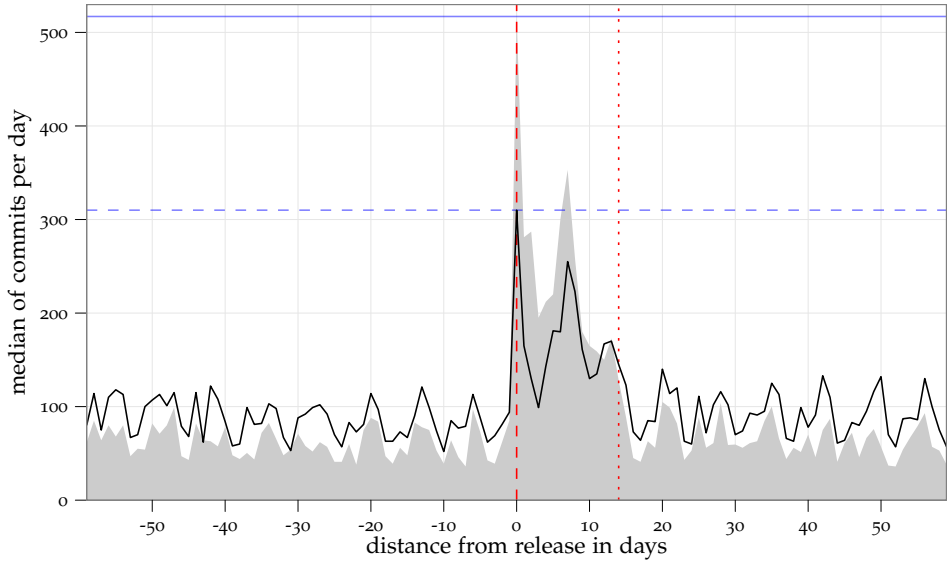
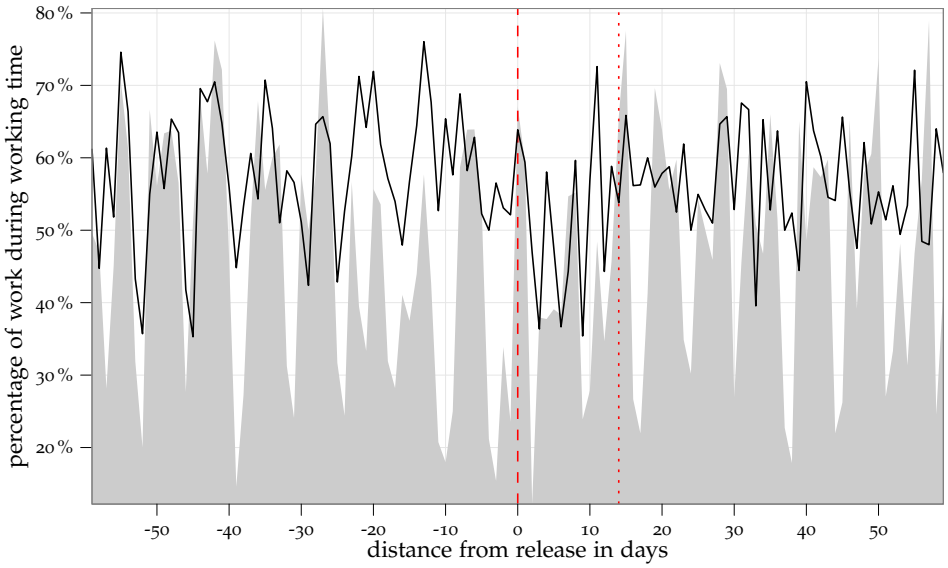


Figure 3.17 depicts the number of commits contributed per day. Before and after the merge window, the activity of authors (solid line) remains high with an average of 89 commits per day. The committers' activity (grey area) during this periods is lower with only 63 commits per day on average. Since the number of changes per release submitted by authors and integrated by committers is equal with only few exceptions, this relationship consequently swaps during the fourteen day period where new code is allowed to be introduced to the codebase. On the first day of it, on average over 500 changes were integrated per release, indicated here by the median for robustness reasons. Over the next fourteen days the committers' activity remains high with 238 commits on average, with a second peak of nearly 350 commits on average exactly 7 days later.

Also the number of authors' commits rises on the first day but not as much as for committers, reaching approximately 300 commits. A reason for that might be that authors keep their changes locally in private repositories and propose them as patches for integration on the mailing list as soon as a new release is announced. Then, the activity drops again to normal level, before reaching a second peak similar to the one described for committers before. An potential explanation for both is the increased testing and fixing of integration issues at that phase, which needs some time. Furthermore, weekends in between that

result in less development activity as shown in Section 3.2 might play a role here, even if new releases were introduced on every day of the week. In total, authors submit 173 commits on average per day during merging time.

Figure 3.18: Average Percentage of Commits during Working Hours



When inspecting patterns regarding the ratio of commits during working time and spare time per day with respect to release windows, as depicted in Figure 3.18, no clear shape can be observed, neither for authors' nor for committers' activities.

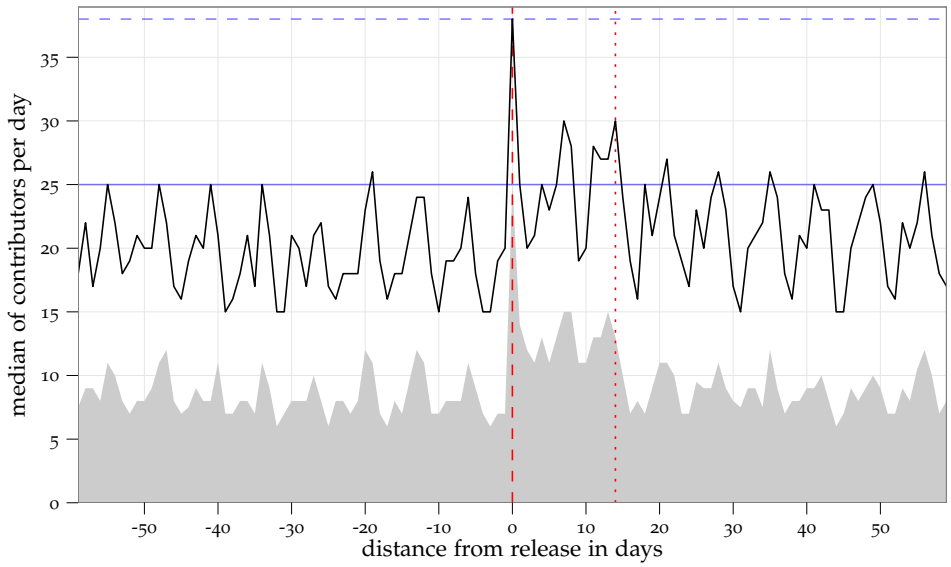
However, both curves drop significantly during the merge window, which might be due to extensive testing around the clock during that period. Here, the average percentage for authors is six percent lower and the one for committers drop by four percent compared to their respective values outside the merge windows. In addition, during the four days before a release is announced, nearly all changes were integrated by committers during their spare time; although this number is quite low as visible in Figure 3.17.

The number of contributors per day is depicted in Figure 3.19. The number of authors, still represented by a solid line, is continuously approximately twice as high as the number of committers, represented by the gray area. Interestingly, this fact remains its validity also during the merge windows. Furthermore, a correlation between both developer types can be noticed as peaks of committers' involvement appear at the same day or one day after the number of authors was high.

3.4.3 *Ratio Authors per Committer*

In contrast to centralistic VCSs, newer distributed solutions keep not only information about the committer of a checked-in piece but also about its author (cf. Section 2.1.1). Consequently, this fact allows the examination of how the

Figure 3.19: Average Number of Contributors



ratio changed over time. In a more generalized form the inspected question is, how many individual authors a single committer supervised by applying the patches of them after review.

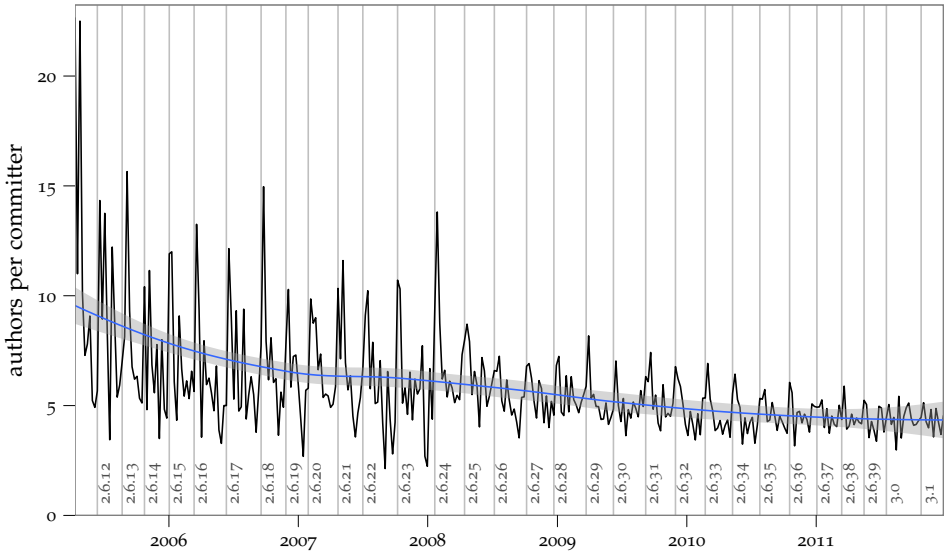
On the following page the changes of this ratio are examined for the Linux kernel project. Furthermore, the results of this might then be used for extrapolation, allowing an estimation of the number of involved authors of projects using Version Control Systems without such a distinction such as CVS or SVN. This becomes relevant, when analyzing a large amount of different FLOSS projects in Chapter 4 of this thesis.

Since Git stores all relevant information for this type of analysis, the ratio of involved authors and committers can be calculated for each point of time by counting the corresponding contributor information stored with each commit. Afterwards the two values, the number of distinct authors and the number of distinct committers, respectively, are divided to get the ratio.

Figure 3.20 depicts the result when aggregating theses values on a weekly basis. Similar to the previous sections, the light blue line plotted on top of the graph represents the result of a LOESS function which in simple terms shows the average of the data. Gray lines in the figure represent the dates of releases. As a result, a strong correlation between releases and a rise of authors per committer shortly afterwards is visible.

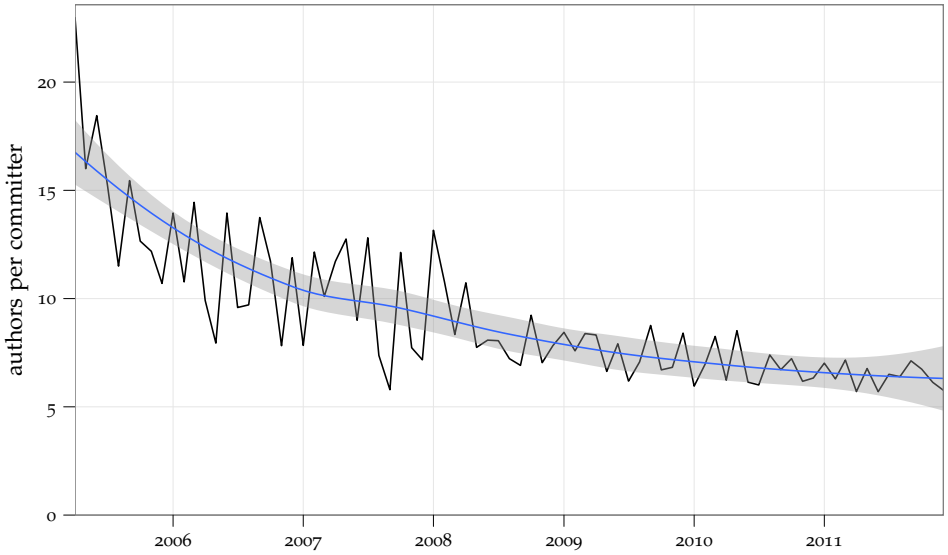
Although, only bug fixes are allowed to enter the official codebase after the merge window, the development on the author side does not stop over the following time. Shortly before a new stable release, the amount of authors' commits drops significantly, indicating on the one hand the code freeze for stabilization and on the other hand providing a direct evidence that patches are restrained until the new merge window is opened.

Figure 3.20: Ratio of *Authors* per *Committer* (weekly)



Interestingly, release candidates seem to not have a high influence with regard to this ratio, supporting the aspects already discussed in Section 3.4.2.

Figure 3.21: Ratio of *Authors* per *Committer* (monthly)



When further reducing the granularity down to a monthly timespan, a conspicuous monotonically decreasing trend of the LOESS curve becomes clearly visible. Also the amplitudes are decreasing and after 2008 only minor deviations from the curve are visible. The number of authors a single committer has to deal with on average falls below ten per month.

A reason for that pattern might be the increasing hierarchization of the Linux kernel project. By introducing ancillary repositories for single subsystems, it

is not only possible to concentrate technical know-how in a specific topic but it also allows to provide more people the rights to evaluate code changes and add them to their copy of the repository. After a while, new code changes are integrated via `pull` requests into the corresponding repository on the next hierarchy level if the maintainer is trusted until it finally ends up in the official one of Linus Torvalds. The information related to a commit are not changed during that "journey" of a commit so that – regardless of how many intermediate stations were involved – the original author and the first committer are preserved [Cor09].

4

OPEN SOURCE WORK RHYTHMS BASED ON OHLOH DATA

4.1 DATA PREPARATION

4.1.1 *Data Source*

In this chapter a large number of open source projects is analyzed. The data was collected by the Ohloh webservice (`http://www.ohloh.net/`), nowadays part of the company *Black Duck Software*, that tries to provide statistics for projects and developers in the area of Free/Libre/Open Source Software (FLOSS), as a way to provide more visibility into software development [Tafo6; Wau10].

For its analysis, Ohloh downloads the project repositories and evaluates the data stored in them using its own toolchain[1].

The web-service started in 2006 with selected projects using the Yahoo! search engine in-links as the main measure [ARo9b]. From that point on, new projects were added by the website's users. The anticipated aim behind that is the collection of all popular FLOSS projects eventually. In an internal mailing conversation June 2007, one of the representatives states:

> In spring of 2006, we seeded the database with about 3000 of the most popular projects from SourceForge, Java.net, RubyForge, and GNU Savannah. We manually added some key projects like Apache, Mozilla, OpenOffice, and the Linux Kernel. Our goal was to quickly cover as many high-profile projects as we could, so that the most common searches on our website would be likely to find a match. Then we unleashed the public on the system, and they've added whatever they pleased.

Ohloh only analyzes projects where the source code is publicly available and managed using one of their supported VCSs; projects that do not fulfill these requirements are ignored.[2] The internal data is updated on a weekly basis by

1 Based on `http://meta.ohloh.net/2007/09/new_subversion_downloader/`, all project were stored and prepared for analyses in Git repositories until 2007. After that SVN repositories were synchronized with built-in functions. In addition, the company uses their own software for source code analysis, `Ohcount`, that was open-sourced in 2008, as written in `http://meta.ohloh.net/2008/01/ohloh_goes_open_source/`. (Both sources were last visited on 07/24/2012)

2 E.g. a blog post at `http://meta.ohloh.net/2006/12/missing_repository/` (visited on 08/22/2012) states: *Ohloh only offers development metrics to projects with openly-accessible source control systems. That way our analyses can go back in time and get a clear picture of how the project was built. Even with only CVS, Subversion and Git source control system support, we get pretty good coverage. However, it's not universal. Given our limited development resources, we'll skip supporting non-repository projects for now.*

downloading new revisions and re-calculating the statistics for each registered project. In addition, commits information preceding the date of when the project was added at the webservice are as well taken into consideration, if the version control provides such historical data.[3]

The database acting as source for the presented analyzes in this thesis is an extraction of the Ohloh database that was made available during the first quarter of 2008. The projects in it represent a wide spectrum of all areas of open source, with no apparent bias towards any particular source or hosting provider. As shown in previous work, the inspected data represents roughly 30 % of the active FLOSS projects during that time [ARo9a].

The company provided a comprehensive extract of their own database in that all user data was anonymized. It allows analyses on a more fine-granular level compared to the options provided by the public web-interface[4], that is available nowadays, and, in addition, has no daily limitation of requests like the latter. Since then, Ohloh has been acquired by GeekNet, the provider of SourceForge, in 2009[5] and by Black Duck Software in 2010[6]. As Conklin, Howison, and Crowston [CHCo5] point out, providing such dumps is not a costless process, as personally identifiable and financial information must be stripped out before delivery. Maybe due to that, no newer database extract of the Ohloh data was made available to the university chair since then. However, it is time consuming and expensive for a research group to build a comprehensive and representative data set by themselves for all open source (the author has no knowledge of any successful attempt). There exist various collaboratively created databases, such as e.g. FLOSSmole [HCCo6]. However, using data from centralized VCSs of multiple sources, these sources do also cope with ambiguous developer identifications[7], and do not provide data that is detailed enough to calculate and provide the insights presented here [RGo9].

Table 4.1: Number of Repositories by VCS

Type	Repositories	
SVN	10 062	(49.1 %)
CVS	9 202	(45.0 %)
Git	1 201	(5.9 %)

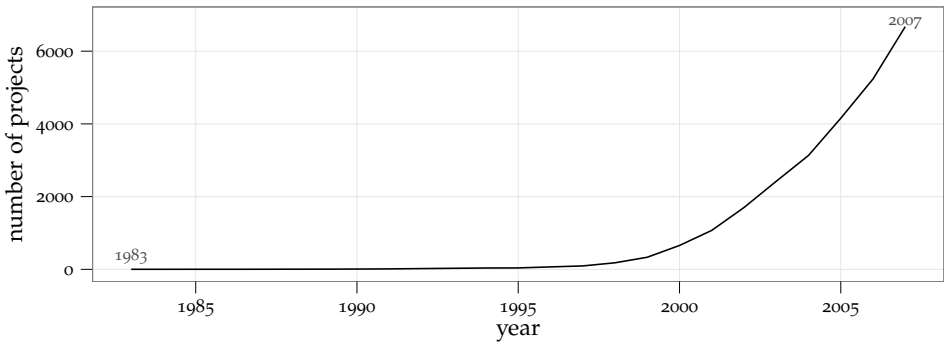
As depicted in Table 4.1, the projects under observation were nearly exclusively managed at that time using the centralized version control systems CVS and SVN (one project might have multiple repositories). Due to that reason, the tables storing contributor information were designed on the least common denominator, namely data that CVS provides.

3 http://meta.ohloh.net/2007/08/worlds_oldest_source_code_repositories/ (visited on 08/22/2012).
4 http://meta.ohloh.net/getting_started/ (visited on 08/21/2012).
5 http://meta.ohloh.net/2009/05/sourceforge_acquires_ohloh/ (visited on 08/02/2012).
6 http://www.blackducksoftware.com/news/releases/2010-10-05 (visited on 08/02/2012).
7 Citing Conklin, Howison, and Crowston [CHCo5]: *Different forges can have projects with the same names; different developers can have the same name across multiple forges; the same developer can go by multiple names in multiple forges.*

Consequently, there exist entries for committers in the database but no author relationships are stored even if they would have been available, e.g. when analyzing Git repositories. It can be assumed that the ratio between authors and committers dropped over time, as shown for the Linux kernel project in Section 3.4.3. However, this cannot be verified due to the fact that some of the repositories are not available anymore today, as well as that no author data is stored in centralized systems and a manual analysis of nearly 20 000 distinct projects would take too much time to be feasible.

The database engine used internally is `PostgreSQL`, storing the various tables. For analysis, the statistical software `R` [R12] is used that gets access to that database with the help of the `RPostgreSQL` package [Con+12]. This allows to do pre-aggregations, such as joining tables, using the Structured Query Language (SQL) [ISO11] on the database side before downloading the results to a local machine for further examination.

Figure 4.1: Number of Distinct Projects per Year



While the database contains commit information back to 1983, this thesis only considers commits after the turn of the millennium as relevant due to the fact that the number before the year 2000 is too small to be representative, as depicted in Figure 4.1. Since the last entry in the database is from February 12th 2008 and, thus, the database only contains a minor part of that year, December 31st 2007 was chosen as the upper bound so that only full years are evaluated. As a result, the data under inspection are 7 811 365 individual commits of 45 870 distinct committers contributing to 9 065 projects over the period of 8 years. In contrast to that, 476 165 commits, representing approximately 5.75% of the total number of entries, are ignored as they are outside of the inspected time frame.

4.1.2 *Data Limitations*

The Ohloh website stores only information about commits that change code. Commits that touch only e.g. images or text files without any programming source code visible for Ohloh's internally used tool `Ohcount`[8] are ignored and

8 Repository at `https://github.com/robinluckey/ohcount` (visited on 07/19/2012).

no entries are created in the database. Consequently, the analysis presented here shows only the behavior of developers involved in FLOSS projects and, thus, might have a slight deviation from the overall trend.

Furthermore, in case that a project switches its repository, e.g. when changing the used VCS, the database keeps both sources and, consequently, collects data and statistics for both repositories. In cases where both sources are kept in sync, the same data is analyzed and stored twice. The topic of data quality is addressed in more detail in Section 4.1.3 below and by Hoffmann [Hof12].

Many of the inspected projects went through an evolution not only of their code base but also by their choice of the used Version Control System. Therefore, despite duplication, another issue with the data source at hand is the fact that some history is already lost forever, in cases where it was not migrated into the new VCS and the old source was deleted before the project was registered at Ohloh. However, the analyses performed in this thesis are focused on recent data, so that the loss of ancient commit history is acceptable.

Another limitation is the assignment of commits to their contributors. Many projects use abbreviations or nicknames as usernames. While some of the latter might be unique, in general these are highly ambiguous, as there exists more than one developer on the world that has e.g. the name *John*. Ohloh tackles this issue by providing an option on their web-interface to claim the ownership of this username. After a registered user clicked on that, the contributions are linked with his or her account. Unfortunately, not all authors of every project registered at Ohloh are members of this webservice and, consequently, the data contains multiple false positives by assigning the data of distinct programmers with the same credential to the same (virtual) user account. As a result, e.g. the username *root* has 182 projects assigned. However, the analyses performed here do not address individuals but try to provide a general overview on when FLOSS is being developed. By filtering out those obvious ambiguities and aligning the unknown ones (cf. Section 4.1.5) most hurdles circumvented.

Even with the limitations mentioned, the database at hand represents the best research source available at the moment.

4.1.3 *Data Cleansing*

The Ohloh data taken from the repositories of various projects and several types of version control systems are aggregated in a central `PostgreSQL` database instance. For normalization purposes it is segmented in multiple tables (52 in total).

As a result of that, a simple mapping between the data of Ohloh's registered users and the account data of the various repositories is not possible. Consequently, the relationship between a contributor and his or her metadata has to be built by a rather complex SQL query, which can be found in listing A.1 in the appendix.

Figure 4.2: Diagram of the Used Database Tables

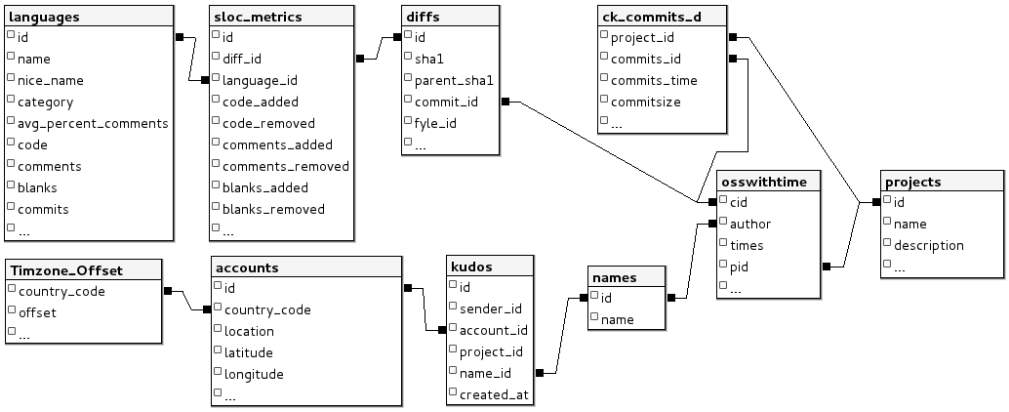


Figure 4.2 shows the relevant database tables for this thesis, of which the table with the name `osswithtime` represents the base for this analyses as it contains an aggregation of the data of all commits as well as their related timestamps. The database table `accounts` contains the meta data of Ohloh's userbase. One of the existing columns of it stores location information – if a user provided it.

Unfortunately for this analysis, the data is entered in a free form field on the website without any validation of the entered text and consequently it is quite error-prone. There are empty strings that only contain zero to up to 8 space characters and location information such as "undefined" that are invalid. Such values are filtered out and are replaced by the data type `NULL` so that their are ignored during the further processing. The same cleanup is done for the other relevant columns in the query result.

During the same SQL query, the timestamp of each single commit in the database is converted into POSIX time format, i.e. the number of seconds since January 1st 1970, as described in Section 2.1.3.3. The reason for that step is that, while the data is internally stored in `PostgreSQL` as UTC timestamp in a column of type *Timestamp without Time Zone*, the output is converted into the local time zone of the requester [PG12a]. The query contains a construct[9] that converts the date into the corresponding integer values instead of the otherwise as default returned string representation and thus keeps the time information in the UTC time zone.

Moreover, this step was necessary since the `RPostgreSQL` plugin [Con+12] that is used for the communication between the database and `R` [R12] has a bug under Microsoft Windows that leads to the fact that the returned time information is cropped in some cases. In the latter, only the date would be available for analysis which would not allow a precise evaluation using the Windows operating system. While this problem does not exist with Unix-like systems, all scripts are required to work on all platforms for further research. The usage of this conversion into integers, hence, solves two issues at once.

9 The exact SQL syntax is: `EXTRACT(EPOCH FROM table.column AT TIME ZONE 'UTC') AS alias` (see sections 9.9.1 and 9.9.3 of [PG12b]).

4.1.4 *Filtering of duplicated data*

After extracting all data from the existing database tables, transferring it to a local workstation, and loading it into an `R` instance, a filter is used to remove duplicated entries that might be in the data. Reasons for that might be e.g. multiple imports due to changes of the underlying version control system when old data was migrated to the new one. As a selection criteria all instances are filtered out where author, timestamp of a commit, as well as the project id – which distinguishes the various stored projects under inspection – are match with another entry.

In addition to the database tables provided by Ohloh additional ones were created, containing information about the multiple time zone offsets (cf. Section 2.1.3.1). Furthermore, they provide a mapping of what countries use which UTC offset. Downloading a copy of that tables thus allows to look up based on his or her metadata in what time zone a user is located.

However, as described above, not every user of an FLOSS project also has an Ohloh profile, or if he or she does, might not have provided that information. In cases, that the user's metadata exists in the table `accounts` a query extracts the country and – in cases where a country spans over multiple time zones – also the state by parsing the string of the related entry's column.

After that step, potential errors are corrected. For example, the used regular expression assigns for the city *Washington DC, USA* the state `DC`. Even if the *District of Columbia* officially is not a part of any US state, for simplicity reasons the data is substituted by the abbreviation of Maryland since it is located in that state and, thus, has the same time zone offset. Moreover, some users are identified manually based on their usernames used in the involved projects. The time zone information of these ones is also adjusted here.

As a result, there is a segmentation into two groups of users: (1) those where the time zone offset could be found based on the stored user information or their username, and (2) as a second group those whose UTC offsets are unknown. The number of commits for the former contains 646 705 entries of 580 users while the latter, i.e. that with unknown UTC offsets, consists of 7 164 660 entries of 45 290 users.

4.1.5 *Calculation of Missing Time Zone Information*

As described in Section 4.1.4, there exists a group of users for whom the time zone difference is not known so far. In the next step, which one might call "educated guessing" and that is described in the following, each of those entries is compared to every known user's behavior.

Based on each person's commit behavior compressed into one week (cf. Section 3.2.1) there are daily peaks dependent on an user's contribution. After normalizing that data, the value 1.0 is assigned to the highest peak of an user and a corresponding fraction of it to every lower value. Thus, all values are

in a range between 0.0 and 1.0 after this step, so that different developers are comparable independently of their total number of contributions.

Afterwards, each individual "behavior curve" is compared by performing a complete enumeration over the same information for every user in a known time zone and calculating the least square distance while the input data is shifted in steps of 30 minutes in a range of ±12 hours. Consequently, each user pair has 48 result values from which the one indicating the lowest differences is returned together with the related offset information.

Finally, it is calculated which UTC offset was returned most often. The latter is then assigned for that particular user, since – based on the law of large numbers [ES10] – it represents the most likely time zone offset.

Subsequent tests with arbitrary values confirmed that more complex algorithms were not need, since the fitted data substantially matched the reference values.

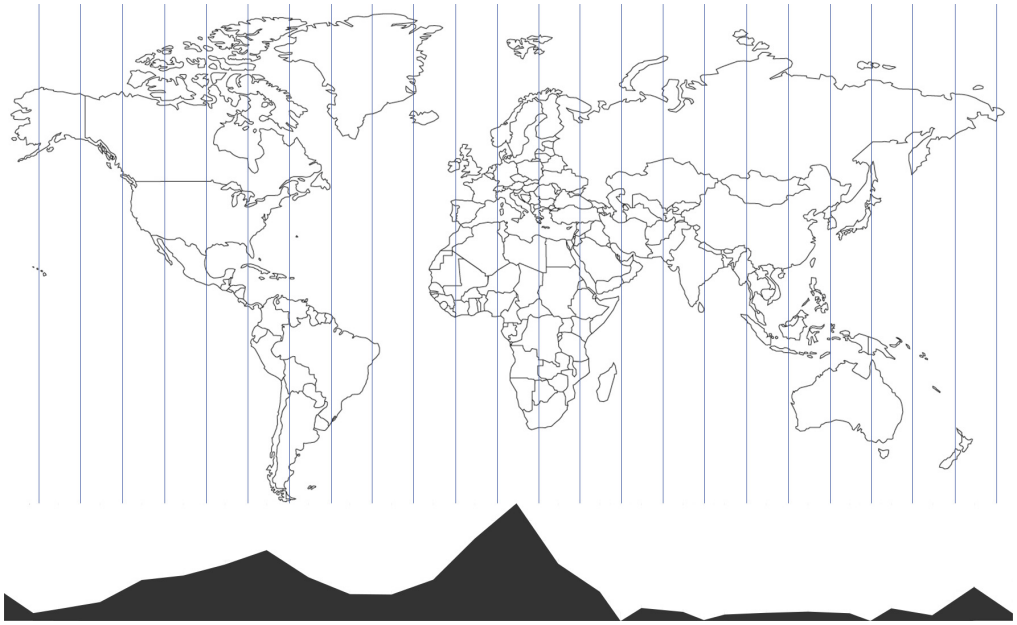
Table 4.2 shows a detailed overview how many contributors are allocated to the distinct UTC offsets while Figure 4.3 depicts the same information graphically on a world map.

Table 4.2: Number of Users per Time Zone in Ohloh

UTC offset	Number of Users		UTC offset	Number of Users	
-12	108 150	(1.38 %)	+2	571 314	(7.31 %)
-11	53 610	(0.69 %)	+3	292 706	(3.75 %)
-10	119 651	(1.53 %)	+3.5	548	(0.01 %)
-9	189 354	(2.42 %)	+4	130 222	(1.67 %)
-8	408 852	(5.23 %)	+5	91 780	(1.17 %)
-7	453 587	(5.81 %)	+5.5	10 237	(0.13 %)
-6	563 183	(7.21 %)	+6	65 501	(0.84 %)
-5	705 957	(9.04 %)	+7	82 576	(1.06 %)
-4	436 968	(5.59 %)	+8	94 324	(1.21 %)
-3	268 474	(3.44 %)	+9	78 140	(1.00 %)
-2	264 630	(3.39 %)	+9.5	2 360	(0.03 %)
-1	412 132	(5.28 %)	+10	126 985	(1.63 %)
0	821 995	(10.52 %)	+11	56 797	(0.73 %)
+1	1 171 846	(15.00 %)	+12	229 486	(2.94 %)

While the group of users with known offsets also contains information about whether they live in a country using DST or not (cf. Section 2.1.3.1), the contributors whose offset is calculated in that way do not. This leads to the minor inaccuracy that a contributor of the latter group who lives in such a described country and worked primarily in summer would be in another time zone (e.g. CEST=UTC+2) than his or her colleagues from the same country who worked primarily during winter months (e.g. CET=UTC+1). Thus, DST is ignored for all calculated offsets.

Figure 4.3: Distribution of Users per Time Zone after "Educated Guessing"



4.2 WEEKLY WORK PATTERN

4.2.1 *Hourly Commits Condensed into One Week*

Similar to Section 3.2.1, the overall activity over the whole inspected period is analyzed in the following by condensing all activities into a single week. Since the data at hand just provides information for committers, but not for authors, only those contributors can be depicted in Figure 4.4. Furthermore, Figure 4.5 shows the data by taking solely commits of those contributors into account for whom the location information could be found in the database.

Both, Figures 4.4 and 4.5, show the combined number of all commits made over the period of eight years in all projects. They are summed up on an hourly basis per day of the week. Consequently, the depicted time frame in both figures ranges from Monday 00:00 until Sunday 23:59.

When eyeballing over these figures, one can see a similar pattern to the one visible in Figures 3.1 and 3.2 for the Linux kernel. Like there, a pattern seems to be repeated for every day of the working week for the Ohloh data. It consists of a steep growth with a small drop around lunch time. Afterwards, the number of commits rises further and peaks in the afternoon before it drops significantly. In the evening a small increase of activity can be observed again before a majority of committers goes to bed, indicated by a number of commits close to zero at that time. The evening activity decreases with every day that the week goes on.

The peaks on the weekend in Figure 4.4 are about a third of the ones on Monday to Friday, also similar to the Linux kernel patterns. However, when inspecting

Figure 4.4: Work Distribution of *All* Contributors

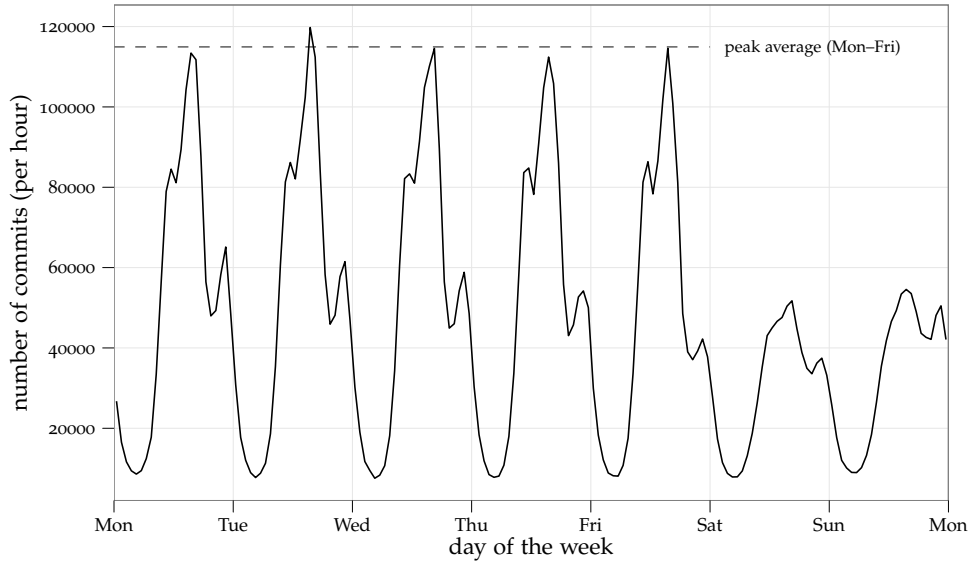
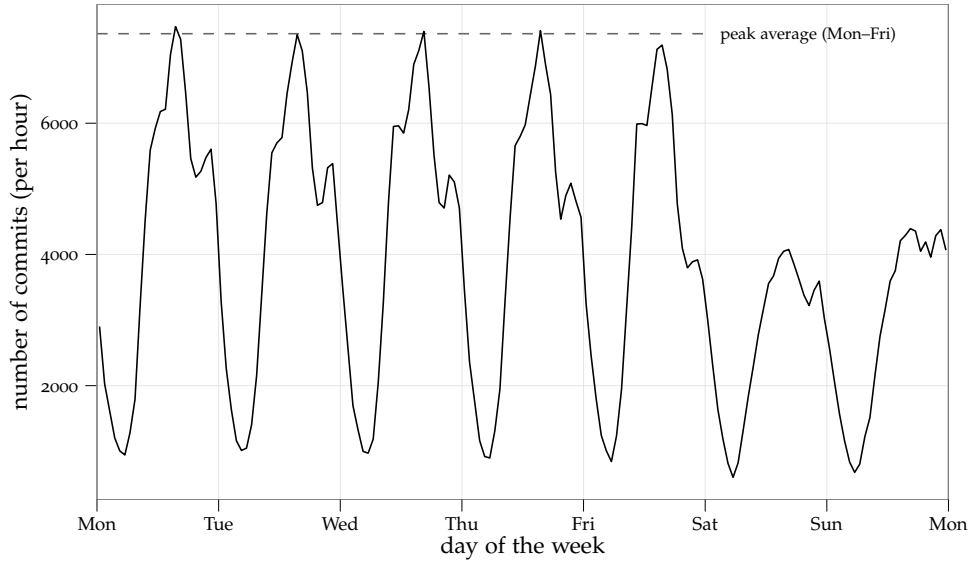


Figure 4.5: Work Distribution of Contributors *with Known Time Zone*



only those committers for whom the time zone information is known for sure, as depicted in Figure 4.5, the peaks at weekend are more than a half of those during the working week days. Consequently, the relative amount of developers working on Saturdays and Sundays, and thus in their spare time, is higher in the latter group. This is also visible on the other days, as – while the working week pattern shows the same rhythms – the daily curves are wider and the evening peaks are higher compared to Figure 4.4.

Again, the average of all peaks from Monday to Friday is depicted in both figures with a dashed lines, as already done in Section 3.2.1. In contrast to the latter, the relative differences here seem to be much smaller since all peaks are

close to that lines. Furthermore, hourly specifics, as e.g. a drop on Thursday afternoon, are not observable.

Table 4.3: Absolute Number of Commits per Day of Week

day of week	all		with TZ data	
Monday	1 279 139	(16.4 %)	104 413	(16.2 %)
Tuesday	1 291 091	(16.5 %)	103 276	(16.0 %)
Wednesday	1 276 207	(16.4 %)	104 292	(16.1 %)
Thursday	1 253 813	(16.0 %)	102 263	(15.8 %)
Friday	1 178 918	(15.1 %)	97 211	(15.0 %)
Saturday	727 424	(9.3 %)	65 232	(10.1 %)
Sunday	804 773	(10.3 %)	70 018	(10.8 %)
Total	7 811 365	(100.0 %)	646 705	(100.0 %)

In general, due to the high number of commits, the curves are smoother then the ones depicted in Section 3.2.1. However, the similarity to the Linux kernel project is highly interesting, since here not only a single but several thousand projects and approximately 27 times more commits are inspected at once. The total numbers of commits for each day of the week are shown in Table 4.3.

When looking at that numbers, one can observe that most commits are done during the first four days of the week. The general activity on Fridays is also high but, with approximately 15 %, slightly lower compared to the other days of the working week. Of all time, the weekend represents around 28.6 % of a 7-day-week; during that time, contributors of the projects registered at Ohloh submitted 19.6 % of their commits, while the smaller group, for whom the time zones are known, submitted 20.92 %. In comparison with Section 3.2.1, these values are mush higher than the 14.3 % and 13.3 % observed there. Consequently, the relative numbers of commits during working week are lower and in general the values are more homogeneously distributed. Based on this fact, it also can be concluded that the general percentage of spare time developers in the Ohloh dataset is higher compared to the one for the Linux kernel project.

4.2.2 *Hourly Commits per Day*

In the following, the week depicted in the previous section is split into its days, allowing an analysis on a 24-hour basis and to show similarities of the distinct days by plotting a line for each of them. Now, areas where similar patterns occur have their data points at the same coordinate and, thus, close to similar line section.

Again, Figure 4.6 shows the data of all contributors while Figure 4.7 only depicts the data of those whose location was stored in the database.

Both figures differ only slightly in their overall shape and are also close to Figure 3.3 shown for the *authors* of the Linux kernel. Due to the diversity of

Figure 4.6: Commits of *All* Contributors per Hour

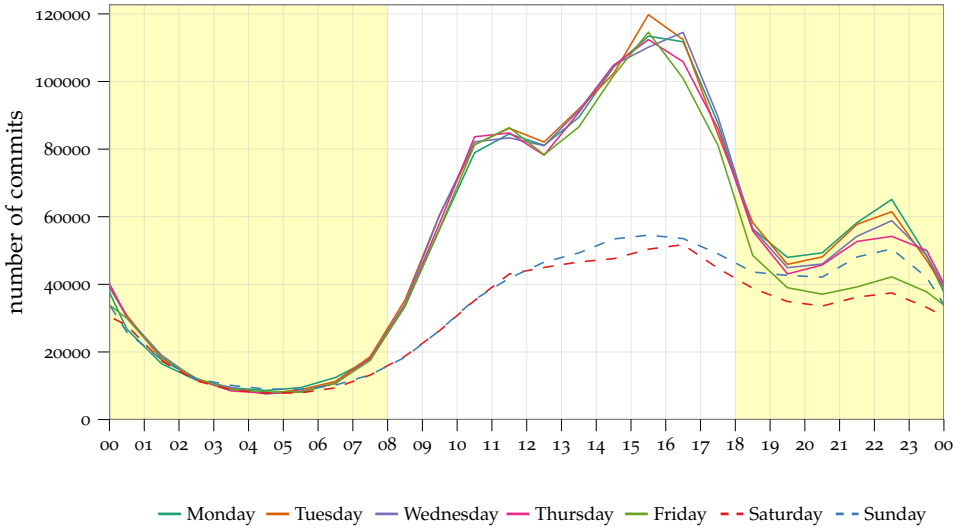
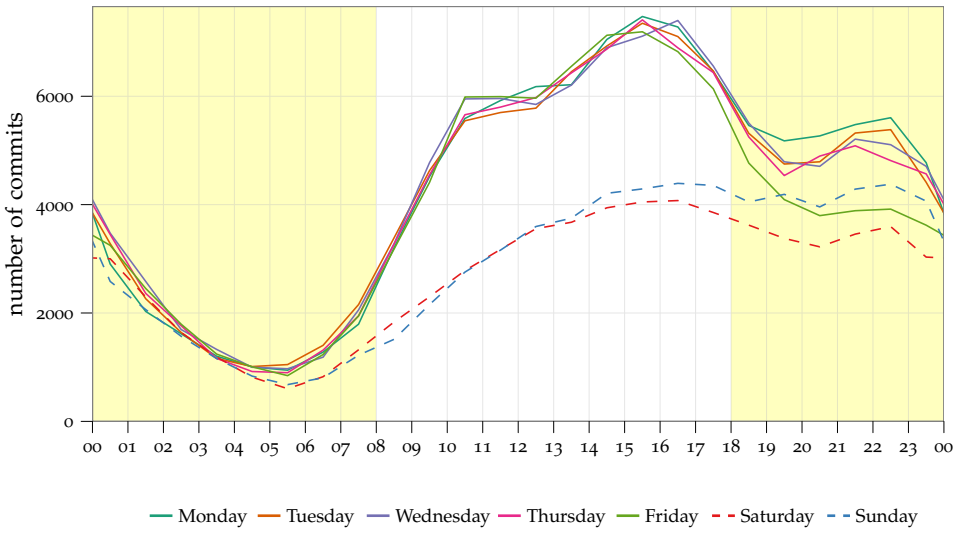


Figure 4.7: Commits of Contributors *with Time Zone Data* per Hour



projects and the resulting higher amount of data, the curves are smoother and more resistant to work rhythms of individual entities (cf. Section 3.2.2). The major work period shown here ranges from 10:00 in the morning until 18:00 in the evening and, thus, supports the defined working time of Section 2.1.3.4.

While the Ohloh data only provides committer information but none for authors, the pattern depicted in Figures 4.6 and 4.7 are closer to the latter than to the former in comparison with the Linux kernel. A reason for this interesting

observation might be the high amount of projects managed with centralistic VCS where contributors have to submit their changes to the central server before they can continue working on the next work item.

Another result of the high amount of commits is the fact, that the lines representing the days of the working week are all on top of each other, except for the hours between 19:00 and 00:00. One can observe that the curve on Friday falls earlier and deeper than the ones of Monday to Thursday. This is in agreement with working patterns in the western countries where the weekend starts already at Friday afternoon, as discussed in Section 3.2.1.

Also both curves representing the days of the weekend are similar to each other. However, in contrast to the ones of the Linux kernel, here, the activity is higher on Sundays. The two lines split at around 11:00, and in the evening hours between 18:00 and 23:00 there are even more commits per hour on Sundays than on Fridays.

When comparing Figure 4.6 directly with Figure 4.7, the previously described differences in the evening hours are catching one's eye. The group of developers with known time zone information shows not such a steep descent, instead the number of commits between 18:00 and 23:00 also stays high. As a result, the latter group consists – relatively – of more commits of spare time developers than when looking at all of Ohloh's data. Since the known committers are a strict subset of all committers, they are also included in that group.

4.2.3 *Commit Size Changes per Hour*

In this section, the commit sizes are analyzed, based on the definition in Section 2.1.2, to provide an overview if the number of commits (previous section) and the number of changed lines of code (this section) correlate. To do so, Figure 4.8 depicts the consolidated commit sizes per hour for all contributors, while Figure 4.9 depicts the same for the subset of those committers with a known time zone.

In general, both figures correspond to the ones shown in the previous section. Especially the sample size for all contributors seems to be too large to show individual patterns, as the two figures are identical in their shapes.

The number of commit size changes per hour for the group of developers with known time zone information is approximately 1.5 times larger than the one observed for the Linux kernel data. Nevertheless, it still seems to be of a size that allows to see several differences in the contributor's behavior:

- Firstly, the number of commits on Monday evenings is higher compared to the other days of the working week, while the commit size changes are on the same level for that days.
- In addition, there is an obvious peak of code changes in Figure 4.9 on Thursdays between 15:00 and 16:00 that cannot be seen when looking at the number of commits. Interestingly, this is exactly that time where the

Figure 4.8: Commit Size Changes by *All* Contributors per Hour

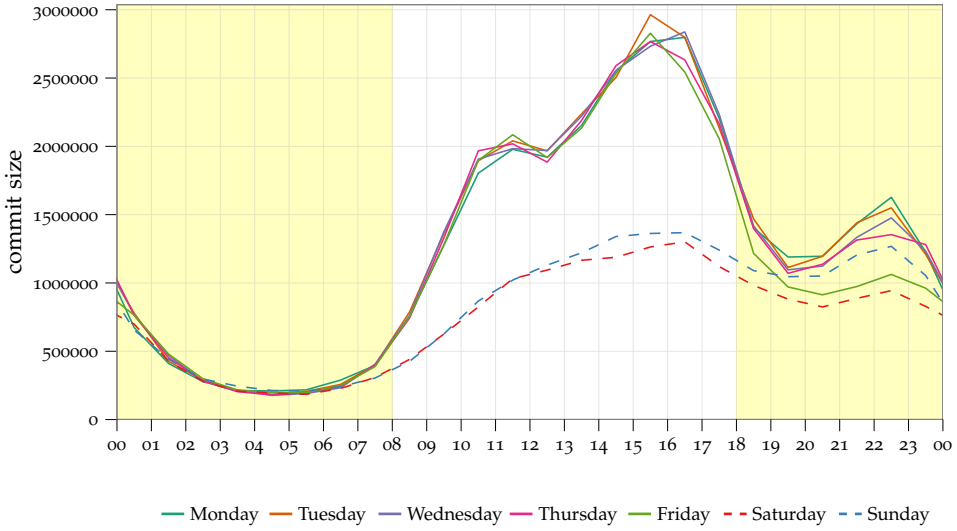
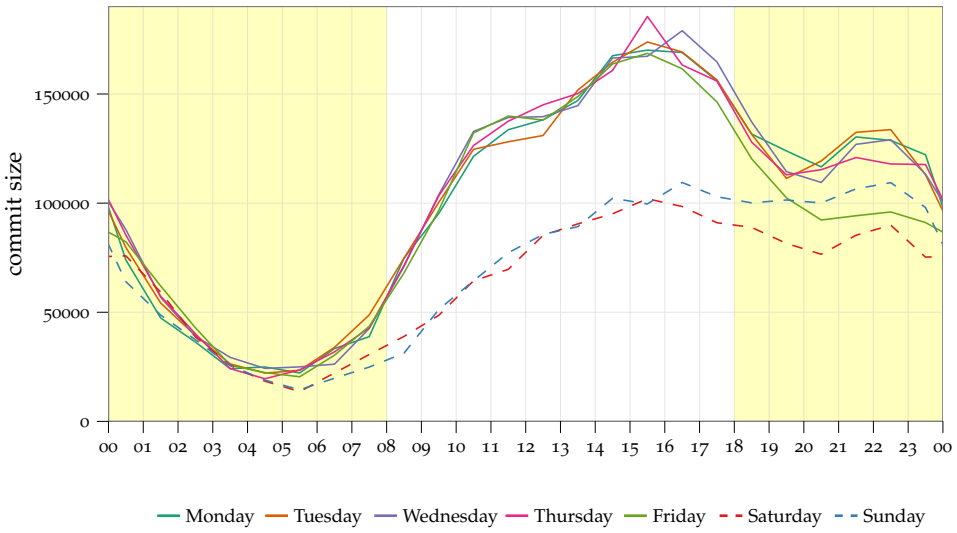


Figure 4.9: Commit Size Changes by Contributors *with Time Zone Data* per Hour



peak was missing for committers when inspecting the Linux kernel data in Section 3.2.3.

- Thirdly, there are more lines of code changed per commit on Wednesdays between 22:00 and 23:00 than on average on that days.

- Furthermore, the drops on Sunday evenings between 18:00 and 19:00 as well as between 20:00 and 21:00, that are present in Figure 4.7, are barely identifiable when inspecting the changes of lines of code.
- The amount of code changes per commit is also higher during weekend's evening hours than through the days.
- Finally, the daily drops between 19:00 and 21:00 are better visible, in general, in Figure 4.9, and the weekend curves are not as smooth as the ones for submitted commits, as depicted in Figure 4.7.

4.3 WEEKLY WORK PATTERN TRENDS

After analyzing commit patterns with all activities condensed into a single week, the trend over the whole inspected period will be analyzed on the next pages. Therefore, the plots contain a single value per week that represents the ratio of commits during working hours in that specific week. For simplicity, a year consists of 52 data points and values of overlapping weeks due to leap years are incorporated into the corresponding last weeks of those particular years. To visualize the overall trend of the time series, a Locally Weighted Scatterplot Smoothing (LOESS) curve is plotted on top of each as a moving average that is robust against fluctuations.

Due to the fact that Section 4.2 showed more fine-granular pattern when inspecting a smaller amount of data, the group of committers with known time zone information is discussed first in the following.

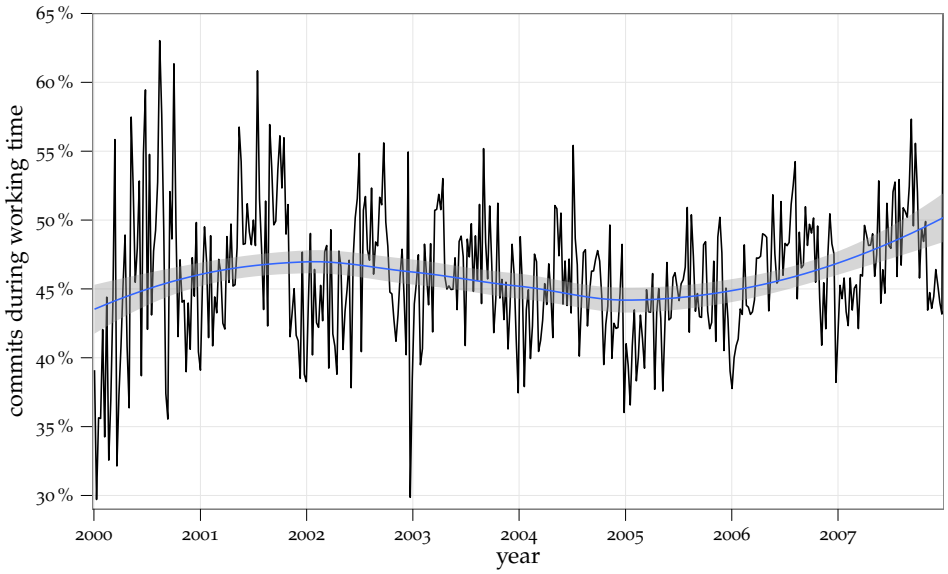
4.3.1 *Trends for Contributors with a Known Time Zone*

At a first glance, the overall trend of the time series depicted in Figure 4.10 can be summarized with that each year seems to start with a quite low percentage of commits during working hours before more and more professionals contribute. Interestingly, the LOESS curve is inverted compared to the ones shown in Figures 3.7 and 3.9 for the Linux kernel.

At the beginning, during the time span from 2000 until 2003, the observed working time percentage fluctuates between 30 % and nearly 65 %. Here, the overall yearly pattern can be described in its shape of being close to linear with large differences between the values of consecutive weeks. Due to the linear growth that has its peak in the third quarter of 2001, the LOESS curve also rises until that time before it again descends (until 2005). The values during the first half of 2002 oscillate uniformly between 38 % and 48 %. In contrast to that the values during the second half of that year do the same but at a higher level (47 %–55 %).

After 2003 the overall pattern changes and stabilizes by only differing between 37 % and 55 %. In addition, the shape converts more to a parabolic arc with its center slightly shifted to the mid of the third quarter of each year. Consequently, the values at the beginning and the end of each year are the lowest ones. A

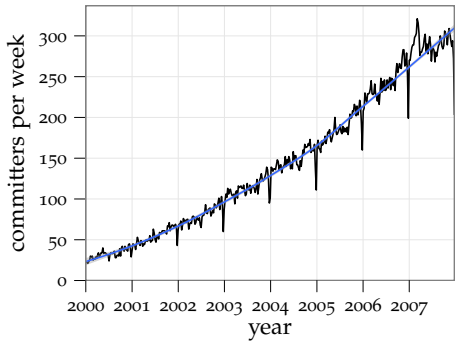
Figure 4.10: Percentage of Commits during Working Time for Committers *with Known Time Zone* on a Weekly Basis



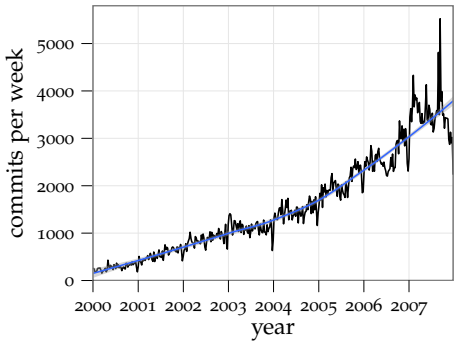
potential explanation for this is the bias towards projects from western countries of the Ohloh platform where Christmas and New Year are important holidays. Furthermore, a reason for the stabilization might be the attainment of a critical number of weekly contributors, which tops 100 committers per week at the end of 2003, as depicted in Figure 4.11a.

Figure 4.11: Additional Time Series Plots for Committers *with Known Time Zone*

(a) Number of Committers per Week



(b) Number of Commits per Week



Taking the two plots of Figure 4.11 as well into account for the analysis, one can observe that the number of involved developers, as depicted in Figure 4.11a, grew close to linear from approximately 25 per week up to over 300 per week during the inspected time span of eight years. Here, drops of approximately a fifth of all developers at the end of each year at Christmas time are visible

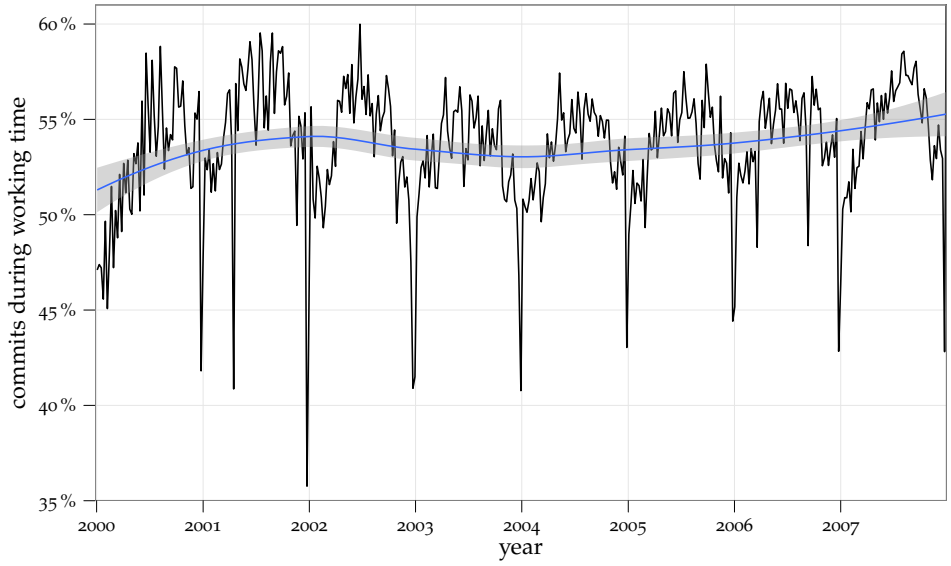
as well, supporting the assumption above of having mainly western projects under inspection.

However, the commits submitted by those developers, as depicted in Figure 4.11b, did not grow with the same factor but show a shallow angle. The time series shows the same parabolic shape, and thus, Figure 4.10 seems to highly correlate with Figure 4.10; much more than the Linux kernel data discussed in Section 3.3.

4.3.2 *Trends for All Contributors*

When looking on the total number of committer information stored by Ohloh, as depicted in Figure 4.12, the shape of the LOESS curve showing a moving average is similar to the one depicted in Figure 4.10 but more flattened. The curve is most of its time in the range between 50 % and 60 % and, thus, clearly more professional work during working hours can be assumed for this data set (cf. Section 4.2.2).

Figure 4.12: Percentage of Commits during Working Time for *All* Committers on a Weekly Basis



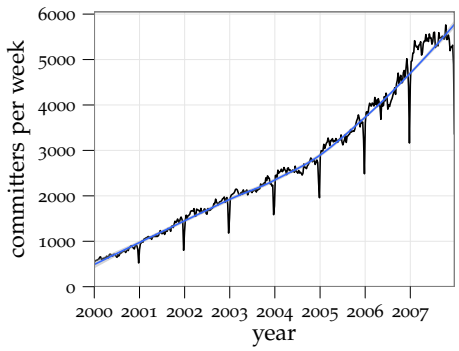
In addition, the yearly pattern of the time series has a parabolic shape, like the data shown in Figure 4.10 in 2006 and 2007 for the committers with known time zone information, and more pronounced rhythm with a sharp decline during the Christmas week. Since the number of inspected commits and developers here is, even at the beginning, more than twenty times higher, there is as much variation as for the smaller group discussed previously. Consequently, it remains stable from the start.

However, looking closer, also an overall drop of the peaks can be observed after 2003 in Figure 4.12, similar the one discussed in Section 4.3.1. This is not

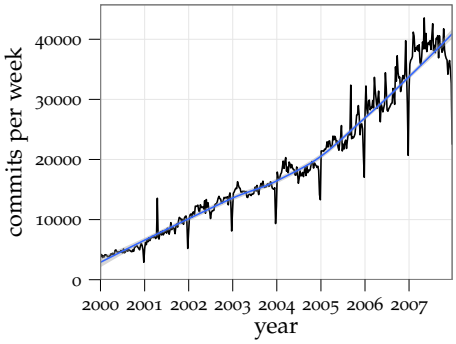
visible when inspecting the number of commits in that time, as depicted in Figure 4.13b.

Figure 4.13: Additional Time Series Plots for *All* Committers

(a) Number of Committers per Week



(b) Number of Commits per Week



This regular pattern can also be seen when inspecting Figures 4.13a and 4.13b. During the yearly Christmas week, the number of committers drops by a third, and the number of commits drops by nearly 50 % each year.

Interestingly, the yearly pattern depicted in Figure 4.12 can be observed for every inspected year, while the number of contributions and developers steadily grows at an exponential rate, as discussed by Deshpande and Riehle [DR08].

4.4 Overall Trends

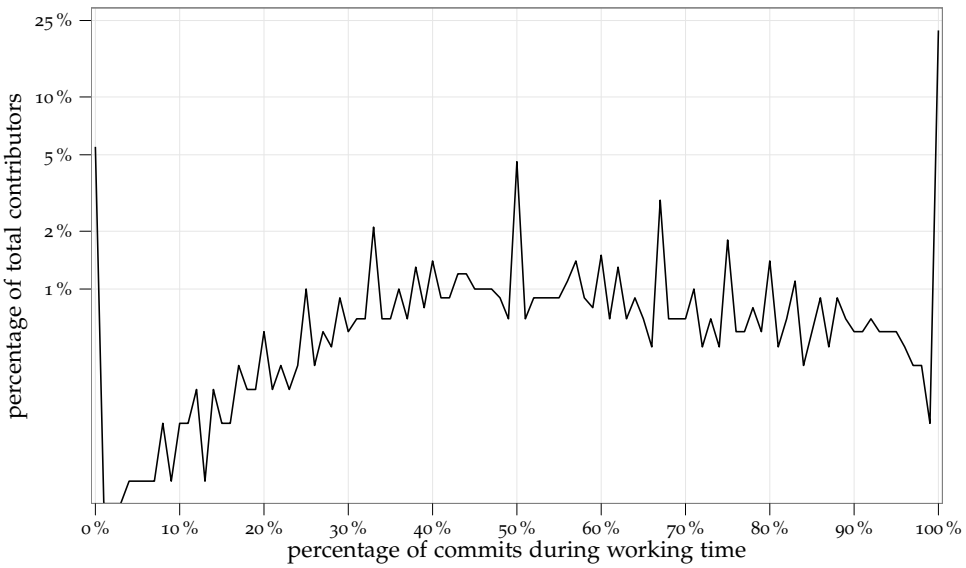
4.4.1 *Distributions*

The last section discussed trends over the whole inspected time frame by considering each week separately. In the following, those trends are further compressed and the distribution of commits during working time over the total period of eight years (2000–2007) will be analyzed. Please note, that the y-axes of the presented figures are logarithmically scaled to show more details even with high variation between the shown values. As before, the group of developers for whom the time zone information could be found in the database are discussed in addition.

Figure 4.14 depicts which percentage of all developers made which percentage of their total commits during working hours. Here, all committers of all projects are shown. Like in Figure 3.12 for the Linux kernel, the values on both edges as well as the 50 % value are significantly higher than the other ones.

Over the inspected eight years, nearly 25 % of all committers submitted their code solely during working hours. At the same time, more than 5 % of all committers performed their work exclusively outside of that time, during their spare time. Moreover, an equally large group of contributors exists that have

Figure 4.14: Distribution of Contributors per Percentage of Commits during Working Time



half of their submissions during working time and half of them during spare time.

Looking closer, one can observe that – even if the projects registered in the Ohloh database are purely open-source – there are interestingly more professional developers than hobbyists since the area below the curve is larger for values on the x-axis over 50 %. However, as Figure 4.16 later shows, a lot of these committers made only a few commits. Consequently, similar to the author's distribution discussed for the Linux kernel, also here peaks at multiples of 25 % and 33 % are clearly distinguishable.

The group of developers with known time zone information, as depicted in Figure 4.15, in contrast, shows nearly an inverted pattern. Again, the edges are higher than the values nearby, however, no clear peaks are prominent. In addition, the edges do not represent the maxima but are outperformed by most numbers in the range between 20 % and 60 % on the x-axis. The area below the curve for values with less than 50 % commits during working hours is much larger than the one above that value, indicating a higher amount of spare time workers in that group.

Comparing both graphics, the range between 0 % and 10 % catches one's eye due to the fact that the total number of committers there is lower than everywhere else.

To manifest the insights from Section 3.4.1, Table 4.4 shows the cumulative number of contributors with respect to their commit behavior. Those authors with exactly 50 % of their commits in working time and in spare time, are counted to both parts.

When inspecting the left column, and hence the all committers of all projects analyzed, barely twelve percent of all contributors submitted more than 75 %

Figure 4.15: Distribution of Contributors *with Known Time Zone* per Percentage of Commits during Working Time

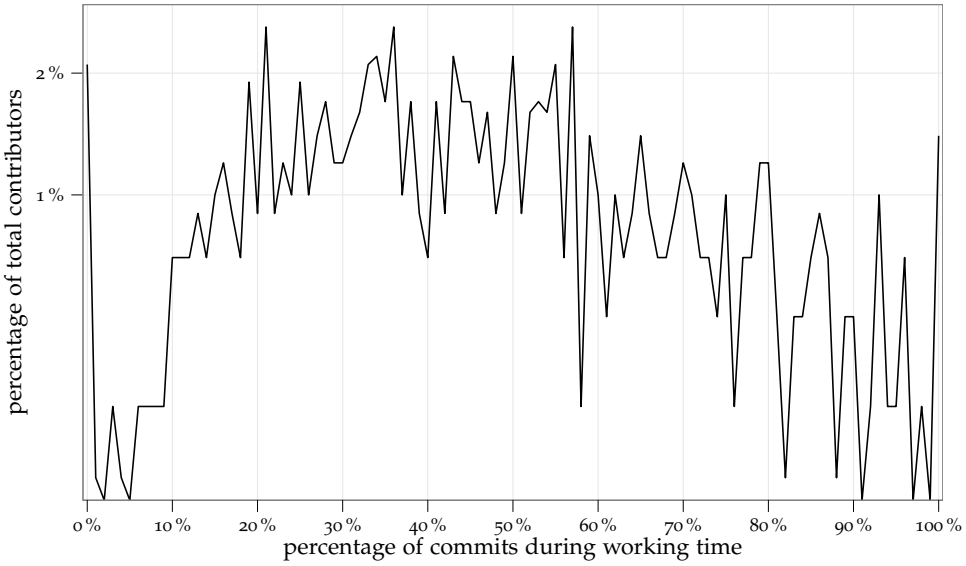


Table 4.4: Cumulative Number of Contributors

	% worktime	all		with TZ data	
Volunteers	0 %	2 538	5.5 %	12	2.1 %
	⩽ 5 %	2 660	5.8 %	16	2.8 %
	⩽ 10 %	2 982	6.5 %	28	4.8 %
	⩽ 25 %	5 399	11.8 %	127	21.9 %
	⩽ 50 %	17 517	38.2 %	346	59.7 %
Professionals	⩾ 50 %	30 474	66.4 %	247	42.6 %
	⩾ 75 %	18 032	39.3 %	86	14.8 %
	⩾ 90 %	12 586	27.4 %	29	5.0 %
	⩾ 95 %	11 158	24.3 %	16	2.8 %
	100 %	10 191	22.2 %	8	1.4 %

of their code changes per person during spare time. Starting from the bottom, approximately the double amount of users contributed during working hours, with at maximum five percent of their commits per person outside of this period. Also, the number of contributors submitting at least 75 % of their code during working hours – and hence can still be classified as highly professional – is higher than the number of those developers performing more than 50 % of their contributions during spare time.

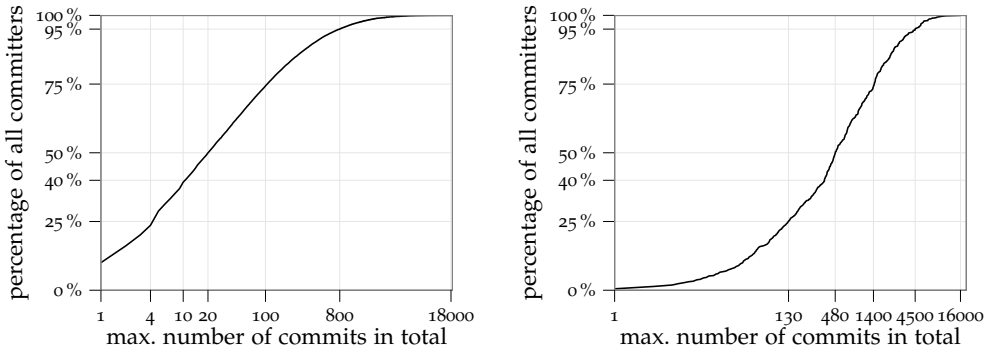
Looking at the column for the subset with time zone information stored in the database, a more gradual increasing trend can be observed on both ends.

However, this changes radically as soon as the twenty five percent border is reached. Starting with the spare time workers, i.e. the upper part of the table, the percentage values for pure hobbyists are less than the half of those when looking at the same group for all contributors. Nevertheless, the relative number of spare time developers who work between 10 % and 50 % of their time during working hours on open source is, relatively, significantly larger than the one of professional workers, as already discussed above. This can also be observed when inspecting both values in the vertical middle of the table, where the proportion is approximately 60:40, compared to the left column where it is roughly 40:60.

Figure 4.16: CDF of Total Number of Commits per Contributor

(a) All Contributors

(b) Contributors with Stored Time Zone Information



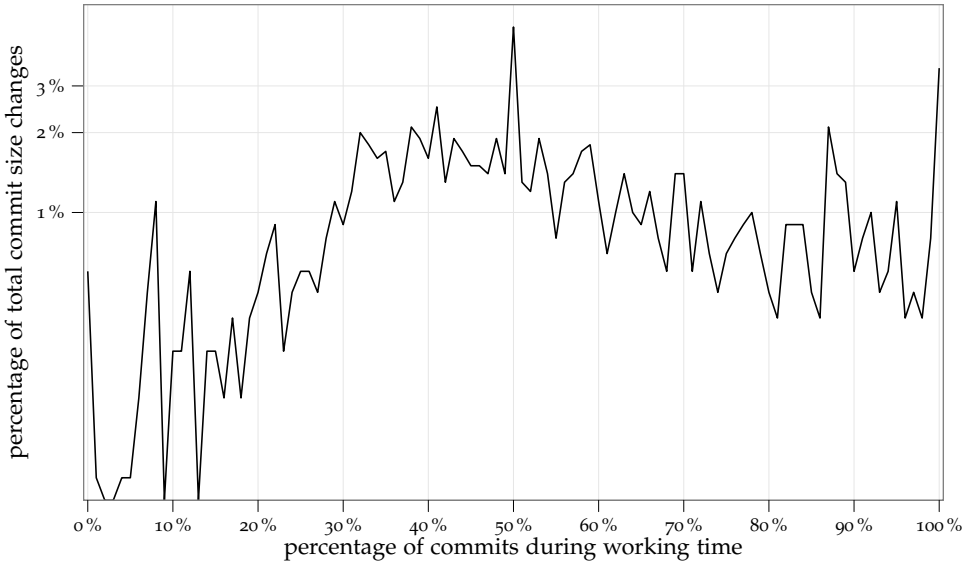
Until now, solely the percentage of commits during working hours per contributor were inspected. However, also the number of commits submitted per person are of interest to get a feeling about the relevance of this inferences. Therefore, Figure 4.16 depicts the Cumulative Distribution Function (CDF) of contributors by their total contribution to all projects stored in the Ohloh data set. Intuitively, the expected two distinct shapes catch the reader's eye.

The one for all committers, depicted in Figure 4.16a, shows a steady, close to linear growth. Since both axes are logarithmically scaled on the same basis, this means that the overall trend double-logarithmic with a very large group of developers with only a few commits and a low number of developers with a large number of contributions. As one can observe, a quarter of all contributors submitted $\leqslant 4$ commits, and the half of all committers whose activity was stored by Ohloh submitted only $\leqslant 20$ changes in total. Consequently, at least half of all contributors in the inspected data (i.e. approximately 23 000 people in this context) have a close to insignificant influence to FLOSS as an individual and represents the variaty of interests and small patches provided to fix specific issues. Only one quarter of all contributors performed more than 100 commits over the period of eight years. The performances of the topmost five percent, consisting of 2 294 contributors, are widely spread and range between 800 and 18 000.

In contrast, Figure 4.16b, depicting the commits of the contributors with certainly known time zone information, has more the shape of the letter "S". This group is only a subset of the other, but having the ascent shifted to the right indicates a larger (relative) number of committers with many contributions.

Only 6.5 percent, or 38 developers, performed $\leqslant 20$ commits over the whole inspected period. Three quarters of this group submitted more than 130 commits and, thus, more changes than three quarters of all committers in the overall data set.

Figure 4.17: Distribution of Contributors per Percentage of Commit Size Changes during Working Time



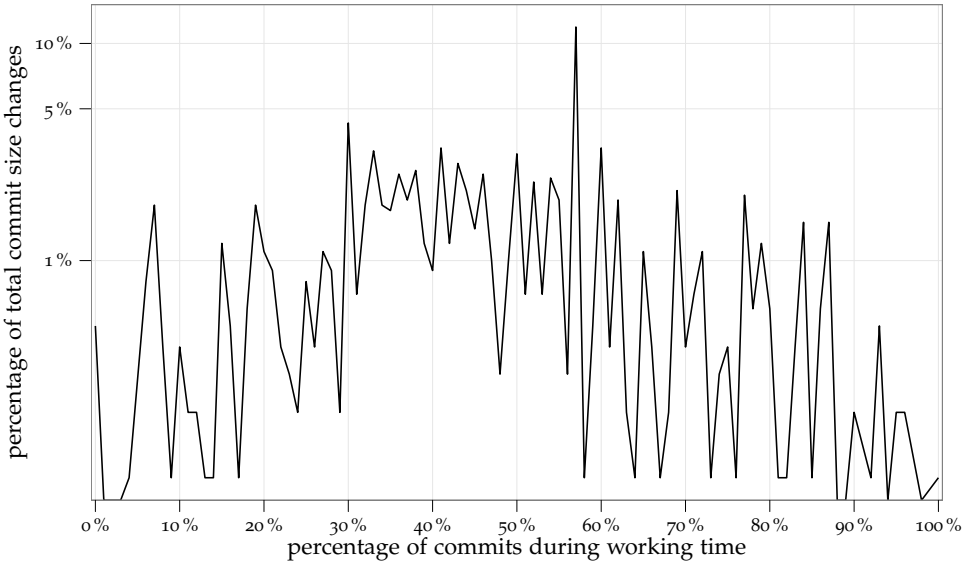
Analyzing the distribution of commit size changes for all stored data, as depicted in Figure 4.17, not only the overall influence is more evenly spread but also a smaller variation between consecutive values can be observed. While the distribution of commits for all contributors, as depicted in Figure 4.14, had its extreme values at the edges, the same is not true for when inspecting the commit size changes. Here, the influence of developers working primarily during their spare time is in fact even smaller than the first impression of Figure 4.17 implied. Again, most changes are introduced by the group of developers with more than 50 % of their commits performed during working hours.

The maximum values are located at 50 % and 100 %. While the latter shows a high professionalization also in FLOSS development, the former might be due to the following reasons:

On the one hand, the number of commits per developer is rather low in this data set, as shown earlier in Figure 4.16a. Consequently, people with a small number of changes followed by a related consecutive patch, e.g. bug fixes or clean-ups of their code, fall in this group if the contribution times balance each other.

On the other hand, having such a large number of projects and developers under inspection here, this large amount of changes at 50 % might also be an indicator for people who contribute to a project during their paid time, and to other projects when they are at home. Nevertheless, the group of those developers has the highest commit size to person ratio, as 4.6 % of all contributors who are in that bin are responsible for 4.4 % of all changed lines of code.

Figure 4.18: Commit Size Changes per Contributor *with Time Zone* Information



Again, the group of those developers whose time zone could be determined with certainty behave a little bit different when analyzing the code changes, as depicted in Figure 4.18. Here, the professional developers, with more than 60 % of their work per person performed during working hours, have only a minor influence.

In addition, the quite high number of contributors working solely during paid time, when looking at Figure 4.15, performed barely no source code changes. However, the high values in Figure 4.18 are evenly distributed and range from eight up to eighty seven percent without much variation. An exception is the peak at 57 %, where a small group of developers (15, or 2.6 %) is responsible for close to twelve percent of all change lines of code in the inspected data.

Figure 4.19 shows the Cumulative Distribution Function (CDF) of commit size changes per percentage of work during working hours. The line representing all committers' work has a smooth "S" shape and passes the diagonal at exactly fifty percent. Also the line of the subset of committers for whom the time zone information is known with certainty shows the same shape, but is more edgily.

Moreover, the latter has a small jump at 57 % that was already discussed above. The line shape indicates that most code changes are performed by the group with a close to balanced proportion of commits during working time and spare time.

Figure 4.19: CDF of Commit Size Changes per Contributor

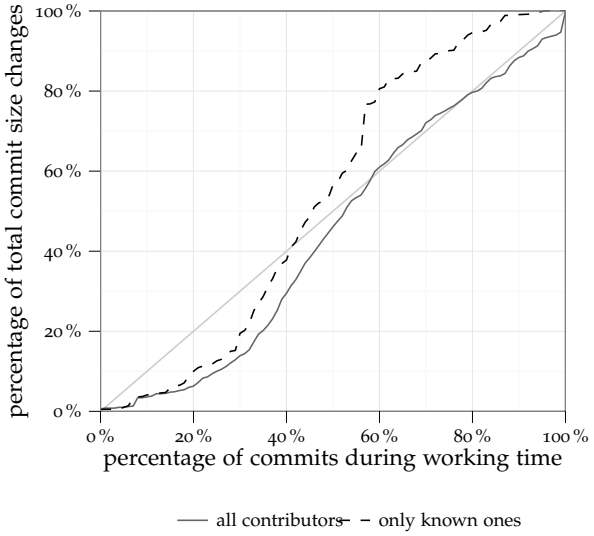


Table 4.5: Total Percentage of Changes during Working Hours

	all	all*	with TZ data
commits	54.0 %	54.1 %	46.4 %
code changes	52.9 %	53.4 %	45.3 %

The overall percentage of changes introduced during working hours to the data set provided by Ohloh over the period of eight years is depicted in Table 4.5. Also here the percentage for the total data set has a slight tendency towards professionalization but can be seen as balanced, even after filtering out bulk imports as well as projects and developers with just a hand-full commits, as indicated by the asterisk in the table.

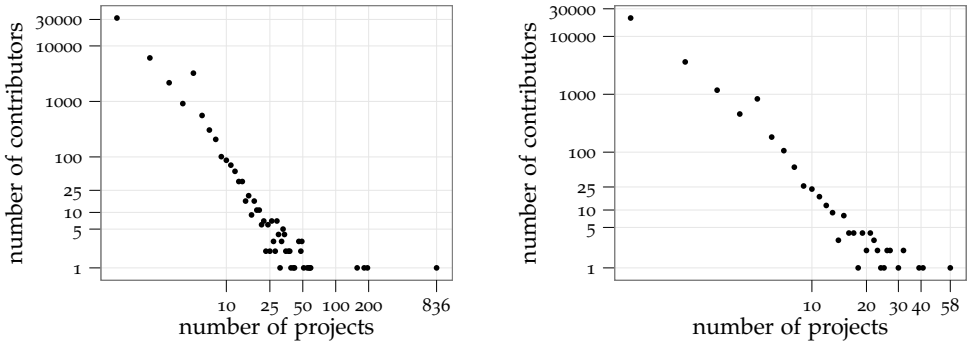
4.4.2 *Project Specific Analyses*

In contrast to the previous sections, now further distributions are inspected by considering each individual developer or project, respectively, as the smallest entity.

The two plots in Figure 4.20 depict how many developers contributed to which number projects in the Ohloh data set. Here, Figure 4.20a shows all developers in the database while for Figure 4.20b only those contributors were considered who submitted at least ten commits in total in a single project over the inspected time frame of eight years. The latter leads to a reduction of relevant developers from 45 870 down to 27 358 (60 %). As one can observe, both figures are double-logarithmic, with a large number of developers who only submitted changes to a single or a few projects.

Figure 4.20: Number of Contributed Projects per Developer

(a) All (b) After Filtering



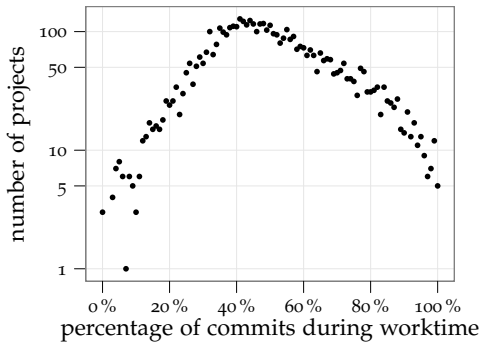
Inspecting Figure 4.20a, one can observe that most developers commit to less than 6 different projects. The preferred way is to just concentrate on a single one, which is true for 31 857 contributors. The next top-most numbers of projects are two (6 075), five (3 230) and three (2 170). The number of developers decreases rapidly the higher the number of involved FLOSS programs rises. Only 453 (roughly 10 %) of all contributors registered in the database participated in more than ten different FLOSS projects over the inspected period of eight years. The five most-diverse developers have 59, 157, 182, 195 and the top one even 836 projects registered were they contributed. However, this insight is heavily biased by the fact that some usernames are ambiguous and assigned to multiple projects. As a result, the four top-performers can not be matched properly and have to be ignored.[10] The three identifiable real developers with the widest spectrum are Stefan Kulow (KDE, 55 projects), Montel Laurent (KDE, 58) , and Dirk Müller (KDE, 59).

After filtering out those developers who performed less than ten commits for each project, the observed trend remains, as depicted in Figure 4.20b. The plotted pattern, while having lower numbers on both axes, can be described as similar to the one described before. Also, those developers who submitted changes more actively prefer to concentrate on a single project (20792, 75 %). However, the filtering process shows that a large group of the developers registered at Ohloh are only providing a handful of commits (cf. Figure 4.16). Only ten percent of the inspected contributors participated in three or more projects. Nevertheless, even if such small contributions, e.g. bug fixes, are not considered, the three most widely contributing real developers submitted changes to 27 (Dirk Müller, KDE and Kohsuke Kawaguchi, Sun) and 41 (Montel Laurent, KDE) projects. Considering their affiliations at that time, this indicates not only a high interest in FLOSS development in general, but also that it is possible to work professionally in this field.

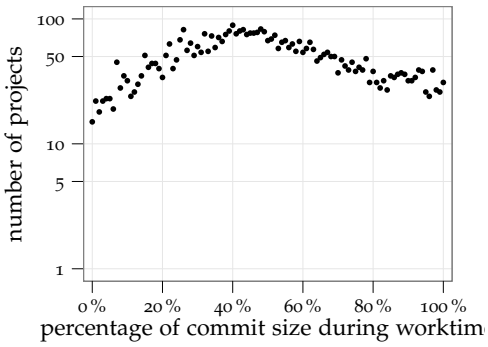
10 The user accounts are definitely do not belong to a single real-world person each, as they are *httpd* (Apache Webserver), *tigrisc* (Tigris CVS web interface), *root* (administrative user under Unix and Linux), and *(no author)* (used in cases where no developer name could be found).

Figure 4.21: Percentage of Professional Work per Project

(a) Percentage of Commits during Working Hours per Project



(b) Percentage of Commit Size Changes during Working Hours per Project



In addition to the distribution of developers per project, Figure 4.21 depicts the level of professionalization for all projects with more than 10 commits per year. The ratio of the total number of commits (or code changes) during spare time and during working time is individually calculated for each project and then all projects with the same percentage are grouped into one bin. Each of the latter is then represented by a single dot in the plots. Here, Figure 4.21a is based on the number of commits, while Figure 4.21b uses the number of source code lines changed.

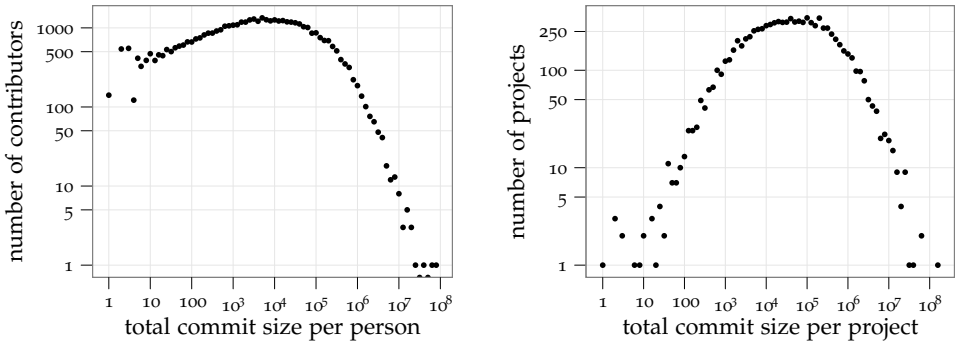
The former has a bell shape, indicating a Gaussian distribution as expected. Most of the projects have between 35 % and 65 % of their commits submitted during working hours, with a slight tendency towards voluntary work. However, one can observe a stronger slope on the spare-time part, indicating only a few pure voluntary projects. In contrast, the curve remains high on the right part of the plot, which shows that the number of professional projects is wider spread.

The latter, concentrating on commit size changes, is interesting when comparing it with the previously discussed plot. The points in it do not form a strong bell shape but the values are more closer to each other. This indicates that different projects have different sizes in their number of lines changed per commit (cf. Table A.1). Projects sticking to small incremental changes, have a high number of commits and, consequently, a higher distribution in the commit times. As shown in previous sections, the individual hour of a day also has an impact on the performance of the individual developers. Due to the larger values on both edges, one can conclude, that there are projects existent that get submissions also during spare time but are mainly developed during working hours, and vice versa.

Finally, the total amount of commit size changes for projects and developers are inspected in Figure 4.22. The two plots have logarithmically scaled axis but need further explanation, as both are different from the ones presented before.

Figure 4.22: Total Number of Commit Size Changes

(a) Per Developer (filtered) (b) Per Project (filtered)



Since the total commit size is heavily prone to minor changes[11], the aggregation was performed as following: Each section of the x-axis is divided into 10 distinct equidistant values[12] and the total commit size of an individual developer or project was rounded appropriately to fit into one of those bins. Then, the number of elements in each bin are summed up and the corresponding value is represented as a dot in each plot.

Figure 4.22a depicts the total number of changed lines of source code (commit size) per developer. One can observe, that most developers changed between 50 and 500 000 lines of code in total over the inspected period of eight years. However, there is also a large group of contributors who submitted less than that – indicating ad-hoc contributions and bug fixes for individual problems. Furthermore, there exists also a small group of developers with more than 500 000 lines of code changed. These roughly 3 % represent a group of FLOSS contributors, with over 10 000 lines changed per day on average, on every day in these eight years. It can be assumed that they mainly evaluate and integrate the work of others (maintainers), as the number of changes is too large to be pure development work of a single person.

Figure 4.22b shows the total number of source code changed (commit size) for projects. Like before, each inspected segment is divided into 10 equally-distanced bins. However, in contrast to the previously described one, this plot has a clear bell shape. Cutting horizontally at 50 projects per bin, approximately 2 % of all projects are in the groups below 400 and above 4 000 000 lines changed in total over eight years, respectively. The majority of all projects lies in between these two values, with a peak in the commit size range between 10^4 and 10^5, representing projects with 3 and 35 lines of code changed on average each day.

11 A developer just needs to have changed a single line of code more or less than another one to have a different value here.

12 As the x-axis is log-scaled, the ten values e.g. for the section between 10 and 100 are: 13, 16, 20, 25, 32, 40, 50, 63, 79, and 100.

4.4.3 *ARIMA*

Sections 4.2 and 4.3 discussed the observable behavior of developers registered in the Ohloh database excerpt in detail.

Until now, only a weekly pattern was taken into consideration and it was shown, that the activity of a developer differs dependent on the day of the week. However, e.g. Figure 4.12 also shows a clear pattern that is repeated for each year. Consequently, it would be interesting to analyze on which factors the weekly activity of developers and in projects is dependent. In contrast to e.g. stock exchanges, that are highly influenced by external events [BM02; Joh+85], the plots discussed so far in this thesis do not show such variety but more a regular rhythm. In addition to that, KOLASSA, SALIM, and RIEHLE [KSR12] point out that a regular development activity can be shown since contributors submit patches often and, thus, changes are not only introduced as big imports but piece by piece.

Inspired by the work of HERRAIZ, GONZALEZ-BARAHONA, and ROBLES [HGR07], the author of this thesis also started examining if the pattern shown so far do also apply when solely inspecting data from a single project individually. A good method for analyzing cyclic patterns is the Autoregressive Integrated Moving Average (ARIMA) [BJ70]. The analysis might provide further insights if the patterns that are observable when doing a general, overall inspection can also be found on a project basis, or if there exists different types of projects, that – when examined at the same time – balance each other.

These analyses needed some preprocessing before the calculations could be performed for each project as well as for each author of the data set at hand. Therefore, each time series was aggregated into weekly data points and then split if it contained an activity break longer than one year (52 weeks). This step was necessary since ARIMA models are based on the arithmetic average and, thus, are prone to outliers, which would lead to wrong outcomes. In cases where such a break appeared, the two resulting data sets were then inspected as individual time series while keeping their name as an indicator to the related part from where they came from.

After cleanup, code of the `astsa` package for `R` [Sto12] was adjusted to fit the needs since the original code had side-effects that were not feasible for the massive parallel execution (parts of the used scripts are shown in listings A.5 and A.6 in the appendix). The ARIMA calculations were performed by iterating over all reasonable input values for each item, resulting in a group of approximately 1 000 output values per data entry. Besides the used input parameters, the results contain the fitted values as well as the error's sum of squares and the error of the Autocorrelation Function (ACF) values as output.

Since these calculations need several days even when utilizing a HPC cluster, a selection of large projects, listed in Table A.1, were examined in advance. For those, it was possible to find parameter combinations, predicting the data quite precisely with a ACF error below 0.1 and a sum of squared error for the ACF residual deviations of 0 in most cases. Consequently, ARIMA models are a

valid technique for inspecting the seasonal impact also on a project basis. With the help of further research on the resulting parameters, it is possible to predict the productivity of projects for the following weeks.

However, it turned out that the calculations for the full data set of approximately 9 000 projects and 45 000 distinct developers exceeded the time restrictions of this work, as especially higher seasonal parameters (P, D, Q) lead to long running calculations. Currently there are computations running on a HPC cluster that are expected to be finished not earlier than after the deadline of this thesis. Due to that, it was not possible to perform additional calculations (e.g. clustering based on the input parameters) with the resulting groups for getting more insights. Having the author's code for the computations described above, this might become a topic for a follow-up work at the university chair which could also take signal processing techniques into considerations, as discussed by HINDLE, GODFREY, and HOLT [HGH09].

5 CONCLUSIONS

5.1 RÉSUMÉ

Free/Libre/Open Source Software (FLOSS) is becoming popular and a variety of the products are well known nowadays. While the movement started with a small group, today thousands of projects are existent and some of them are the market leaders since years in their business areas. However, the development effort needed to stay competitive against proprietary software has to be sponsored somehow.

This thesis analyzed if the influence of commercial enterprises in FLOSS can be shown, based on the commit behavior of developers. Here, a special focus was laid on the work performed during general (western world) working hours. A large sample of open source projects in the time frame from 2000 until 2008 (exclusively) was inspected, covering a wide spectrum of software topics and sizes (Chapter 4). Additionally the Linux kernel project was chosen as a prominent example and the recent history was inspected by using its VCS data, covering the period from 2005 until the end of 2011 (Chapter 3).

Based on this input, the commit frequency by day of the week and by hour of a day was discussed. Interestingly, both data sets showed identical patterns. The development activity during working hours on Mondays to Fridays per day was threefold the amount of that on weekend days. Also the daily performance curves were similar. Here, Ohloh's committer data closer matched the authors' curve of the Linux kernel. A reason for that might be the focus of the Ohloh data set on sources providing easy access to centralistic VCS, that contains a large amount of smaller projects which are not as strongly regulated as e.g. the Linux kernel. Consequently, in those cases authors and committers might be identical and the commits might be directly applied without a prior discussion on mailing lists or the like. However, providing all calculations for individual groups of projects in detail exceeded the focus of this thesis due to computational requirements and, thus, the author can only guess a reason here.

The percentage of commits during working time on a weekly basis over the total time-frame showed two diverse patterns.

The Linux kernel data had a strong correlation between peaks and releases. The fluctuation decreased over time and is in a 20 % range nowadays, with nearly all values over 50 %. Consequently, the Linux kernel project can be seen as a FLOSS project that is highly backed by companies, which is in line with [CKM12].

In contrast to that, the plots for Ohloh showed a clear constant pattern over the inspected period of eight years, that is repeated in each year. This is interesting, since the underlying data grew exponentially during that time. One can deduce from that fact, that the ratio of professional, i.e. paid, work to volunteer contributions in the development in this field can be assumed as being close to stable. In addition, the data shows that FLOSS long has become normal work for many developers, as also here most of the values are above 50 %, even when inspecting a large variety of projects from different domains. Having the empirical evidence, this insight can be used to further convince people and companies in replacing their proprietary solutions, against the prejudgement of getting a product that is solely developed by volunteers on their (rare) spare time and that is of lower quality – as e.g. stated by Nichols and Twidale [NT03] or VanHilst et al. [Van+11] – which itself dates back to times where even the World Wide Web did not yet exist [BBV86]. In this context, Table A.1 provides an except of the top well known projects in the Ohloh dataset based on their number of commits with detailed information, aggregated over the period from 2000 until the end of 2007.

Due to the usage of Git, detailed information about releases and author activity are available for the Linux kernel, which allowed inspecting additional topics.

Firstly, Section 3.4.2 provided an insight in the field of patch submissions when using strict rules for release management. Even if ancillary repositories for staging processes exist, an increased activity of authors and committers could be observed during the merge window. This might help companies and volunteers getting a better impression of the development and support the official documentation of the project.

Secondly, having also the authors' submission times for patches, not only differences in the behavior could be highlighted but in Section 3.4.3 also the relation of both types of contributors was analyzed. Here, one could observe that while the number of authors tripled over the inspected time-frame the support ratio decreased, so that a single committer – on average – had to review the submissions of approximately six authors per week over the last two years.

5.2 LIMITATIONS TO FINDINGS

For the empirical work presented in this thesis the assumption is made, that paid work is performed in the time from Monday to Friday, 8:00–18:00, in the respective local time zone. This is based on a generalization of working time used in Western countries. Consequently, a cultural bias is in the selected working time definition, since some political regions have another work period per week than Mondays–Fridays. However, as several sources as well as an analysis of the data at hand clearly indicated that for a majority of FLOSS developers it can be assumed that they are located in the Western countries.

Furthermore, the time of a commit does not precisely describe the actual time when development was performed. Instead, it just points to the time when it was submitted [Rob07]. As Kolassa, Riehle, and Salim [KRS10] showed, the

mean time between two commits of the same developer averages at about 100 minutes. Consequently, knowing this fact, commits were taken as a proxy for the work performed.

However, this thesis solely inspected the development activity using data stored in the VCS repositories. This represents only those patches that were accepted and integrated. In addition, the FLOSS ecosystem contains several other aspects besides the pure development of a software product, such as bug reporting or artwork, that were not taken into consideration here [Rob+05].

While the data used for the analyses presented in Chapter 3 are publicly available, Chapter 4 is base on a undisclosed database provided by the Ohloh webservice. This fact hinders reproducing the results outlined here [GR12; Kit08]. As Rodriguez, Herraiz, and Harrison [RHH12] state *one of the hardest tasks is to preprocess the data. However, trusting the preprocessed data from others is a poisened chalice.* Since recently, Ohloh provides a possibility to access parts of their database.[1] However, the access is restricted to selected tables and limited in the number of requests allowed per day.

The snapshot of Ohloh's database dates back to 2008 and, thus, does not show the evolution of the last five years. It represents a comprehensive data source that is less biased as e.g. FLOSSmole [HCC06], due to the continuous updates and additions from the community after the initial seeding. Consequently, it does not have a focus on specific projects or hosting providers. One can argue, that a bias, if it exists, is towards popular projects, which is more positive than negative for this work. Nevertheless, a replication of the calculations on other data sources might provide further valuable insights.

The data provided by Ohloh is solely based on the analysis of repositories using one of the three supported VCS. This covers a majority of the popular FLOSS projects at that time. Nowadays, the service supports two additional VCSs, namely Mercurial and Bazaar. This shows a trend away from centralized towards distributed solutions. However, a collection of VCSs at Wikipedia[2] lists several more proprietary and open source products for revision control, that are all not yet supported by Ohloh. Especially the analysis of author information, like e.g. supported by Git, would be a large benefit for further research in the area of work rhythms.

Moreover, it became obvious during the creation of this document that a better cleanup of the data in the Ohloh excerpt is needed. Not only does it contain duplicated entries (cf. Section 4.1.3) but also anomalies, such as project history from times where the project was not yet open-sourced [Hof12]. While this would mean a high computational effort, methods like copy-paste detection or manual author filtering could enhance the database further.

Deshpande and Riehle [DR08], who worked with a previous version of the Ohloh data, point out that in addition to the general data quality, several projects migrated their VCS over time. They found that many of these dropped

1 See http://meta.ohloh.net/2012/07/open-data-and-ohloh/ and http://www.heise.de/developer/meldung/Daten-zu-500-000-Open-Source-Projekten-verfuegbar-1648857.html (both visited on 07/20/2012)

2 http://en.wikipedia.org/wiki/List_of_revision_control_software

their historical information of the former VCS when switching to the other. Consequently, only the recent commit history is available while the old one is lost. However, as already mentioned above, the former is considered as more interesting here than the latter. In addition, when inspecting this large amount of FLOSS projects over a wide time frame, as done in this thesis, the individual significance of single project is quite low.

5.3 FURTHER RESEARCH

During the work on this thesis, several options for further research were discovered. These are discussed in the following.

In April 2005, ROBLES et al. [Rob+05] analyzed the evolution of the Linux kernel. Instead of the announcement of the 2.7 release, as predicted by the authors, the 2.6 release line was continued and, instead, the Linux kernel project migrated from the VCS BitKeeper to Git for managing the source code. Consequently, it would be interesting to examine the influence of Git on the project and to research if changes are observable in the workflow of the project. The repository inspections might be combined with an evaluation of the mailing list activity before and after the switch, to see if patches can be easier submitted nowadays and if they are applied faster. This is in line with BIRD et al. [Bir+09] who plan to answer similar questions for projects in general that switch from centralized to distributed VCS.

In addition to that, the annual Linux Report already discusses the involvement of companies in the development of the Linux kernel [CKM12]. However, using more sophisticated methods, the results can be enhanced to provide better insights in this topic. Furthermore, the reports and this thesis discuss only applied changes. By taking also bug tracking information and the history of the mailing list into consideration, the analysis of how much work and refinement is needed until a patch enters the kernel might be of interest for companies. As a first approach in this direction, MACHT [Mac12] describes what steps are needed for the submission of a new feature and his participation with the community as an exemplary case study for schedule and budget driven company structures.

Looking at the Ohloh database snapshot, a manual filtering of all approximately 45 000 committers and tagging of invalid entries might be a valuable but also work-intensive improvement. In general, a more comprehensive filtering, e.g. based on the definition of "active projects" [Daf07], might help concentrating on groups of developers and projects of higher relevance.

With respect to the calculations performed in this thesis, it might be also useful to calculate the key figures on a yearly or even monthly basis, instead of the holistic perspective taken here. In addition, research might be done to find out if the behavior of the developers in primarily *volunteer* and *professional* projects differs and are visible, especially when evaluating the data over time. While Christian holidays, such as Christmas, where clearly visible in the data,

the influence of catastrophes and other global events, e.g. war or presidential elections, might be observable when inspecting only a part of the data at once.

When analyzing the repositories, Ohloh does not only store the number of changed lines of code per commit but also the programming language of each changed file. Furthermore, the tools recognize if a changed line was program code or a comment. In combination with the here presented aspects, it might be interesting to research, which programming language is preferred by professionals and volunteers by analyzing the code changes per project and individual contributor, while taking percentage of work during working hours into consideration. Moreover, one could research which influence bulk imports (that where filtered out in this thesis) have and at what time they are usually made. Also, the influence of the used Version Control System might provide further insight into the behavior of developers in FLOSS projects.

As discussed earlier, the database snapshot does not cover the recent five years. During that time, the number of repositories observed by the web-service increased from 20 000 (2008) up to 250 000 today. Since the tool-set was also improved since then, a new database dump might not only be more precise and of higher quality without the issues mentioned above, but might also help to show the influence of recent hypes and trends in software by providing access to a vast number of projects. In combination with the current excerpt, this might help to understand how the work rhythms nowadays differ from those common 5 or even 10 years ago.

In a paper published this year, Rodriguez-Bustos and Aponte [RA12] examined the influence of used VCSs to the Mozilla Firefox project before and after switching from CVS to the distributed VCS Mercurial and found that the core developer team nearly completely changed after that step. Consequently, not only cultural events but also the choice of the used development and project management tools and approaches has an impact on the behavior of the involved developers. The mentioned paper seems to be the first one addressing this topic. Also here, a software archaeology might provide new insights on how the behavior of developers changed over time.

Conklin, Howison, and Crowston [CHC05] were the first who described an approach of collecting and pre-processing such data also for academic purposes in 2005. While the price for storage can be seen as largely insignificant for this, in contrast to ten years ago, the amount of FLOSS data to analyze also increased at an exponential rate [DR08]. Consequently, the pure computational requirements might be unfeasible when doing such a project alone, but might become possible when working collectively. Here, universities and researchers need to improve the communication and bundling of resources. In some cases, also the industry supports such steps, like e.g. GitHub, by providing its project logs for public access.[3] By combining modern approaches, such as Map-Reduce [Sha09] or facilitation of calculations with the help of graphics cards [Nag+12], handling the increased number of available FLOSS data might be remain feasible in the future.

3 http://www.githubarchive.org/

A

APPENDIX

Listing A.1 shows the SQL query used to aggregate and retrieve the raw data of the Ohloh dataset out of the PostgreSQL database for further processing.

In a sub-query entries with invalid user provided content are preprocessed, so that an additional cleanup of that issue after downloading becomes obsolete. Furthermore, a regular expression extracts the state from the location column, since e.g. the United States or Canada are spanning over multiple time zones.

Listing A.1: SQL Query To Get Ohloh Committer Data

```
SELECT
  DISTINCT owt.cid, owt.author, owt.pid,
  EXTRACT(EPOCH FROM owt.times AT TIME ZONE 'UTC') AS times,
  akn.author_name, akn.country_code, akn.state AS st, akn.location AS loc
FROM
  public.osswithtime AS owt
LEFT OUTER JOIN
(
  SELECT
    n.id AS nID, n.name AS author_name,
    NULLIF(TRIM(BOTH FROM a.country_code), '') AS country_code,
    NULLIF(TRIM(BOTH FROM a.location), '') AS location,
    NULLIF(TRIM(BOTH FROM
      SUBSTRING(a.location FROM ', ([A-Za-z]*)')), '') AS state
  FROM
    public.accounts a,
    public.kudos k,
    public.names n
  WHERE a.id = k.account_id
  AND n.id = k.name_id
) AS akn ON owt.author=akn.nID
ORDER BY
  owt.author, owt.pid, times
```

Listing A.2 depicts the R code that is used to preprocess the data extracted from the local Git repository of the Linux kernel. As one can see, the raw data is cached, so that it does not have to be retrieved from Git in the next iteration.

Furthermore, a preprocessed file is used, if available, that contains already cleaned-up entries for the code statistics (lines added, deleted) per commit. The function `wdckms` is used to provide a consistent interface when working with multiple machines, so that only a single variable pointing to the local working directory has to be adjusted.

Listing A.2: Code For Loading Linux Data

```
lk.file.name <- wdckms("data/caching/linux-gitlogs.Rdata")
if(!file.exists(lk.file.name)) {
  source(wdckms("r-scripts/git.R"))
  repo.path <- "~/repos/linux_kernel/"
  message("loading Git log")
  lk.log <- gitLog(repo.path)
  message("loading tags")
  lk.tags <- getTags(lk.log)
  stats.file <- wdckms("data/linux-shortlog.csv")
  if(!file.exists(stats.file)) {
    message("loading stats with Git")
    lk.shortStat <- gitShortStat(repo.path)
  } else {
    message("loading stats from CSV")
    lk.shortStat <- gitParseShortStatFile(stats.file)
  }
  save(lk.log, lk.tags, lk.shortStat, file=lk.file.name)
  message("done")
  rm(repo.path, stats.file)
} else {
  load(lk.file.name)
}
rm(lk.file.name)

# remove head tag
lk.tags <- lk.tags[-nrow(lk.tags),]
lk.tags$is.rc <- grepl("-rc", as.character(lk.tags$ref.name), fixed=TRUE)

lk.log$ref.name <- as.character(lk.log$ref.name)
lk.log[nrow(lk.log), "ref.name"] <- NA
lk.log[nchar(lk.log$ref.name) == 0, "ref.name"] <- NA
# convert time
lk.log$author.realtime <- localTime(data.frame(times=lk.log$author.utc,
   Offset=lk.log$author.utc.offset, dst=FALSE))
lk.log$committer.realtime <- localTime(data.frame(times=lk.log$committer.
   utc, Offset=lk.log$committer.utc.offset, dst=FALSE))
lk.tags$committer.realtime <- localTime(data.frame(times=lk.tags$
   committer.utc, Offset=lk.tags$committer.utc.offset, dst=FALSE))

# calculate if commit was done in sparetime
now <- format(Sys.time(), '%Y-%m-%d')
wt.start <- as.POSIXct(paste(now, worktime.start))
wt.end <- as.POSIXct(paste(now, worktime.end))
lk.log$author.sparetime <- ((as.POSIXct(paste(now, format(lk.log$author.
   realtime, '%H:%M'))) < wt.start | as.POSIXct(paste(now, format(lk.log$
   author.realtime, '%H:%M'))) > wt.end) | format(lk.log$author.realtime,
    '%a') %in% c('Sat', 'Sun'))
lk.log$committer.sparetime <- ((as.POSIXct(paste(now, format(lk.log$
   committer.realtime, '%H:%M'))) < wt.start | as.POSIXct(paste(now,
   format(lk.log$committer.realtime, '%H:%M'))) > wt.end) | format(lk.log
   $committer.realtime, '%a') %in% c('Sat', 'Sun'))
rm(now, wt.start, wt.end)
```

As an example for the code used for graphics generation, Listing A.3 shows the file content needed to create the plot in Figure 4.8.

Here, one can observe, that common functions are encapsulated in additional files for re-use. Consequently, the calculation can be reduced to a little bit over 10 lines, plus plot creation code. As described in Section 2.2.1, eventually, Ti*k*Z is used for the creation of native LaTeX output.

Listing A.3: Code Used To Create Figure 4.8

```
source('additional-header.R')
source('additional-ohloh.R')
library(tikzDevice)
# max lines of codes to still count as "normal" commit
max.loc.per.commit <- 175

my.stats <- merge(oh.log, getCommitSizes(), by.x="cid", by.y="commits_id"
   , sort=FALSE)
# since the merge produces duplicates, clean up
my.stats <- deleteDuplicates(my.stats, .columns=c("cid", "author", "pid")
   )

# aggregation
per.hour <- data.frame(
  hour=round2hour(prorateTime(data.frame(times=my.stats$realtime), 4)),
  wday=factor(format(my.stats$realtime, '%A'), levels=wdays.en.long,
     ordered=TRUE),
  commit.size=my.stats$commitsize)
rm(my.stats)

# filter out bulk commits
per.hour <- per.hour[per.hour$commit.size < max.loc.per.commit, ]
limits <- range(per.hour$hour)
# normally it ranges only until 23:00, thus add one hour to max
limits[2] <- limits[2] + 60*60
# add on both sides so that the curve does not end in 0 at the edges
tmp1 <- tmp2 <- per.hour
tmp1$hour <- tmp1$hour + (24 * 60 * 60)
tmp2$hour <- tmp2$hour - (24 * 60 * 60)
per.hour <- rbind(tmp1, per.hour, tmp2)
rm(tmp1, tmp2)
# points per hour at XX:30 for better readability
per.hour$hour <- as.POSIXct(format(per.hour$hour, "%Y-%m-%d %H:30:00"),
   tz="UTC")
# calculate sum per hour
per.hour <- ddply(per.hour, c("wday", "hour"), function(x) {
  return(c(commit.size=sum(x$commit.size)))
}, .progress="text")

p.all <- base.plot(per.hour) +
  geom_rect(xmin=as.numeric(limits[1]), xmax=as.numeric(as.POSIXct(paste(
     format(limits[1], "%Y-%m-%d "), worktime.start, ":00"), tz="UTC")),
     ymin=0, ymax=1.025*max(per.hour$commit.size), fill="#FFFFBF", alpha
     =1/100) +
```

```
  geom_rect(xmin=as.numeric(as.POSIXct(paste(format(limits[1], "%Y-%m-%d
     "), worktime.end, ":00"), tz="UTC")), xmax=as.numeric(limits[2] + 24
      * 60 * 60), ymin=0, ymax=1.025*max(per.hour$commit.size), fill="#
     FFFFBF", alpha=1/100) +
  geom_line(aes(x=hour, y=commit.size, col=wday, linetype=wday)) +
  opts(axis.title.x=theme_blank(), legend.position="bottom", legend.title
     =theme_blank()) +
  scale_x_datetime("", breaks="1 hour", minor_breaks=NULL, labels=my_date
     _format("%H")) +
  scale_y_continuous("commit size", minor_breaks=NULL, labels=my_num_
     format()) +
  coord_cartesian(xlim=limits, ylim=c(0, 1.025*max(per.hour$commit.size))
     ) +
  scale_colour_manual(name="weekday", values=c(brewer_pal(palette="Dark2"
     )(5), brewer_pal(palette="Set1")(2))) +
  scale_linetype_manual(name="weekday",values=c(rep("solid", 5), rep("
     dashed", 2)))

tikz(file="../graphics/ohloh-hourly-freq-commitsize-all.tex", width
   =5.118, height=3.15) #15x8 cm
print(p.all)
dev.off()
rm(per.hour, limits, p.all)
```

Another non-trivial graphics is the Hollerith card, depicted in Section 3.4.2. The code, as shown below in Listing A.4, is relatively straightforward. First, the used functions and data are loaded. Then, the tags are filtered and releases are identified and isolated. With the latter at hand, the data is aggregated and formatted accordingly, before it is fed into the plotting function and saved into a Ti*k*Z file.

Listing A.4: Code Used To Create Figure 3.16

```
source('additional-header.R')
source('additional-linux.R')
library(tikzDevice)

# prepare raw data
my.tags <- lk.tags[, c("ref.name", "is.rc")]
my.log <- merge(lk.log, my.tags, all.x=TRUE)
rm(my.tags)
my.log$release <- (!is.na(my.log$is.rc) & !my.log$is.rc)

releases <- my.log[my.log$release, ]
releases <- as.POSIXct(format(releases$committer.realtime, "2011-08-0%u %
   H:00:00"), tz="UTC")

input <- data.frame(
  day=factor(format(releases, format='%a'),
             levels=wdays.en, labels=wdays.en, ordered=TRUE),
  hour=factor(format(releases, format='%H:00'),
              levels=sprintf('%02d:00', 0:23), ordered=TRUE))
input <- melt(as.data.frame(table(input)), id=c('day', 'hour'))
names(input)[names(input) == 'value'] <- 'freq'
```

```
input[input$freq == 0, 'freq'] <- NA
input$freq <- factor(input$freq, ordered=TRUE)

p <- base.plot(input) +
  geom_point(aes(x=hour, y=day, size=freq), na.rm=TRUE) +
  opts(axis.title.x=theme_blank(), axis.title.y=theme_blank(), legend.
     position = 'none', axis.text.x = theme_text(size=FONTSIZE * 0.8,
     angle=45, hjust=1, colour='black'))

tikz(file="../graphics/linux-releases-time.tex",
     width=5.118, height=1.97) #15x5 cm
print(p)
dev.off()
rm(releases)
```

Listing A.5 depicts a typical batch script that is needed to perform calculations on the HPC cluster at the LRZ in Munich. For privacy reasons, the user data is stripped out and replaced with pointers encapsulated in angle brackets. A description of the different parameters set in this file can be found at the corresponding website[1].

Listing A.5: Batch Script Used On The LRZ Cluster

```
#!/bin/bash
#SBATCH --time=48:00:00
#SBATCH --cpus-per-task=100
#SBATCH -J ArimaAdjAu
#SBATCH -o /home/hpc/<add username here>/arima_adj/arimaAdj_log.%j.%N.out
#SBATCH -D  /home/hpc/<add username here>/arima_adj
#SBATCH --clusters=uv3
#SBATCH --nodes=1-1
#SBATCH --get-user-env
#SBATCH --mail-type=all
#SBATCH --mail-user=<add email address here>

source /etc/profile.d/modules.sh
module load R/parallel/2.13
R_HOME=/lrz/sys/applications/R/2.13.1_static

R --no-save -q < sarima_authors_HPC.R
```

Listing A.6 shows a simplified version of the code used to calculate the ARIMA values for each developer. First, the libraries for parallel computing are loaded and the number of available cores for parallel calculation is set. Afterwards the adjusted method as well as the preprocessed input data is loaded into memory. Finally, the latter is processed is chunks, to be able to restart the process after the 48 hour limit without having to re-calculate existing results. For the calculation of this file, each used core utilizes at least two Gigabyte of memory to hold the required temporary results.

1 http://www.lrz.de/services/compute/linux-cluster/batch_parallel/

Listing A.6: ARIMA Calculation Code (simplified)

```
library("doMC")
library("plyr")
registerDoMC(100)

# see astsa::sarima
sarima = function(xdata,p,d,q,P=0,D=0,Q=0,S=-1,details=TRUE,tol=sqrt(.
   Machine$double.eps),no.constant=FALSE)
{
  # code skipped here...
  # removed plotting code since this is run in parallel
  return(list(fit=fitit, error.sum.of.sqares=sum((abs(ACF[which(abs(ACF)
     > L)]) - L)^2), error=sum(ACF^2)))
}
# load list containing a time series for each author
load("authors.as.ts.Rdata")

offset <- 1
while(offset <= length(authors.as.ts)) {
  file.name <- paste("arima-fit.authors.", offset, ".Rdata", sep="")
  message("### starting with authors, offset ", offset, " ###")
  if(!file.exists(file.name)) {
    my.proj <- authors.as.ts[offset:min(length(authors.as.ts), (offset +
       999))] # inspect max. 1000 at once
    arimas <- llply(my.proj, function(x) {
      years <- length(x) / frequency(x)
      result <- NULL
      for(p in 0:3) { for(d in 0:2) { for(q in 0:2) {
        for(P in 0:ceiling(years/3)) { for(D in 0:2) { for(Q in 0:2) {
          try({
            a <- sarima(x, p, d, q, P, D, Q, frequency(x))
            tmp <- data.frame(p=p, d=d, q=q, P=P, D=D, Q=Q, sSqL=a$error.
               sum.of.sqares, error=a$error)
            result <- rbind(result, tmp)
          })
        }}}
      }}}
      return(result)
    }, .parallel=TRUE)
    # store results
    save(arimas, file=file.name)
  }
  # logging so that progress is also visible at run time in cluster log
  message("### offset ", offset, " done ###")
  offset <- offset + 1000
}
```

Table A.1, shown on the next two pages, gives an impression, which kind of projects are included in the Ohloh dataset. The table shows 42 selected, well-known FLOSS projects that have more than 10 000 commits registered for the period from 2000 until 2008.

Table A.1: Statistics for selected FLOSS projects in the Ohloh dataset with more than 10 000 commits

name	**commits**	**% wt**	**commit size**	**% wt**	**dev's**	**cs/c**	**c/dev**	**cs/dev**	**project url** (as of 2008)
GNOME	220 270	50,9 %	57 436 373·5	48.6 %	1 218	260.8	180.8	47 156·3	http://www.gnome.org
NetBeans IDE	118 735	78.4 %	31 663 764·5	73.1 %	510	266.7	232.8	62 085·8	http://www.netbeans.org
OpenOffice.org	117 420	70.4 %	22 628 818·5	73.1 %	409	192.7	287.1	55 327·2	http://www.openoffice.org
Linux Kernel 2.6	87 732	53.1 %	22 930 496·5	48.3 %	4 398	261.4	19.9	5 213·8	http://www.kernel.org
Mono	56 053	52.6 %	11 757 883·5	54.3 %	256	209.8	219.0	45 929·2	http://www.mono-project.com
Android	53 746	54.7 %	10 712 929·5	26.7 %	3 203	199.3	16.8	3 344·7	http://code.google.com/android/
GCC	48 232	51.7 %	12 577 644·5	56.3 %	300	260.8	160.8	41 925·5	http://gcc.gnu.org
KOffice	41 663	42.3 %	7 785 913·5	46.5 %	302	186.9	138.0	25 781·2	http://www.koffice.org
PHP	37 184	45.3 %	5 621 858·0	41.0 %	380	151.2	97.9	14 794·4	http://php.net
Wine	35 859	52.0 %	4 693 699·0	49.6 %	772	130.9	46.4	6 079·9	http://www.winehq.org
Eclipse JDT	34 253	84.8 %	4 232 194·0	82.0 %	58	123.6	590.6	72 968·9	http://www.eclipse.org/jdt/
wxWidgets	30 696	41.9 %	13 129 495·0	51.3 %	58	427.7	529.2	226 370·6	http://wxwidgets.org
LLVM	27 907	52.7 %	2 271 483·0	51.5 %	59	81.4	473.0	38 499·7	http://llvm.org
Moodle	27 397	50.0 %	4 273 094·0	56.9 %	127	156.0	215.7	33 646·4	http://moodle.org
MediaWiki	26 638	40.7 %	25 540 250·0	41.2 %	136	958.8	195.9	187 796·0	http://www.mediawiki.org
ScummVM	25 124	31.1 %	3 566 683·0	26.8 %	67	142.0	375.0	53 234·1	http://www.scummvm.org
Evolution	24 468	62.8 %	3 556 928·5	58.9 %	204	145.4	119.9	17 435·9	http://www.gnome.org/projects/evolution/
WebKit	19 851	59.3 %	5 494 817·5	48.5 %	106	276.8	187.3	51 837·9	http://webkit.org
Samba	19 644	61.4 %	4 385 696·5	54.4 %	70	223.3	280.6	62 652·8	http://www.samba.org
Python	19 448	48.0 %	3 376 216·0	33.2 %	143	173.6	136.0	23 609·9	http://www.python.org
Firebird	18 985	53.2 %	8 654 854·5	30.3 %	60	455.9	316.4	144 247·6	http://www.firebirdsql.org
Wireshark	18 696	45.0 %	7 408 273·5	35.8 %	39	396.3	479.4	189 955·7	http://www.wireshark.org

continued on next page

Table A.1 – continued from previous page

name	**commits**	**% wt**	**commit size**	**% wt**	**dev's**	**cs/c**	**c/dev**	**cs/dev**	**project url** (as of 2008)
VLC	17 494	37.3 %	2 382 725·0	30.8 %	81	136.2	216.0	29 416·4	http://www.videolan.org/vlc/
GIMP	15 994	46.3 %	5 251 817·5	39.8 %	98	328.4	163.2	53 590·0	http://www.gimp.org
Subversion	15 950	56.9 %	1 949 556·5	54.2 %	109	122.2	146.3	17 885·8	http://subversion.tigris.org
Adium	14 193	39.9 %	2 596 707·5	43.3 %	55	183.0	258.1	47 212·9	http://www.adiumx.com
Horde	13 434	51.7 %	1 065 433·5	46.9 %	27	79.3	497.6	39 460·5	http://www.horde.org
AbiWord	13 194	44.5 %	1 818 604·5	46.7 %	64	137.8	206.2	28 415·7	http://www.abisource.com
PostgreSQL	12 763	51.0 %	2 409 640·5	48.9 %	24	188.8	531.8	100 401·7	http://www.postgresql.org
MythTV	12 584	39.2 %	2 558 283·0	35.9 %	42	203.3	299.6	60 911·5	http://www.mythtv.org
X.Org	12 331	57.0 %	2 752 893·5	58.4 %	316	223.3	39.0	8 711·7	http://www.x.org
ArgoUML	12 293	32.2 %	2 769 075·5	26.1 %	51	225.3	241.0	54 295·6	http://argouml.tigris.org
GNUstep	12 249	44.5 %	3 878 168·0	51.9 %	61	316.6	200.8	63 576·5	http://www.gnustep.org
Eclipse SWT	11 966	87.5 %	2 334 961·0	87.4 %	30	195.1	398.9	77 832·0	http://www.eclipse.org/swt/
Azureus	11 213	44.8 %	1 292 757·0	45.4 %	32	115.3	350.4	40 398·7	http://azureus.sourceforge.net
Thunderbird	11 039	61.0 %	1 039 258·5	58.7 %	271	94.1	40.7	3 834·9	http://www.mozilla.org/projects/thunderbird/
Parrot	11 025	47.7 %	1 849 871·5	48.4 %	69	167.8	159.8	26 809·7	http://www.parrotcode.org
GlassFish	10 878	74.2 %	9 361 908·0	66.3 %	148	860.6	73.5	63 256·1	https://glassfish.dev.java.net
Blender	10 535	40.9 %	7 574 648·5	50.1 %	68	719.0	154.9	111 391·9	http://www.blender3d.org
xine (video player)	10 431	37.9 %	2 128 690·0	33.7 %	53	204.1	196.8	40 164·0	http://www.xinehq.de
eZ Publish	10 338	81.1 %	2 812 861·0	63.7 %	77	272.1	134.3	36 530·7	http://ez.no/ezpublish
FFmpeg	10 303	40.6 %	1 133 005·0	52.1 %	68	110.0	151.5	16 661·8	http://www.ffmpeg.org

BIBLIOGRAPHY

[And05] Jeremy Andrews. *Feature: No More Free BitKeeper*. 2005. URL: http://kerneltrap.org/node/4966 (visited on 06/29/2012).

[AR09a] Oliver Arafat and Dirk Riehle. "The Comment Density of Open Source Software Code." In: *31st International Conference on Software Engineering-Companion Volume*. 2009, pp. 195–198.

[AR09b] Oliver Arafat and Dirk Riehle. "The Commit Size Distribution of Open Source Software." In: *Proceedings of the 42nd Hawaiian International Conference on System Sciences*. 2009.

[AS09] Brian de Alwis and Jonathan Sillito. "Why are software projects moving from centralized to decentralized version control systems?" In: *Proceedings of the ICSE Workshop on Cooperative and Human Aspects on Software Engineering*. 2009, pp. 36–39. DOI: 10.1109/CHASE.2009.5071408.

[ASF12] Apache Software Foundation. *Apache Subversion – "Enterprise-class centralized version control for the masses"*. 2012. URL: http://subversion.apache.org/ (visited on 06/21/2012).

[ASM12] Khaled Aslan, Hala Skaf-Molli, and Pascal Molli. "Connecting Distributed Version Control Systems Communities to Linked Open Data." In: *International Conference on Collaboration Technologies and Systems*. 2012, pp. 242–250. DOI: 10.1109/CTS.2012.6261056.

[BBV86] Theodore Bergstrom, Lawrence Blume, and Hal Varian. "On the Private Provision of Public Goods." In: *Journal of Public Economics* 29.1 (1986), pp. 25–49. DOI: 10.1016/0047-2727(86)90024-1.

[BdJ12] Bundesministerium der Justiz. *Arbeitszeitgesetz*. 2012. URL: http://www.gesetze-im-internet.de/arbzg/ (visited on 07/16/2012).

[Ben12] James Bennett. *Android Smartphone Activations Reached 331 Million in Q1'2012 Reveals New Device Tracking Database from Signals and Systems Telecom*. 2012. URL: http://www.prweb.com/releases/2012/5/prweb9514037.htm (visited on 08/29/2012).

[Bir+06] Christian Bird et al. "Mining Email Social Networks." In: *Proceedings of the 3rd International Workshop on Mining Software Repositories*. 2006, pp. 137–143. DOI: 10.1145/1137983.1138016.

[Bir+09] Christian Bird et al. "The promises and perils of mining git." In: *Proceedings of the 6th International Working Conference on Mining Software Repositories*. 2009, pp. 1–10. DOI: 10.1109/MSR.2009.5069475.

[BJ70] George Box and Gwilym Jenkins. *Time series analysis: Forecasting and control*. 1970.

[BM02] Wolfgang Bessler and James P Murtagh. "The stock market reaction to cross-border acquisitions of financial services firms: an analysis of Canadian banks." In: *Journal of International Financial Markets, Institutions and Money* 12.4–5 (2002), pp. 419–440. doi: 10.1016/S1042-4431(02)00022-7.

[BN12] Christian Bird and Nachiappan Nagappan. "Who? Where? What? Examining Distributed Development in Two Large Open Source Projects." In: *Proceedings of the 9th Working Conference on Mining Software Repositories*. 2012, pp. 237–246. doi: 10.1109/MSR.2012.6224286.

[Cad12] Jose M. Cadenas. "A tool to manage low quality datasets." In: *International Conference on Fuzzy Systems*. 2012, pp. 1–8. doi: 10.1109/FUZZ-IEEE.2012.6251223.

[CCDP07] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. "Identifying Changed Source Code Lines from Version Repositories." In: *Proceedings of the 4th International Workshop on Mining Software Repositories*. IEEE Press, May 2007, pp. 14–21. doi: 10.1109/MSR.2007.14.

[CCDP09] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. "Ldiff: An enhanced line differencing tool." In: *Proceedings of the 31st International Conference on Software Engineering*. 2009, pp. 595–598. doi: 10.1109/ICSE.2009.5070564.

[Cen10] The CENATIC Team. National Open Source Competency Centre. National Open Source Software Observatory. *Report on the International Status of Open Source Software*. 2010. url: http://www.cenatic.es/publicaciones/onsfa?download=39%3Areport-on-the-international-status-of-open-source-software-2010 (visited on 08/14/2012).

[CH05] Kevin Crownston and James Howison. "The social structure of free and open source software development." In: *First Monday* 10.2 (2005). url: http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/1207/1127 (visited on 09/01/2012).

[Cha09] Scott Chacon. *Pro Git*. Apress, 2009. isbn: 978-1430218333. url: http://git-scm.com/book/ (visited on 06/13/2012).

[CHC05] Megan Conklin, James Howison, and Kevin Crowston. "Collaboration Using OSSmole: a repository of FLOSS data and analyses." In: *Proceedings of the 2nd international workshop on Mining software repositories*. 2005, pp. 116–120. doi: 10.1145/1083142.1083164.

[CKM12] Jonathan Corbet, Greg Kroah-Hartman, and Amanda McPherson. *Linux Kernel Development – How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It*. 2012. url: http://go.linuxfoundation.org/who-writes-linux-2012 (visited on 06/29/2012).

[CN09] M. Cataldo and S. Nambiar. "On the relationship between process maturity and geographic distribution: an empirical analysis of their impact on software quality." In: *Proceedings of ACM SIGSOFT symposium on the foundations of software engineering*. 2009, pp. 101–110. DOI: 10.1145/1595696.1595714.

[Coh08] Bram Cohen. *The BitTorrent Protocol Specification*. 2008. URL: http://www.bittorrent.org/beps/bep_0003.html (visited on 09/01/2012).

[Con+12] Joe Conway et al. *RPostgreSQL: R interface to the PostgreSQL database system*. R package version 0.3-2. 2012. URL: http://CRAN.R-project.org/package=RPostgreSQL.

[Cor08] Jonathan Corbet. *How to Participate in the Linux Community*. Tech. rep. August. 2008.

[Cor09] Jonathan Corbet. *How patches get into the mainline*. 2009. URL: http://lwn.net/Articles/318699/ (visited on 08/20/2012).

[CVS12] CVS Team. *CVS – Concurrent Versions System*. 2012. URL: http://www.nongnu.org/cvs/ (visited on 06/21/2012).

[Daf07] Carlo Daffara. *Estimating the number of active and stable FLOSS projects*. 2007. URL: http://robertogaloppini.net/2007/08/23/estimating-the-number-of-active-and-stable-floss-projects/ (visited on 09/01/2012).

[DKW09] Dieter Dowe, Karlheinz Kuba, and Manfred Wilke. *FDGB-Lexikon. Funktion, Struktur, Kader und Entwicklung einer Massenorganisation der SED (1945-1990) – Sachteil A: Arbeitszeit*. 2009. URL: http://library.fes.de/FDGB-Lexikon/texte/sachteil/a/Arbeitszeit.html (visited on 08/13/2012).

[DPG09] Massimiliano Di Penta and Daniel M. German. "Who are Source Code Contributors and How do they Change?" In: *Proceedings of the 16th Working Conference on Reverse Engineering*. 2009, pp. 11–20. DOI: 10.1109/WCRE.2009.41.

[DR08] Amit Deshpande and Dirk Riehle. "The Total Growth of Open Source." In: *Proceedings of the 4th Conference on Open Source Systems (OSS 2008)*. 2008.

[Dri09] Mark Driver. *Key Issues for Open-Source Software, 2010*. 2009. URL: http://www.gartner.com/DisplayDocument?id=1359127 (visited on 06/24/2012).

[EO07] Paul Eggert and Arthur David Olson. *Sources for Time Zone and Daylight Saving Time Data*. 2007. URL: http://www.twinsun.com/tz/tz-link.htm (visited on 06/24/2012).

[ES10] Brian S. Everitt and Anders Skrondal. "Law of large numbers." In: *The Cambridge Dictionary of Statistics*. 4th ed. 2010, p. 244.

[EU93] Council of the European Union. "Council Directive 93/104/EC of 23 November 1993 concerning certain aspects of the organization of working time." In: *Official Journal L 307* (1993), pp. 18–24.

[EY11] Ernst & Young. *Open Source Software im geschäftskritischen Einsatz*. 2011. URL: http://www.ey.com/Publication/vwLUAssets/Open_Source_Software_2011/$FILE/OpenSourceSoftware_2011.pdf (visited on 07/25/2012).

[Fal07] Erin Falconer. *Why the 9 to 5 Office Worker Will Become a Thing of the Past*. 2007. URL: http://www.pickthebrain.com/blog/why-the-9-to-5-office-worker-will-become-a-thing-of-the-past/ (visited on 06/15/2012).

[Fin09] Jim Finkle. *Oracle to buy Sun Micro, enters hardware market*. 2009. URL: http://www.reuters.com/article/2009/04/20/us-oracle-sun-idUSTRE53J2DD20090420 (visited on 08/29/2012).

[Gib10] George Gibbs. *Hudson, George Vernon – Biography*. In: *Dictionary of New Zealand Biography*. 2010. URL: http://www.TeAra.govt.nz/en/biographies/3h42/1 (visited on 06/24/2012).

[Git12] Junio C. Hamano et al. *Git – the stupid content tracker*. 2012. URL: http://git-scm.com/ (visited on 06/21/2012).

[GR12] Jesus M. Gonzales-Barahona and Gregorio Robles. "On the reproducibility of empirical software engineering studies based on data retrieved from development repositories." In: *Empirical Software Engineering* 17.1–2 (2012), pp. 75–89. DOI: 10.1007/s10664-011-9181-9.

[GS09] Georgios Gousios and Diomidis Spinellis. "A platform for software engineering research." In: *Proceedings of the 6th International Working Conference on Mining Software Repositories*. 2009, pp. 31–40. DOI: 10.1109/MSR.2009.5069478.

[GS12] Georgios Gousios and Diomidis Spinellis. "GHTorrent: Github's Data from a Firehose." In: *Proceedings of the 9th Working Conference on Mining Software Repositories*. 2012, pp. 12–21. DOI: 10.1109/MSR.2012.6224294.

[Han+04] Il-Horn Hann et al. "An empirical analysis of economic returns to open source participation." unpublished work. Carnegie Mellon University, 2004. URL: http://citeseerx.ist.psu.edu/viewdoc/similar?doi=10.1.1.89.6697&type=ab (visited on 08/29/2012).

[HBS10] Hans-Böckler-Stiftung. *Streiks und Aussperrungen – Kurzchronik 1945 bis heute*. 2010. URL: http://www.boeckler.de/wsi-tarifarchiv_4866.htm (visited on 08/29/2012).

[HCC06] James Howison, Megan Conklin, and Kevin Crowston. "FLOSSmole: A collaborative repository for FLOSS research data and analyses." In: *International Journal of Information Technology and Web Engineering* 1 (2006), pp. 17–26.

[Hec78] Paul Heckel. "A technique for isolating differences between files." In: *Communications of the ACM* 21.4 (1978), pp. 264–268. DOI: 10.1145/359460.359467.

[HGH07] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. "Release Pattern Discovery: A Case Study of Database Systems." In: *IEEE International Conference on Software Maintenance*. Oct. 2007, pp. 285–294. doi: 10.1109/ICSM.2007.4362641.

[HGH09] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. "Mining recurrent activities: Fourier analysis of change events." In: *2009 31st International Conference on Software Engineering - Companion Volume*. 2009, pp. 295–298. doi: 10.1109/ICSE-COMPANION.2009.5071005.

[HGR07] Israel Herraiz, Jesus M. Gonzalez-Barahona, and Gregorio Robles. "Forecasting the number of changes in Eclipse using time series analysis." In: *Proceeding of the 29th International Conference on Software Engineering Workshops*. January. 2007, pp. 9–10.

[HGS09] Jeffrey S. Hammond, Mary Gerush, and Justinas Sileikis. *Open Source Software Goes Mainstream – Convert Your Cost-Cutting Crisis Into An OSS Opportunity*. 2009. url: http://www.forrester.com/rb/Research/open_source_software_goes_mainstream/q/id/54205/t/2 (visited on 06/24/2012).

[HM76] J. W. Hunt and M. D. McIlroy. *An Algorithm for Differential File Comparison*. Tech. rep. 41. Bell Laboratories, 1976.

[Hof12] Gottfried Hoffmann. "Open Source Licenses and Project Growth." Diploma thesis. Friedrich-Alexander University, 2012.

[HR09] Philipp Hofmann and Dirk Riehle. "Estimating Commit Sizes Efficiently." In: *Open Source Ecosystems: Diverse Communities Interacting*. Vol. 299. 2009, pp. 105–115. doi: 10.1007/978-3-642-02032-2_11.

[ISO04] International Organization for Standardization. *8601:2004(E) Data elements and interchange formats – Information interchange – Representation of dates and times*. 3rd ed. 2004.

[ISO11] International Organization for Standardization. *9075-1:2011 Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)*. 4th ed. 2011. url: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=53681.

[ISO96] International Organization for Standardization. *9945-1 Information technology – Portable operating system interface (POSIX) – Part 1: System application programming interface (API) (C Language)*. 1996.

[Joh+85] W. Bruce Johnson et al. "An analysis of the stock price reaction to sudden executive deaths: Implications for the managerial labor market." In: *Journal of Accounting and Economics* 7.1–3 (1985), pp. 151–174. doi: 10.1016/0165-4101(85)90034-5.

[Kit08] Barbara Kitchenham. "The role of replications in empirical software engineering — a word of warning." In: *Empirical Software Engineering* 13.2 (2008), pp. 219–221. doi: 10.1007/s10664-008-9061-0.

[KRS10] Carsten Kolassa, Dirk Riehle, and Michel Salim. "The Commit Size Distribution of Open Source Projects." In: *ACM Transactions on Software Engineering and Methodology*. 2010.

[KSR12] Carsten Kolassa, Michel Salim, and Dirk Riehle. "The Empirical Commit Frequency Distribution of Open Source Projects." 2012.

[Lan11] Rob Landley. *Here's a unified kernel git repo covering 0.0.1 to the present*. 2011. URL: https://lkml.org/lkml/2011/7/22/386 (visited on 07/03/2012).

[Lom10] Claudia N. Lombardo. *Shorter Workweek in a Tough Economy*. 2010. URL: http://www.hrhero.com/hl/articles/2010/02/04/shorter-workweek-in-a-tough-economy/ (visited on 07/16/2012).

[LV89] R.K. Lind and K. Vairavan. "An Experimental Investigation of Software Metrics and Their Relationship to Software Development Effort." In: *IEEE Transactions on Software Engineering* 15 (1989), pp. 649–653. DOI: 10.1109/32.24715.

[Mac12] Holger Macht. "A Case Study in Open Source Patch Submission." Seminar paper. Friedrich-Alexander University, 2012.

[McV98] Larry McVoy. *A solution for growing pains*. 1998. URL: https://lkml.org/lkml/1998/9/30/122 (visited on 07/03/2012).

[MFH02] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. "Two case studies of open source software development: Apache and Mozilla." In: *ACM Transactions of Software Engineering and Methodology* 11.3 (2002), pp. 309–346. DOI: 10.1145/567793.567795.

[MS07] Andrew Mertz and William Slough. "Graphics with PGF and TikZ." In: *Proceedings of the Practical TEX2006 Conference*. 2007, pp. 91–99. URL: http://www.tug.org/TUGboat/tb28-1/tb88mertz.pdf (visited on 09/04/2012).

[MW11] Quinn E. McCallum and Stephen Weston. *Parallel R: Data Analysis in the Distributed World*. O'Reilly Media, 2011. ISBN: 1449309925.

[MyS08] MySQL Website. *Sun to Acquire MySQL*. 2008. URL: http://www.mysql.com/news-and-events/sun-to-acquire-mysql.html (visited on 08/29/2012).

[Nag+12] Rina Nagano et al. "Using the GPGPU for scaling up Mining Software Repositories." In: *Proceedings of the 34th International Conference on Software Engineering*. 2012, pp. 1435–1436. DOI: 10.1109/ICSE.2012.6227077.

[Net12] Netcraft Ltd. *June 2012 Web Server Survey*. 2012. URL: http://news.netcraft.com/archives/2012/06/06/june-2012-web-server-survey.html (visited on 06/22/2012).

[NT03] David Nichols and Michael Twidale. "The Usability of Open Source Software." In: *First Monday* 8.1 (2003). URL: http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/1018/939 (visited on 09/01/2012).

[OEC12] Organisation for Economic Co-operation and Development. *Average annual working time 2012 – Hours per worker*. 2012. DOI: 10.1787/20752342-table8. (Visited on 07/15/2012).

[OSI] Open Source Initiative. *The Open Source Definition*. URL: http://www.opensource.org/docs/osd (visited on 08/29/2012).

[PG12a] PostgreSQL Documentation Team. *PostgreSQL 8.4 Documentation – Chapter 8. Data Types – Date/Time Types – Time Stamps*. 2012. URL: http://www.postgresql.org/docs/8.4/static/datatype-datetime.html#AEN5360 (visited on 06/24/2012).

[PG12b] PostgreSQL Documentation Team. *PostgreSQL 8.4 Documentation – Chapter 9. Functions and Operators – Date/Time Functions and Operators*. 2012. URL: http://www.postgresql.org/docs/8.4/static/functions-datetime.html (visited on 06/24/2012).

[Pon10] Wouter Poncin. "Process mining software repositories." Master thesis. Eindhoven University of Technology, 2010.

[Pon11] Wouter Poncin. "Process Mining Software Repositories." In: *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*. 2011, pp. 5–14. DOI: 10.1109/CSMR.2011.5.

[Pre05] David S Prerau. *Seize the Daylight: The Curious and Contentious Story of Daylight Saving Time*. New York: Thunder's Mouth Press, 2005. ISBN: 978-1560256557.

[R12] R Development Core Team. *R: A Language and Environment for Statistical Computing*. ISBN 3-900051-07-0. R Foundation for Statistical Computing. Vienna, Austria, 2012. URL: http://www.R-project.org/.

[RA12] Christian Rodriguez-Bustos and Jairo Aponte. "How Distributed Version Control Systems impact open source software projects." In: *Proceedings of the 9th Working Conference on Mining Software Repositories*. 2012, pp. 36–39. DOI: 10.1109/MSR.2012.6224297.

[RG05] Gregorio Robles and Jesus M. Gonzales-Barahona. "Developer identification methods for integrated data from various sources." In: *Proceedings of the 2nd International Workshop on Mining Software Repositories*. 2005, pp. 106–110. DOI: 10.1145/1083142.1083162.

[RG09] Gregorio Robles and Jesus M. Gonzales-Barahona. "Geographic Location of Developers at SourceForge." In: *Proceedings of the 6th International Workshop on Mining Software Repositories*. 2009, pp. 144–150. DOI: 10.1145/1137983.1138017.

[RGH09] Gregorio Robles, Jesus M. Gonzales-Barahona, and Israel Herraiz. "Evolution of the core team of developers in libre software projects." In: *Proceedings of the 6th International Working Conference on Mining Software Repositories*. 2009, pp. 167–171. DOI: 10.1109/MSR.2009.5069497.

[RHH12] Daniel Rodriguez, Israel Herraiz, and Rachel Harrison. "On Software Engineering Repositories and Their Open Problems." In: *First International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*. 2012, pp. 52–55. DOI: 10.1109/RAISE.2012.6227971.

[Rie06] Dirk Riehle. "Geld verdienen mit Open-Source." In: *OBJEKTspektrum* 6 (2006), pp. 15–18.

[Rie07] Dirk Riehle. "The Economic Motivation of Open Source: Stakeholder Perspectives." In: *Computer* 40.4 (2007), pp. 25–32.

[Rie11] Dirk Riehle. "Controlling and Steering Open Source Projects." In: *Computer* July (2011), pp. 91–94. DOI: 10.1109/MC.2011.206.

[Rob+05] Gregorio Robles et al. "Evolution and Growth in Large Libre Software Projects." In: *Eighth International Workshop on Principles of Software Evolution*. 2005, pp. 165–174. DOI: 10.1109/IWPSE.2005.17.

[Rob07] Romain Robbes. "Mining a Change-Based Software Repository." In: *Proceedings of the 4th International Workshop on Mining Software Repositories*. 2007, p. 15. DOI: 10.1109/MSR.2007.18.

[Rou08] Margaret Rouse. *History of the punch card*. 2008. URL: http://whatis.techtarget.com/reference/History-of-the-punch-card (visited on 08/27/2012).

[Rup10] Nayan B. Ruparelia. "The history of version control." In: *SIGSOFT Software Engineering Notes* 35.1 (2010), pp. 5–9. DOI: 10.1145/1668862.1668876.

[Sca05] Walt Scacchi. "Socio-Technical Interaction Networks in Free/Open Source Software Development Processes." In: *Software Process Modeling*. Springer, 2005.

[Sha09] Weiyi Shang. "MapReduce as a general framework to support research in Mining Software Repositories." In: *Proceedings of the 6th International Working Conference on Mining Software Repositories*. 2009, pp. 21–30. DOI: 10.1109/MSR.2009.5069477.

[SKB12] Emad Shihab, Yasutaka Kamei, and Pamela Bhattacharya. "Mining challenge 2012: The Android platform." In: *Proceedings of the 9th Working Conference on Mining Software Repositories*. 2012, pp. 112–115. DOI: 10.1109/MSR.2012.6224307.

[SMG12] Vibha Sinha, Senthil Mani, and Monika Gupta. "Mince: Mining change history of Android project." In: *Proceedings of the 9th Working Conference on Mining Software Repositories*. 2012, pp. 132–135. DOI: 10.1109/MSR.2012.6224271.

[Spi06] Diomidis Spinellis. "Global Software Development in the FreeBSD Project." In: *Proceedings of the international workshop on Global Software development for the practitioner*. 2006, pp. 73–79. DOI: 10.1145/1138506.1138524.

[Ste12] Harlan Stenn. *BitKeeper diffs*. 2012. URL: http://lists.ntp.org/pipermail/bk-ntp-stable-send/2012-March/000556.html (visited on 07/03/2012).

[Sto12] David Stoffer. *astsa: Applied Statistical Time Series Analysis*. R package version 1.0. 2012. URL: http://CRAN.R-project.org/package=astsa.

[Taf06] Darryl K. Taft. *Startup Helps Assess Open-Source Projects*. 2006. URL: http://www.eweek.com/c/a/Linux-and-Open-Source/Startup-Helps-Assess-OpenSource-Projects/ (visited on 08/23/2012).

[TH10] Yuri Takhteyev and Andrew Hilts. *Investigating the Geography of Open Source Software through Github*. 2010. URL: http://takhteyev.org/papers/Takhteyev-Hilts-2010.pdf (visited on 09/01/2012).

[THZ09] Ran Tang, Ahmed E. Hassan, and Ying Zou. "Techniques for identifying the country origin of mailing list participants." In: *Proceedings of the 16th Working Conference on Reverse Engineering*. 2009, pp. 36–40. DOI: 10.1109/WCRE.2009.46.

[Top12] TOP500. *Operating Systems Family – Systems Share Development Over Time*. 2012. URL: http://i.top500.org/overtime (visited on 06/22/2012).

[Tor07a] Linus Torvalds. *Google Tech Talk: Linus Torvalds on git*. 2007. URL: http://www.youtube.com/watch?v=4XpnKHJAok8 (visited on 08/02/2012).

[Tor07b] Linus Torvalds. *Re: Trivia: When did git self-host?* 2007. URL: http://marc.info/?l=git&m=117254154130732 (visited on 06/29/2012).

[Tor+12] Linus Torvalds et al. *List of maintainers and how to submit kernel changes*. 2012. URL: https://github.com/torvalds/linux/blob/master/MAINTAINERS (visited on 08/29/2012).

[Tor91] Linus Torvalds. *Free minix-like kernel sources for 386-AT*. 1991. URL: https://groups.google.com/forum/?fromgroups#!msg/comp.os.minix/4995SivOl9o/GwqLJlPSlCEJ (visited on 06/29/2012).

[Van+11] Michael VanHilst et al. "Measuring effort in a corporate repository." In: *Proceedings of the International Conference on Information Reuse and Integration*. 2011, pp. 246–252. DOI: 10.1109/IRI.2011.6009554.

[Wau10] Robin Wauters. *Geeknet Sells Open Source Directory Ohloh To Black Duck Software*. 2010. URL: http://techcrunch.com/2010/10/05/geeknet-sells-open-source-directory-ohloh-to-black-duck-software/ (visited on 08/23/2012).

[Wei12] Florian Weikert. "Product Features in Commercial Open Source Software." Bachelor thesis. Friedrich-Alexander University, 2012.

[Wic09] Hadley Wickham. *ggplot2: elegant graphics for data analysis*. Springer New York, 2009. ISBN: 978-0-387-98140-6. URL: http://had.co.nz/ggplot2/book.

[YR07] Ligou Yu and Srini Ramaswamy. "Mining CVS Repositories to Understand Open-Source Project Developer Roles." In: *Proceedings of the 4th International Workshop on Mining Software Repositories*. 2007, pp. 8–11. DOI: 10.1109/MSR.2007.19.

[Yua+10] Lin Yuan et al. "Mining Roles of Open Source Software." In: *Proceedings of the 2nd International Conference on Software Engineering and Data Mining*. 2010, pp. 548–554.