Re-Allocation of Resources during Releases [*]

Md Tajmilur Rahman
Concordia University
Montreal, QC H3G 1M8
438-932-2288, +1
mdt_rahm@encs.concordia.ca

Peter C. Rigby
Concordia University
Montreal, QC H3G 1M8
514-848-2424, +1
peter.rigby@concordia.ca

ABSTRACT

This section will be written at the end.

Categories and Subject Descriptors
K.6.3 [**Software Management**]: Software Development, Software Resource Management, Resource Reallocation

General Terms
Experiment, Human Factors, Resource Management, Reallocation

Keywords
Resource Reallocation, Code Ownership, Defect Density, Software Releases

1. INTRODUCTION

Software projects are notorious for going over budget and schedule. Rush periods are often get seen before a major release that turn the developers into dinosaurs as Frederick Brooks likens in his benchmark study "The Mythical Man Month" [5]. This "Rush To Release (RTR)" can be prompted either by external forces such as decisions by management to include new features in the release or to release earlier to beat a competitor. Alternatively, the rush may simply be due to inappropriate or unrealistic scheduling. Whatever the reason is it is an obvious. Regardless of the causes, the rush to release stresses developers and often requires developers to work on unusual, high priority or critical areas of the system. In this paper we study how RTR may effect project organization and introduce technical debt. We want to propose a method to observe, analyze and summarize the results of revisions found around releases. We intend to infer behavior of the developers related to the process around release time. The key research questions that we expect to answer with our methodology are described in section 1.1.

[*]Copyright note

During the merge window, developers are free to provide any change that is ready for release. During the release candinate period, we would expect to see fixs on the files that were changed during the previous merge window. We measure this by comparing the Jaccard distance of merge window with the subsequent rc period, compared with the merge window and subsequent merge window.

We observe the developers' working areas to understand the allocation of the resources within an open source software development project to understand the reallocation or distribution takes place in a software development process which may cause disruptive events. We attempt to identify the project's different release times and calculate the difference between different development periods within a releases cycle to identify which new areas are being worked on and what is the behavior of the developers between normal development period and the period when RTR is prompted. The historical data from git that we are working on for this purpose will help us to extract a lot of information like calculating the developers' working areas and time-frame of each release as well as different periods in a stable release period. This information will help us to identify the criteria of the resources, their roles and behaviors, code ownerships of the domains they are working in.

We have organized this paper as follows. In Section 2, we describe some background and motivations followed by the summaries of related works in Section 3. Section 4 will describe methodology of our work and explanation of the data we are working with for this research. In section 5 we will dicuss the results of the findings. Finally section 6 will conclude the paper.

1.1 Research Questions

We have set our research questions focusing the key properties of software development process, allocation of developers throughout the process, especially for open source software development. First couple of research questions are to understand the basic structure and strategies of the development and release and merging process practiced by Linux Kernel development team. Rest of the questions are to understand the distribution of developers' working areas and the behavior of developers in different segments of a release period.

Q1: What is the release process used by the project?
We will be giving a qualitative description of Linux

Kernel development in brief to give anser to this question.

Q2: How many developers are allocated troughout different segments of a release period? What significant difference can be observed between the segments of an entire release period?
Answer to this question will give us a big picture of Linux Kernel development community as well as the contrubution of developers in different periods within a stable release period. We will calculate Jaccard Distance to differentiate the behavior of developers during RTR and normal development period.

Q3: Do developers work on different areas of the system around the time of release?
We want to see if developers are looking at different types of files in the code-base around the time of release which is being considered as the merge period or merge window[12]. We are expecting this would be the RTR are in a release time where technical debt may be introduced.

Q4: Do developers work in others' code in a high proportion during the rush period? Many developers are working on many files. We will find out which files has been churned mostly by which developer so that we can understand how developers are distributed among the system.

Q5: Are there certain areas of the system that receive increased attention (i.e. do developers focus on a smaller set of files around releases)?
To answer this question we will look into the frequency of file churns made into files. Difference or distance of the files between normal development period and the RTR period or merge period of a release.

2. BACKGROUND

There may have lower developers productivity [15, 8] which may cause inefficient run in the rush moments in a release period. There is a substantial and important body of literature on risk in software engineering. Boehm identified the most important risks encountered by software project managers and described successful risk management practices [4, 16, 3]. Some of the risks identified are related to disruptive events, such as the introduction of a new technology, but most are macro risks associated with running a project, such as developing the wrong functionality. General risk mitigation strategies can be difficult to apply to specific disruptive events. There may be various kinds of disruptive events for example, as a release approaches; developers take shortcuts that introduces technical debt. If it is not repaired, the long term quality of the system will suffer. Another example can be placed, if a lead developer who owns an important part of the code-base leaves and if steps to train other developers were not taken, it will become a dead area of the system and will be difficult to modify and maintain. Also often management reorganizes the developers on a company's projects during the rush period around the release time, with the result that developers move to code-bases for which they have less experience. The reorganization introduces new perspectives and expertise that can lead to faster release; however, it can also result in a drop in productivity and the unnecessary re-writing of large portions of the system that the new developers do not understand. In this paper, we plan to take the measures on this last example among them mentioned above.

3. RELATED WORKS

Hindle worked on release pattern discovery via partitioning [1] to propose an approach to characterize a project's behavior around the time of major and minor releases while we are trying to study the behavior of the development resources around the time of release. In this research they proposed a method of observing, analyzing and summarizing the results of metrics of revisions found near releases. They have characterized a project's behavior around the time of major and minor releases. This is done by partitioning the observed activities like the artiffect check-ins around the dates of major and minor releases, then look for reasonable patterns. Hindle divided the revisions in each release in 4 different classes, Source Code, Testing, Building, and Documentations. On the other hand Cook did an interesting job, he inserted sensors and monitors into the development process but Hindle analyzed the data to understand what happened in the past [13]. Basically Hindle worked in a reverse way than Cook did. Another research work we would like to mention was done by Damian where they have worked on the role of domain knowledge and cross functional communication among the OOS development teams [9]. Posnett did some dual ecological measures of focus in software development [10]. Posnett's measure was for the more general view that unifies developer focus and artifact ownership. He analyzed i) developer artifact contribution to network to a predator-prey food web ii) drew upon ideas from ehology to produce a novel and iii) conceptually unified view of measuring focus and ownership. Another study was done by F. Rahman about the authorship of the code-bases in OSS development [11].

4. METHODOLOGY AND DATA

This section presents our methodology for discovering information which can give us the idea to get the answers to our research questions. We have collected the development history data of Linux kernel. It is a data source containing the the historical data of Linux kernel development since 2005. We are going to present the steps involved in this process and then we will follow up with an application of our methodology in a case study. In order to address our research questions, we obtain key measures of project evolution from the archival data collected from the distributed version contron (DVC) system of Git where all information of the OSS project "Linux Kernel Development" is recorded in electronic form. Many other OSS projects archive similar data, so the techniques used here can be replicated on any such project. We used the data elements extracted from the archival source to construct a number of measures on the commit log records to understand the behavior of the developers.

Our methodology can be summarized as: Extracting Data for revisions and releases (Section 4.1); Partitioning the release version numbers (Section 4.2); Finding the commits associated to a version of Kernel (Section 4.3); Get timespan between each release (Section 4.4); Calculate developer areas (Section 4.5); Finding code ownership (Section 4.6).

4.1 Data Extraction

We went for the VCS of a target project and either mirror the repository or download every revision and commit log history data. From DVCSs such as Git we extract the revisions and release information. We wrote Perl scripts to extract date that was further processed to obtain details. Manual inspection was used to resolve problems and things like that in cases where all automated techniques failed. We then put them into a database. We have used PSQL database to create tables in, to store our extracted data. These extracted data will be analyzed by us later on. Per each revision the information extracted includes the commit id, tree id, author of the revision, date of revision, the name of the revised file, parent and child info for the revision and the detail log information. Once extraction is completed we are ready to partition the version numbers (Section 4.3) and duration of each release (Section 4.4).

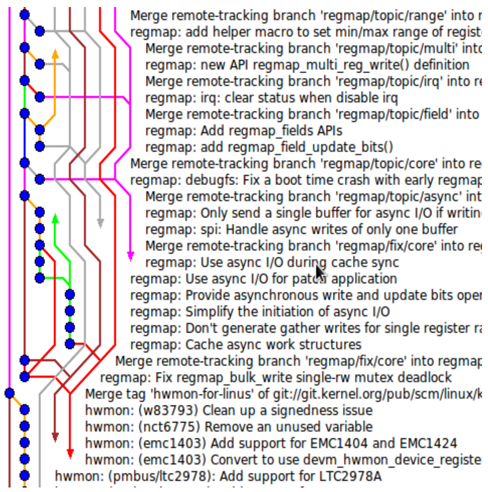
4.1.1 Explanation

A Linux kernel development release version having maximum length of information does look like **linuxvA.B.C-rcP** where A, B, C, P are numeric. If a release versioning looks like "linuxv2.6.13" it tells us that this particular release is the 13th minor release in the seriese of the 6th major release of kernel version 2. "linuxv2.6.13-**rc1**" says that after the 12th stable minor release had published, development for the next release has been started and rc1 is the first candidate on the way to the next release. It also ensures that next release is going to be another minor release which is "linuxv2.6.13". We have last couple of release candidates for "linuxv2.6.12" and first couple of release candidates for "linuxv3.12" which is not useful for our calculation. So we are having 340 releases including major, minor, micro and rc; 39 stable releases from "linuxv2.6.11" to "linuxv3.11" having complete data with us. We have total 400441 distinct commits and for 372943 of them have change information for 77082 different files in our hand where we have 14599 developers' working history among 14621 distinct developers. Surprisingly 14569 individual files didn't have any change (addition or deletion) but was being committed at least once. So we found them in our released wise datasets.

4.2 Partitioning Release Numbers

We stored the extracted git commit log data into the tables git_commit and git_revision where all the basic and detail log information for a particular commit was mentioned in the first table and second one containing which commit belongs to which version of Linux kernel development and change details like path modified, new path created due to the change, how many addition and how many deletion occurred in a particular commit etc. By joining these tables we can easily get the dates of each version and from the version number which is a combination of different types of releases we can determine which commit belongs to which version and release, and also what type of release that is.

Commits those are made for pushing a release have release tags associated to them by which we can understand which commit belongs to which version of release; is this a major release or minor or a micro stable release. Another information we have captured that is **rc** which indicates that the particular commit was a release candidate. These information and splitting off release numbers will help us accumulating commits for different releases.



```
Merge remote-tracking branch 'regmap/topic/range' into r
regmap: add helper macro to set min/max range of regist
  Merge remote-tracking branch 'regmap/topic/multi' into
  regmap: new API regmap_multi_reg_write() definition
  Merge remote-tracking branch 'regmap/topic/irq' into re
  regmap: irq: clear status when disable irq
  Merge remote-tracking branch 'regmap/topic/field' into
  regmap: Add regmap_fields APIs
  regmap: add regmap_field_update_bits()
Merge remote-tracking branch 'regmap/topic/core' into re
regmap: debugfs: Fix a boot time crash with early regmap
  Merge remote-tracking branch 'regmap/topic/async' int
  regmap: Only send a single buffer for async I/O if writin
  regmap: spi: Handle async writes of only one buffer
  Merge remote-tracking branch 'regmap/fix/core' into reg
  regmap: Use async I/O during cache sync
regmap: Use async I/O for patch application
regmap: Provide asynchronous write and update bits oper
regmap: Simplify the initiation of async I/O
regmap: Don't generate gather writes for single register ra
regmap: Cache async work structures
 Merge remote-tracking branch 'regmap/fix/core' into regmap
 regmap: Fix regmap_bulk_write single-rw mutex deadlock
Merge tag 'hwmon-for-linus' of git://git.kernel.org/pub/scm/linux/k
hwmon: (w83793) Clean up a signedness issue
hwmon: (nct6775) Remove an unused variable
hwmon: (emc1403) Add support for EMC1404 and EMC1424
hwmon: (emc1403) Convert to use devm_hwmon_device_registe
hwmon: (pmbus/ltc2978): Add support for LTC2978A
```

Figure 1: *Branching in git DAG for linux kernel development.*

4.3 Finding Commits for Releases

We have to sort out and accumulate all the commits together those are belonging to any of the releases that we have. We have all 400441 commits in our hand but though we can't plot them into a single time domain to assign different set of commits to different release periods because development goes in parallel on different branches in git so commits also go in parallel. If there are multiple branches, surely there are multiple merges and the commits prior to a crisscross merge are all ancestors of the heads of all subsequent branches [7]. Figure 1 shows how a git DAG looks like having so many crisscross merges in a busy branching snap-shot of the git DAG of Linux Kernel development. So it is possible to make a commit months ago from today while several releases may have already been made in the mean time but that commit can belong to any future release. Commits that are being made to a branch may get merged with other branch or the head branch during the release period. This is why we had to track all the way down the trees for each of the 340 releases in the git DAG to find out which commit contributes to which release. We could do this using Git DAG Traversing (GDT) algorithm as shown in Algorithm 1. After applying GDT we collected 381152 out of 400441 commits those are well organized in the git DAG for Linux Kernel development. As we mentioned earlier that we have last couple of release candidates for "linuxv2.6.12" and first couple of release candidates for "linuxv3.12". We are choping out them because we don't have complete release commit data for version 2.6.12-rc as well as 3.12-rc releases. Out of 400441 commits 19289 are belonging to those releases. So finally we have 381152 commits for 340 releases in our database to progress our work on.

4.4 Time-Span of Releases

Another information that we require is what are the durations of the releases of Linux kernel development. To achieve that we joined the table "git_refs_tags" where all the releases are stored including the splitted release numbers and the ta-

Algorithm 1 GDT: to find Commits within Releases

```
Require:
 1: commit id for all releases
Ensure:
 2: for all commits do
 3:   start from new commit, release R
 4:   store commit in "git_commit_release"
 5:   if has parent commit then
 6:     for each parent commit do
 7:       if if parent commit has a release tag then
 8:         repeat step 1
 9:       else if parent commit already stored then
10:         compare release versions
11:         if release version not smaller than R then
12:           update existing_release ← R
13:         end if
14:       else
15:         store parent commit in "git_commit_release"
16:         commit id ← parent commit id
17:         repeat step 3
18:       end if
19:     end for
20:   else if no parent commit then
21:     repeat step 1
22:   end if
23: end for
```

ble "git_commit" to give us all the dates of commits for each and every releases, so that we can easily find out the duration between two consecutive releases (including release candidates RC). This information will significantly help us because we need to see what is the general development period and what are the merge periods which can be treated as RTR period within a release period. We also need to understand how developers are working, what is the impact of their changes made in the code-base during a particular release period later on. Table 1 shows a small portion of this information. Preior to calculate the developers' areas in secion 4.4 we require this information.

Table 1: Different Releases

Release	Type	Start Date	End Date
linuxv2.6.12-rc2	rc	-	2005-04-16
linuxv2.6.12-rc3	rc	2005-04-16	2005-04-20
linuxv2.6.12-rc4	rc	2005-04-20	2005-05-07
linuxv2.6.12-rc5	rc	2005-05-07	2005-05-24
linuxv2.6.12-rc6	rc	2005-05-24	2005-06-06
linuxv2.6.12	micro	2005-06-06	2005-06-17
linuxv2.6.13-rc1	rc	2005-06-17	2005-06-29
linuxv2.6.13-rc2	rc	2005-06-29	2005-07-05
linuxv2.6.13-rc3	rc	2005-07-05	2005-07-13
...	...	...	...
...	...	...	...
...	...	...	...
linuxv3.11	minor	2013-08-25	2013-09-02

Note that the very first row has no information for the "Start Date" because we don't have the complete information prior to stable release "linuxv2.6.12" and we have chopped them out. This will not affect our methodology and measurements. From the extraction we found complete records for releases "linuxv2.6.13" to "linuxv2.6.39" then it is "linuxv3.0" to "linuxv3.11". Our farther works will be proceeded based on these 39 releases only. As we don't have the complete release information for 3.12 or later as well as for 2.6.12 and earlier ones.

4.5 Calculate Developers' Area

Developers are working throughout the release time. Here "Developers' Area" means the files developers work in within a release cycle. It doesn't refer to the file itself but it is important to understand which files are being touched by which developers. We see that authors are committing files almost everyday throughout a release but this does not make sure that these commits are going to the next release because commits are being made in different branches and all branches are not supposed to get merged to the next release. We have the commit log data in our hand and we know what is the duration for each and every release (i.e. start and end dates). Now it is possible to find out which commits were made for which files by which developer within a certain time range of a release.

A relatively straightforward discipline of Linux Kernel Development team is followed with regard to the merging of patches for each release [12]. At the beginning of each development cycle, the "merge window" is said to be opened. At that time, code which is deemed to be sufficiently stable (and which is accepted by the development community) is merged into the mainline kernel. The bulk of changes for a new development cycle (and all of the major changes) will be merged during this time, at a rate approaching 1,000 changes ("patches," or "changesets") per day. Strategically linux starts a new kernel release with the merging to get their development branch ready to start development for the next release. So there are two main segments in a release period, one is merge window or merge period (MP) where all the developments works get merged together from different branches. So we can say this is the main development period of a release. A merge period remains opened exactly for two weeks [12], many developers are working in many other branches to develop new features once they are ready to push it to the main kernel then they wait for the upcoming merge window and once it's merge period they all contribute their works for the next release during this Merge Period. Some developers who are mainly responsible to merge new developments verify and accept the new changes and merge them into the main kernel. Another perod in a release cycle is the period of candidate releases which gets opened once the merge window is closed where mainly the bug fixing of minor enhancement works are to be worked on and pushes by small candidate releases. Once all the fixing and enhancements are done for all the codes that had been merged into the main kernel, the release is published. We are calling this period of development as the release period (RP) because no new feature development works do not get accepted during this time, it is just the period prior to publishing a new release.

4.5.1 *Developers' Area in a Merge Period*

Merge Perid can be determined from the previous release date (of any of major, minor or micro releases) to the date of rc-1 (i.e. date of the first candidate release). For example if the date of release for "linuxv2.6.12" is 2005-06-17 and date of pushing the rc-1 for the next release “linuxv2.6.13” is 2005-06-29 then this time period of of 12 days is going to be being called the merge window. We are considering this segment as a Rush To Release or RTR because thousands of commits gets pushed in the main branch during this time and developers remain in a rush to pack up merging all the branches those are to go with this release. Table 2 shows merge periods of some releases.

Table 2: Release Merging Periods

Release	Start Date	End Date
linuxv2.6.13	2005-06-17	2005-06-29
linuxv2.6.14	2005-08-28	2005-09-12
linuxv2.6.15	2005-10-27	2005-11-11
linuxv2.6.16	2006-01-02	2006-01-17
linuxv2.6.17	2006-03-20	2006-04-02
linuxv2.6.18	2006-06-17	2006-07-06
linuxv2.6.19	2006-09-19	2006-10-04
linuxv2.6.20	2006-11-29	2006-12-13
linuxv2.6.21	2007-02-04	2007-02-20
linuxv2.6.22	2007-04-25	2007-05-12
linuxv2.6.23	2007-07-08	2007-07-22
linuxv2.6.24	2007-10-09	2007-10-23
linuxv2.6.25	2008-01-24	2008-02-10

We have the commits contributed to releases and every record explains us who committed the file when did this get churned. If we run an SQL query then we can easily find out for every author and for every particular file how many times a it has been committed by an author and how many churns (addition + deletion = churn) developer has made. We calculate developers’ working areas and store the information into a table. A part of that table is shown in table 3.

Table 3: Developers’ Areas in Merge Period

Author	File Path	Commits	Churn
D. S. Miller	include/.../pci.h	1	8
V. Hanquez	arch/.../cpu.c	1	14
A. Bunk	drivers/.../shmem.c	1	2
J. Juhl	arch/.../generic.c	1	3

In this table we see in a merge period within a release cycle D. S. Miller has made change in pci.h file only once where he has made 8 churns. In the similar approach we calculate developers’ areas in release period and also for merge period.

4.5.2 *Developers’ Areas in Release Period*

As like merge period we can extract developers working in code files during release period. Release Period here in Linux Kernel development process is being considered as since the date of rc-1 release (i.e. first release candidate opening the gate for the development of the next release) to the date of the next stable release. For example if the date of rc-1 for “linuxv2.6.13” after releasing “linuxv2.6.12” is 2005-06-29 and the date of publishing the release “linuxv2.6.13” is 2005-08-28 then we are calling this 60 days of development period for the release as the Release Period. Table 4 is showing some development periods of different releases.

These information above will greatly help us to answer our first couple of research questions and to progress our further works.

Table 4: Release Periods

Release	Start Date	End Date
linuxv2.6.13	2005-06-29	2005-08-28
linuxv2.6.14	2005-09-12	2005-10-27
linuxv2.6.15	2005-11-11	2006-01-02
linuxv2.6.16	2006-01-17	2006-03-20
linuxv2.6.17	2006-04-02	2006-06-17
linuxv2.6.18	2006-07-06	2006-09-19
linuxv2.6.19	2006-10-04	2006-11-29
linuxv2.6.20	2006-12-13	2007-02-04
linuxv2.6.21	2007-02-20	2007-04-25
linuxv2.6.22	2007-05-12	2007-07-08
linuxv2.6.23	2007-07-22	2007-10-09
linuxv2.6.24	2007-10-23	2008-01-24
linuxv2.6.25	2008-02-10	2008-04-16

4.6 Finding Code Ownership

So many developers are working from different development communities throught different development branches. We tried to investigate the ”code ownership” [2] evolved in Linux Kernel development. Here code ownership is going to be represented as a percentage of making churns or commits by a developer with respect to the total churns or commits made to a particular file. From the data that we have for complete releases (“linuxv2.6.13” to “linuxv3.11”) we found 75138 distinct files that have been churned or commited by developers. We investigate the total number of commits, total number of churns for each and every file, we also investigate how many developers contributed for these commits and changes. This information yet does not tell about the ownership, we need to find out ownership for each developer in different development areas in a release cycle. We already have the collection of developers working in merge period and release period within a release cycle and stored them with corresponding files they have worked on, how many commits a developer made to a particular file and how many changes he/she has made. We now update those information with the ownership information. Equation 1 is the base for calculating ownership.

$$\omega = \frac{n}{N} * 100 \tag{1}$$

Where ω is the ownership and n is the number of total changes made by an atuthor of a file, N is the total number of changes made for a particular file. To determine a developer be a owner of a file we are considering that if the developer makes changes more than 80% of the total churn that has been made by all contributors so far for that file, then that developer can be called as a owner of that file, similarly as C. Gutwin did [6] for finding out the main developers’ group for a project. We already mentioned that 14569 individual files didn’t have any change (addition or deletion) but was being committed at least once. So we found them in our release wise datasets that’s why in addition to Gutwin here we are involving commits to determine ownerships but only for those files that do not have churn information. If a file has no addition or deletion information in it’s commit log but has been committed by some one at least once then that or those authors may have the ownership for that file. So, we have made a little change in equation 1 which becomes like:

$$\omega = \begin{cases} n/N & \text{if } N > 0 \\ c/C & \text{if } N = 0 \end{cases} * 100 \qquad (2)$$

Where c is the number of commits made by the author and C is the total number of commits made for the particular file. We calculate code ownership for both Merge Period and Release Period. We also canculate general ownership for all the developers for all the files they have worked ever. Table 6 represents a part of the data which shows the ownerships of the developers in each release cycle.

Table 5: Code Ownership in RP

Author	Linuxv	Path	ω (%)
Thomas Gleixner	2.6.24	.../numa_64.h	0.0000
Jeff Kirsher	3.1	.../ethtool.c	0.0000
Sathya Perla	2.6.29	.../hwlib.h	100.00
Roel Kluin	2.6.24	.../innovator.h	100.00
David S. Miller	2.6.24	.../visasm.h	100.00
Stephen Hemminger	2.6.24	.../qla3xxx.c	12.083
Randy Dunlap	2.6.24	..._core.c	100.00

The example given above is calculated for finding ownerships of developers worked for different files in different releases. Ownership is calculated for all the developers in release periods as well as in merge period. In addition we have calculated which file is owned by which developer and also the percentage of working with owned files during merge meriod and release period seperately. These extractions will help us drawing the results from our findings to answer our 3rd and 4th research questions.

5. RESULTS

In this section we present results from several quantitative analysis of the archival data from the Linux Kernel development project. The measures we derive from these data are well-suited to answer our research questions.

What is the release process used by the project?

In the early 1990's the Linux kernel development was not so busy affair [12]. There were very small number of users and developers involved at that time. Developers use a loosely time-based release process. Linux publishes it's major kernel release in every two or three months. If we look at figure 2 the upper Line represents how the stable releases are going on since 2005. From the figure we can see the average duration of each stable release is 62.47 although "linuxv2.6.24" was significantly hight as exception, it took 92.8 days. The bottom line which is much steady than the upper one is representing the durations of merge period in each stable release. If we put the releases in a time domain we can see a picture shown in figure 3 where we see that the stable releases are all going on maintaining almost a similar frequency and similar pattern. The highest picks are the merge periods then the smaller picks between the highest ones are representing the rc releases. For more clear view we have plotted only 100 releases of all 340 releases.

Linux Kernel development uses Rolling Release Development Model. Usually every micro release (ex. linuxv2.6.x) is a major kernel release with new features, internal API changes, and more. A minor release (ex. linuxv3.11) can have around 10 thousand changesets with several hundred thousands of churned lines of codes (LOC). At the beginning of each stable release the merge window i.e. merge period gets opened and all the accepted and stable codes from the community are merged having around 1000 of churns everyday. Merge period approximately for 14 days as shown in the figure 2. The bottom line is representing the merge windows of every releases. We don't have the merge period data for the very first one that's why it's starting from 0.

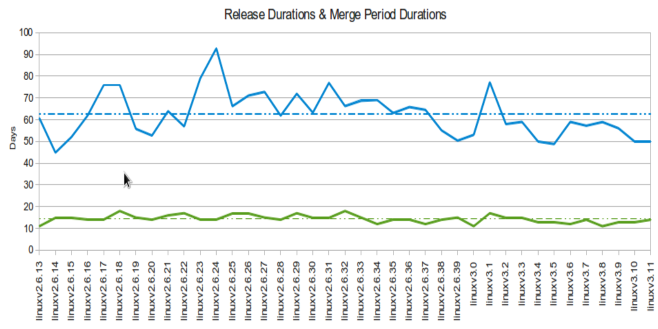


Figure 2: *Durations of Merge Window and Release Cycle*

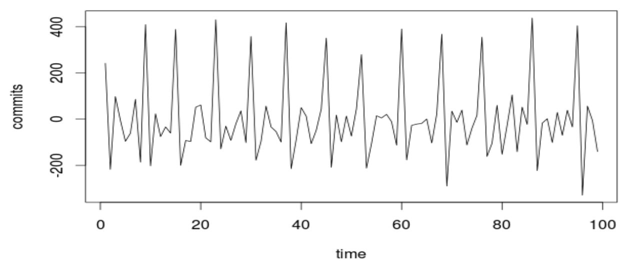


Figure 3: *Time Domain for Releases*

After getting all the major works merged development period for the next release is announced to be opened. If the next release is going to be linuxv2.6.14 then the release pushed at the end of the merge window will be called linuxv2.6.14-rc1. **-rc1** release indecates that the time to stabilize the next kernel has begun. These candidate releases continu to be pushed for once a week. If there is any bug fix or any high-priority change only then that goes to the main line during this period. With the continuous rc releases when the new kernel seems to be stable then the it goes for the stable release. Again after the core features are pushed to the main line, merge window gets opened and accepts all other feature developments, patchsets from other branches.

As a summary, the description above helps to address some of the questions about how Linux Kernel development is organized. Answer to this first research question provides

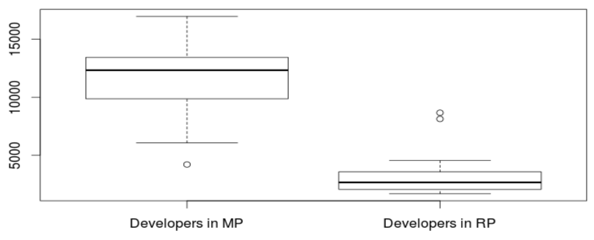


Figure 4: *Developers working in MP & RP*

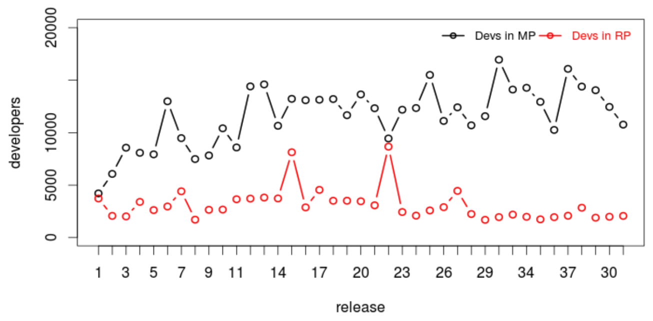


Figure 5: *Increasing contribution in RDP*

essential background for understanding our quantitative results. We will look into more detail of the development process, distribution of developers among the release segments as well as will try to find answers for our other research questions in the next sections.

5.1 Difference between release segments

How many developers are allocated troughout different segments of a release period? What significant differ-ence can be observed between the segments of an entire release period?

In order to see how many developers contribute to the new featcher developments and how many contribute in merging and fixing, review and accepting codes from the community during the merge period, we observed the developers' area in Merge Period and developers' area in Release Period. Out of all 381152 commits belonging to all 340 releases 292926 commits were made during the time of merge period which is 76.85% (more than 2/3) of the total. These commits were made by 14621 individual developers. We now keeping only the records belong to release 2.6.13 to 3.11 because we have complete information for thse releases, so we had to remove the commit records belong to linuxv2.6.12 and linuxv3.12 and later. 11209 developers work in merge period and 7625 developers work around the release period. We found that 4908 developers contribute in both merge period and the release period that means near about half (43.78%) of the developers work during the merge period are shared that means they work in both merge and release periods.

In figure 4 we see the boxplot showing that a very large number of developers contribute during the merge period which is in a sense the main development period because all the development branches get merged together during this time, all the new features come into the main branch of the kernel. In release period significantly small number of developers work and this number looks pretty steady for all the releases only a couple of high number appeared in two releases as we see in figure 4. The number of contributions during the main development period over releases looks going up in an increasing manner as shown in figure 5. On the other hand the bottom line for the release developers is pretty steady for all the releases.

Once we found different releases and the release periods as well as we calculate the developers' areas and code ownerships for developers we also would like to know what is the difference being created by making changes between the two main segments within an entire release development cycle (i.e. Merge Period MP and Release Period RP). To observe this we want to calculate the difference using Jaccard Similarity Coefficient [14] which is represented by equation 3.

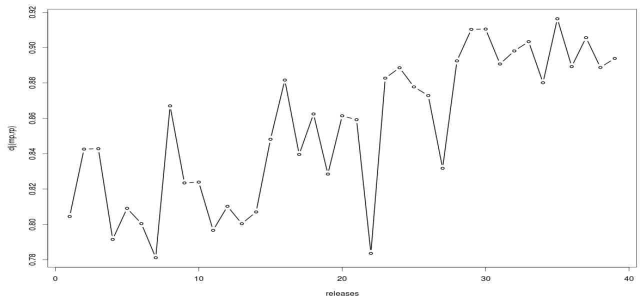


Figure 6: *Jaccard Distance between the Merge Period and Release Period*

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{3}$$

The Jaccard Distance can be obtained by subtracting the Jaccard Coefficient from 1.

$$d_j(A,B) = 1 - J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} \tag{4}$$

Here A and B are two sets. For calculating Jaccard Distance between two releases we are considering the files that have been worked for, in a particular release. If set A is the set of files worked in MP and files worked for in another release is set B then In figure 6 the Jaccard distances between MP and RP for each release can be represented. The average jaccard distance between merge period and release period for all the release cycles is 0.85 and we see the jaccard distance is also going upward by releases except the only exception . So we can say that developers work in very different areas once the merge window is closed and all core developments, feature developments, merging other branches regarding the previous release are done. Regarding the increase in the Jaccard Distance for the minor releases (since linuxv3.0) we can say that in minor releases developers work in more versatile areas than micro releases.

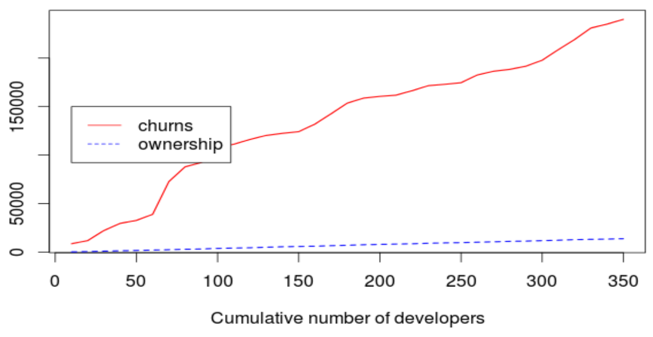


Figure 7: *Cumulative Number of Churns vs Percentage of Owned Files in release 2.3.13*

5.2 Developers' contributions around the time of release

Do developers work on different areas of the system around the time of release?
Do developers work in others' code in a high proportion during the rush period?

We are supposed to find the answers for the two research questions together in this section. Developers work for the software continuously. Some are working for the core development some people are for the fixing and minor enhancements and some for feature developments, planning to merge new features with any up coming release but we would like to understand what number of files they change during the time of regular development period and how many during the time of RTR. How many of them they really own and how many files they work that they do not own. We also want to know about the code ownership of the developers around the time of release and general code ownership i.e. which files a developer has worked making more than 80% chruns in lifetime.

Since the release of "linuxv2.6.13" till "linuxv3.11" 2889 developers out of 14621 worked for the files they owned. Although they have worked for other files that they do not own. So the percentage of working with owned files is 19.82% through out the life time (release 2.6.13 to 3.11). We examine the contribution in writing code and making churns by the developers to understand the behaviour and to see if the impact of this 19.82% on the graph. Figure 7 plots the cumulative representation of churnes files and percentage of owned files per developer for a particular release "linuxv2.6.13". If I say in details, in the increase of every 10 developers it is observed that developers work increasingly more unknown files than the files they own or they are very familier to. We see the similar picture for other releases too. So we can say that in Linux Kernel development developers work in more unfamilier files than the most familier (i.e. more than 80%) ones.

To understand the distribution of the developers around the system we also found that during these release periods 8057 developers have experience of working in the merge meriod and 14621 developers worked in the release development period as we already mentioned earlier but we noted that 4884

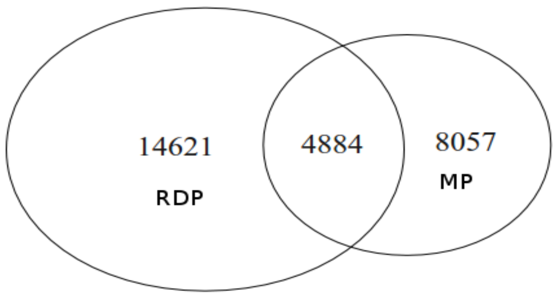


Figure 8: *Developers' Distribution in Merge Period and Development Period*

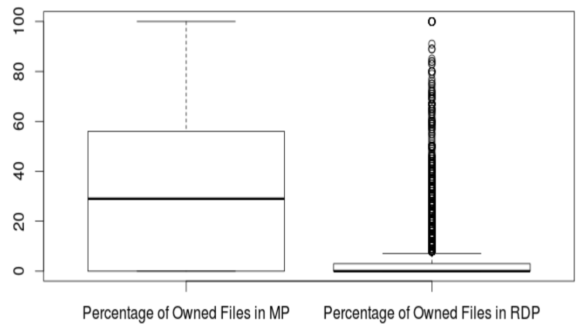


Figure 9: *Percentage of working in owned files during Merge Period and the that of life time*

developers work in both the parts. So almost half of the developers that work in merge period also work in the release development period. Figure 8 shows the Venn diagram.

If we compare the percentage of working in owned files (ωP) in merge period and the percentage of working in owned files in release period (ΩP) for the life time for each developer works in the merge period then we see a significant difference as shown in figure 9.

Now, to predict if developers work in others' code in a high proportion during the rush period we would like to observe the correlation of the % of working in owned files in merge period with other independent variables "churn percentage in merge period ($\c{c}P$)", "churn percentage in general ($\c{C}P$)", "average ownership (ωAvg)" and % of wirking in owned files in general (ΩP). Table 6 shows the Spearman rank correlation among the % of working with owned files and other variables. We can see there is no statistically significant positive relationship between the measures and the % of working in owned files

All three correlations Kendall, Pearson and Spearman calculates that the percentage of working with owned files doesn't to any other variables strongly as we predicted seeing figure 7 having a very poor percentage of contribution (19.82%) on owned files during the merge period of time. But we can see a significant strong relation between the % of churn in general and % of churn in merge period which is 0.7199 according to Kendall, 0.7028 according to Pearson and 0.8446 according to Spearman. Also the correlation between the % of churn in general and the average ownership is strong according to Spearman model which is 0.7982. We are giving the priority to Spearman model because Spearman correlation is a more robust correlation technique [17]. If we consider the following models with our variables:

M1: Predictors: (ωP), ΩP

M2: Predictors: (ωP), ΩP, ωAvg

M3: Predictors: (ωP), ΩP, ωAvg, $\c{c}P$

M4: Predictors: (ωP), ΩP, ωAvg, $\c{c}P$, $\c{C}P$

then we see the linear regression model for the % of working with owned code measure using all the predictors has an R-squared value 0.1132 with residual standard error is 32.52 on 8053 degrees of freedom, Adjusted R-squared value is 0.1139. F-statistic is 342.8 and the p-value is $< 2.2\text{e-}16$. R-square is a measure of variance in the dependent variable that is accounted for by the model built using the predictors [17]. R-square is a measure of the fit for the given data set. It cannot be interpreted as the quality of the dataset to make future predictions.

Table 6: Cross Correlations

Measures	ωP	ΩP	ωAvg	$\c{c}P$	$\c{C}P$
ωP	1.0000	0.2897	0.4496	0.3325	0.2826
ΩP	–	1.0000	0.7110	0.3979	0.5075
ωAvg	–	–	1.0000	0.6754	**0.7982**
$\c{c}P$	–	–	–	1.0000	**0.8446**
$\c{C}P$	–	–	–	-	1.0000

So, finally we can say that in merge periods developers work in more native codes while during the regular release development period the percentage of working with native code is very low in number although this percentage of working in owned code during merge period does not significantly relate to other co-factors.

5.3 Developers' focus on code-base around the time of release

Are there certain areas of the system that receive increased attention (i.e. do developers focus on a smaller set of files around releases)? Answer to the following questions are going to be looked for to answer this research question:

q1: How many times a file has been committed during merge period?

q2: How many authors made those commits?

q3: How many churns were made to the files during each commit?

q4: What is the time interval for the commits for a file?

q5: How many files each developer deals with during merge period?

q6: How many files each developer deals with during release development period?

q7: What is the jaccard distance J(mp1, rdp1), J(rdp1, rdp0) and J(rdp0, mp1)?

We found the relation J(mp1, rdp1) > J(rdp1, rdp0) < J(rdp0, mp1) for the minor releases is significant but among the micro stable releases this relation is not significant at all.

6. CONCLUSIONS

This section will be written at the end.

7. REFERENCES

[1] R. C. H. A. Hindle, M. W. Godfrey. Release pattern discovery via partitioning: Methodology and case study. In *Proceedings of the Fourth International Workshop on Mining Software Repositories at ICSE Workshops MSR '07*, volume 35, page 19, Minneapolis, MN, USA, 2007. MSR.

[2] J. H. A. Mockus, R.T. Fielding. A case study of open source software development: the apache server. In *Proceedings of the 2000 International Conference on Software Engineering, 2000.*, pages 263–272, June 2000.

[3] B. Boehm. Get ready for agile methods, with care. *ACM Computer*, 35(1):64–69, 2002.

[4] B. W. Boehm. Software risk management: Principles and practices. *IEEE Softw.*, 8(1):32–41, January 1991.

[5] F. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975.

[6] K. S. C. Gutwin, R. Penner. Group awareness in distributed software development. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, pages 71–81, Seattle, Washington, USA, 2004. ACM.

[7] E. T. B. Christian Bird, Peter C. Rigby. The promises and perils of mining git. In *6th IEEE International Working Conference on Mining Software Repositories, 2009. MSR '09*, pages 1–10, 2009.

[8] J. S. Daniel Damian, L. Izquierdo. Awareness in the wild: Why communication breakdowns occur. pages 81–90, 2007.

[9] K. I. Daniela Damian, Helms Remko. The role of domain knowledge and cross-functional communication in socio-technical coordination. In *35th International Conference on Software Engineering, ICSE*, pages 442–451, 2013.

[10] P. D. Daryl Posnett, Raissa D'Souza. Dual ecological measures of focus in software development. In *Proceeding ICSE '13 Proceedings of the 2013 International Conference on Software Engineering*, pages 452–461, 2013.

[11] P. D. F. Rahman. Ownership, experience and defects: a fine-grained study of authorship. In *Proceeding of the 33rd international conference on Software engineering, ACM*, pages 491–500, 2011.

[12] L. K. O. Inc. The linux kernel archives. https://www.kernel.org/doc/Documentation/development-process/2.Process.

[13] A. L. W. J. E. Cook. Automating process discovery through event-data analysis. In *Proceedings of the 17th international conference on Software engineering in ICSE âĂŹ95*, volume 35, pages 73–82, Seattle, Washington, USA, April 1995. ACM Press.

[14] P. Jaccard. Distribution of alpine flora in the basin dranses and some neighboring areas. In *Bulletin of the Waldensian Society*, volume 37, pages 241–272, 1901.

[15] J. H. M. Cataldo, P. Wagstrom. Identification of coordination requirements: implications for the design of collaboration and awareness tools. *CSCW*, pages 353–362, May-June 2006.

[16] K. L. M. Keil, P. E. Cule. A framework for identifying software project risks. *Commun. ACM*, 41(11):76–83, November 1998.

[17] T. B. Nachiappan Nagappan. Use of relative code churn measures to predict system defect density. In *ICSE 2005. Proceedings. 27th International Conference on Software Engineering*, pages 284–292, 2005.

APPENDIX

A. HEADINGS IN APPENDICES

Apendix section will be written later with the following subsections may be:

A.1 Introduction

...

A.2 Background and Motivation

A.2.1 Related Works

...

A.2.2 Methodology and Data

...

A.3 Conclusions

...

A.4 Acknowledgments

...

A.5 References

... ...