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 s[illegible] before a major release that turn the developers into di[illegible]urs as Frederick Brooks likens in his benchmark study "The Mythical Man Month" [**?**]. 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 inappropri[illegible]or unrealistic scheduling. Whatever the reason is it is an [illegible]ous. Regardless of the causes, the rush to release stresses d[illegible]opers and often requires developers to work on unusual, high priority or critical areas of the system. In this paper we study how RTR effects project organization and introduces technical debt. The key research questions that we expect to answer with our methodology are as follows:

1. Do developers work on different areas of the system around the time of release?
2. Are there certain areas of the system that receive increased attention (i.e. do developers focus on a smaller set of files around releases)?
3. Do the areas of code that are modified around the time of release have higher defect densities than code that is modified during normal development?

We observe a realloca[illegible]on of the resources among the software development tea[illegible] a large project to identify that an improper reallocation or inappropriate reorganization causes a disru[illegible]e event take place in a software development process. [illegible]attempt to identify a project's different release times and calculate the difference between two consecutive releases to discover which are the new areas have been worked between two consecutive releases. The comm[illegible]g data that we are working on for this purpose will he[illegible]s to extract a lot of information like calculating the developers' working areas and time-frame of each release. This [illegible]mation will help us to identify the criteria of the resou[illegible], their roles, file ownership and nativeness in the domain. This knowledge of nativenes[illegible]ll guide us understanding the method of reallocation. [illegible]y few research works hav[illegible]en performed regarding the re-allocation of resources. [illegible]ert van Engelen worked for similar kind of a research to understand the resource allocation dynamics across the software projects [**?**]. He mainly tried to reallocate development resources amongst projects for increasing the satisfactory level of consumer or customer while we are focusing on the impact on code-base like the complexity of script files. Robert proposed a project-entropy metric in his work to understand if there is any limit for a particular reallocation does not lead to user satisfaction. Here entropy is to represent disorder and chaos to understand degradation of software and its inherent complexity. In his work resources may not just be the developers but also can be any othe[illegible]ources necessary for a software project development. [illegible] have organized this paper as follows. In Section 2, we describe some background and motivations followe[illegible] some summaries of related works in section 3. Section 4[illegible] describe about the ownership of files and ownership of a set of files or a directories. We will try to understand how native a code-base is to a developer or a development team. In Section 5, some analysis to determine reallocation has been performed in a release or where reallocation needs to be p[illegible]med will be presented. What changes in nativeness (âĹ[illegible]) occurs after the reallocation. Section 6 will give us the result to show how change in nativeness puts impact on the outcome of a

*Copyright note

software. Finally section 7 will give us an idea of our future work followed by the conclusion in section 8.

2. BACKGROUND

Sometimes project members with interdependent tasks usually may not communicate effectively; coordination breakdowns occur, which results in integration failures [?]. There may have lower developers productivity [?, ?] which may cause inefficient work 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 [?, ?, ?]. Some of these 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 introduce 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 companyâĂŹs projects, 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 innovation; 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. We want to study that proper re-organization or efficient re-allocation of resources based on meaningful criteria can bring better outcome of a software development project by identifying quantify outcomes (Example: number of defects found).

3. RELATED WORKS

Many people have worked with pretty relevant ideas but I didn't find many very similar to our motivation. Hindle worked on release pattern discovery via partitioning [?]. 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 art-effect checkins 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. Actually this paper worked in a reverse way than Cook did [?]. Cook inserted sensors and monitors into the development process but Hindle and Michael analyzed the data to understand what happened in the past. 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 [?]. Posnett did some dual ecological measures of focus in software development [?]. Posnett's measure was for the more general view that unifies developer focus and artifact ownership. Posnett 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 [?].

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. Actually it is a database containing the all the commit log records by the Linux kernel developers 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 archival data had been preserved throughout the history of the Linux Kernel development project. All information on the OSS project 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 development resources (mainly developers). Our methodology can be summarized as: Extracting Data for revisions and releases (Section 4.1); Partitioning the version numbers (Section 4.2); Get time-span between each release (Section 4.3); Calculate developer areas (Section 4.4); Finding code ownership (Section 4.5). Calculating Jaccard Distance for different development areas.

4.1 Extracting Data

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 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 complete we are ready to partition the version numbers (Section 4.2) and duration of each release (Section 4.3).

4.2 Partitioning Release Numbers

We stored the git commit log extracted data into a table named git commit and git revision where all the basic and 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 get determine which commit belongs to which version and release, and also what type of release that is as well.

```
Find out the release candidates:
update git_refs_tags set rc = cast(
  substring(
    path from
      position( '-rc' in path )
        + 3 for 2
  ) as integer
) where path ~ E'-rc';
```

Find out major release version number:

```
update git_refs_tags set major = cast(
  substring(
    path, 'v([0-9]+)\.?'
      ) as integer
);
```

Find out minor release version number:

```
update git_refs_tags set minor = cast(
  substring(
    path, 'v[0-9]+\.([0-9]+)\.?'
      ) as integer
);
```

Find out micro release version number:

```
update git_refs_tags set micro = cast(
  substring(
    path, 'v[0-9]+\.[0-9]+\.([0-9]+)\-?'
      ) as integer
);
```

We are getting the release dates also with every commit record. So we can understand which commit belongs to which version of release, is this a major release or minor or micro. Another information we have captured that is **rc** which means that the particular commit was for a release candidate.

4.2.1 Explanation

A Linux kernel development release version having maximum length of information looks 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 under the 6th major release of kernel version 2. "linuxv2.6.13**-rc1**" says that after the 13th minor release been released development for the next release has been started and rc1 is the first candidate on the way to the next release. It doesn't ensure that next release is also going to be another minor release.

4.3 Get Time-Span Between Release

Another information that we require is what are the durations of the releases of Linux kernel development. To achieve that we joined the table in figure 1 and the table "git_commit". The following psql did the job pretty easily for us:

```
select
grt.path as release, grt.major, grt.minor,
grt.micro, grt.rc, gc.committer_dt
from
git_refs_tags grt, git_commit gc
where
gc.commit = grt.commit
order by
gc.committer_dt
```

This query gives us all the dates of commit for each and every release, so we can easily find out the duration between two consecutive releases (all including release candidates RC). This information is more important to us because we need to see what is the actual development period and what is the merging or other fixing period within a release time-span and also we need to understand how developers are working, what is the impact of their changes made in the code-base during a particular release period. Table 1 shows a small portion of this information. Preior to calculate the areas of the developers 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
linuxv2.6.13-rc4	rc	2005-07-13	2005-07-28
linuxv2.6.13-rc5	rc	2005-07-28	2005-08-02
linuxv2.6.13-rc6	rc	2005-08-02	2005-08-07
linuxv2.6.13-rc7	rc	2005-08-07	2005-08-23
linuxv2.6.13	micro	2005-08-23	2005-08-28
linuxv2.6.14-rc1	rc	2005-08-28	2005-09-12
linuxv2.6.14-rc2	rc	2005-09-12	2005-09-19
linuxv2.6.14-rc3	rc	2005-09-19	2005-09-30
linuxv2.6.14-rc4	rc	2005-09-30	2005-10-10
linuxv2.6.14-rc5	rc	2005-10-10	2005-10-20

Note that the very first row has no information for the "Start Date" because we don't have any data prior to stable release "linuxv2.6.12". From the extraction we found complete records for releases "linuxv2.6.13" to "linuxv2.6.39" then "linuxv3.0" to "linuxv3.11". We have last couple of release candidates and the release date 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; 40 stable releases from "linuxv2.6.12" to "linuxv3.11". 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.

4.4 Calculate Developers' Area

Developers are working throughout the release time. Developers' Are (DA) means the files developers work in. It

doesn't refer the to 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. We have the commit log data in our hand and we know what is the duration for each and every releases (i.e. start and end dates). Now it is possible to find out which commits were made for which files by a developer within a given time range of a release.

4.4.1 DA in a Release Period

We select all the release information from the table where we stored the releases with their corresponding time-span. After that we select author name and other relevant information from the joining of "git_revision" and "git_commits" tables for every release. We insert these information into another table so that we can use the data for our further analysis.

The following pseudo code shows us the way of achieving the desired result in this section:

Algorithm 1 Calculate Developers Area

Require:
  $commit\_date \geq release\_start\_date$
  $commit\_date < release\_end\_date$
Ensure:
  $getallreleases$
  while $prev\_release\_date \leftarrow release\_start\_date$ and $release \leftarrow release\_version$ **do**
    if $release$ **then**
      if $commit\_date \geq release\_start\_date$ and $commit\_date < release\_end\_date$ **then**
        $selectauthor, path, commits, churn$
        $insertdataintotabledev\_area\_rel$
      end if
    end if
  end while

A relatively straightforward discipline of Linux Kernel Development team is followed with regard to the merging of patches for each release [**?**]. 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 1000 changes ("patches," or "changesets") per day. Strategically linux starts a new kernel release with the merging and fixing 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) and another is development window what we are going to call Release Development Period (RDP).

Similarly we have calculated DA in both MP and RDP as described in the following subsections.

4.4.2 DA in a Merge Period

Merge Perid can be determined from the previous release (any of major, minor or micro releases) date to the first RC date. For example if the date of release for "linuxv2.6.12" is 2005-06-17 and date of pushing the first RC for the next release "linuxv2.6.13-rc1" is 2005-06-29 then this time period of of 12 days is being called the merge window [**?**] or MP. Table 2 shows some merging periods of different releases.

Table 2: Release Merging Periods

Release	Start Date	RC 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 understand DA or Developers' Area is the files in the code-base, developers are working from the dataset that we have in our hand. We have the commit logs and every record says about who committed the file when did he/she worked on the file. If we run an SQL query then we can easily find out for every author and for every particular file how many times it's been committed by an author and what changes (here, addition + deletion = churn) he/she has been made. So we are calculating the DA and storing the information into a table. A part of that table is shown in table 3.

Table 3: DA 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 merging period inside a release period D. S. Miller has made change in pci.h file only once wich the churn number 8. In the similar way we calculate DA in RDP and RTR period as well.

4.4.3 DA in a RD Period and RTR

As like MP we can extract the RDP from the. RDP here in Linux Kernel development process is being considered as from the release date of RC1 (i.e. first release candidate opening the gate for the development of the next release) to the next release (major, minor or micro) date. For example if the date of release for "linuxv2.6.13-rc1" after releasing "linuxv2.6.12" is 2005-06-29 and date of publishing the next release "linuxv2.6.13" is 2005-08-28 then this time period of of 60 days is being called the RDP. Table 4 is showing some development periods of different releases.

So along these RDs of each release period developers get involved in the development for the next release. As we mentioned earlier RTR may happen as the release approaches.

Table 4: Release Development Periods

Release	RD Start Date	Release 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

Keeping this in our aim list we noticed an important slot in these RDs of releases. We see in between two consecutive releases there is a merging period of time, after that development starts by pushing RCs and once RC releases stops and it takes some days of time for the preperation to publish the release. We are calling this last part as RTR Period because we asume RTR may happen in this period of time before release. Table 5 represents some releases with their corresponding RTR (i.e. from end date to rc and date of release).

Table 5: RTR Periods

Release	RC End Date	Release Date	Days
linuxv2.6.13	2005-06-29	2005-08-28	5
linuxv2.6.14	2005-09-12	2005-10-27	8
linuxv2.6.15	2005-11-11	2006-01-02	9
linuxv2.6.16	2006-01-17	2006-03-20	8
linuxv2.6.17	2006-04-02	2006-06-17	12
linuxv2.6.18	2006-07-06	2006-09-19	7
linuxv2.6.19	2006-10-04	2006-11-29	14
linuxv2.6.20	2006-12-13	2007-02-04	10
linuxv2.6.21	2007-02-20	2007-04-25	7
linuxv2.6.22	2007-05-12	2007-07-08	8
linuxv2.6.23	2007-07-22	2007-10-09	9
linuxv2.6.24	2007-10-23	2008-01-24	5
linuxv2.6.25	2008-02-10	2008-04-16	8

These information above will help us to answer our first research-question.

4.5 Finding Code Ownership

So many developers are working from different development communities. We tried to investigate the "code ownership" [**?**] evolved in Linux Kernel development. Here code ownership is going to be represented as percentage value. Considering the data that we have for complete releases ("linuxv2.6.12" to "linuxv3.11") we filtered the code files and we found 75138 distinct files that have been worked on. We investigate the total number of commits, total number of churn for each and every file also 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 every developers in different development areas in an RP. We already have the collection of developers working in different areas MP, RDP, RTR within an RP and stored them with corresponding file 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. For calculating ownership Equation 1 is applied.

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

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 considering that if the developer makes changes more than 80% of the total change that the file has got, then that developer can be treated as a owner of the codes of that file similarly as C. Gutwin did [**?**] 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 considering commits also to determine ownership but only for files without any add or remove lines information. If a file has no addition or deletion information (we are calling it churn) in it's commit log but has beet committed by some one at least once then that or those authors may have the ownership for that file. So the equation 1 becomes like:

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

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 all different parts MP, RDP, RTR and also for an entire release period RP. Table 6 represents a part of the data where ownerships is calculated.

Table 6: 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 the developers worked for different files in different releases. Ownership is calculated for all the developers in release periods. In the same way we have calculated what is the ownership value for a developer to a particular file in the merge period (MP) in a release, development period (RDP) in a release and RTR period ind a release.

5. PRELIMINARY 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. Once we found different releases and the release periods as well as we

calculate the work areas and code ownerships for developers we also want to know what is the difference being created by making changes between two consecutive releases. We want to see what is the difference among the releases also among the different sections MP, RDP, RTR in a release period. To observe this we want to calculate the difference using Jaccard Similarity Coefficient [**?**] which is represented by the following equation.

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

So the Jaccard Distance can be obtained by subtracting the Jaccard coefficient from 1.

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

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 as the set A and files worked for in another release is set B. We found that Jaccard distance between two consecutive releases are not too high. For example jaccard bwteen release "linuxv2.6.13" and "linuxv2.6.14" is 0.925446 and bwteen "linuxv2.6.14" and "linuxv2.6.15" is 0.9191968. The higest Jaccard distance we have measured is 0.940 which is between 2.6.20 and 2.6.21 on the other hand the lowest distance is 0.906 between 2.6.30 and 2.6.31. Table 7 shows some Jaccard distances for release pairs and in Table 8 we see the Jaccard distance within the same releases between different perilds MP, RDP and RTR.

Table 7: Jaccard Distance between Releases

From	To	Jaccard Distance
linuxv2.6.13	linuxv2.6.14	0.925446
linuxv2.6.14	linuxv2.6.15	0.919196
linuxv2.6.15	linuxv2.6.16	0.917945
linuxv2.6.16	linuxv2.6.17	0.916900
linuxv2.6.17	linuxv2.6.18	0.929941
linuxv2.6.18	linuxv2.6.19	0.937535
linuxv2.6.19	linuxv2.6.20	0.935265
linuxv2.6.20	linuxv2.6.21	0.940366
linuxv2.6.21	linuxv2.6.22	0.920760
linuxv2.6.22	linuxv2.6.23	0.921652
linuxv2.6.23	linuxv2.6.24	0.929194
linuxv2.6.24	linuxv2.6.25	0.910464

Table 8: Jaccard Distance between Releases

Release	J(MP, RDP)	J(RDP, RTR)	J(MP, RTR)
linuxv2.6.13	0.952060	0.921850	0.996786
linuxv2.6.14	0.958213	0.839313	0.988072
linuxv2.6.15	0.947005	0.936153	0.996408
linuxv2.6.16	0.949774	0.919361	0.991275
linuxv2.6.17	0.938539	0.835925	0.971014
linuxv2.6.18	0.946127	0.865967	0.990750
linuxv2.6.19	0.960241	0.845649	0.993054
linuxv2.6.20	0.967823	0.804856	0.995071
linuxv2.6.21	0.941798	0.835575	0.989031
linuxv2.6.22	0.947115	0.928818	0.997208
linuxv2.6.23	0.939658	0.909090	0.995797
linuxv2.6.24	0.946806	0.871875	0.989284

In table 8 last three columns representing the Jaccard distance between merge period and release development period, between release development and rush to release period, between merge period and rush to release period respectively.

6. CONCLUSIONS

This section will be written at the end.

7. ACKNOWLEDGMENTS

... ...

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

... ...