Re-Allocation of Resources during Releases *

Md Tajmilur Rahman
Concordia University
Montreal, QC H3G 1M8
438-932-2288, +1
rupak.karmadhar@gmail.com

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, 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" [**?**]. 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 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 reallocation of the resources among the software development teams in a large project to identify that an improper reallocation or inappropriate reorganization causes a disruptive event take place in a software development process. We attempt to identify a projectâĂŹs 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 commit log data 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. This information will help us to identify the criteria of the resources, their roles, file ownership and nativeness (âĹĘÆđ) in the domain. This knowledge of nativeness (âĹĘÆđ) will guide us understanding the method of reallocation. Very few research works have been performed regarding the re-allocation of resources. Robert 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 other resources necessary for a software project development. We have organized this paper as follows. In Section 2, we describe some background and motivations followed by some summaries of related works in section 3. Section 4 will 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 performed will be presented. What changes in nativeness (âĹĘÆđ) occurs after the reallocation. Section 6 will give us the result to show how change in âĹĘÆđ puts impact on

the outcome of a 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 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 [?, ?, ?]. 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 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. 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 area owners (Section 4.5). Finding merging time and development time within a release period that we found in section 4.1 (Section 4.6).

4.1 Extracting Data

We went for the VCS of a target project and either mirror the repository or download each revision and commit log history data individually. From VCSs such as CVS we extract the revisions and release information. We then put them into a database. We have used here 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 Partioning 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. Figure 1 shows the picture of the data how it looks like.

```
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';
```

commit	path	major	minor	micro	rc	date
02f8c6aee8df3cdc935e9bdd4f2d020306035dbe	linuxv3.0	3	0	0	0	2011-07-21 22:17:23-04
6887a4131da3adaab01161377dd865f4bcfb5678	linuxv3.5-rc5	3	5	0	5	2012-06-30 19:08:57-04
6b16351acbd415e66ba16bf7d473ece1574cf0bc	linuxv3.5-rc4	3	5	0	4	2012-06-24 15:53:04-04
8e0ee43bc2c3e19db56a4adaa9a9b04ce885cd84	linuxv2.6.29	2	6	29	0	2009-03-23 19:12:14-04
4a6908a3a050aacc9c3a2f36b276b46c0629ad91	linuxv2.6.28	2	6	28	0	2008-12-24 18:26:37-05
3fa8749e584b055f1180411ab1b51117190bac1e5	linuxv2.6.27	2	6	27	0	2008-10-09 18:13:53-04
bcef7793daec3e05ec5c570d2457b81fe7b5725	linuxv2.6.26	2	6	26	0	2008-07-13 17:51:29-04
4b119e21d0c66c22e8ca03df05d9de623d0eb50f	linuxv2.6.25	2	6	25	0	2008-04-16 22:49:44-04
49914084e797530d9baaf51df9eda77babc98fa8	linuxv2.6.24	2	6	24	0	2008-01-24 17:58:37-05
bbf25010f1a6b761914430f5fca081ec8c7accd1	linuxv2.6.23	2	6	23	0	2007-10-09 16:31:38-04
de46c33745f5e2ad594c72f2cf5f490806lb16ce1	linuxv2.6.21	2	6	21	0	2007-04-25 23:08:32-04
77b67063bb6bce6d475e910d3b886a606d0d91f7	linuxv3.7-rc5	3	7	0	5	2012-11-11 07:44:33-05
15c03dd4859ab16f9212238f29dd315654aa94f6	linuxv3.12-rc3	3	12	0	3	2013-09-29 18:02:38-04
4a10c2ac2f368583138b774ca41fac4207911983	linuxv3.12-rc2	3	12	0	2	2013-09-23 18:41:09-04
272b98c6455f00884f0350f775c5342358ebb73f	linuxv3.12-rc1	3	12	0	1	2013-09-16 16:17:51-04
d8dfad3876e4386666b759da3c833d62fb8b2267	linuxv3.11-rc7	3	11	0	7	2013-08-25 20:43:22-04

Figure 1: Releases extracted from Version

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 shown in figure 1 and git_commit. The following psql query 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. 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: Nonlinear Model Results

Release	Type	Start Date	End Date
linuxv2.6.12-rc2	rc	2005-04-16	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

4.4 Calculate Developers' Area

Developers are working throughout the release time. Developers' are means the files developers work in. But it doesn't refer the to file itself but 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 of each 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.

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 a table named "dev_area_rel" so that we can use the data for our further analysis.

The following pseudo code may show the path to achieve 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 $prev\_release\_date \leftarrow release\_start\_dateandrelease \leftarrow release\_version$ **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

4.5 Finding Code-Area Owners

... ...

4.6 Jaccard Distance

5. CONCLUSIONS

This section will be written at the end.

6. 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

... ...