Tamas Sztanka-Toth

Cached Bug Prediction for Python repositories on GitHub

Computer Science Tripos - Part II

Homerton College

May 10, 2016

Proforma

Name:	**Tamas Sztanka-Toth**
College:	**Homerton College**
Project Title:	**Cached Bug Prediction for Python repositories on GitHub**
Examination:	**Computer Science Tripos - Part II, 2016**
Word Count:	**11993**[1]
Project Originator:	Advait Sarkar
Supervisor:	Advait Sarkar

Original Aims of the Project

To implement the FixCache bug prediction algorithm in Python, for Git repositories. Once the algorithm is implemented, analyse and evaluate it on five different open-source Python projects on GitHub and compare these results with the ones found by the original authors who analysed Java and C/C++ projects. Finally, implement a new evaluation technique for FixCache and evaluate the algorithm with this new method.

[1] This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Work Completed

The core algorithm, together with other supporting modules was implemented on time. It was tested and analysed with the repositories outlined in the project proposal, and results were then compared to previous research findings. Lastly, the performance of FixCache was measured using a new technique, using the notion of 'windowed-repositories'.

Special Difficulties

None.

Declaration

I, Tamas Sztanka-Toth of Homerton College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

1 Introduction **1**

1.1 Motivation 2
1.2 Previous work 2
1.3 Terminology 4

2 Preparation **5**

2.1 Overview FixCache 5
2.1.1 Abstract view of the algorithm 6
2.1.2 Hits, misses and hit-rate 7
2.1.3 Cache-localities 8
2.1.4 Variables 10
2.1.5 Pre- and post-processing 10
2.2 Identifying fixing commits: the SZZ algorithm 11
2.3 Git 11
2.3.1 Overview 11
2.3.2 Git snapshots 12
2.3.3 Git File structure 13
2.3.4 Branches in Git 13
2.4 GitPython 17
2.4.1 Overview 17
2.4.2 The object database 18

3 Implementation **21**

3.1 Design overview 21
3.2 Back-end modules 23
3.2.1 Parsing module 23
3.2.2 File-management module 25
3.2.3 Cache implementation 29
3.3 The Repository module 31

3.3.1 Implementing the SZZ algorithm . . . 31
3.3.2 Cache size initialisation . . . 31
3.3.3 Setting variables . . . 32
3.3.4 Initialising commits . . . 32
3.3.5 Running FixCache . . . 33
3.4 Versions and errors . . . 35
3.5 Implementation difficulties . . . 36
3.5.1 Using GitPython . . . 36
3.5.2 Bottleneck: sorting . . . 36
3.6 Speed-up . . . 37

4 Evaluation **39**
4.1 Repositories analysed and evaluated . . . 39
4.2 Evaluation modules . . . 40
4.2.1 Analysing FixCache . . . 40
4.2.2 Displaying results . . . 40
4.3 Evaluation over hit-rate . . . 41
4.3.1 Results . . . 41
4.3.2 Criticism . . . 45
4.4 How good is FixCache? . . . 45
4.4.1 Methodology . . . 46
4.4.2 Precision, recall and F_2 score . . . 47
4.4.3 Results . . . 49
4.5 Summary . . . 51

5 Conclusion **53**
5.1 Summary of work . . . 53
5.2 Novel Contributions . . . 54
5.3 Limitations and future work . . . 54

Bibliography **54**

A Modules **57**
A.1 `filemanagement.File` class . . . 57
A.2 `filemanagement.Distance` class . . . 61
A.3 `cache` module . . . 64

B Project Proposal **68**

List of Figures

1.1 Examples of data that might be used by different bug prediction approaches . . . 3

2.1 Abstract view of how FixCache proceeds when a bug-introducing change is encountered. . . . 6
2.2 Distributed revision-control with Git . . . 12
2.3 Snapshots in Git . . . 13
2.4 Git file-structure for a given commit. . . . 14
2.5 Basic branching in Git . . . 14
2.6 Merging branches in Git . . . 15
2.7 Comparing Git branching with other VCS . . . 16
2.8 Showing time-to-run of backends `GitDB` (version_5) and `GitCmdObjectDB` (version_6) for repositories boto.git and facebook-sdk.git when FixCache is run for different cache-ratios. Here cache-size is computed directly from the cache-ratio, which will be explained in Chapter 3 in detail. . . . 18

3.1 Core modules and the interaction between them. . . . 22
3.2 Data-flow through the Parsing module . . . 23
3.3 Example of changes between revisions. . . . 24
3.4 Data-flow through the File-management module . . . 26
3.5 Adding new files to the Cache and processing them. . . . 29
3.6 Control-flow of FixCache inside the Repository module. . . . 33
3.7 Speedup of boto.git and boto3.git between version_1 and version_5 with different cache-ratios . . . 38

4.1 Evaluation of FixCache according to hit-rate for different repositories 41
4.2 Fixed cache-ratio analysis for boto.git . . . 42
4.3 Showing all cache-ratio analyses for facebook-sdk.git, raspberryio.git, boto3.git and boto.git. . . . 43
4.4 Window and horizon presented on a timeline of commits . . . 46

4.5 Precision and recall. 48
4.6 Evaluating boto.git: recall, precision and F_2 score 49
4.7 Evaluating django.git: recall, precision and F_2 score 50

Acknowledgements

I would like to thank my supervisor, Advait Sarkar, for his guidance and advices throughout the project. This work would have been much harder without his valuable and helpful feedback, comments and suggestions.

Chapter 1

Introduction

This dissertation describes and analyses FixCache [1], a bug prediction approach which analyses the version history of a repository to identify bug-prone files. First it will discuss what preparatory work was done before the implementation, in order to acquire the necessary background knowledge. Then, it will show the process of implementing this algorithm in Python, for open-source Python repositories on GitHub.

The implemented algorithm was be analysed and evaluated against *hit-rate*, a metric introduced by Kim *et al.* [1]. Results show that the repositories in this dissertation perform the same way in terms of hit-rate, as the projects which Kim *et al.* looked at. Thus, it can be deduced that the algorithm indeed works for Python open-source repositories on GitHub.

In addition to evaluating against hit-rate, a new evaluation technique is introduced. When using this new technique, it was found that FixCache's performance is only high for a short period of time in the future, and that after that, it's performance drops by roughly 30-40%.

This chapter will further talk about motivation behind the dissertation itself, and briefly describe the previous work done in the field of bug prediction.

1.1 Motivation

Testing and debugging has always been one of the most important parts of Software Development. Companies and developer teams spend a lot of time and effort when finding and fixing bugs prior to releasing a stable version of a product.

As projects grow bigger every day, their complexity also rises, which further means that testing and modelling also becomes harder. As follows, developers have to write more complex unit-tests and debugging tools, which will cover a wider range of use cases.

However even with the most advanced testing tools and review techniques there still are bugs discovered only after deployment, and therefore bug prediction techniques have gained importance. These, as opposed to static paradigms (like unit-testing) will use data associated with a software (or its repository) to identify and predict bugs.

1.2 Previous work

There are several approaches to bug prediction to date, which can be put into two main categories: Change-log approaches and Single-version approaches [2]. These approaches usually use different types of data for prediction (as seen in figure 1.1), but ultimately all of them try to achieve the same result: to narrow down the set of files from a code-bas to files, classes, functions which are more bug-prone than the others, or are more likely to contain a bug in the future.

Single-version The Single-version approaches use the current state of the repository and calculate variety of metrics to predict faulty behaviour. The most commonly used metric is the so-called Chidamber and Kremmer metrics suite[3]. The CK metrics were proposed for Object-oriented programs (OOP), and they quantify the state of a program using different data. For example, one metric is the Weighted Method Count per Class (WMC): the sum of the *complexities* of the methods in that class. The definition of the per method *complexity* measure is left open by Chidamber and Kremmer, they only suggest it should be an additive property [3].

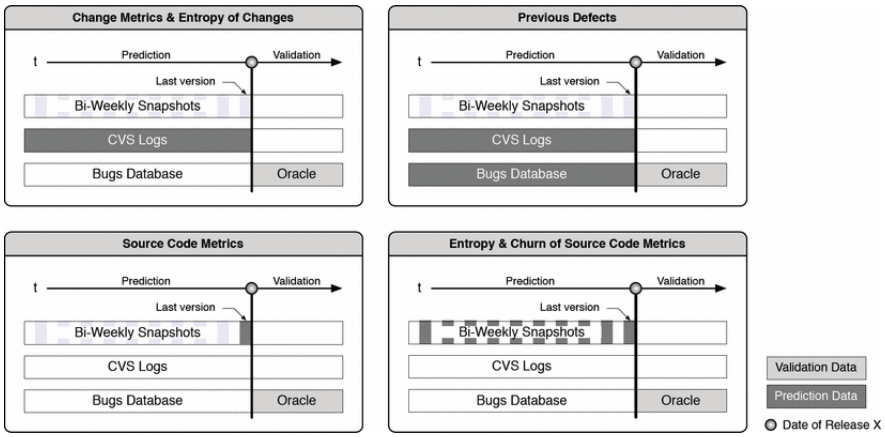

Figure 1.1: Examples of data that might be used by different bug prediction approaches. Figure taken from D'Ambros *et al.* [2].

.

There are several other OOP metrics used, which weren't suggested as a part of CK metrics. A simple example might be, the Lines Of Code per file, or method (LOC).

These metrics are then used as a part of a bug prediction or bug-identification scheme. Several such schemes were proposed, for reference and examples one should look at the listings described by D'Ambros *et al.* [2].

Change-log In contrast, the Change-log approaches look at how the repositories evolved over time, and try to predict where the future defects will occur. For example Hassan's algorithm calculates the complexity of each change and calculates the entropy of these changes which then is used for bug prediction [4].

FixCache is also a change-log approach to bug prediction: the algorithm looks as a history as a whole, and using how a repository evolved, it tries to identify more bug-prone entities. It was introduced by Kim *et al.* in 2007, who implemented it for both files and methods [1]. These two approaches differ only in that the first looks at files while the second looks at methods as the basic entities the algorithm is ran on. This dissertation will only consider the file-level approach.

Since its introduction, FixCache has been researched and tested heavily: Sadowski *et al.* analysed how the algorithm performs on the file-level, and how does hit-rate perform over time [5]. Rahman *et al.* compared the algorithm to other

prediction techniques, and found it to be better than a naive prediction technique, but not significantly [6].

Previous implementations [1][5], except one [6] did not use Git [7] as their back-end Version Control System (VCS), but rather they used an alternative one such as Subversion (SVN) or CVS. These Version Control Systems differ in many aspects, which will be discussed more broadly in section 2.3. Also, as these implementation were solely ran on C/C++ and Java products rather than Python, hence my motivation to implement FixCache and run it on public Python repositories, on GitHub.

1.3 Terminology

FixCache can be implemented for two differented entities: files and methods. The first will identify buggy files, while the second will identify buggy methods. As this dissertation discusses only FixCache implemented for files as entities, so each time I use the word 'entity' it refers to a file.

In Git `revision` and `commit` are different notions: the first refers to a pointer by which one can reference a Git object (for instance a commit or a branch), whereas the second is used when talking about commit-objects. In this dissertation I will use them as synonyms, whenever I write `revision` I mean `commit`.

Chapter 2

Preparation

This chapter summarizes the work done before implementation: it discusses relevant literature and concepts used by FixCache. It first discusses FixCache and it's background; then it describes Git [7] software management system and how can it be used to implement FixCache. Finally, in this chapter will introduce a development tool used in this dissertation: GitPython, a Python library used to acces Git repositories.

2.1 Overview FixCache

The algorithm itself is called FixCache as it uses a notion of a cache, which contains a fixed subset of files from a repository (fixed subset means that the size of our cache is limited, and constant for a single run).

During one run, we go through all the commits in a repository, and at each commit we modify our cache so that at each point it contains the files which are believed to be more bug-prone than others. After the algorithm has completed, the files which ended-up in the cache are said to be more bug-prone, and they are the ones which will more likely to have bugs n the future [1].

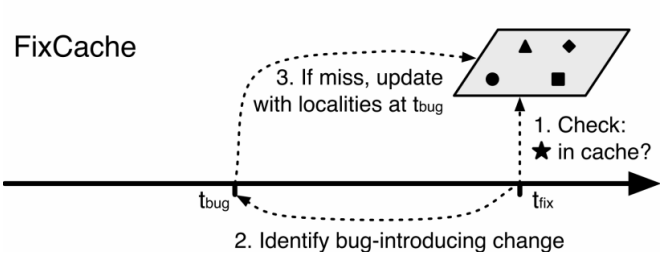

Figure 2.1: Abstract view of how FixCache proceeds when a bug-introducing change is encountered.

.

2.1.1 Abstract view of the algorithm

For a given repository, the algorithm works as follows:

1. Pre-load the cache with initial files.
2. For each commit C in the repository, starting from the first commit:
3. If C is identified as a fixing-commit (a bug-fixing commit):
   (a) For each file in C, we check if they are in the cache. If a file is in the cache, we record a hit; else we record a miss.
   (b) For each file F which was not in the cache, we identify the commits which introduced the bug(s) fixed by C. We take the closest files to F at these bug-introducing commits, and put them into the cache.
4. We put the new-files and changed-files at C into the cache.
5. We remove the files from the cache, which were deleted at C.

Figure 2.1 shows how the algorithm proceeds at step 3, when we identify a fixing-commit.

So far, we only talked about adding files to the cache, but our cache has a limit of how many files can it store. To ensure that new files can be added, behind the scenes the algorithm should take care of cache pollution, and should remove files from the cache to make space for new arrivals.

Cache-replacement policies

As the number of files in the cache (the cache-size) is fixed, we need to take out files from the cache if the cache is polluted. There exist different cache-replacement policies [1]:

- *Least-recently-used (LRU)*: When removing files from the cache we first remove those which were used least-recently. That is we remove the files which were added/changed earlier in our commit history.
- *LRU weighted by number of changes (CHANGE)*: This approach will remove files which were changed the least, as the intuition is that more frequently used files are more bug-prone, so we want to keep them in the cache.
- *LRU weighted by number of faults*: This is similar to the approach before, the only difference is that instead of removing files with least changes it removes files with least faults.

In the implementation itself LRU has been used as it is the simplest, and there was no significant difference identified between the different replacement policies by neither of the previous implementations [1][5][6].

2.1.2 Hits, misses and hit-rate

We only look-up the cache at the so-called bug-fixing commits (step 3a in the abstract view). To flag a commit as a bug-fixing one, we use a set of regular expressions (which are listed in more detail in section 3.2.1) and parse the commit message. If a message matches any of our regular expressions, it is a bug-fixing commit (following the idea used in [8]). Some examples of bug-fixing commit messages[1]:

- `Simple json module import.  Fixes #106.`
- `bugfix:  changed key of id from user to id in the user dict stored in the session`
- `Final fix for Python 2.6 tests.`
- `PEP8 fixes.  Make sure that code is compliant with pep8 1.3.3.`

[1]All examples taken from identified commit-messages of facebook-sdk.git repository, explained later.

At each bug-fixing commit, for each file involved in that commit, we make a lookup in the cache. If a file is already in the cache, we note a **hit**. Otherwise we score a **miss**.

Hit-rate

Once we have the number of hits and misses, we can define **hit-rate**, as the ratio of hits over all lookups, that is:

$$\text{hit-rate} = \frac{\#hits}{(\#misses + \#hits)}$$

Since at each bug-fixing commit we either increase hit-count or miss-count (or both), hit-rate is a cumulative indicator of how good our algorithm is. Kim *et al.* used hit-rate (at the end of a FixCache run) to evaluate the algorithm for several different repositories [1].

It might be argued that this evaluation strategy is poor, as even with high hit-rate it is not known what is the window of our prediction, that is how early/late will the files in our cache contain bugs. In Chapter 4 FixCache will be evaluated against hit-rate, but also a new, more sophisticated evaluation technique will be introduced.

2.1.3 Cache-localities

Following the logic of having a 'cache' as a central element of our algorithm, we can also define a notion of localities: spacial, temporal, new-entity and changed-entity. We use these localities to put files into the cache, in order to increase the algorithm's accuracy.

Temporal locality

As with physical caches, temporal locality in FixCache is used following the idea that an entity which had a fault recently is likely to have a fault in the near future. That is, at each bug-fixing commit, we will load all the files involved in that commit to the cache.

Spatial locality

Again, the idea of spatial locality comes from the world of physical caches: we assume that if a file had a fault it is likely that files 'near' to that will have a fault in the near future.

To define 'nearness', we first need to define the notion of distance between two files in. The distance for any two files f_1 and f_2 at revision/commit n is defined as (following Kim *et al.* [1]):

$$distance(f_1, f_2, n) = \frac{1}{cooccurrence(f_1, f_2, n)}$$

The $coocurrence(f_1, f_2, n)$ returns an integer: how many times were the files f_1 and f_2 used (committed) together from revision 0 to revision n.

This locality is only used when we lookup a file in our cache, and it is missed. This is similar to physical caches when we only load a new cache line into the cache when a cache-miss occurs. If a file (say f_t) is missed during a bug-fixing commit, we go back to the bug-introducing commit(s) and fetch the closest files to f_t at each bug-introducing commit identified. We can identify the bug-introducing commits using the SZZ algorithm [9].

It is important to understand the distinction between bug-fixing commits and bug-introducing commits. The former are identified by parsing the commit message, while the latter are identified by the SZZ algorithm.

Changed-entity-locality and new-entity-locality

At each commit we put the encountered files into the cache. The encountered entities can be further divided into two categories: new files (new-entity-locality) and files changed between the commit and its parent commit (changed-entity-locality).

Summary of cache-localities

Spatial and temporal localities have they equivalent in physical caches. These will only be used at bug-fixing commits.

New-entity and changed-entity localities are introduced by FixCache, and they will be used at every commit.

2.1.4 Variables

Cache-size

The main parameter of FixCache. It specifies a boundary on the cache, how many files are allowed to be in it at any given commit. It stays constant for one run.

Distance-to-fetch (block-size)

This parameter is used to define how many of the closest files will be loaded to the cache together with our missed file, according to the spatial locality defined above.

Pre-fetch-size

This variable is used for both changed-entity and new-entity localities to specify how many files should be fetched. At each revision files with higher Lines Of Code (LOC) are put in the cache first. That is if the pre-fetch-size is 5, than we will load at commit c_n the 5 new-files with highest LOC and the 5 changed-files with the highest LOC.

2.1.5 Pre- and post-processing

Initalising the cache

To encounter a small number of misses at the beginning, we need to pre-load the cache with some files at commit 0. This is done by using the new-entity-locality: each file is new, so each file is considered, and we will load files with the highest LOC, according to the pre-fetch-size, discussed above.

Cleaning-up

At each commit we remove the deleted files from the cache, to save space for further insertions and to avoid having having non-existent files in the cache.

2.2 Identifying fixing commits: the SZZ algorithm

The SZZ algorithm was introduced by J. Śliverski, T. Zimmermann and A. Zeller to identify which commit(s) introduced a bug in a file, when viewing the file at a bug-fixing commit [9]. The algorithm works the following way:

1. At a fixing commit C_{fix}, we first need to identify the set of lines L, which were deleted between C_{fix} and it's parent. We assume that all of these lines were deleted as a result of the bug-fixing procedure (which resulted committing C_{fix}), so that they contributed to the bug itself.
2. Once we know L, we say that the set of commits $C_{intr,L}$ which introduced the lines in L is the set of bug-introducing commits we are looking for.

   In Git, getting which lines were introduced by which commit is quite straightforward using the `git blame` command.
3. Do steps 1-2. for each bug-fixing commit.

2.3 Git

This section will first provide a quick introduction into Git [7], then it will describe it's pitfalls when using it for code-mining, and finally it will explain how these pitfalls are tackled when Git is used for FixCache.

2.3.1 Overview

The main difference between Git and other VCSs, say Subversion (SVN), is that Git is fully distributed: each developer's machine has a valid Git repository to which the developer can commit to, without accessing any 'main' repository.

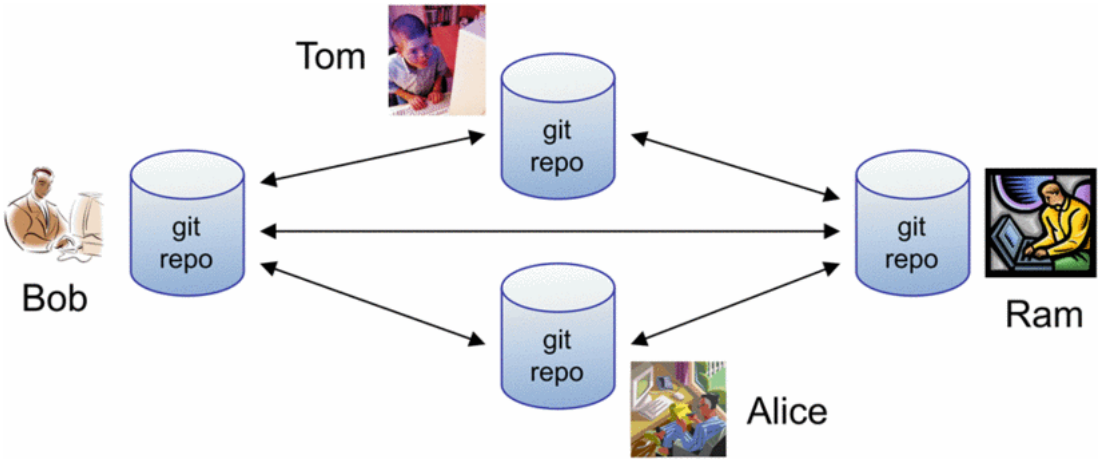

Figure 2.2: Distributed revision-control with Git. Figure taken from [7].

Later the developer can push changes from the local repository to any other machine which it has access to, as seen in the figure 2.2.

Despite this, usually there is a dedicated 'origin' computer which holds the up to date version of a repository, and with which the developers are communicating. This approach makes it easier to develop software when internet connectivity is poor (as we can still work on a project locally), but introduces other issues, such as merge-conflicts (more on this later).

Since its development in 2005, Git has been widely adopted by variety open-source projects as their primary VCS, such as the Django Project[2], Ruby on Rails[3] or jQuery[4].

2.3.2 Git snapshots

Git stores data differently to other major VCSs. Rather than storing for each file the list of changes, git stores the whole repository at each point in the history. For efficiency if a file hasn't been changed at a commit, rather then storing the file again, Git only stores a pointer to the last identical file in the previous revision as in figure 2.3.

We can think of a Git repository as an ordered list of snapshots. As follows, version-control is simply done by creating snapshots (by committing) of our repos-

[2]https://www.djangoproject.com/
[3]http://rubyonrails.org/
[4]https://jquery.com/

itory. To view differences (using the `git diff` command) Git simply looks up the difference between snapshots

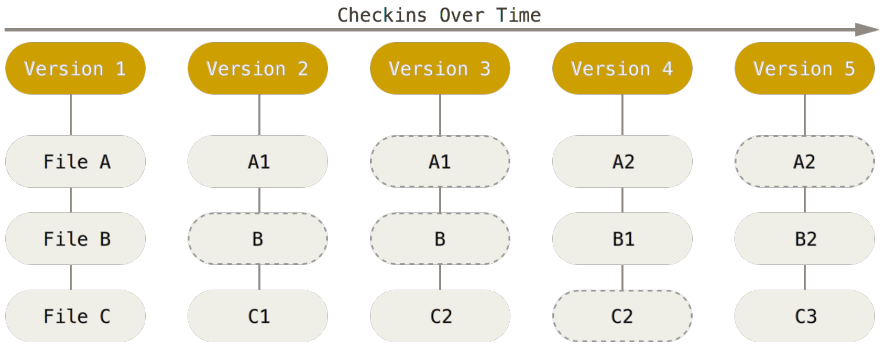

Figure 2.3: Snapshots in Git, pointers to files have striped lines. Figure taken from https://git-scm.com/book/

2.3.3 Git File structure

In Git, we can think of a repository as a small file structure (one for each snapshot). Objects in this file structure can either be *blobs* or *trees*: blobs correspond to files whereas trees correspond to directories and subdirectories under the repository's root directory. At each point in the history of a repository Git stores a commit object, which stores the meta-data (time committed, author) about that commit.

Each commit object has pointer to a tree object, corresponding to the root directory of the repository at a given snapshot, as seen in figure 2.4. Each tree can have a pointer to many more trees or blobs, whereas blobs are leaf nodes. Each commit has a pointer to it's parent(s), except the special initial commit which does not have parents.

2.3.4 Branches in Git

Branches are an important feature of Git, but they make mining repositories harder, as they introduce new problems to be tackled. They allow non-linear development in a decentralised manner, meaning that developments can make their own changes locally, and later join/merge these changes together.

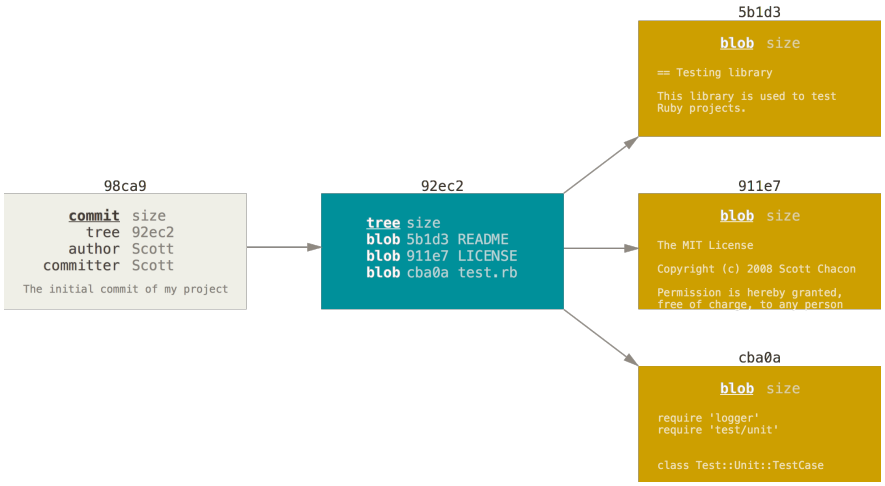

Figure 2.4: Git file-structure for a given commit. Figure taken from https://git-scm.com/book/.

As discussed before each commit is simply an object with some meta-data, which stores a pointer to a snapshot. A branch is simply a pointer to one of these commits.

The branch named 'master' is the default branch (although this can be changed); after initialising a repository you are given a 'master' branch which will store a pointer to the latest commit made. When a new branch is created, Git creates a new pointer to the commit viewed at that moment. Git knows which branch is active at any time by storing a special pointer called 'HEAD' which is a pointer to the active branch.

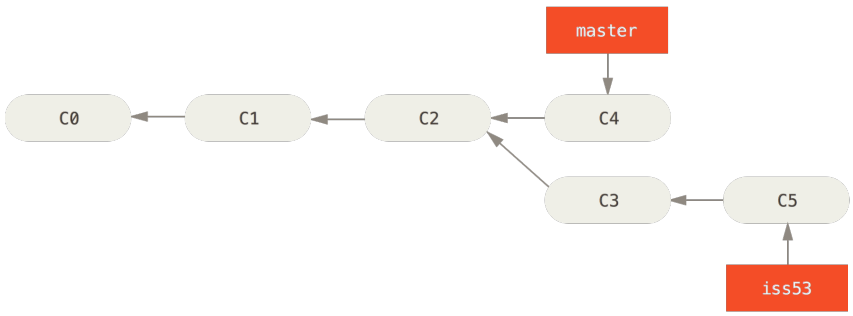

Figure 2.5: Basic branching in git, with two branches: 'master' and 'iss53'. Figure taken from https://git-scm.com/book/.

Diverging branches

It is possible to diverge two branches, as seen in figure 2.5 above. There starting from commit C2, some changes were made on branch 'master' which resulted in commit C4. Then, from C2 also some changes were made on branch 'iss53', which resultsed in branches C3 and C5. We can see from the diagram that C4 and C3 have the same parent, C2.

Merging branches

Following this logic, you can also merge two branches, that is for example include the changes (that is the set of commits) made on branch 'iss53' to the changes made on 'master'. If the two branches are on the same linear path, we can simply 'fast-forward', that is point the pointer of 'master' to the same place when 'iss53' is pointing. Otherwise, we need a real merge, which can be tricky to do automatically.

Figure 2.6 shows how merging works in Git. We are merging 'iss53' into 'master', so the commit C5 has to appear on the 'master' path as well.

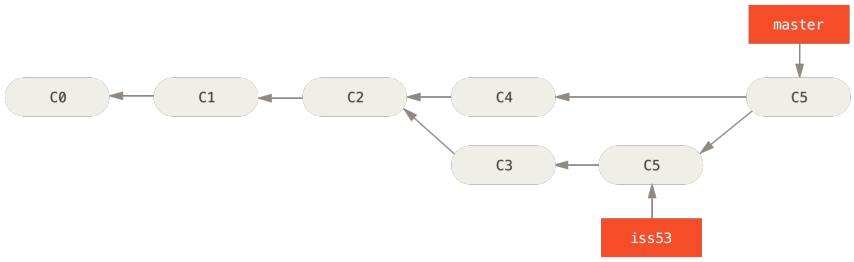

Figure 2.6: Merging 'iss53' branch into the 'master' branch. Figure taken from https://git-scm.com/book/.

When performing a merge, there are two possibilities:

1. There are no merge-conflicts. This means that there is no single file which were changed/overwritten on both branches. In this case Git will handle merging automatically through a so-called "recursive-merging" by default.

2. There are merge-conflicts. This means that there is at least file which has been modified by both branches 'master' and 'iss53'. In this case Git will

merge the files that it is able to merge, and for those where the conflict arose it will place some readable text to tell the user where the conflict is. The user will have to manually go to each file and resolve the conflict, and commit the new changes later. There exist automated tools for merge-conflict resolution, which automate the last two steps during the merging command itself.

When a merge occurs Git will create an automated merge commit, as seen in figure 2.6. If there are no merge-conflicts (as seen above) a new merge-commit is created (copy of C5) which will contain the same changes which were made at C5, but it will have two parents: C5 on 'iss53' and C4 on 'master'. In this case, changes made by C5 will appear twice: once on branch 'iss53' and once on branch 'master'.

If there are merge-conflicts, our new merge-commit will contain the changes which resolved that conflict, so C5 will only appear once on branch 'iss53'.

A key difference between Git and other VCS is that in Git we can treat the repository as a Directed Acyclic Graph (DAG) of commits, as the branching data is preserved, as we can see in figure 2.7

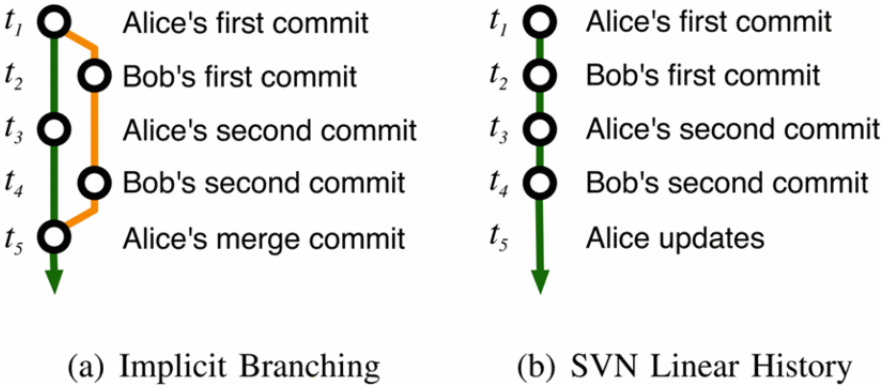

Figure 2.7: Comparing Git branching with other VCS. (a) is the DAG produced by git, while (b) is the linear history produced by SVN. Figure taken from [10]

Alice's merge commit will have two parents: Alice's second commit and Bob's second commit, thus preserving the DAG information, which is lost in SVN. This DAG representation comes handy when analysing how exactly each user contributed to a repository, but it makes backtracking a commit history more complex, as we might want to go through all the paths in the DAG. This will

increase the complexity of some code-mining and repository-mining algorithms [10].

Using Git for FixCache

Luckily, it is also possible to view the commit history as a linear list of commits (even when with merges present). We simply take each commit in the DAG and order them by their committed time, regardless on which branch were they made.

However, this linear representation will contain duplicates in Git (for the merges without merge-conflicts), as seen in figure 2.5: C5 will appear twice in our linear representation, once as a merging commit and once as a part of 'iss53'. We will use this linear representation when implementing FixCache.

2.4 GitPython

2.4.1 Overview

GitPython[5] is a Python package developed by Sebastian Thiel[6] as a pure Python high-level tool for interacting with Git repositories. It provides a mechanism for reading and writing Git objects by using Python code. Also it allows us to use Git functionality (creating new repositories, commiting changes, checking-out branches etc.) through function calls in Python.

When implementing FixCache, GitPython will only be used to access for the following tasks:

- Getting the list of commits (for the `master` branch) in chronological order, by their committed time.
- Determining differences in files between two commits.
- Keeping track of each file's history: number of changes, faults and lines at each commit.

[5] https://github.com/gitpython-developers/GitPython
[6] https://github.com/Byron

2.4.2 The object database

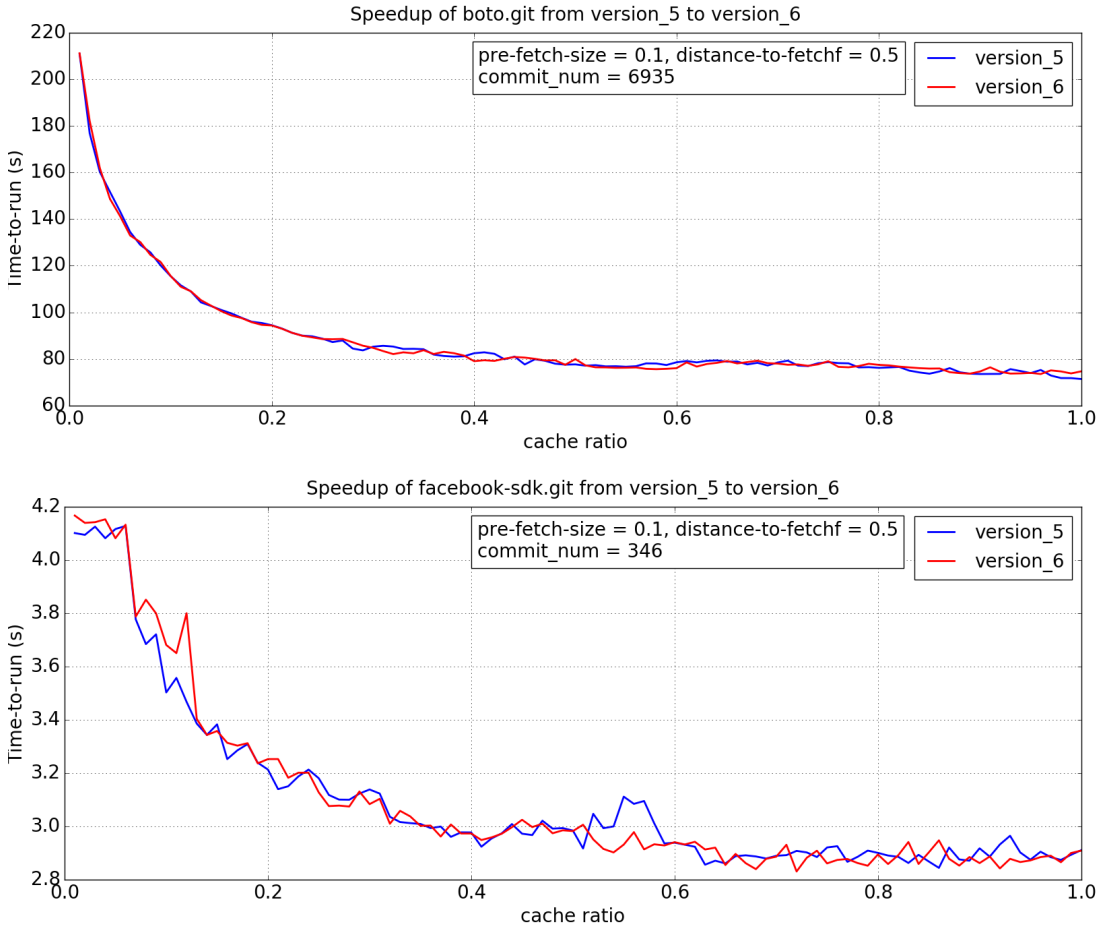

Figure 2.8: Showing time-to-run of backends `GitDB` (version_5) and `GitCmdObjectDB` (version_6) for repositories boto.git and facebook-sdk.git when FixCache is run for different cache-ratios. Here cache-size is computed directly from the cache-ratio, which will be explained in Chapter 3 in detail.

Behind the scenes GitPython is using a Git object database when accessing a Git repository. We have two options provided by GitPythonw: either we use the default `GitDB` (first implemented in the package gitdb[7], then added to GitPython) or we use `GitCmdObjectDB`. The first one, according to the Thiel *"uses less memory when handling huge files, but will be 2 to 5 times slower when extracting large quantities small of objects from densely packed repositories"* while second one *"uses persistent git-cat-file instances to read repository information. These*

[7] https://pypi.python.org/pypi/gitdb

operate very fast under all conditions, but will consume additional memory for the process itself. When extracting large files, memory usage will be much higher than the one of the `GitDB"`.

Figure 2.8 shows the time-to-run in seconds for both. As it can be seen, there is not a significant difference in the running time, so the default (`GitDB`) back-end was used for all further analyses.

The reason why it is more efficient to use gitdb is due to how gitdb handles memory usage. Rather than loading all the Git objects to the memory, it will operate on streams of data, which will ensure that our object database can handle objects of any size.

Chapter 3

Implementation

This chapter explained the main components implemented to run FixCache. It first explores a very high level view of all modules, and then it describes all components in more detail, together with any implementation difficulties. Finally, this chapter will talk about the bottleneck of the first implementation, and how this bottleneck was tackled to achieve a speed-up.

3.1 Design overview

The algorithm is implemented in Python using its 2.7.6 release. The implementation has several modules which handle different parts of the algorithm, the main module being the Repository module which implements the algorithm itself. We use several other back-end modules, such as Parsing, File-management and Cache. Figure 3.1 on the next page shows these main modules and how do these modules depend on each other.

Parsing module Accepts data from Repository (as can be seen in figure 3.1), and then it processes that data and sends the parsing result back to the Repository. It is essentially a collection of functions implementing different parsing functionalities used by FixCache: flagging important lines, parsing `git diff` messages to get the line numbers of deleted lines and flagging bug-fixing commits from the commit message.

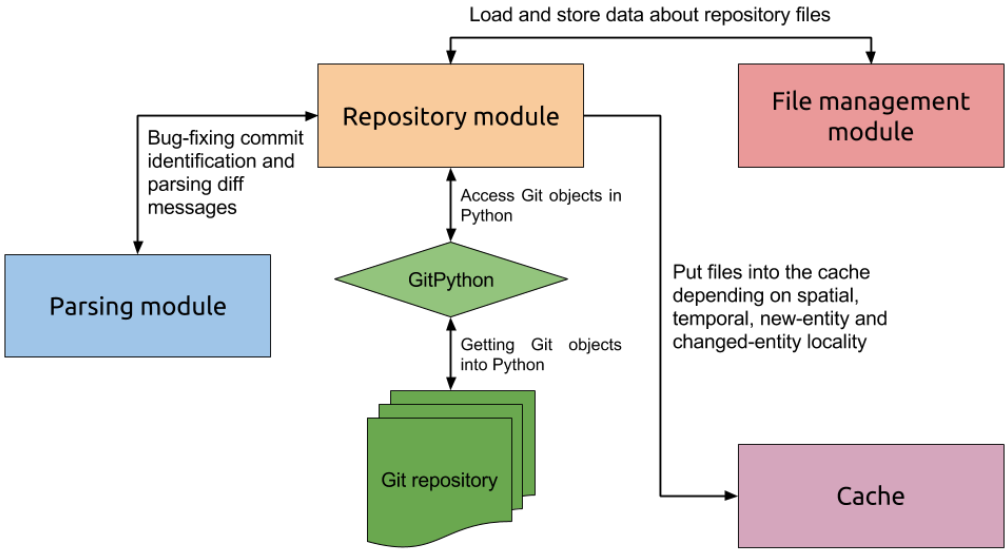

Figure 3.1: Core modules and the interaction between them. Lines indicate in which direction the data flows between modules.

File-management module Implements a layer for representing files, file-sets and distances between files. The classes implemented by this module are used both by the Cache module and by the Repository module, however only the latter accesses File-management explicitly. We can think of this module as a database which can be queried by the Repository module for different types of data such as file-metrics and relationships between files.

Cache module This module contains the `Cache` class which implements the cache functionality used by FixCache. Repository will look-up files in the Cache module and it will send new files to our Cache module. Cache will handle these additions: it will add the arriving files, and it will remove files if the cache is filled to make space for arrivals. Removing is handled implicitly by our module, hence the one-way line figure 3.1.

Repository module This is the main module which runs FixCache itself. It will communicate with all the other modules, also it will monitor the communication of these module between themselves. Moreover, this module will be in charge of accessing the Git repositories, through GitPython.

3.2 Back-end modules

Back-end modules are used by the Repository module, when running FixCache. They are the layer which handles the abstraction of objects and object metrics used by FixCache.

3.2.1 Parsing module

The Parsing module parses data arriving from the Repository module, as it can be see in the figure 3.2 representing the data-flow through the module. The three key functionalities implemented are: deciding whether a line is "important" or not; getting the numbers of deleted lines from the output of a `git diff` command; parsing the commit messages to identify fixing commits.

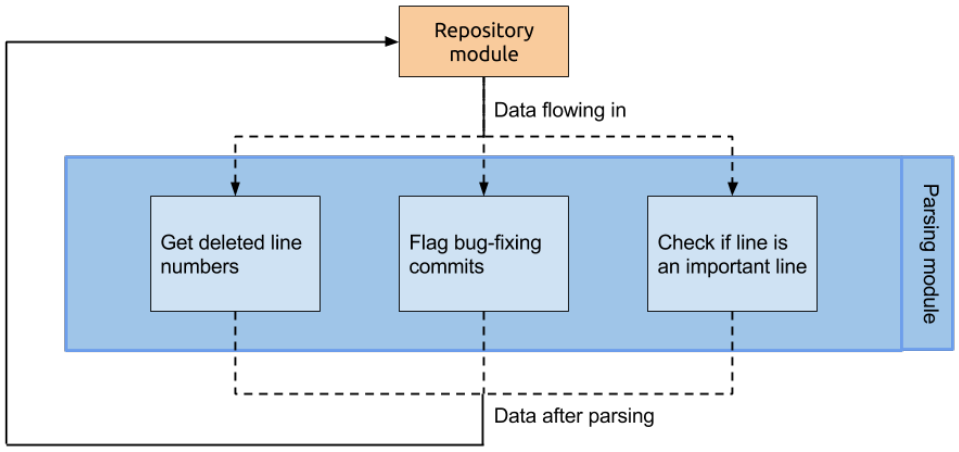

Figure 3.2: Data-flow through the Parsing module

Check if line is important

When identifying bug-introducing commits, we are using the SZZ algorithm (explained in section 2.2), which has a step of finding which lines are contributing to the bug. The algorithm assumes that each deleted line between a fixing commit C_{fix} and it's parent contributed to the bug fixed by C_{fix}.

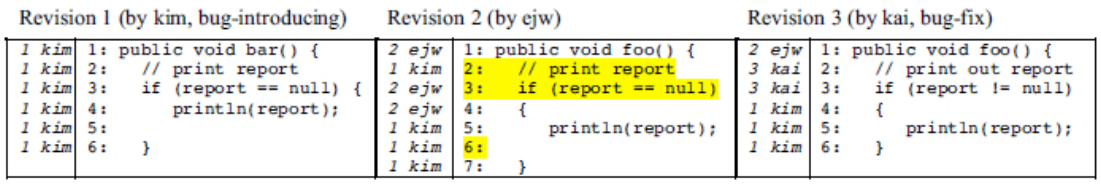

Figure 3.3: Example of changes between revisions (3rd fix is a bug-fix change). Figure taken from [8].

This assumption has been criticised by Kim *et al.* [8] as it treats all lines equally important, which might ultimately introduce plenty of false-positive lines, that is lines which were deleted between C_{fix} and it's parent, but had no contribution to the fixed bug whatsoever.

Figure 3.3 provides an example. Looking at the bug-fixing revision (Revision 3) we can see that it changed lines #2 and #3 (that is removed #2 and #3 and inserted new lines to that position) and it removed line #6. SZZ would identify all of these lines as buggy lines, whereas we can see that truly only #3 should be treated as a buggy line, because #2 is a comment line and #6 is blank line.

Several improvements have been proposed by Kim *et al.* to reduce the number of false-positives, out of these we are only checking if a line is blank/empty or if it is a comment line [8]. As this implementation is for Python repositories, to identify comment lines we parse each line and identify whether it starts with `#` (the comment symbol in python). Multi-line comments are ignored, as python uses the same syntax for them and for mutli-line strings.

This important line checker merely outputs an approximation for the lines which are truly important (that is they contributed to the bug fixed), usually it will return a bigger set. Further approximation of the true set would require more techniques mentioned by Kim *et al.*, but getting the exact set is computationally impossible [8].

Getting deleted line numbers

An essential part of the FixCache algorithm is identifying which lines were removed between two revisions. This information is later used by SZZ algorithm when identifying bug-introducing commits.

Our module will accept the output of a `git diff` command for each file in the commit, and it will pars this output to get the line numbers which were deleted in that `git diff` (where the difference is what changed between a commit and

it's parent commit), that is the line numbers of all the lines starting with "-" in our `git diff`.

Identifying bug-fixing commits

For each commit the algorithm looks at, we need to decide whether it is a fixing-commit or not. To identify these commits, we need to parse the commit message itself. If the commit message is accepted by any of the below regular expressions (following Sadowski *et al.* [5]), we flag it as a bug-fixing commit.

Regular expressions used to flag bug-fixing commits:

- `r'defect(s)?'`
- `r'patch(ing|es|ed)?'`
- `r'bug(s|fix(es)?)?'`
- `r'(re)?fix(es|ed|ing|age\s?up(s)?)?'`
- `r'debug(ged)?'`
- `r'\#\d+'`
- `r'back\s?out'`
- `r'revert(ing|ed)?'`

3.2.2 File-management module

When running FixCache we need to somehow store data about the files we looked at so far (data to store: number of faults, changes, their LOC) and also we need to store the distance between any two files in our system, at any commit. A good implementation would be able to tell for any two files f_1 and f_2 what is their co-occurrence (which is inversely proportional to their distance) at any commit n. Figure 3.4 shows a high level view of the File management module, and how does data flow through it.

As it can be seen from the figure, Repository requests objects (`filemanagement.File` or `filemanagement.Distance` objects) from the File-management module, and the module will return the requested objects from a set of files or from the distance database.

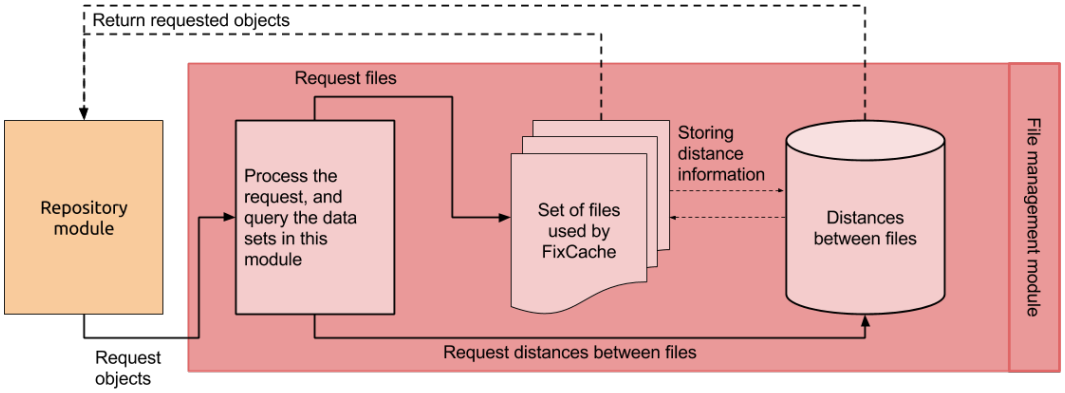

Figure 3.4: Data-flow through the File-management module

Processing requests

When requesting files, the Repository module accesses them only by their file-path. That is, it queries every `filemanagement.File` object by it's file-path, and the File-management module will either return the corresponding file, or if it did not exist before, it will create a new file with the queried path, and return this newly created file. We can think of this as a dictionary functionality with an extension that if a key looked-up is missed, this module will create the corresponding value, rather than throwing a `KeyError`.

Similarly for file-distances, Repository will query File-management by file-path, only this time the request will contain two paths (of the files we want to know the distance of). From these two paths, the Distance-database will create a key, and as before, it will return the corresponding value, or create a new `filemanagement.Distance` object. The key-generation is fairly simple:

$$\forall f_1, f_2 \; files$$

$$key(f_1, f_2) = \begin{cases} f_1.path + f_2.path & if \;\; f_1.path > f_2.path \\ f_2.path + f_1.path & if \;\; f_1.path < f_2.path \\ error & otherwise : two \; paths \; equal, f_1 == f_2 \end{cases}$$

That is we will append the shortest path to the longest path for any two files. If the two paths are equal, it means that the two files looked at are also equal, and we need to throw an error. Generally, this will never happen in this FixCache implementation, as we never query the distance with the same path.

Storing file information

For each file in the repository looked at when running FixCache we create a `filemanagement.File` object indexed by it's path, which will store the basic information needed by FixCache as mentioned above. Whenever an file-related event occurs during a FixCache run we need to change or update the internal data of the files involved in the event so that they have a correct state when used by FixCache. There are three three major file-events:

- *Changed file*: whenever a file is committed, we need to record this change at that commit. First we add the commit number to the changed-commit-list (which stores all the commits at which the file was changed). Then we need to update the LOC for that file, where the amount of lines changed will be equal to $added\_lines - deleted\_lines$.
- *Fixed fault*: at each bug-fixing commit C_{fix}, for each file involved, we mark that there was a fault associated with that file, and that this was fixed at C_{fix}. This is later not used, as we are using LRU cache-replacement policy, but it could have been used with different cache-replacement policies which take the number of faults and their occurrence in time into account.
- *Reset the file*: To complete an analysis we need to run FixCache 1500 times (for the cache-ratio analysis). If our repository, say has a 1000 files, we would create 1.5 million `filemanagement.File` instances (not taking into account any garbage-collector). This would be rather inefficient, so instead we reset all the objects present in our File-set and Distance-database: we set their internal parameters to their initial values.

Rather than requesting each file individually at these events, the output of a `git diff --stat` command is passed as data to the File-management module, which will update and return files accordingly. In Git, this output looks like this:

```
setup.py | 11 ++++++++++-
 1 file changed, 10 insertions(+), 1 deletion(-)
```

Luckily we do not need to parse this, as GitPython already has a parser, which will parse it and create a nested Python dictionary from the parsed data. For the output above, this dictionary will look like this:

```
{
    u'setup.py': {
        'deletions': 1,
        'lines': 11,
        'insertions': 10
    }
}
```

Each changed file's path will be a key in this dictionary, and the corresponding value will be an another dictionary with three keys: `deletions, lines, insertions`. Our module will use this parsed `diff` data to update and create new files in our File-management module. Furthermore, File-management will clean-up deleted files, based in this data: we assume that each file is deleted when their LOC becomes zero.

Once File-management finished updates and deletes, it will return the `filemanagement.File` instances for all the files involved in this update.

Distance-database

In order to implement the temporal locality used by FixCache, we need to know the co-occurrence of any two files at any commit. To store that, for each two files committed together a `filemanagement.Distance` instance will be created and stored in the Distance-database. This instance will have a list of commit numbers representing the commits when the two files were changed together. That is, for each two files in a commit, we either update the corresponding `filemanagement.Distance` object, or we create a new one if it has not yet existed.

Whenever two files are committed together (say at commit C_n, the Repository module will query the Distance-database (using the key generated as explained in 3.2.2) to get the `filemanagement.Distance` object, and it will update the co-occurrence for that object, that is add C_n to the commit list.

The Distance-database is also important when determining which files are the closest ones to a given file f at a given commit C_n. In this case, the Repos-

itory queries the Distance-database with one file path ($f.path$ in this case), and asking for k closest, files, and the database will respond with a list of k `filemanagement.File` objects with the highest co-occurrence with f (that is lowest distance to f).

3.2.3 Cache implementation

FixCache uses the notion of a file cache, which has some limited number of files in it. We can think of it as a bucket, where we can put some files in, and when the bucked is filled, it will automatically remove some files in order to make space for new arrivals. The `cache.Cache` class is responsible for keeping track of files which are in the cache, it also counts cache-hits and cache-misses, and as mentioned before, it will remove files when the cache is filled (according to the LRU policy). The figure 3.5 shows the cache handles the addition of new files.

Preprocessing Before putting the files into the cache, there is a pre-processing step, which will check if either of the files is already in our cache. If so, we will simply discard those files to avoid duplicates. This step is important as our cache needs to know exactly how much free space is required upon new arrivals.

The preprocessing step may also include sorting, if the cache is smaller than the number of files we want to insert. In this case, we need to sort them so that later they obey the LRU cache-replacement policy.

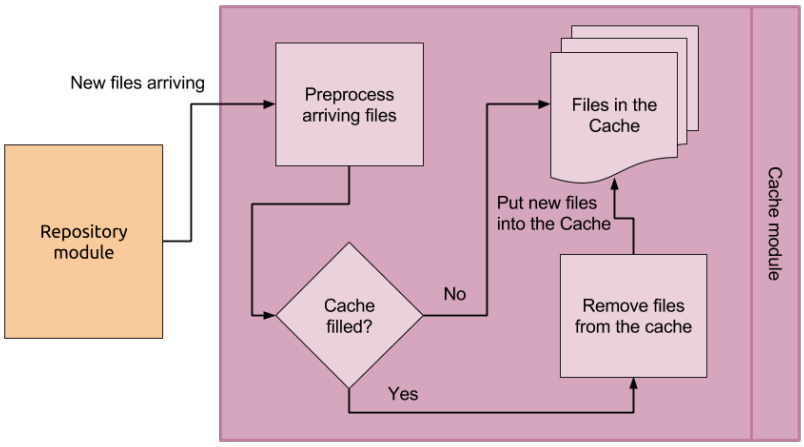

Figure 3.5: Adding new files to the Cache and processing them.

Adding new files After preprocessing we can add the arriving files to our cache. We first look-up how much space do we need, and remove files accordingly. Once the required space has been freed, we can safely add the new arrivals.

Removing files We are not removing any files explicitly. The implementation itself will take care of clean-up, so that it will only discard files when our cache needs more space. As we are using the LRU cache-replacement policy, if we need a space of k files, this module will select the top k least recently used (changed/committed) files and remove them from the cache.

Limiting the number of files Internally, files are stored in Python `set()` object, which does not allow duplicates, but it does not have a size limit by default. To actually have a limit in our cache implementation, all addition methods first look up how many files are currently in the cache, and subtract that value from the cache size to get how big is the currently available space. Removing files works accordingly to this: if we need more space, we remove as many files as necessary to free up the cache.

3.3 The Repository module

This module is the central element of FixCache implementation. It is connecting all modules together, and uses them to run and analyse the algorithm for different repositories. A `repository.Repository` class is instantiated for each Git repository we want to run FixCache on. Also, this module has two more classes, used only for evaluation purposes.

3.3.1 Implementing the SZZ algorithm

The SZZ algorithm in this implementation is implemented inside the `repository.Repository` class, rather than a standalone module, as it is a core part of FixCache. The two functions which are implementing it are:

- `_get_diff_deleted_lines(self, commit1, commit2)`

  Returns the deleted line-numbers (per file) between any two commits. Used to get the deleted lines between a commit and it's parent.

- `_get_line_introducing_commits(self, line_list, file_path, commit)`

  Once we have the line numbers which were deleted between two commits (commit and it's parent) we can for each file get the list of bug-introducing commits. This method uses the `git blame` (which, in GitPython outputs a nested list, where the first value of a list is a commit object, and the second value is a list of lines last changed - introduced - by that commit), and produces a list of bug-introducing commits.

The SZZ algorithm is used for identifying commits which introduced a bug, that is it is only called for bug-fixing commits.

3.3.2 Cache size initialisation

A key variable used by our algorithm is the cache-size. Upon each initialisation we need to set the cache-size which will be stay static throughout a single run of the algorithm. To calculate the cache-size, we need to set a variable called cache-ratio (a number between between 0.0 and 1.0), and from this we will calculate the cache-size the following way: we take the number of files in the repository at the last commit, and we multiply this number by the cache-ratio. We then take

the floor of this number (if the floor is zero, we add one, as the cache-size cannot be zero) to arrive at the desired cache-size. That is:

$$cr = file\_count\_at\_last\_commit * cache\_ratio$$

$$cache\_size = \begin{cases} 1 & if\ floor(cr) = 0 \\ floor(cr) & otherwise \end{cases}$$

$$where\ cache\_ratio \in (0.0 \ldots 1.0]$$

To calculate the variable file-count-at-last-commit we need to traverse the Git tree at the last revision, and count the number of 'blobs' (objects representing files) in the tree.

3.3.3 Setting variables

Once we calculated and set cache-size we can calculate other variables, such as distance-to-fetch (block-size) and pre-fetch-size as discussed in 2.1.4. In the original paper all these variables are given as a percentage of file-count-at-last-commit, that is if file-count-at-last-commit = 100, then distance-to-fetch = 5% means that we will fetch 5 files each time we use our spatial locality.

Rather than using percentages, this implementation for both distance-to-fetch and pre-fetch-size uses between 0.0 and 1.0. Also, they are not ratios of file-count-at-last-commit, but rather of cache-size. That is if file-count-at-last-commit is 100, cache-ratio is 0.1 and distance-to-fetch is 0.5, then the cache-size is 10 ($100 * 0.1$) and we will fetch $10 * 0.5 = 5$ number of files each time we use our spatial locality.

3.3.4 Initialising commits

In Git each commit's identifier is a 40-digit long hexadecimal number generated by SHA-1 from the commit data, it is impossible to know just from the hash value what is the order of commits. Our FixCache is using integers as commit identifiers (the bigger the commit number, the later it happened), so a dictionary needs to initialised prior to running our algorithm. We first iterate through the list of commits (which is provided by GitPython) and at each commit we store the hash value of that commit as key, and it's order in the list as the value. Each time we want to access the order of any commit object, we can lookup this dictionary.

3.3.5 Running FixCache

To run FixCache we need to call `run_fixcache` on a `repository.Repository` instance; calling this method will run FixCache with the currently set parameters. However, before running the algorithm itself, we need to find the repository for which we want to run it on.

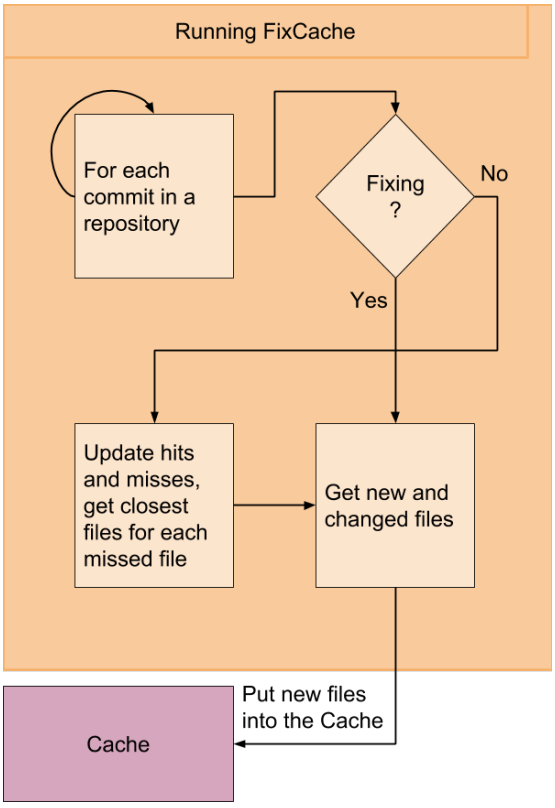

Figure 3.6: Control-flow of FixCache inside the Repository module.

The Repository module will accept a string as an input, which will be the name of the repository we want to look at. The module will look for a repository with this name (under a predefined directory for repositories) and it will connect to it with GitPython, or throw an error if the repository is not found. Once we have connected to the desired repository we can run FixCache. The figure 3.6 shows how the control-flow inside the Repository module works for a single run.

Between any two runs, we do not create a new connection with the repository, but rather we call a `reset()` function, which will implicitly reset the data found in the modules discussed so far. This reset propagation will save memory, and also save some time as it is not necessary to redo the initialization step before each run.

Once we have all these modules, implementing FixCache is quite straightforward. Following the figure 3.6, a more detailed pseudo code would look like the algorithm presented on the next page.

```
Data: A Git repository; Initial settings
Result: A cache with files projected to have bugs in the future
for each commit C in the commit list do
    if C has no parents then
        Pre-load the cache with some initial files, starting with ones with
        the highest LOC
    else if C has one parent then
        if C is a bug-fixing commit then
            for each file F committed by C do
                if F is in the cache then
                    Increase the number of hits.
                else
                    Increase the number of misses and add F to the cache.
                    Find when the bug-introducing commits for this file, and
                    put the closest-files to F at those commits to the cache
                    (number of files to get defined by distance-to-fetch).
                end
            end
        Add some part of changed and created files at C to the cache
        (number of files to insert defined by pre-fetch-size).
    else
        C is a merging commit, so it may be disregarded.
    end
end
```

Algorithm 1: FixCache

Cache additions are handled by our Cache module, while other operations (such as functions implementing SZZ) are handled by internal functions of `repository.Repository`.

In the above pseudo-code we have two if-statements: firstly how many parents does a commit have. If it has zero, it means that we are viewing the initial commit, so we need to pre-load the cache with some initial files (files with the biggest LOC). If it has one parent we proceed normally. If it has two or more parents it means that we have reached a merge commit.

If this merge commit resulted from a merge without conflicts, it is simple a duplicate of a commit which occurred before (as discussed in 2.3.4), so it can be disregarded. On the other hand, if the merge commit resulted from a merge with conflicts, it will contain the changes which resolved the merge conflict itself.

These changes are not considered to be bug-fixing changes (as they fixed an error made by wrongly designed work-flow rather than a software error), so in this case we can disregard the merge commit as well.

The second if-statement occurs when we are going through all files in a bug-fixing commit. If the file f was already present in the cache, we increase the hit-count. Otherwise, we increase the miss-count and and this file to the cache (temporal locality). Furthermore, we identify the commits which introduced (lets say: $C_{intr,1}, C_{intr,2} \dots C_{intr,n}$ the bug-fixed in this commit, and add the closest files to f at $C_{intr,1}, C_{intr,2} \dots C_{intr,n}$ (spatial locality).

Furthermore, for each commit we add the changed and updated files to the cache (new-entity and changed-entity locality). All these additions are bounded by the specified by the pre-fetch-size and distance-to-fetch however these are omitted from the pseudo-code.

3.4 Versions and errors

There are several versions implemented: from version_1 to version_6, which differ in various aspects. Versions 1 to 4 are incorrect, these were the part of the development process, in which different bugs were discovered in different places, to list just a few:

- Negative indexing in Python: rather than throwing an error, Python allows negative indexing, which was an issue as the initial line-difference-counter was broken, and it was accessing negative line numbers in a file (which is a list of lines in Python). Everything worked, but it was functionally incorrect.
- Broken diff-message parser: The output of a `git diff` command should be quite easy to parse, but the first implementation only looked for the first `@@` characters (which mark a meta-data line in the diff output), while several such characters might exist.
- Localities were broken at the beginning: Rather than getting the files with biggest LOC the algorithm was getting the ones with smallest LOC. Similarly with last-used vs. least-recently-used.

Version 5 and 6 only differ in that they use different object-database, as explained in 2.4.2, and there is no difference in speed between them.

Although versions 1 to 4 are functionally incorrect, it still makes sense to compare them to the functionally correct versions (5,6) in terms of how fast they are relative to each other, as they have roughly the same number of computations.

3.5 Implementation difficulties

3.5.1 Using GitPython

When using 3rd party software it is always key if the software we are using have a good or bad documentation (if it has a documentation at all). I found that GitPython has generally good a documentation, although some pieces (object reference, internal methods and variables of classes) lack proper definitions and clear type descriptions. There were times when the only way of finding out something about a certain class was to dive deep into the source-code of GitPython and try to figure out how is the certain class implementing functionalities provided by Git itself. This resulted in a slow implementation speed at the beginning of the project.

3.5.2 Bottleneck: sorting

During one run of FixCache there are several times when we have to select the "top objects from a set of objects". The "top objects" might mean "files with biggest LOC" or "least recently used files" or "closest files to a file at a commit". Usually the order does not matter after selection, we do not care whether the k number of files are sorted after selection, as long as all selected files are "bigger" (ie. "have greater LOC" or "were used least recently" or "are closer than") than any file which was not selected. Also, for each of these selections the k does not change throughout a single run of FixCache, as usually it's value is pre-calculated during initialization of the algorithm. On the other hand, n changes from commit to commit.

Initial implementation Initially all these selection procedures were implemented by: first sorting the whole set of n files, and then selecting the top k elements. This is rather inefficient in two cases:

1. If $k << n$, then we are wasting time on an $\mathcal{O}(n \log(n)$ operation, while we could do in $\mathcal{O}(n \log(k))$ using a selection algorithm explained on page 37, which for large n and small k is clearly more efficient than sorting.
2. If $k >= n$, then we could just return the whole set of n files, which is $\mathcal{O}(1)$ as the order after selection does not matter, thus we are wasting time again on sorting ($\mathcal{O}(n \log(n))$ instead of doing a constant-time operation.

From the two possibilities above we can see that if k approaches n from above, we can not really improve on the algorithm itself.

3.6 Speed-up

To achieve speed-up a "select top k elements from n objects" algorithm (mentioned in the previous section) was implemented. This can be done Using a binary-min-heap, the pseudo-code looks like this:

```
Data: A list of n sortable objects and k an integer.
Result: A list of k top objects from the list of n.
HEAP = BinaryHeap()
if if k is bigger or equal to n then
    return the input list of objects.
else
    for ITEM in the object list do
        if HEAP has less elements then k then
            push ITEM onto the HEAP.
        else if HEAP has k elements then
            if ITEM bigger than HEAP.min() then
                pop the smallest object from the HEAP, and push ITEM
                onto the HEAP.
    end
    return all the objects in the HEAP as a list.
end
```

Algorithm 2: Return the top k elements from a list of n sortable objects.

We need to have a binary min-heap, to keep track of what is the minimal item in the currently selected k objects. If we find anything that is bigger than the current minimum, we either simply push that onto the heap, or if there is no space in the heap (that is the heap has k elements) we pop the smallest item, and then

insert the new one. The time complexity of insertion and pop operations for a heap of size k is $\mathcal{O}(\log k)$, and since we need to traverse all the objects, the above function has a time complexity bounded by $\mathcal{O}(n \log k)$ as required.

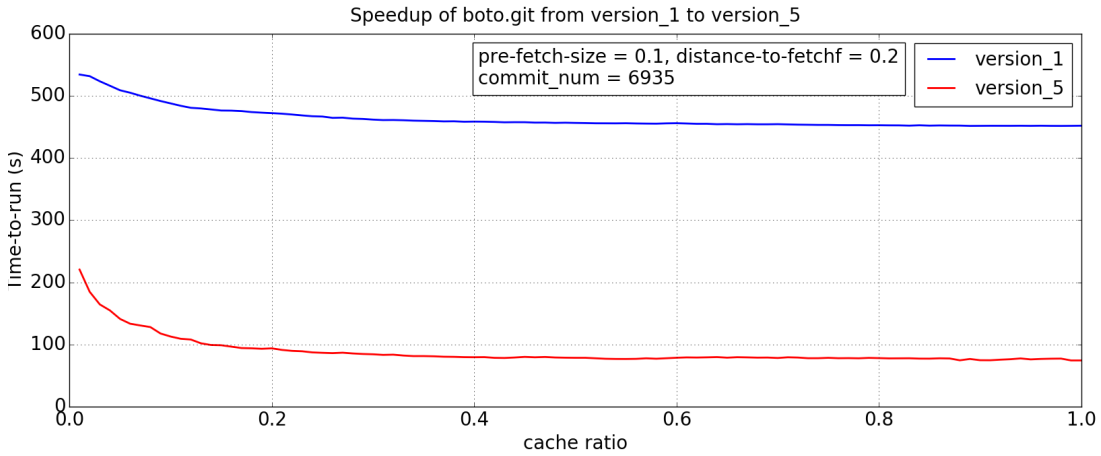

Figure 3.7: Speedup of boto.git and boto3.git between version_1 and version_5 with different cache-ratios

We can see on figure 3.7 on the next page, how good is fast did it take for FixCache to run for a given cache-ratio. The figure is showing two versions: version_1 (with sorting) and version_5 (with "select top k from n"), and its clear that version_5 is faster. This speed-up is caused mainly by replacing sorting, as we will be using more linear operations (if $k >= n$) and $\mathcal{O}(n \log k)$ operations, where for a single run k is fix, and n is changing.

Chapter 4

Evaluation

This chapter first introduces the repositories evaluated, and their properties. Then we will look at the results according to the evaluation method used by the original authors of FixCache. Finally, a new evaluation method is introduced, and two of the repositories are evaluated accordingly.

4.1 Repositories analysed and evaluated

The algorithm was evaluated and analysed using five repositories: facebook-sdk.git[1], raspberryio.git[2], boto3.git[3], boto.git[4] and django.git[5]. All these repositories were cloned, and FixCache was run on the locally: facebook-sdk.git was mainly used for testing purposes; raspberryio.git, boto3.git and boto.git was analysed completely; boto.git and django.git were used for evaluation. The repositories were chosen to have different number of commits, and to be mostly pure Python (except raspberryio.git which is only 43% Python), also they have been used in earlier projects I was involved in, hence the idea to analyse them.

As all the repositories were cloned, the data presented here (such as the number of commits for a repository) resembles the data for the date when the repository was cloned, that is: 11-10-2015 for facebook-sdk.git; 22-04-2014 for raspberryio.git; 12-01-2016 for boto3.git; 18-01-2016 for boto.git; 28-03-2016 for django.git.

[1] https://github.com/mobolic/facebook-sdk
[2] https://github.com/python/raspberryio
[3] https://github.com/boto/boto3
[4] https://github.com/boto/boto
[5] https://github.com/django/django

4.2 Evaluation modules

4.2.1 Analysing FixCache

To analyse and evaluate FixCache we need to run it several times each time with different parameters. The Analysis module contains different functions to perform different types of analysis. Out of the three key variables (cache-ratio, distance-to-fetch and pre-fetch-size) we fix one or two, and later we display the results found. The different analysis types implemented are listed below.

`analyse_by_cache_ratio`

Here, at a single analysis we fix both pre-fetch-size and distance-to-fetch, and run FixCache for 100 different cache sizes, that is cache-ratio $\in \{0.01, 0.02, 0.03 \ldots 0.99, 1.00\}$ We run this for different pre-fetch-size and distance-to-fetch values, namely pre-fetch-size $\in \{0.1, 0.15, 0.2\}$ and distance-to-fetch $\in \{0.1, 0.2, 0.3, 0.4, 0.5\}$ That is, a complete analysis of this type will require $100 * 5 * 3 = 1500$ runs of FixCache.

`analyse_by_fixed_cache_ratio`

Following the previous examples, here we fix the cache-ratio, and are interested in the best possible pre-fetch-size and distance-to-fetch for a given cache-ratio. We look at different cache-ratios, to be specific: $cache\_ratio \in \{0.05, 0.1, \ldots 0.5\}$.

4.2.2 Displaying results

We store all the results, for each analysis in .csv files. Once the .csv files were created, we read them and display the data using the matplotlib[6] Python library. The Graph module is responsible for reading, and displaying the .csv files created by our Analysis module, as .png images using matplotlib.

[6] http://matplotlib.org/

4.3 Evaluation over hit-rate

4.3.1 Results

In the original paper FixCache is evaluated and analysed against the hit-rate, that is the ratio of cache hits over all cache lookups. It was found for several projects (such as Apache1.3, PostgreSQL, Mozilla and others in [1]) that the hit rate is between 0.65 and 0.95 for cache-rate of 0.1, pre-fetch-size of 0.1 and distance-to-fetch of 0.5.

For the same variables, similar results were found: hit-rate was always between 0.6 and 0.83. Also, for repositories with higher number of commits we usually had a bigger hit-rate, as we can see on the diagram on page 41. At cache-size = 0.2, we get an even better result of hit-rate which will be between 0.75 and 0.9. It seems that for bigger repositories (such as boto.git) the curve is smoother than for smaller ones, as one might expect.

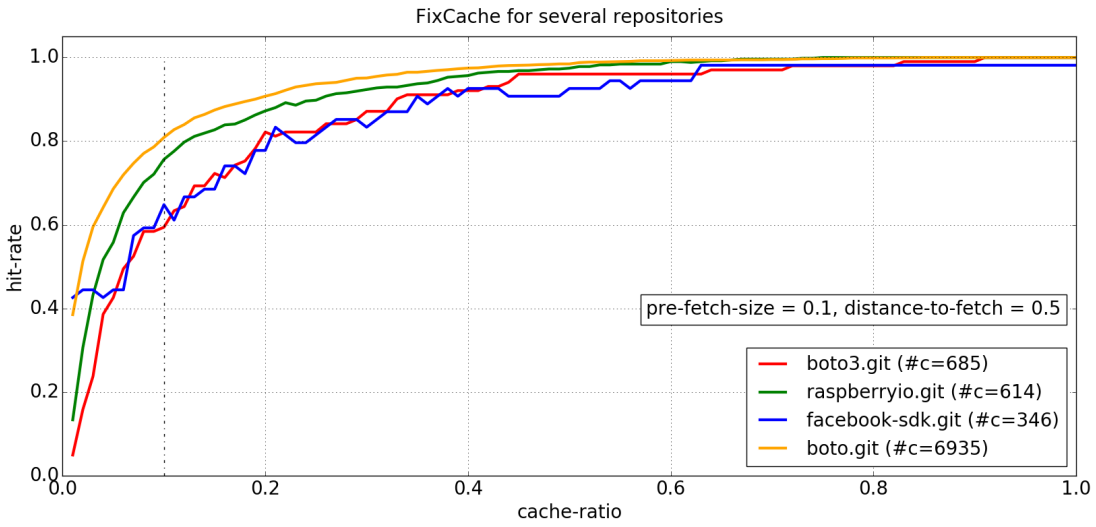

Figure 4.1: Evaluation of FixCache according to hit-rate for different repositories

What is the "best" cache-ratio? From the figure 4.1 it can be deduced that for cache-ratio of 0.2 we already get a really good hit-rate, so 0.2 can be considered the "best" cache-ratio. For this ratio, for all possible distance-to-fetch and pre-fetch-size values the raspberryio.git has a hit-rate between 0.86 and 0.89, and

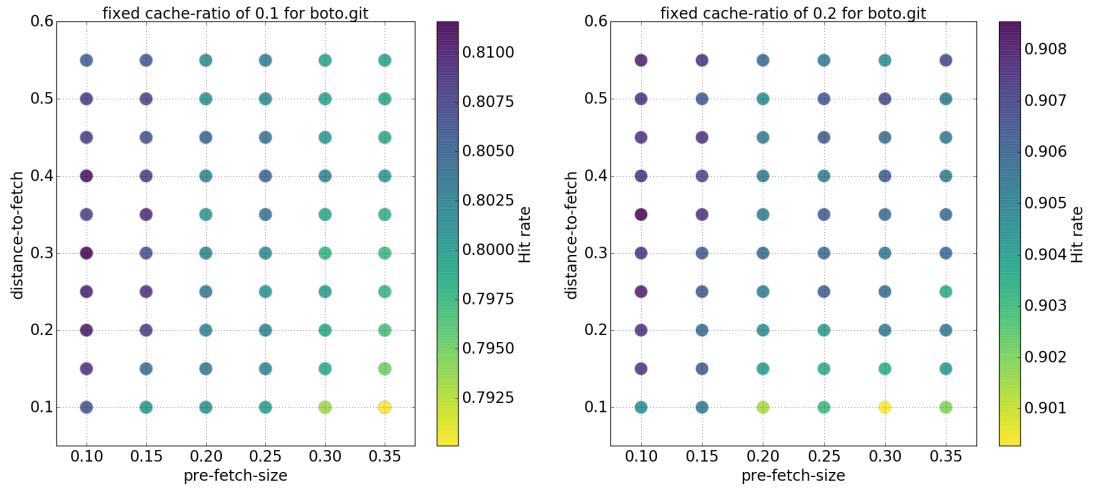

Figure 4.2: Fixed cache-ratio analysis for boto.git

boto.git has a hit rate between 0.9 and 0.92. Furthermore, if we fix pre-fetch-size and distance-to-fetch (0.1 and 0.5 respectively) all repositories looked at will have a hit-rate between 0.78 and 0.96. From this it seems that we get a closer result to the original evaluation for a higher hit rate. An interesting thing to note is that pre-fetch-size and distance-to-fetch does not change drastically our hit-rate as can be seen on the figure 4.2. This result was also found by Rahman *et al.*: they found that temporal locality has a significantly bigger impact on cache performance than new-entity-locality, changed-entity-locality (pre-fetch-size) and spatial locality (distance-to-fetch). That is, putting the most-recently faulty files to our contributes to our hit-rate the most from all [6].

Finding the best pre-fetch-size and distance-to-fetch Even though pre-fetch-size and distance-to-fetch are not the highest predictors of the cache-performance, it is still worthwhile to look at for which values of these variables is the hit-rate the biggest. In the in figure 4.2 we can see that for boto.git the results of a fixed-cache-ratio analysis, where we fixed the cache-ratio to 0.1 and to 0.2. The results indicate that for both we will get the best results when pre-fetch-size is low and distance-to-fetch is either 0.3 or 0.4 or 0.5. These results are in line with what was found in previous implementations [1].

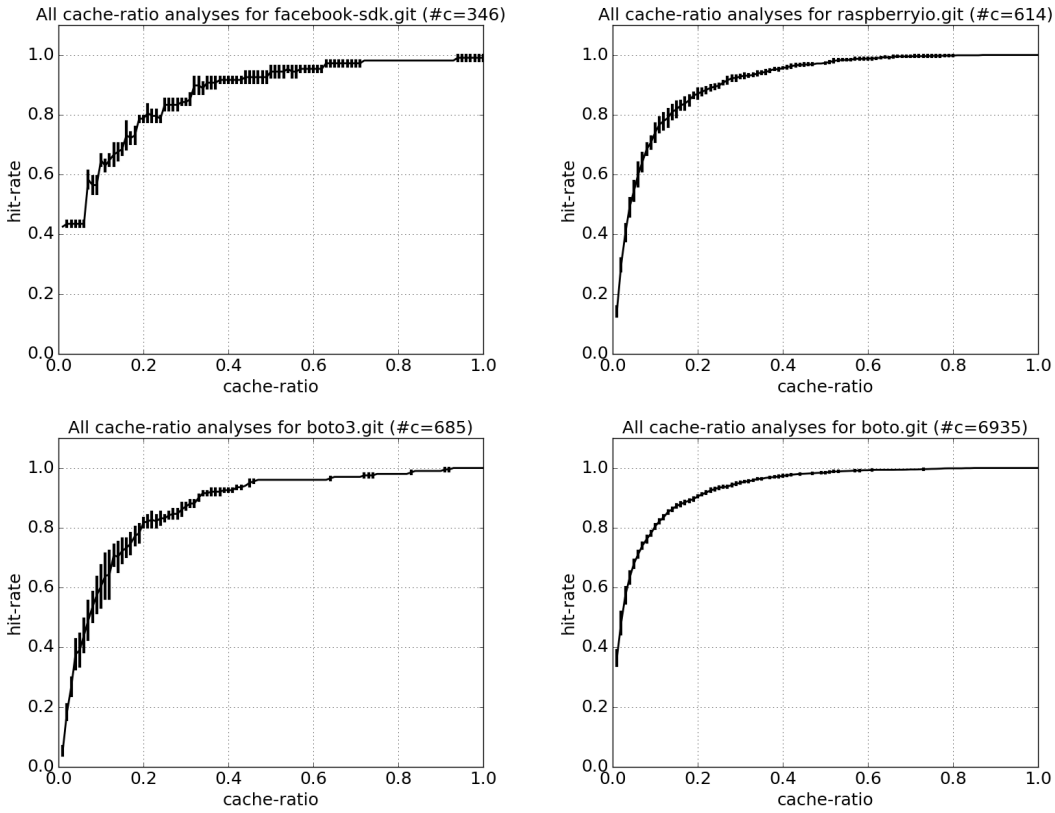

Figure 4.3: Showing all cache-ratio analyses for facebook-sdk.git, raspberryio.git, boto3.git and boto.git.

Variance in hit-rate for different variables How big is the difference between all the plots for a given repository, at any given cache-ratio? Figure 4.3 shows all the analyses for facebook-sdk.git, boto3.git, raspberryio.git and boto.git. The figure itself does not plot each and every analysis (taking all prefetch-size and distance-to-fetch values it would mean 15 curves), rather it plots vertical bars for each cache-ratio, where the bottom of the bar starts at the lowest hit-rate and ends at the highest hit-rate from all the analyses. That is, the bars are cover the area which was touched by any of the curves, and when a bar is taller, then the variance of hit-rates is also bigger for that repository at that cache-rate. As we can see boto3.git (which has a lot less commits than boto.git) has a higher variance in the hit-rate results, than boto.git has. The biggest difference for boto3.git it will be 0.158 at cache-ratio of 0.12, for boto.git it will be 0.073 at cache-ratio of 0.02. Similarily the other two repositories with less commits than boto.git, raspberryio.git and facebook-sdk.git also have a high variance in this graph, so we might reason that the less commits a repository has,

the bigger it's variance in the output will be. 0.076 at 0.06 for raspberryio. 0.092 at 0.16 for facebook-sdk.

What do these results mean? When comparing with the original paper, the results are similar (0.65-0.95 vs 0.6-0.83), although the top hit-rate boundary (0.95 for 0.1 cache-ratio) was not achieved, the highest was 0.83 for boto.git. This top boundary is reached for the cache-ratio of 0.2, which might mean that this implementation works the same for bigger cache-ratios. It is important to note that when making such a comparison one does need to remember that this implementation differs from the original in two key things: firstly, it was implemented for Git rather than Subversion and CSV and secondly it was run on open-source Python repositories rather than on C/C++ or Java projects.

From these results, the following points could be stated:

1. For a given repository R, the relationship between the number of faulty files in R and the size of R is not linear, as for a bigger R the variance decreases with variables staying the same (percentage of R's size).
2. The results found do not differ significantly from the results found in the original paper, although the same boundaries are reached for a higher cache-ratio of 0.2. Other than that, the variance between the hit-rates of different repositories is also quite big for cache-ratio of 0.1 (0.6-0.83 vs. 0.65-0.95 in the original paper), and this result seems to be connected to the number of commits in a repository (that is the history of a repository).
3. There might be several possible explanations for these differences in cache-ratios:
   (a) Python programmers make more mistakes (as opposed to C/C++ or Java developers) as the language is structured differently (loose type system: duck-typing, and possibly other differences), so in general Python repositories are more vulnerable
   (b) Open-source projects are more bug-prone as they are less "looked-after" and the development process is more chaotic than in well-organised companies and teams
   (c) As Git is a distributed, de-centralized version control system, programming with the help of Git might be less structured and organised than programming with the help of Subversion or other centralised systems.

Further analysis of these differences was out of the scope of this dissertation, but they definitely would be interesting to investigate further.

4. From these results we can see that FixCache works for Git, and also it works for open-source Python repositories, however the hit rate for the same parameters is slightly lower on-average.

4.3.2 Criticism

Authors of FixCache claimed that after running the algorithm, our cache will contain the files which "are more likely to have a bug in the future" than others. However, they did not specify what is "future" ie. how many commits/days/weeks ahead does FixCache predict? To know this, we have to introduce the notion of a windowed-repository to our implementation, which will be explained later, on page 45, and we find that FixCache indeed works for the first few commits in the future, but later it's accuracy drops.

Another criticism is that FixCache treats files as atomic entities whereas this is hardly the case. Files have several metrics on which they differ, such as LOC, or code density. If our algorithm identifies say 10% of our file base, this 10% might account for 20% of all lines in the repository. This will be generally the case, as longer files are expected to have more impact and importance on our project, hence they'll have more bugs. Sadowski and colleagues found that even if this is the case, FixCache finds the files with the highest "defect densities" (files which are more bug dense: LOC/#bugs), so the algorithm is indeed still helpful [6].

4.4 How good is FixCache?

Evaluating over hit-rate leaves several questions open, for example "Is FixCache identifying files which truly have bugs in the future?", "For how many commits in the future will FixCache work?".When trying to answer these questions a notion of a windowed-repository was created. This means, that rather than running FixCache for all commits until now, we divide the commit history to a window and a horizon, and run FixCache on the window and we check on the horizon whether it actually worked. That is, say our window is of size 0.9, our horizon then is 0.1, and say we have 1000 commits, then we would run FixCache on the first 900 commits, and then check the following 100 whether it actually worked.

If we look at figure 4.4, t0 would be at commit #1 and tNow would be at commit #1000. At each point in our timeline below, $C_{tX} = floor(tX)$, as the time axis below corresponds to the commit order from commit 1, to the latest commit.

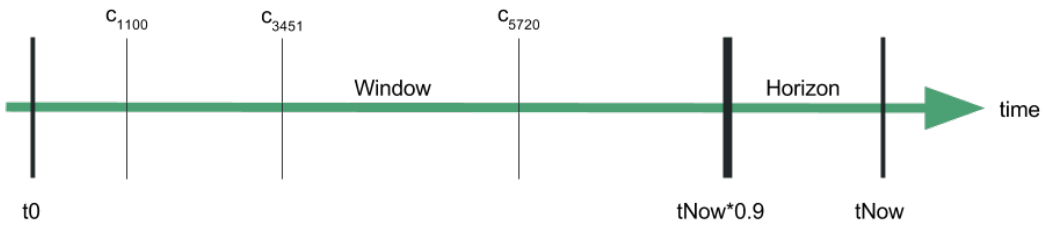

Figure 4.4: Window and horizon presented on a timeline of commits

4.4.1 Methodology

As mentioned before, we run FixCache for each repository, but we stop at tNow*0.9 number of commits (the window). By default when performing such an evaluation (using `evaluation` module's `evaluate_repository` mehtod) our default parameters are: window-ratio of 0.9, cache-ratio of 0.1, pre-fetch-size of 0.1 and distance-to-fetch of 0.5.

At the time we stop, we have some files in our cache, and this cache would be left unchanged as we walk through our horizon, lets call the set of files in the cache CH. Let tHStart = tNow*0.9, then at each point tX in our horizon (that is $\forall c_{tX} \in \{c_{tHStart}, \ldots, c_{tNow}\}$ we can define the following two sets:

- Normal-file-set at tX: Files which were changed (committed) between tHStart and tX, but were not part of a bug-fixing commit in this interval. Lets call this set NF.
- Faulty-file-set at tX: Files which changed in the same interval, but they were at least once changed as a part of a bug-fixing commit. Lets call this set FF.

From this, we can define the following properties:

- True-positives at tX (TP): The files in our cache, which indeed were faulty in [tHStart:tX]. That is $TP = FF \cap CH$.
- False-positives at tX (FP): The files in our cache, which were not faulty in [tHStart:tX]. That is $FP = NF \cap CH$.

- True-negatives at tX (TN): The files which are in our Normal-file-set in [tHStart:tX], and are not in our cache. That is $TN = NF \setminus CH$.
- False-negatives at tX(FN): The files which are in our False-file-set in [tHStart:tX], and are not in our cache. That is $FN = FF \setminus CH$.

From this we can see that all changed files in [tHStart:tX] are equal to $TP \cup FP \cup TN \cup FN$, and also that these sets are distinct.

Performance metrics Following the definition of TP, TN, FP and FN we can define several performance metrics on which we will evaluate our windowed-repositories[11]:

(a) Precision: $\frac{|TP|}{|TP|+|FP|}$

(b) False discovery rate: $\frac{|FP|}{|TP|+|FP|}$

(c) Negative predictive value: $\frac{|TN|}{|TN|+|FN|}$

(d) False omission rate: $\frac{|FN|}{|TN|+|FN|}$

(e) Recall (or sensitivity) $\frac{|TP|}{|TP|+|FN|}$

(f) Specificity: $\frac{|TN|}{|TN|+|FP|}$

(g) Accuracy: $\frac{|TP|+|TN|}{|TP|+|FP|+|TN|+|FN|}$

(h) F_β score:

$$(1+\beta^2) * \frac{precision * recall}{(\beta^2 * precision) + recall} = \frac{(1+\beta^2) * |TP|}{(1+\beta^2) * |TP| + \beta^2 * |FN| + |FP|}$$

Where β specifies whether we weight precision or recall higher.

4.4.2 Precision, recall and F_2 score

Out of the metrics above, we are mostly interested in precision, recall and F_2 score at each bug-fixing commit in our horizon. Figure 4.5 shows these metrics in a Venn-diagram. In this diagram the *relevant elements* corresponds to fixed-files (that is ones with bugs) whereas *selected elements* corresponds to files in our cache. Following this, we can define precision as the ratio of *selected and*

[7] https://en.wikipedia.org/wiki/Precision_and_recall

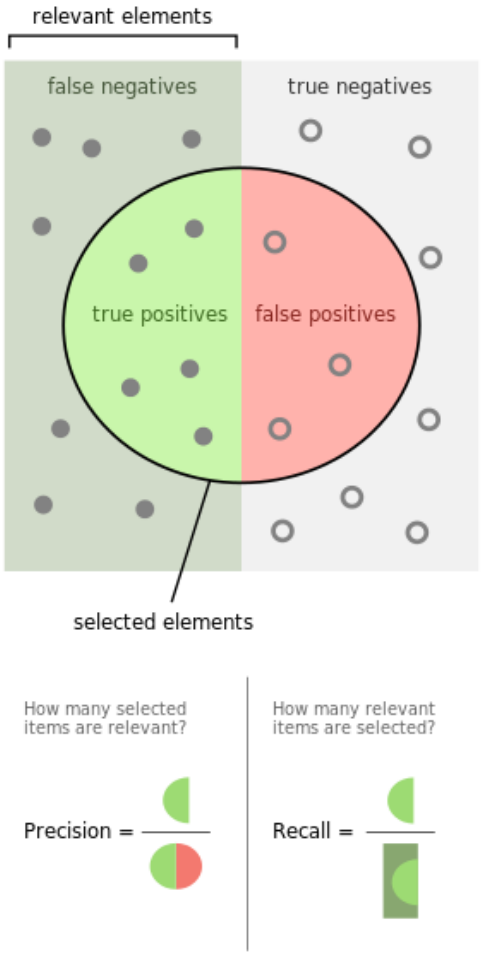

Figure 4.5: Precision and recall. Figure taken from Wikipedia[7].

relevant files and *selected but not relevant* files and recall as the ratio of *selected and relevant* files and *relevant* files. That is precision is tells us the percentage of files in the cache which are indeed buggy at each bug-fixing commit in our horizon while recall tells us the percentage of correctly predicted buggy files.

As defined above the F_β score weight precision and recall according to the parameter β. Since FixCache supposed to identify files which are 'more likely to contain a bug in the future' recall should be weighted higher, as when finding buggy files it should be more important to correctly identify the biggest subset of truly faulty files than to have some false-positives, therefore F_2 score will be used for evaluation. After substituting 2 for β we get that:

$$F_2 = \frac{5 * precision * recall}{4 * precision + recall} = \frac{5 * |TP|}{5 * |TP| + 4 * |FN| + |FP|}$$

4.4.3 Results

Evaluating boto.git

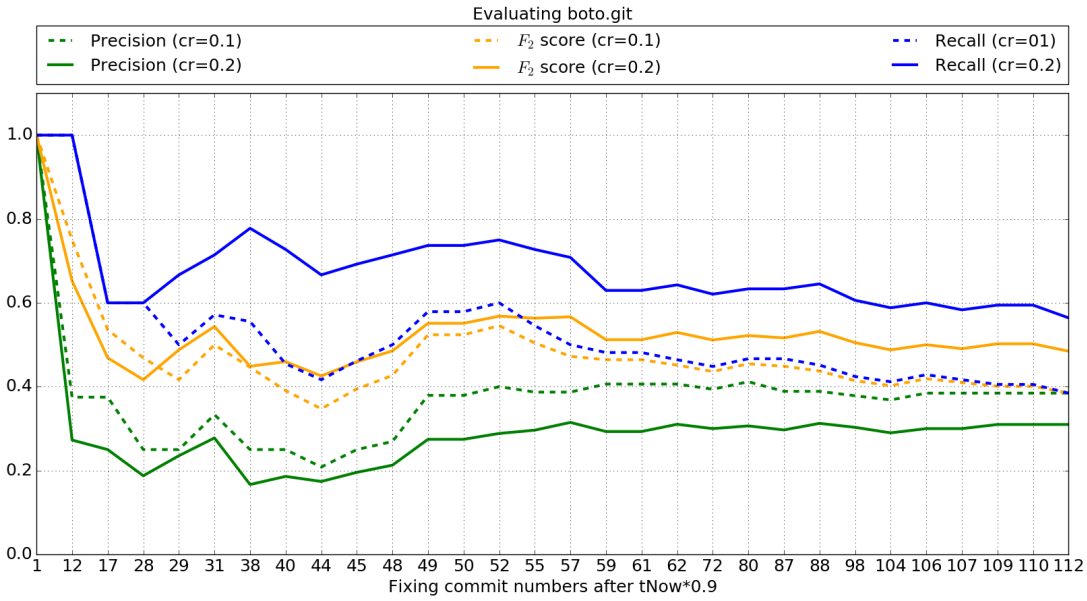

Figure 4.6: Evaluating boto.git with precision, recall and F_2 score for the first 30 commits in the horizon after tNow*0.9.

Figure 4.6 shows precision, F_2 score and recall for the first 30 fixing commit in the horizon for boto.git (at cache-ratio of 0.1 and 0.2), which at the time of evaluation has 6935 commits. For cache-ratio of 0.2 we can see that for the first fixing-commit in the horizon (that is for C_1) all the metrics are equal to 1.0. That is our algorithm identified buggy files fixed at C_1 correctly. However, at the second fixing commit, there is a massive drop in all three metrics, except for recall, when this drop occurs between C_{12} and C_{17} rather than between C_1 and C_{12}. If we look at recall, it always stays above 0.6 (except for the last few commits), that is we will identify at least 60% of the truly buggy files. Furthermore, after the sudden drop, it will have it's peak of 0.78 at C_{38}, after which it will continue to drop.

Recall at cache-ratio of 0.1 is lower or equal to the recall at 0.2, which is expected. However, precision is higher, which is due to the higher number of false-positives identified when having a bigger cache.

From this figure, we can deduce that for boto.git FixCache does only perform well for the first two commits in terms of recall, and does poorly afterwards (except

the peak at C_{38}; similarly it only works for the first commit for the other two metrics, after which precision and F_2 score will also drop. This means that in this case FixCache should not be used as a long-term bug prediction technique, but rather as a source for finding recently introduced bugs. Also, we can see that the performance is better with cache-ratio of 0.2.

Evaluating django.git

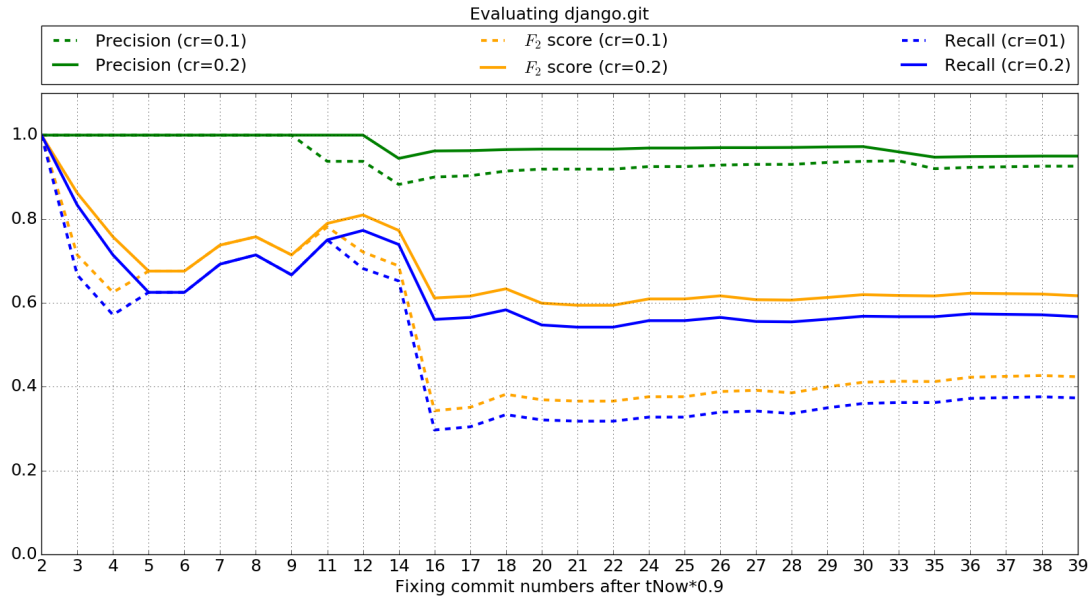

Figure 4.7: Evaluating django.git with precision, recall and F_2 score for the first 30 commits in the horizon after tNow*0.9.

Figure 4.7 shows the same data as 4.6 but this time for django.git, which has 22352 commits at the time of the evaluation. Also, a difference to observe is that nearly every commit in the horizon looked at is a fixing-commit (as we arrive at commit #39 which is the 30^{th} fixing-commit), that is fixes occur much more frequently. This is likely due to the fact that django.git is already a complete product, and all these fixing-commits are responses to issues reported by users rather than by the developers themselves. That is the implementation of django.git is finished (at least on the master branch) and hence nearly all the commits are fixing ones.

Again as with figure 4.6 the metrics are higher for cache-ratio of 0.2 (even for precision this time). However, the difference between cache-ratios is much bigger for the F_2 score after the drop happens (here at C_{12}). We can see that the

difference between F_2 scores for cache-ratio of 0.1 and 0.2 will be at least 0.2 after this point. This difference was never as dramatic in figure 4.6.

Another key difference between this figure and figure 4.6: precision stays high for all the commits. That is, if a file is identified as a buggy-file here, it is likely to indeed be buggy: our cache indeed has faulty files in it. Other than this, recall also drops after the first two commits, after which it also climbs back to a peak at around 0.78 (here at C_{12}), and after this peak it will drop below 0.6, so it is slightly worse than recall for boto.git.

4.5 Summary

When evaluating our repositories over hit-rate similar results were found to those of the original paper, however generally they were slightly lower on-average for cache-ratio of 0.1. For cache-ratio of 0.2 the results resembled more the findings of Kim *et al.* for their 10% cache-size [1].

The parameters (for a fixed cache-ratio) for which we have the best hit-rate possible were also found to be close to the ones identified in the original paper.

With the new evaluation technique, it has been found that FixCache as a predictor only performs well for the first few commits in the horizon (future) after which it's performance drops.

Chapter 5

Conclusion

5.1 Summary of work

This dissertation presents an implementation of FixCache, an algorithm which from a repository's commit-history identifies a subset of files which are more-likely to have a bug in the future. The algorithm was implemented in Python for Python Git repositories.

It was also explored how Git can be used for repository analysis, specifically for FixCache, and how any pitfalls related to this approach should be tackled. Overall, the project goals outlined in the project proposal were met, and all the components were completed on schedule.

The algorithm was evaluated against open-source Python repositories publicly accessible via GitHub. It was found that these repositories perform slightly poorer in terms of FixCache's hit-rate when compared to the repositories evaluated by Kim *et al.* [1]. It is yet to be discovered whether this difference is due to to Python itself, or the fact that this dissertation only looked at open-source public Git repositories.

5.2 Novel Contributions

Most of the previous implementations were implemented using Subversion or CVS as their back-end Version Control System [1][5], only Rahman *et al.* used Git (repositories from GitHub) before [6]. None of these implementations however were evaluated on Python repositories, all of them used C/C++ and Java projects, so evaluating the algorithm on Python is a novelty of this dissertation.

Another contribution was the introduction of the 'windowed-repository' evaluation. It has been found that using this evaluation paradigm that FixCache only works for the few first commits in the future in terms of precision (django.git is an exception here, where the precision was always near 1.0), recall and F_2 score. This means, that FixCache should only be used for immediate predictions, as it's performance drops in the long-term, so it should be re-run after each bug-fixing commit to always have an up-to-date prediction.

5.3 Limitations and future work

A limitation of this FixCache implementation is, that it is quite heavy-weight: for over 22 thousand commits of django.git, one run required up-to 3 hours to complete, so for large projects it might not be a feasible solution for bug prediction. This limitation however might only be due to the fact that this version was implemented in Python, which is considered to be slower than other highly used languages as Java or C/C++.

Further work could be done in achieving a faster implementation, to make it usable for larger repositories with several tens-of-thousands of commits.

Also, a major improvement in terms of hit-rate could be the implementation of a more sophisticated per line analysis of lines 'which truly contribute to a bug' (following the methods proposed by Kim *et al.* [8]).

Finally, it would be also interesting to implement a method-level FixCache for Git, in order to compare the results with what was found in that matter by Kim *et al.* [1].

Bibliography

[1] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[2] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31–41, May 2010.

[3] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, Jun 1994.

[4] A. E. Hassan. Predicting faults using the complexity of code changes. In *2009 IEEE 31st International Conference on Software Engineering*, pages 78–88, May 2009.

[5] Caitlin Sadowski, Chris Lewis, Zhongpeng Lin, Xiaoyan Zhu, and E. James Whitehead, Jr. An empirical analysis of the fixcache algorithm. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 219–222, New York, NY, USA, 2011. ACM.

[6] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. Bugcache for inspections: Hit or miss? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 322–331, New York, NY, USA, 2011. ACM.

[7] Linus Torvalds and Junio Hamano. Git: Fast version control system. http://git-scm.com, 2010.

[8] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *Proceedings*

of the 21st IEEE/ACM International Conference on Automated Software Engineering, ASE '06, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[9] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[10] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The promises and perils of mining git. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[11] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

Appendix A

Modules

A.1 `filemanagement.File` class

```
1  class File(object):
2      """File object used by the FileSet class.
3
4      Represents a file in the fixcache algorithms backend.
5      """
6
7      def __init__(self, path, commit=0, line_count=0):
8          """File initialization."""
9          try:
10             self.path = path
11             self.faults = 0
12             self.changes = 0
13             self.last_found = commit
14             self.line_count = line_count
15         except ValueError as ve:
16             logging.warning(ve)
17             raise FileError("Error during initialization
                   of file")
18
19     def __str__(self):
20         """File string representation."""
21         return self.path
22
```

```
23      @property
24      def path(self):
25          """The absolute path to a File in the repository.
                """
26          return self._path
27
28      @path.setter
29      def path(self, value):
30          if value == "":
31              raise ValueError("Path of a File cannot be
                    empty")
32          self._path = value
33
34      @property
35      def line_count(self):
36          """The line count of a file. This value evolves.
                """
37          return self._line_count
38
39      @line_count.setter
40      def line_count(self, value):
41          self._line_count = value
42
43      @property
44      def faults(self):
45          """The number of faults for a file. This value
                evolves."""
46          return self._faults
47
48      @faults.setter
49      def faults(self, value):
50          if value < 0:
51              raise ValueError("A File cannot have negative
                    number of faults")
52          self._faults = value
53
54      @property
55      def changes(self):
56          """The number of changes for a file. This value
```

```
            evolves."""
57          return self._changes
58
59      @changes.setter
60      def changes(self, value):
61          if value < 0:
62              raise ValueError(
63                  "Number of changes cannot be negative for
                        %s" %
64                  (value,))
65          self._changes = value
66
67      @property
68      def last_found(self):
69          """The last commit number when the file was found.
                """
70          return self._last_found
71
72      @last_found.setter
73      def last_found(self, value):
74          if value < 0:
75              raise ValueError("Last-found Commit number
                    cannot be negative")
76          self._last_found = value
77
78      def changed(self, commit):
79          """Called when file was changed."""
80          try:
81              self.changes += 1
82              self.last_found = commit
83          except ValueError as ve:
84              logging.warning(ve)
85              raise FileError("Error during calling change()
                    on file")
86
87      def fault(self, commit):
88          """Called when file had a fault.
89          """
90          try:
```

```
91              self.faults += 1
92          except ValueError as ve:
93              logging.warning(ve)
94              raise FileError("Error during calling fault()
                    on file")
95
96      def reset(self, line_count=0):
97          """Reset the given file, called when analysis
                restarted."""
98          self.changes = 0
99          self.last_found = 0
100         self.faults = 0
```

A.2 filemanagement.Distance class

```
1
2  class Distance(object):
3      """An abstract view of the distance object.
4
5      Stores two file pointers to two files, and their co-
           occurrence.
6      """
7
8      def __init__(self, file1_in, file2_in):
9          """Initialization."""
10         self.files = {
11             'file1': file1_in,
12             'file2': file2_in
13         }
14         self.occurrence_list = []
15
16     def reset(self):
17         """Reset the distance between two files."""
18         del self.occurrence_list
19         self.occurrence_list = []
20
21     def increase_occurrence(self, commit):
22         """Increse the occurrence.
23
24         Called when two files are present together in a
               commit.
25         """
26         if commit < 0:
27             raise DistanceError("commit cannot be negative
                   ")
28
29         if len(self.occurrence_list) == 0:
30             self.occurrence_list.append(commit)
31             return
32
33         if commit not in self.occurrence_list:
34             if commit < self.occurrence_list[-1]:
```

```
35                  for i in range(len(self.occurrence_list)):
36                      if self.occurrence_list[i] > commit:
37                          self.occurrence_list.insert(i,
                               commit)
38                          break
39              else:
40                  self.occurrence_list.append(commit)
41
42      def get_distance(self, commit=None):
43          """Return the distance which is 1/occurrence."""
44          raise DeprecatedError
45          try:
46              occurrence = self.get_occurrence(commit)
47              return 1.0 / float(occurrence)
48          except ZeroDivisionError as zde:
49              logging.warning(zde)
50              raise DistanceError(
51                  "Division by zero occurred during
                       get_distance()." +
52                  "The occurrence is 0")
53
54      def get_occurrence(self, commit=None):
55          """Return the occurence for a commit number in
               linear time."""
56          if commit is None:
57              return len(self.occurrence_list)
58
59          counter = 0
60          for commit_num in self.occurrence_list:
61              if commit_num > commit:
62                  return counter
63              elif commit_num == commit:
64                  return counter + 1
65              else:
66                  counter += 1
67
68          return counter
69
70      def get_other_file(self, file_in):
```

```
71          """Given a file path return the other file in the
               distance."""
72          if self.files['file1'].path == file_in.path:
73              return self.files['file2']
74          elif self.files['file2'].path == file_in.path:
75              return self.files['file1']
76          else:
77              raise DistanceError(
78                  'The file with path: %s is not in this
                       Distance object'
```

A.3 cache module

```
1  import logging
2  import helper_functions
3
4
5  class AbstractCache(object):
6      _hit = True
7      _miss = False
8
9      def __init__(self, size):
10         logging.debug('Cache initialized')
11         self.size = size
12         self.file_set = set()
13
14     @property
15     def hit(self):
16         return self._hit
17
18     @property
19     def miss(self):
20         return self._miss
21
22     @property
23     def size(self):
24         return self._size
25
26     @size.setter
27     def size(self, value):
28         if value < 1:
29             raise ValueError("Cache size cannot be less
                   than 1")
30         self._size = value
31
32     def _filled(self):
33         return len(self.file_set) == self.size
34
35     def _find_file_to_remove(self):
36         raise NotImplementedError
```

```
37
38      def _get_files_to_remove(self, number):
39          raise NotImplementedError
40
41      def _remove_multiple(self, number=1):
42          if number >= self.size:
43              # empty the whole file set
44              self.file_set = set()
45          else:
46              remove_file_set = set(self.
                    _get_files_to_remove(number))
47              self.file_set -= remove_file_set
48
49      def _remove(self):
50          len_ = len(self.file_set)
51
52          if len_ < 1:
53              return None
54          elif len_ == 1:
55              file_ = self.file_set.pop()
56              return file_
57          else:
58              file_ = self._find_file_to_remove()
59              self.file_set -= {file_}
60              return file_
61
62      def _get_free_space(self):
63          space = self.size - len(self.file_set)
64
65          assert space >= 0
66
67          return space
68
69      def _preprocess_multiple(self, files):
70          files = filter(lambda x: x not in self.file_set,
                files)
71          return files
72
73      def file_in(self, file_):
```

```
74          if file_ in self.file_set:
75              return self.hit
76          else:
77              return self.miss
78
79      def add(self, file_):
80          if self._filled():
81              self._remove()
82
83          self.file_set.add(file_)
84
85      def add_multiple(self, files):
86          files = self._preprocess_multiple(files)
87
88          len_ = len(files)
89
90          if len_ == 1:
91              self.add(files[0])
92          elif len_ <= self._get_free_space():
93              self.file_set = self.file_set | set(files)
94          elif len_ <= self.size:
95              to_remove = len_ - self._get_free_space()
96              self._remove_multiple(to_remove)
97              self.file_set = self.file_set | set(files)
98          else:
99              files_to_sort = helper_functions.
                    get_top_elements(
100                     [(x.last_found, x) for x in files], self.
                            size)
101             files_to_insert = [x[1] for x in files_to_sort
                    ]
102             del self.file_set
103             self.file_set = set(files_to_insert)
104
105     def remove_files(self, files):
106         for file_ in files:
107             self.file_set.discard(file_)
108
109     def flush(self):
```

```
110         del self.file_set
111         self.file_set = set()
112 
113     def reset(self, size=None):
114         self.flush()
115 
116         if size is not None:
117             self.size = size
118 
119 
120 class Cache(AbstractCache):
121     def _find_file_to_remove(self):
122         file_to_remove = None
123         for file_ in self.file_set:
124             if file_to_remove is None:
125                 file_to_remove = file_
126             else:
127                 if file_to_remove.last_found > file_.
                       last_found:
128                     file_to_remove = file_
129 
130         return file_to_remove
131 
132     def _get_files_to_remove(self, number):
133         file_list = list(self.file_set)
134         file_tuple_list = helper_functions.
               get_top_elements(
135             [(-x.last_found, x) for x in file_list],
                   number)
136         file_list = [x[1] for x in file_tuple_list]
137 
138         return file_list
```

Appendix B

Project Proposal

Tamas Sztanka-Toth
Homerton College
ts579

Computer Science Part II Project Proposal

Cached Bug Prediction for Python repositories on GitHub

23/10/2015

Project Originator: Professor A. Blackwell and Mr A. Sarkar

Resources required: See relevant section.

Project supervisor: Mr A. Sarkar

Signature:

Director of studies: Dr B. Roman

Signature:

Overseers: Professor J. Crowcroft and Dr. T. Sauerwald

Signature:

Introduction and Description of the Work

Testing and debugging has always been one of the most important parts of Computer Science, especially in Software Development. As projects and repositories grow bigger and bigger every day, more and more time is spent on writing unit-tests, debugging and code review. However, even with the most advanced testing tools, review techniques there still will be bugs and errors, which will only be found after deployment, hence bug prediction algorithms have gained importance. The aim of this project is to implement and investigate one of such algorithms, FixCache [1], and use it to predict bugs in Python repositories on GitHub. Previous implementations were used on C, C++, Java and Javascript projects, rather than Python, and they were mostly mining softwares repositories with different version control system: Subversion and CVS [1][2]

There are several approaches to bug-prediction to date, which rely on different information, and they can be put into two main categories: Change-log approaches and Single-version approaches [3]. Ultimately, all of them try to achieve the same results: to identify files, classes, functions which more contain a bug more likely than others. The Single-version approaches use the current state of the repository and calculate variety of metrics to predict faulty behavior. The most commonly used metric is the so-called Chidamber and Kremmer metrics suite [4]. In contrast, the Change-log approaches look at how the repositories evolved over time, and try to predict where the future defects will be. Hassan's algorithm calculates the complexity of each change and file to calculate the entropy of changes which then is used for bug-prediction [5]. The FixCache algorithm, on the other hand, approaches this problem by looking at the history of a repository and by identifying which commits have introduced a bug, and which of them have fixed it. Then, using the history of the repository, the algorithm can infer a set of bug-prone files.

Although this algorithm meant to work on every repository independently of language, I could not find any research which specifically focused on Python project (the vast majority of implementations was run on C, C++ and Java projects). As a result, my project will primarily focus on running FixCache on open-source, public, Python projects found on GitHub.

Resources Required

No special resources required. This project will entirely require the use of my personal laptop and the resources available on the MCS machines. The project will be implemented in Python.

Starting Point

- Part IA Object Oriented Programming course, Part IA Algorithms course.
- Summer internships at NRICH (after Part IA) and RealVNC (after Part IB), during both worked on projects which required advanced knowledge of Python.

Success Criteria

For the core of the project:

- To implement the FixCache algorithm in Python.
- To apply the FixCache algorithm to open-source, public, Python repositories found on GitHub and compare the results found with other implementations and repositories (eg., Java, C, C++) and deduce any language specific differences identified.

Possible extensions:

- Modify the algorithm using ideas from other bug-prediction approaches, in order to increase the correctness and performance.

Substance and Structure of the Project

1. Preparation: Background reading on bug-prediction techniques, approaches, especially technical details about FixCache algorithm.
2. Implementation: Start off by implementing the the bug-commit module which can be thought of as a parsing module, in a sense that it will prepare the data for FixCache. It will responsible for: (i) identifying when did bug-fixes occur, and which files were associated with them, (ii) during which commits were these bugs introduced, (iii) finally to sort the bug-introducing commits. Once this module is done, the core algorithm can be implemented, using data which was created by the bug-commit module.
3. Evaluation: This can be broken into two parts.
    1. Firstly, it is essential to evaluate whether the algorithm is working correctly. That is whether the files found by it, are really those which are more bug-prone than others. To evaluate this, a simple evaluation-module will be written, which will work as follows: given two points in the time, T (some point in the past) and N (present), the module will run the algorithm up until T, and then it will check whether the files in the list returned are present in any of bug-fixing commits between T and N.
    2. Secondly, the results found can be compared with previous research, and also with implementations ran on other languages' (C, C++ and Java) repositories.Testing and Improvement: Test the algorithm on several repositories, and try improving it.

4. Testing and Improvement: Test the algorithm on several repositories, and try improving it. The algorithm will be tested on the repositories listed above, in chronological order. The first repository was chosen as it has below 500 commits, so it will suffice when testing the commit parsing and bug-identifying parts of the algorithm. The latter three were picked randomly from the top thirty most starred Python repositories found on GitHub.

    1. https://github.com/pythonforfacebook/facebook-sdk
    2. https://github.com/boto/boto3
    3. https://github.com/boto/boto
    4. https://github.com/django/django

5. Write up the dissertation.

Timetable and Milestones

Week 1-2 (October 10 - October 23)

Contact project supervisor and arrange a meeting. Discuss the project ideas and start reading about bug-prediction, FixCache algorithm in particular. Write the Draft Proposal, and send it to the Overseers. Write the Project Proposal and submit it.

Milestones: Project Proposal submitted, project identified.

Week 3-4 (October 24 - November 6)

Research more deeply into the FixCache algorithm, especially the techniques to identify commits which introduce bugs into repositories. Identify any implementation differences for Python repositories.

Milestones: Theoretical background sufficient to start the implementation.

Week 5-6 (November 7 - November 20)

Implement the bug-commit identifying module of the algorithm, and test it if it works correctly on both small and large projects. Use this module to sort the bug-introducing commits in time.

Milestones: Bug-commit identifying module implemented.

Week 7-8 (November 21 - December 4)

Implement the core algorithm using the bug-commit module implementation from before. Run the algorithm on both big and small repositories and store the results in a structured manner.

Milestones: first version of FixCache algorithm implemented and working.

Week 9-10 (December 5 - December 18) - *Christmas Vacation*

Write the evaluation module for the algorithm, which will be used to empirically test how effective the algorithm is when identifying bug-prone files.

Milestones: Evaluation module implemented.

Week 11-14 (December 19 - January 15) - *Christmas Vacation*

Test the implementation using the evaluation module, and see if it works correctly. Compare the results with results found by other implementations, ran on other languages' repositories. By modifying the parameters of the algorithm, see how the results change accordingly. A short vacation is also planned in this period.

Milestones: Tested implementation, comparisons with other results made.

Week 15-16 (January 16 - January 29)

Prepare the 'Progress Report' and prepare a presentation about the work done so far.

Milestones: Progress Report submitted, presentation created.

Week 17-18 (January 30 - February 12)

Try modifying the algorithm in order to increase performance. Identify any possible language specific findings, differences (as a results of this implementation or of comparing it to other implementations).

Milestones: investigated the possibility of modifying the algorithm in order to increase performance.

Week 18-19 (February 13 - February 26)

Start writing up the project, starting with the 'Preparation' and 'Implementation' chapters of the dissertation.

Milestones: draft 'Preparation', 'Implementation' chapters written

Week 20-21 (February 27 - March 11)

Finish the 'Preparation' and 'Implementation' chapters, and start writing up the 'Evaluation' chapter.

Milestones: 'Preparation', 'Implementation' chapters finished.

Week 22-23 (March 12 - March 25) - *Easter Vacation*

Finish the 'Evaluation' chapter, write 'Conclusions' and 'Introduction'. Send over the first draft dissertation to supervisor, overseers and DoS.

Milestones: first draft dissertation completed, and sent to the supervisor.

Week 24-26 (March 26 - April 15) - *Easter Vacation*

Improve the dissertation quality relying on comments made by the supervisor, overseers and DoS. During this period revision for the exams will also be done.

Milestones: second draft dissertation completed.

Week 27-30 (April 16 - May 13)

Final four weeks. This serve as a buffer period for any unexpected difficulties. Complete the dissertation, send it over to supervisor, overseers and DoS, and submit it at least two weeks before the deadline (Friday the 13th of May). Revision for exams is continued in this period.

Milestones: Dissertation submitted.

Bibliography

1: S. Kim, T. Zimmermann, E. J. Whitehead, Jr., A. Zeller; *Predicting Faults from Cached History*; ICSE '07 Proceedings of the 29th international conference on Software Engineering, 2007

2: C. Sadowski, C. Lewis, Z. Lin, X. Zhu, E.J. Whitehead, Jr.; *An empirical analysis of the FixCache algorithm*; Proceedings of the 8th Working Conference on Mining Software Repositories, 2011

3: M. Lanza, R. Robbes; *An extensive comparison of bug prediction approaches*; Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on, 2010

4: S.R. Chidamber, C.F. Kemerer; *A metrics suite for object oriented design*; IEEE Transactions on Software Engineering, 1994

5: A. E. Hassan; *Predicting faults using the complexity of code changes*; Proceedings of the 31st International Conference on Software Engineering, 2009

Resource Declaration

Hardware

- My personal computer: Samsung Ultrabook series 5 (256 GB SSD, Intel Core i3 1.90GhZ x4, 4GB DDR3 Memory) for code development and writing dissertation. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.
- MCS machines as back-up, in case my personal computer fails.
- Back-up storage: External HDD with 500GB, plus a Git repository.

Software

- Sublime Text 3 editor for development.
- PyCharm IDE as a back-up for development (with student account).
- GitHub repository for version control and back up (private repository with student account).