Tamas Sztanka-Toth

Cached Bug Prediction for Python repositories on GitHub

Computer Science Tripos - Part II

Homerton College

May 2, 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, July 2016**
Word Count:	**X**[1]
Project Originator:	Advait Sarkar
Supervisor:	Advait Sarkar

Original Aims of the Project

Work Completed

Special Difficulties

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

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

List of Figures

Acknowledgements

Chapter 1

Introduction

This dissertation explores one of the bug-prediction techniques used for Source Code Management (SCM) systems: FixCache[1]. It will first explain the idea behind this algorithm, together with necessary background knowledge in Chapter 2; then it will talk about the implementation details and difficulties in Chapter 3; finally it will explore the analysis results for five different Python repositories and evaluate the results found in Chapter 4.

This chapter will further talk about motivation behind the dissertation itself, and describe some previous work done in the field of bug-prediction and mining software repositories.

1.1 Motivation

Testing and debugging during software development has always been one of the most important parts of Computer Science, especially in Software Development. An extremely important part of the software-development cycle is finding and fixing bugs prior to the production version of a product. As these projects and repositories grow bigger and bigger every day, more and more time is spent on writing unit-tests, debugging and code review. However, as many of the bugs are functional bugs rather than compilation errors, 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.

If we could narrow down the file base to a smaller subset of files which are more bug-prone, it would speed-up the revision process of a faulty program. This would not only save time, so that developers could focus on more advanced problems, but also would save money for the companies as they would spend less money on debugging and releasing hot-fixes. Previous FixCache implementations[1][2], except one[3] did not use Git[4] as their back-end SCM, but rather they used an alternative one such as Subversion (SVN) or CVS. Also, these were mostly 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.

1.2 Previous work

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[5]. 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[6]. 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[7].

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. The algorithm was introduced by Sunghun et al. in 2007[1]. Since then it has been researched and tested heavily: Sadowski et al.[2] analysed how the algorithm performs on the file-level, and how does the hit-rate (introduced by [1]) perform over time. Rahman et al.[3] compared the algorithm to other prediction techniques, and found it to be better than a naive prediction technique, but not significantly.

Chapter 2

Preparation

This chapter summarizes the work done before implementation, relevant literature and concepts used by FixCache algorithm. It first discusses FixCache and it's background; then it describes Git[4] software management system and how can it be used to implement FixCache. Finally, in this chapter we will look at GitPython: a Python library used to acces Git repositories.

2.1 Overview of FixCache algorithm

2.1.1 History

FixCache is a bug-prediction algorithm implemented in 2007 by researchers at MIT[1]. The original algorithm was implemented for Subversion and CVS, whereas my implementation is for Git. These Source Code Management systems differ in lot of aspects, the key difference between Git and the others, is that Git is distributed, that is it does not require a central 'parent' or 'master' repository. This means that we do not need a remote server access every time we want to commit a new change, we simply commit them locally, and later we can push those commits to a remote location. This feature speeds up programming, but also it introduces new problems to tackle when one is mining repositories[8].

The authors argued in the original paper that after FixCache is ran for a repository it will correctly identify some subset of the repositories file-base which subset is said to contain files which are "more likely to have a bug in the future". This dissertation will further explore this proposition, will discuss the best parameter

for FixCache and in the Evaluation chapter will compare results with the original paper.

FixCache can also be implemented for method-level bug-prediction[1]. The difference is that rather than being interested in files as entities, our algorithm looks at standalone methods. This dissertation, however, only implemented the file-level version of FixCache.

2.1.2 Abstract view of the algorithm

The algorithm works as follows:

1. Preload the cache with initial items (new-entity locality).
2. For each commit C in the repository, starting from the first commit:
3. If the commit is identified as a fixing-commit:

   For each file uploaded by that commit, we check if they are in the cache. If a file is in the cache, we record a hit, else we record a miss.
4. For each file F which were not in the cache, we go back to the commits when the bugs in those files were introduced (say revision n), and take some closest files to F, at revision n (spacial locality).
5. We select some subset of new-files and changed-files at C, and put this subset to the cache (new-entity locality and changed-entity locality).
6. We remove those files from the cache, which were deleted at C (clean-up).

So far, files were only added to the cache. What happens is that the algorithm in the background takes care of cache pollution, and removes files from the cache to make space for new arrivals.

Cache-replacement policies

The algorithm itself is called FixCache as it uses a notion of a cache, which stores a subset of files in a repository. Following this logic, the algorithm defines a notion of locality: spacial, temporal, new-entity and changed-entity. We use these localities to put files into the cache, to increase its accuracy. 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 they were added/changed earlier in the 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 I have used the LRU cache replacement policy, as it is simple, and there is not significant difference between the different replacement policies[1][2][3].

2.1.3 Hits, misses and hit rate

We only look-up the cache at the so-called bug-fixing commits. 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 [9]). 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.`

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. Then we define hit-rate, as the ratio of hits over all lookups, that is:

$$hitrate = \frac{\#hits}{(\#misses + \#hits)}$$

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

Since at each commit we either increase the hit-count or miss-count (or both), the hit-rate itself is a cumulative indicator of how good our algorithm is.

At each commit, the cache contains both fault and non-faulty files. What is really important in FixCache, is that only truly faulty files score hits, so there is a relation between the hit-rate and True-positives (files that are faulty and are in the cache) in the cache itself, that is there is a relation between hit-rate and the number of identified faulty files. Some might argue that this evaluation strategy is poor, as it is not known what is the window of our prediction, that is how early/late will the files in the cache contain bugs. There exists other, more sophisticated evaluations, by other authors such as [2] and [3] which look at: examples here....

2.1.4 Cache-localities and variables

As mentioned before there are four different localities when talking about FixCache: temporal, spatial, changed-entity and new-entity.

Temporal-locality

As with physical caches, temporal-locality in FixCache is used following the idea that files which were used (had a fault) will be used (will have a fault) in the near future. At each bug-fixing commit, we load all the files involved in the cache, regardless of whether we recorded a hit, or miss for them.

Spatial-locality

Again, the idea of spatial-locality comes from the world of physical caches: when a file is used (has a fault) it is likely that other files "near" to that file will be used (will have a fault). To define nearness, we first need to define the notion of distance for two files in FixCache. Distance is for any two files f_1 and f_2 at revision/commit n is defined as (following [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.

Distance-to-fetch (block-size): this parameter (variable) is used to define how many files will be fetched when loading the closest files to a file at revision n.

This locality is used every time we have a cache-miss. 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[10]. Note that bug-fixing commits are not the same as bug-introducing commits. The former were identified by parsing the commit message, while the latter are identified by the SZZ algorithm.

Changed-entity-locality and new-entity-locality

At each revision we put newly encountered files into the cache. These can be further divided into two categories: new files (new-entity-locality) and files changed between the revision viewed and the previous revision (changed-entity-locality).

Pre-fetch-size: this variable is used for both changed-entity and new-entity locality 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 revision n the 5 files with highest LOC.

Initalising the cache

To encounter a small amount of misses at the beginning, we need to pre-load the cache with some files at revision 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 revision 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.

2.1.5 Identifying fixing commits: the SZZ algorithm

The SZZ algorithm was introduced by J. Śliverski, T. Zimmermann and A. Zeller [10] to identify which commit introduced a bug in a file, when viewing the file at a bug-fixing commit. The algorithm works the following way:

1. At a fixing commit, identify which lines were deleted at this between this commit and it's parent commit. We assume that all of these lines were deleted as a result of the bug-fixing procedure, so that they contributed to the bug itself.
2. Once we know the lines, we need to check where were these lines introduced. We take all these commits, and say that these are the bug-introducing commits for a given bug-fix. 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.

The assumption made by SZZ, that each deleted line in fact was a 'buggy' line is a really strong one, and it leads to plenty false positive lines. There are several techniques to lower this number, which are discussed in more detail by [9]. Out of those, as explained later in section 3.2.1, this SZZ implementation is only using the comment-line and blank-line reduction.

2.2 Git

2.2.1 Overview

Git[4] is a modern code version-control system which uses distributed revision-control. The main difference with other version-control systems, say Subversion is, that Git is fully distributed. That is, a developer can make some changes on their own machine, and without accessing the main repository they can commit their changes to the local machine, as the local machine will have a valid Git repository. Later the developer can push changes from the local repository to any other machine which it has access to, as seen in the figure 2.1. 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. It was

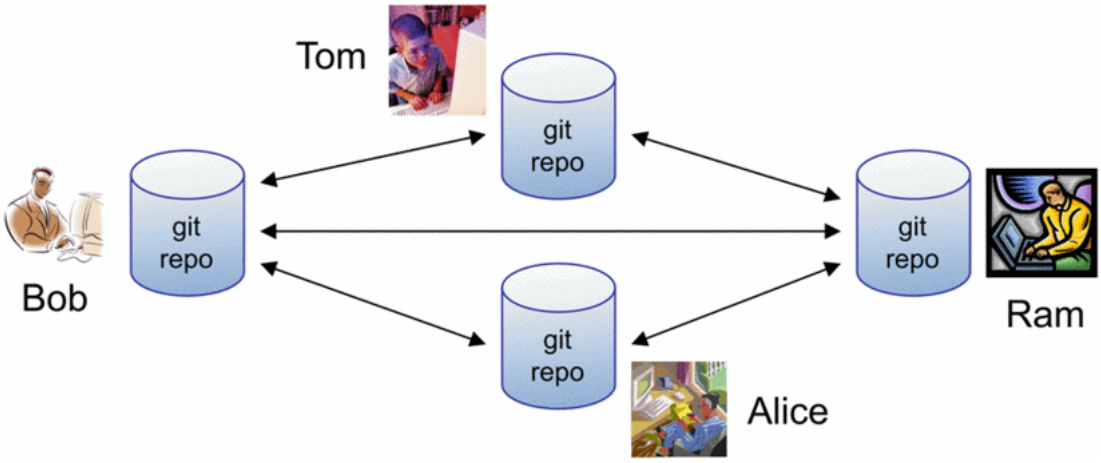

Figure 2.1: Distributed revision-control with Git. Figure taken from [4].

developed to help the Linux kernel developers with proper version control system. Since its development in 2005, it has been widely used by open-source projects as their primary VCS, such as the Django Project[2], Ruby on Rails[3] or jQuery[4].

2.2.2 Git snapshots - how does the version control works

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 revision 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 2.2. We can think of a Git repository as an ordered list of snapshots. To view differences (using the `git diff` command) Git looks up the differences in a repository between two snapshots. In this definition, version control is simply done by storing (committing) at various times the state of our repository, directory structure.

2.2.3 Git File structure

At each revision point in a Git repository we can think of data as a small file structure. Objects in a Git snapshot can either be blobs or trees: Blobs corre-

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

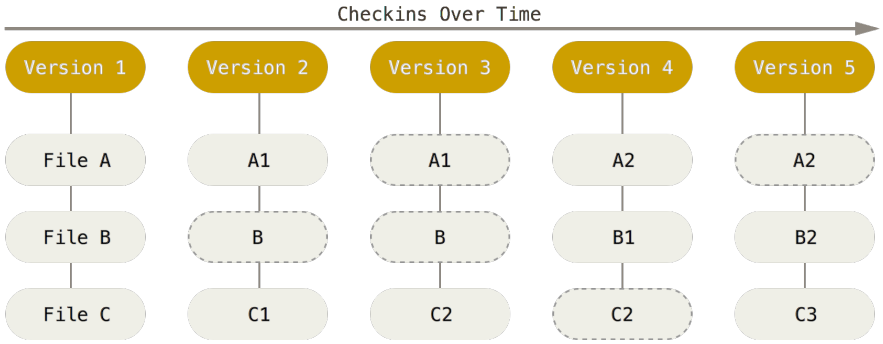

Figure 2.2: Snapshots in Git, pointers to files have striped lines. Figure taken from https://git-scm.com/book/

spond 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 about that commit, for example: the author, the time committed. Furthermore, this commit object has pointer to a tree object, corresponding to the root directory of the repository as seen in figure 2.3. Furthermore, each tree can have a pointer to many more trees or blobs. Blobs, as they correspond to files, do not have any children. Each commit has a pointer to it's parent(s), except the initial commit which doesn't have parent(s).

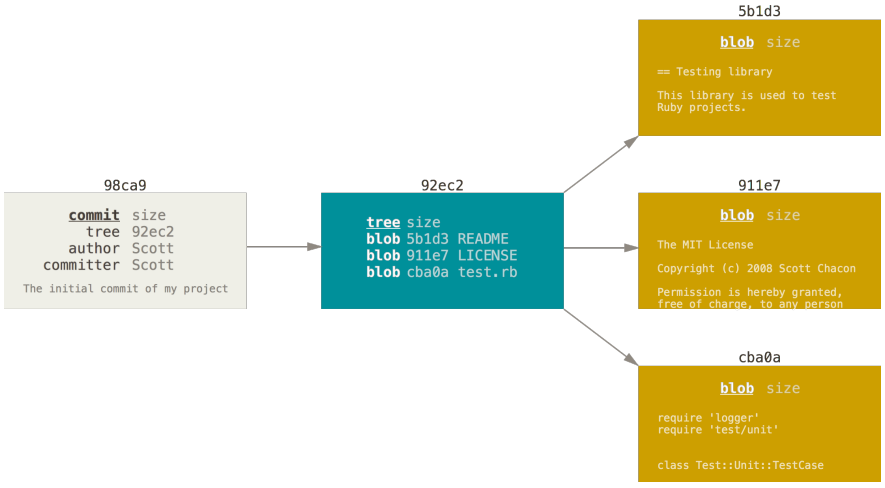

Figure 2.3: Git file-structure for a given commit. Figure taken from https://git-scm.com/book/.

2.2.4 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 a non-linear development in a decentralised manner, meaning that developments can make their own changes locally, and later join/merge these changes together.

As 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 object.

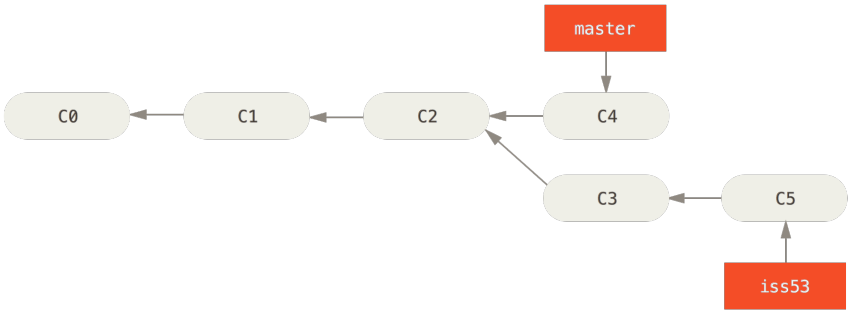

Figure 2.4: Basic branching in git, with two branches: ‘master‘ and ‘iss53‘. Figure taken from https://git-scm.com/book/.

Diverging branches

It is possible to diverge two branches, as seen in figure 2.4. 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 2.5 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. When performing

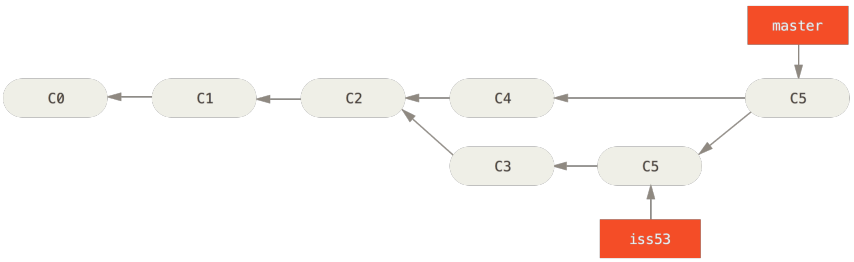

Figure 2.5: Merging ‘iss53‘ branch into the ‘master‘ branch. Figure taken from https://git-scm.com/book/.

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" algorithm.

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, which will contain the information about the which commits were merged together, that is this commit will have at least two parents. The changes contained by a merge

commit are also visible on the branch they were really made, this is an important feature if you want to view the repository as a linear history of commits.

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 the figure 2.6 Alice's merge commit will have

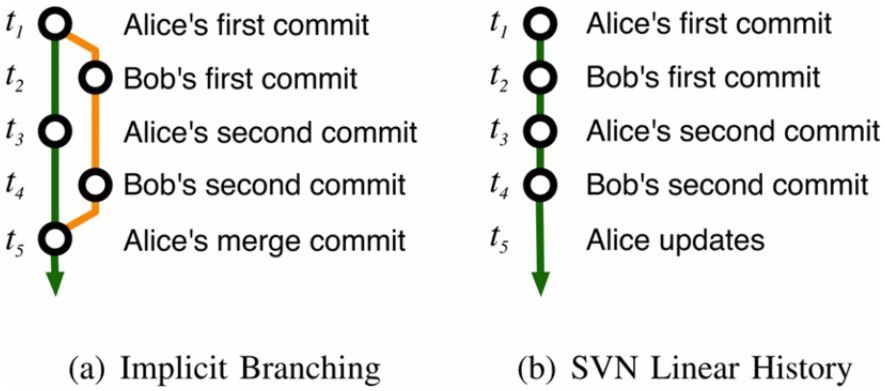

Figure 2.6: Comparing Git branching with other VCS. (a) is the DAG produced by git, while (b) is the linear history produced by Subversion. Figure taken from [8]

two parents: Alice's second commit and Bob's second commit, thus preserving the DAG information, which is lost in Subversion. This DAG representation comes handy when analysing different user activities, but it introduces several new problems, because when we are mining a repository we might want to check each path in a repository.

Luckily, it is also possible to view each branch as a linear set of commits (even when some branches were merged into this branch). We will simply take each commit in the DAG and order them by committed time, regardless on which branch were they made. However, this linear representation will also contain all the merge commits, which will be the ones with at least two parents. This means, that when going through this history merged changes will be visible twice: once in their original commit (made on some branch B1) and once in the merge commit of branches B1 and B2 say. We will use this linear representation when implementing FixCache.

2.3 GitPython

2.3.1 Overview

GitPython[5] is a python package developed by Sebastian Thiel[6] as a purely python high-level tool for interacting with Git repositories. By using it, one is able to do everything via Python: create new repositories, commit changes, checkout branches etc. Each Git object has its own representation in GitPython, and each object is stored in a object database to achieve lower memory overhead. In fixcache, GitPython will only be used to access for the following tasks:

- Getting the list of commits (for the `master` branch) in chronological order
- To determine differences in files between two commits
- To keep track of each files history: number of changes, faults and lines at each commit.

2.3.2 The object database

Behind the scenes GitPython is using gitdb[7] for accessing a git reposiotry. This is an efficient way of accessing data, as gitdb only operates on streams of data rather than the whole objects, so it actually requires small amount of memory. There are two options here, either we use the default `GitDB` (implemented by the gitdb package) or the `GitCmdObjectDB` which was only added to the GitPython. 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 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 2.7 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.

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

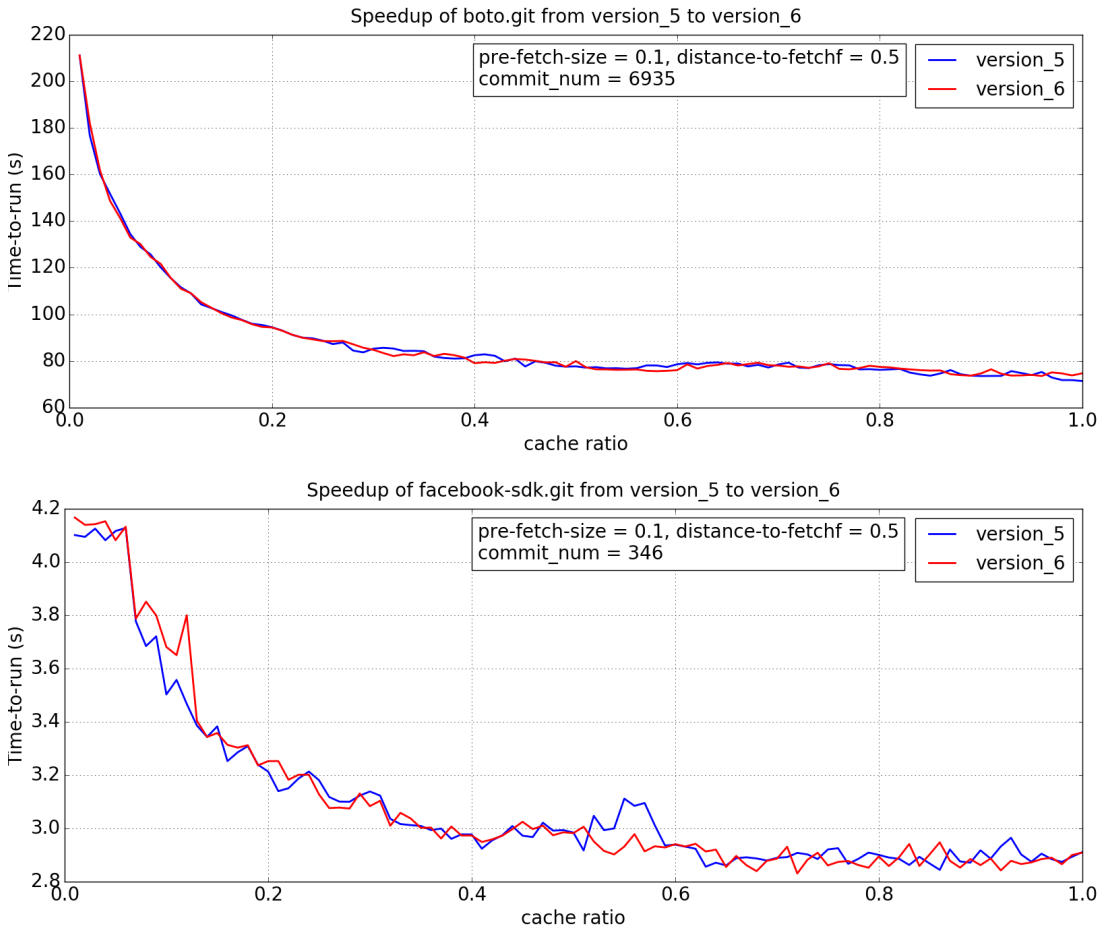

Figure 2.7: Showing time-to-run of backends `GitDB` (version_5) and `GitCmdObjectDB` (version_6) for repositories boto.git and facebook-sdk.git.

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 is the Repository which implements the algorithm itself. It uses several back-end modules, such as Parsing, File management and Cache. Figure **??** 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 on figure **??**, and then it processes that data and sends the parsing result back to the Repository. It is essentially a collection of functions which do different parsing 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.

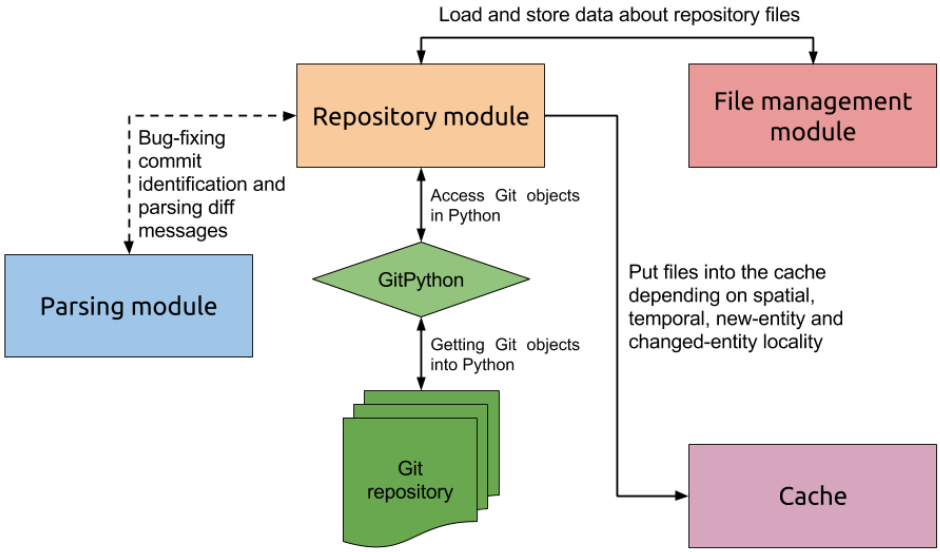

Figure 3.1: Core modules and the interaction between them.

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 represents our cache when running FixCache in our `repository` module. Instances of this class will store how good the algorithm is, that is they will keep track of number of hits and number of misses. The Repository module will send new files to our Cache module, which will handle additions to cache, and removals if the cache is filled. The removals are handled implicitly, hence the one-way line in figure **??**.

Repository module This is the main module which runs FixCache itself. It communicates with all the other modules (requesting existing files, adding new files, updating the cache, parsing data), and handles the communication of these module between themselves. Also it will read the the Git repository (through GitPython on which we are currently running FixCache itself. The communi-

cation of Repository with other modules differ by module. In the figure ?? the striped line means that there are no `File` or `FileDistance` (these are described later) instances involved. The continuous line on the other hand, means that we are sending and receiving these objects from other modules. For example, Repository will request `File` and `FileDistance` objects based on files' paths and File management will return the correct instances. Similarly, Repository will send new `File` instances for the Cache module to process, which will insert them into the cache itself.

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.

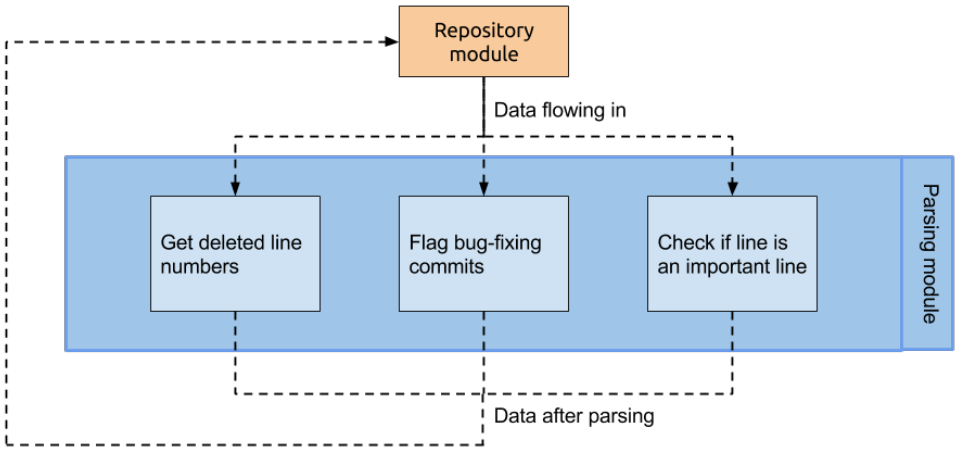

Figure 3.2: Data-flow through the Parsing module

Check if line is important

When identifying bug-introducing commits, we are using the SZZ algorithm[10] (explained in section 2.1.5), which has a step of finding which lines are contributing to the bug. This algorithm has been criticised by Kim et al.[9] as when identifying these lines, they are all treated equal. This will ultimately introduce plenty of false positive lines, that is lines which were changed during a bug-fix but had no contribution to the bug whatsoever.

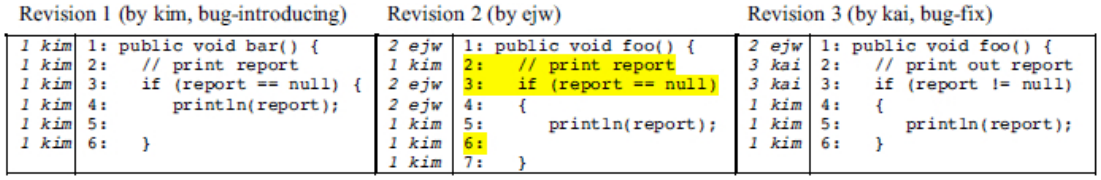

Figure 3.3: Example of changes between revisions (3rd fix is a bug-fix change). Figure taken from [9].

For example if we look at the figure 3.3 we can see three revisions. When we look at the bug-fixing revision (Revision 3) we can see that it changed lines #2 and #3 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[9], out of these we are only checking if a line is blank/empty or if it is a comment line. 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 approximating the true set would require more techniques mentioned by Kim et al., but getting the exact set is computationally impossible[9].

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 when identifying the

so-called "bug-introducing" commits with the SZZ algorithm.

Our module will accept the output of a `git diff` command for each file in the commit, and it will parser this output to get the line numbers which were deleted in that `git diff` (where the difference is what changed between the 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.[2]), we flag it as a bug-fixing commit.

Regular expressions used to parse commit messages:

- `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 (their 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 that what is their co-occurrence (which is reversely proportional to their distance) at any commit n. The figure 3.4 below shows a high level view of the File management module, and how data-flow through this module works.

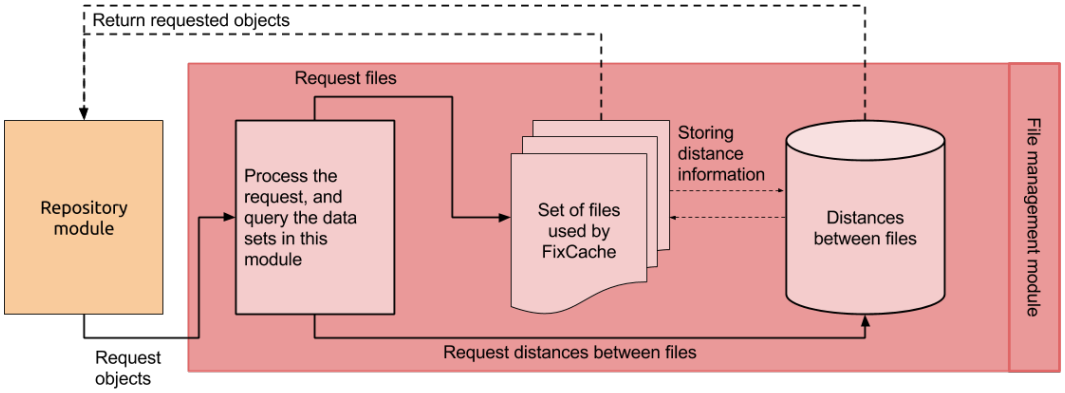

Figure 3.4: Data-flow through the File management module

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

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 the newly created file. We can think of this as a dictionary functionality implemented by File management, 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 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 our FixCache run we need to change or update the internal data of the file involved in the event so that it has 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 added 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 bfc, for each file involved we mark that there was a fault associated with that file, and that this was fixed at bfc. 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.Files`. 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.

When updating these objects after an event occurred we do not request every file, and do the update one-by-one. Rather, 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 the following:

```
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. 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 the files (changes, faults and LOC) in our File-set.: if they already existed update, if they did not create a new `File` instance. Furthermore we clean-up deleted files: we assume that each file is deleted when their LOC (lines-of-code) becomes zero.

After updating the data for each file and creating new files, it will return a list of tuples, where the second value of a tuple will be the actual `File` instance, and the first value will be either `'changed'` or `'deleted'` or `'created'` dependant on what happened to that file between the `diff`-ed commits we are viewing.

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 do 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 `Distance` object, or we create a new one if it has not yet existed.

The Repository module queries these `Distance` objects by a key generated from the two file paths of which the distance we are looking for (as explained in 3.2.2). Again, when a request is sent, either we return a already existing object, or we create a new one and return that. Whenever two files are committed together, the

Repository module will query the Distance-database to get the `Distance` object, and it will update the co-occurrence for that object.

The Distance-database is also important when determining which files are the closest ones to a given file f at a given commit c. In this case, the Repository 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 `File` objects (at most k number) which have the higher co-occurrence with the queried file.

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 take care of cleaning up some other 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, counting hits, counting misses and removing files (according to or 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.

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 files and remove them from the cache.

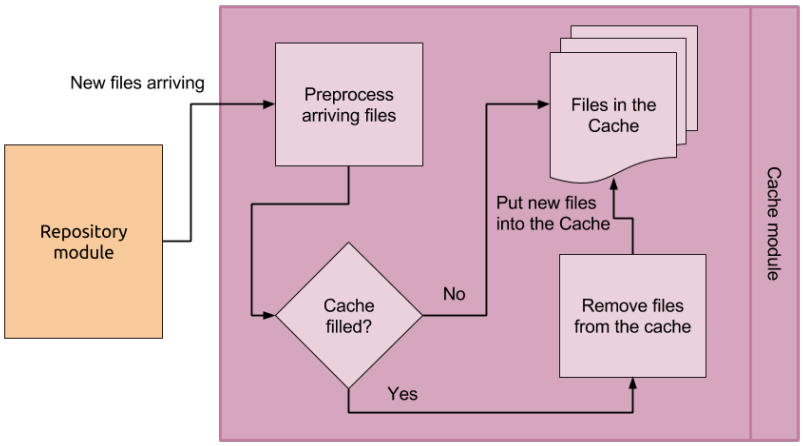

Figure 3.5: Adding new files to the Cache and processing them.

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 get it.

3.3 The Repository module

This module is the central element of the 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. That is for each fixing change, we want to identify the starting points of a bug fixed by that change.

3.3.2 Cache size initialisation

A key variable used by our algorithm is the cache-size. Upon each initialisation (or after reset) 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). 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. A strong assumption is made here, namely that the repository will always increase as history moves forward.

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 spacial-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 = 100, cache-ratio = 0.1 and distance-to-fetch = 0.5 then we will fetch 100∗0.1∗0.5 = 5 number of files.

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 parameters currently set to some value. However, before running the algorithm itself, we need to find the repository for which we want to run it on. 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 pre-defined 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.

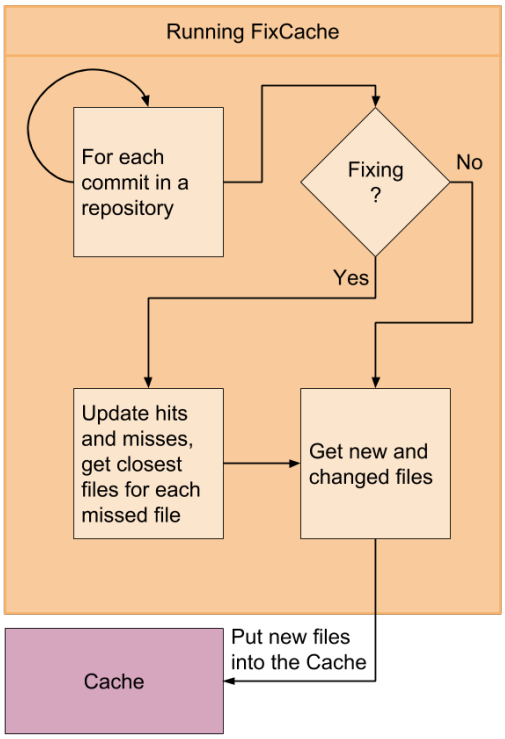

Figure 3.6: Control-flow of FixCache inside the Repository module.

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 all the other modules discussed earlier. This reset propagation will save memory, and also save some time as it is not necessary to redo the initialization step before each run.

Given all other 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 in the commit list do
    if commit has no parents then
        Pre-load the cache with some initial files, starting with ones with
        the highest LOC
    else if commit has one parent then
        if commit is a bug-fixing commit then
            for each file included in this commit do
                if file is in the cache then
                    Increase the number of hits
                else
                    Increase the number of misses and add file to the cache
                    Find when the bug-introducing changes for this file, and
                    put the closest-files at those commits to the cache
                end
            end
        Add some part of changed and created files at this commit to the
        cache.
    else
        This commit is a merging commit, so it may be disregarded.
    end
end
```

Algorithm 1: Pseudo-code for 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 merging commit, which we can disregard as the changes listed there already have been listed somewhere earlier in our commit history, as discussed in 2.2.4.

The second if-statement is when we are going through all files in our 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: $bci_1, bci_2 \ldots bci_n$

the bug-fixed in this commit, and add the closest files to f at $bci_1, bci_2 \dots bci_n$ (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 some numbers 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-recently-used vs. least-recently-used.

Version 5 and 6 only differ in that they use different object-database, as explained in 2.3.2, and version 5 is slightly faster.

As these versions are mostly only functionally incorrect, it still makes sense to compare them in terms of how fast they are relative to each other, as they have roughly the same number of computations. As it can be seen in the figure on page 34 there is a massive speed-up between version_1 and version_5.

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 have found that GitPython has generally good 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 this 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 33, which for large n 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

3.6.1 Alternative for sorting

To achieve speed-up we need to implement a "select top k elements from n objects", mentioned in the previous section. 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 the item onto the heap
        else if heap has k elements then
            if item bigger smaller 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.

We can see on figure 3.7 on the next page, how good is fast did it take for

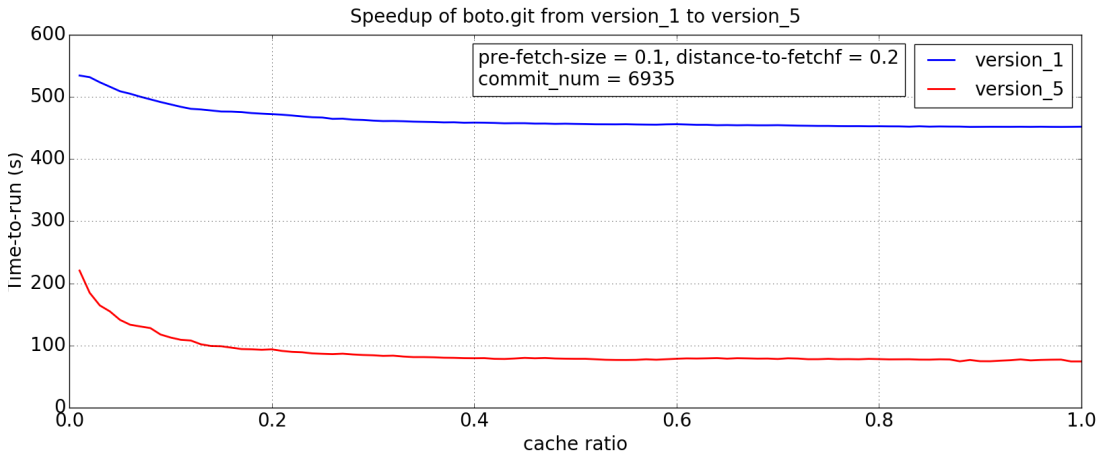

Figure 3.7: Speedup of boto.git and boto3.git between version_1 and version_5 with different cache-ratios

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. A reason for this speedup is that whenever we increase the cache-ratio we also increase the cache-size, and the bigger the cache size, the bigger the pre-fetch-size and distance-to-fetch, so we will fetch more files, hence our "get top k elemens from n objects" algorithm explained in 3.6 will run in $\mathcal{O}(1)$ time more often, as k (the number of files we are fetching) will increase with n unchanged.

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 37. 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.

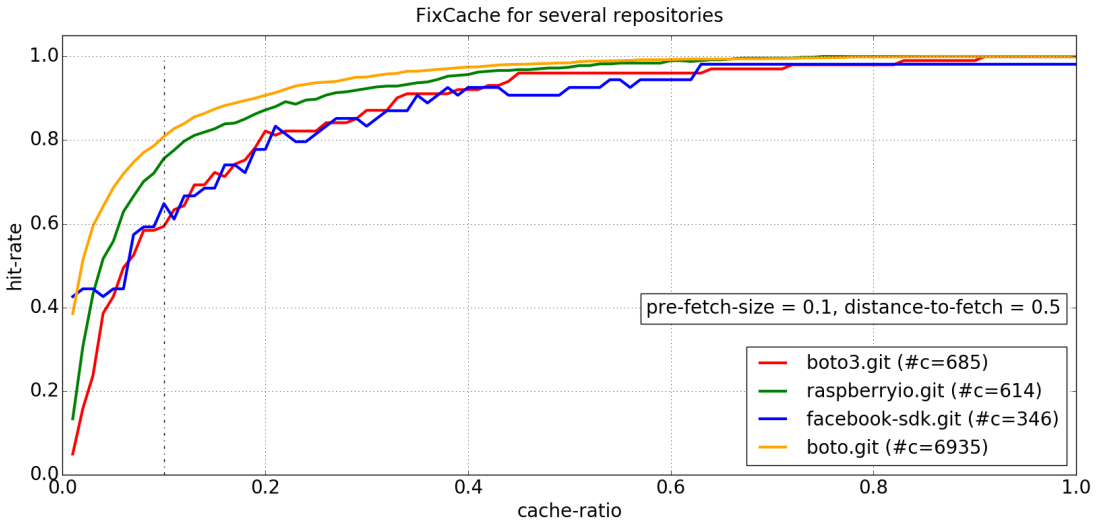

Figure 4.1: Evaluation of FixCache according to hit-rate for different repositories

What is the best cache size? From the figure 4.1 it can be deduced that the best hit-rate occurs around 0.2 of 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 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)

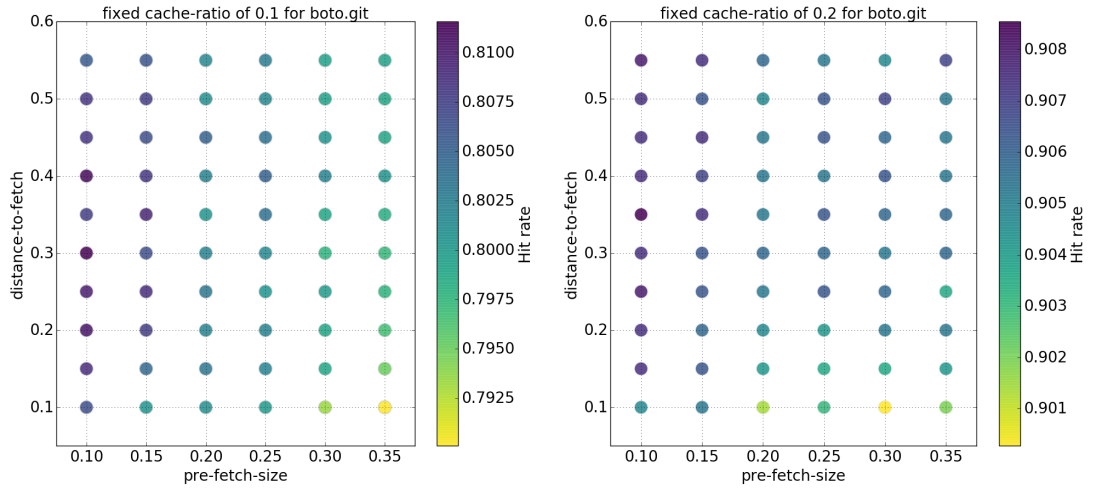

Figure 4.2: Fixed cache-ratio analysis for boto.git

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 not 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 [3]: 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.

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].

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 on the next page shows all the analyses for boto3.git and boto.git. The figure itself does not plot each and every analysis (taking all pre-fetch-size and distance-to-fetch values it would mean 15 curves), rather it plots vertical bars for each

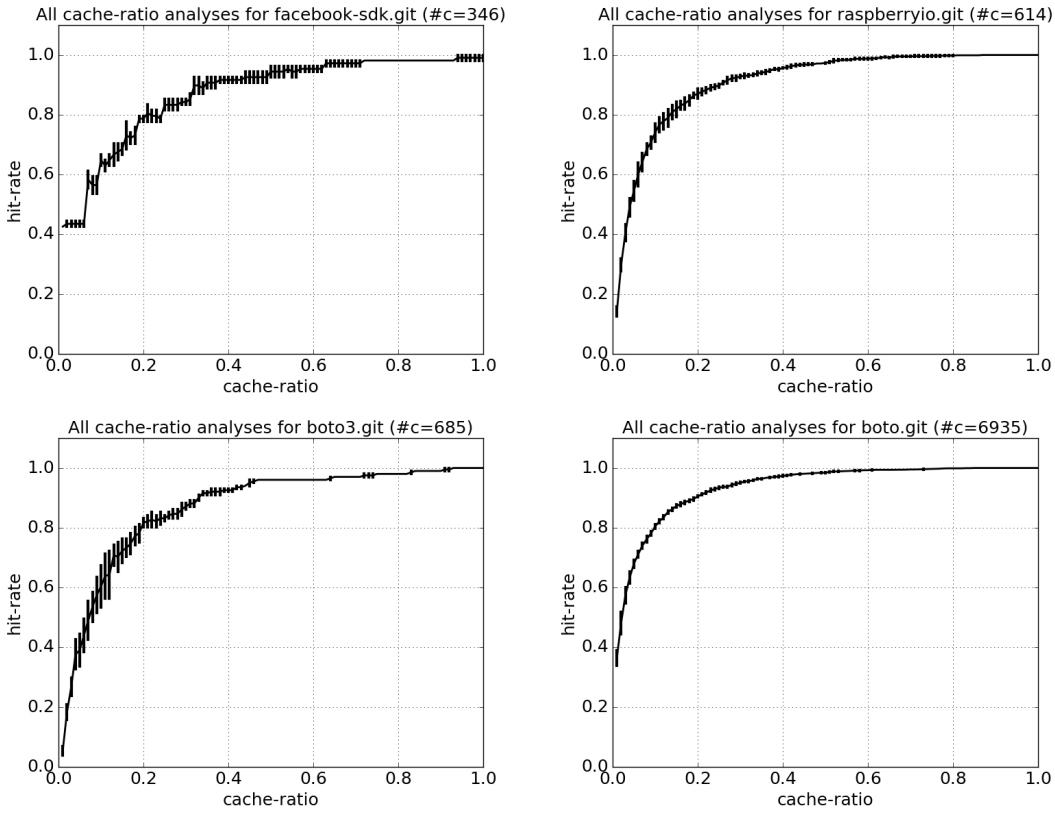

Figure 4.3: Showing all cache-ratio analyses for facebook-sdk.git, raspberryio.git, boto3.git and boto.git

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 beFor boto3.git it will be 0.158 at cache-ratio of 0.12, and for boto.git it will be 0.073 at cache-ratio of 0.02. Similarily, raspberryio.git and facebook-sdk.git would have also 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 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 forthese 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 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 oth-

ers. 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 41, 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 [3] 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.

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, and our horizon then is 0.1, and 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 **??** below, 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.

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

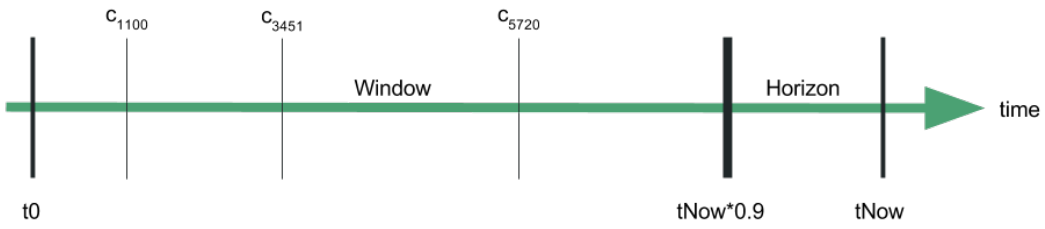

Figure 4.4: Window and horizon presented on a timeline of commits

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. The figure **??** on the right 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 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

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

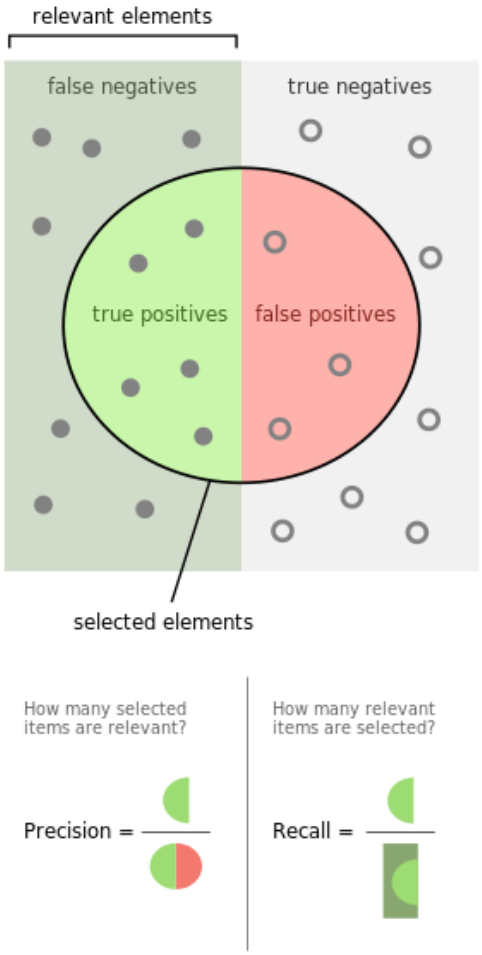

Figure 4.5: Precision and recall. Figure taken from Wikipedia[7].

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 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

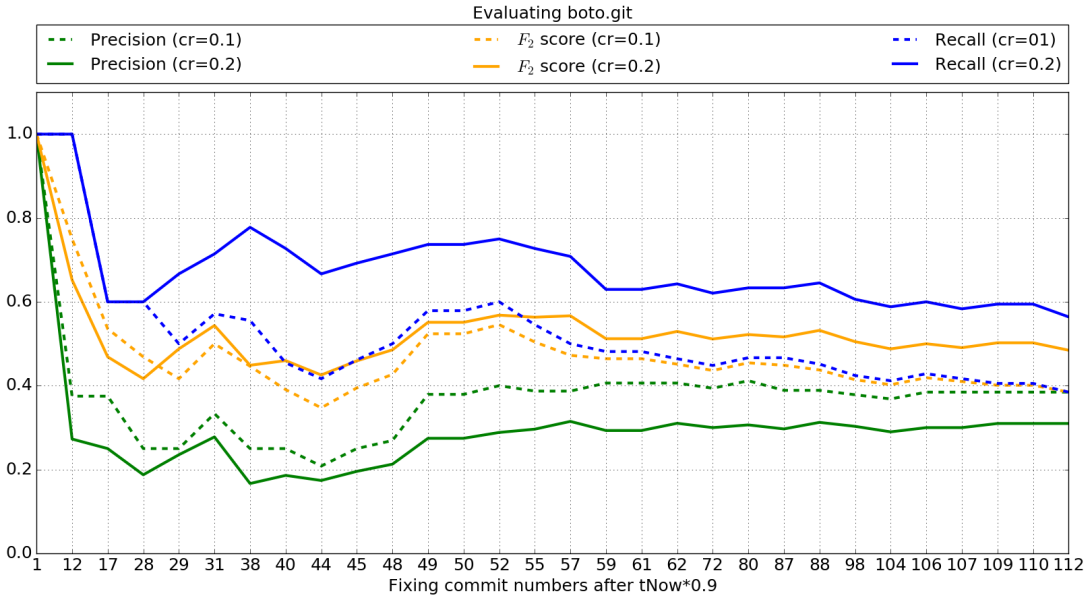

Figure 4.6: Evaluating boto.git with precision, recall and F_2 score for the first 30 commits in the horizon after tNow*0.9.

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

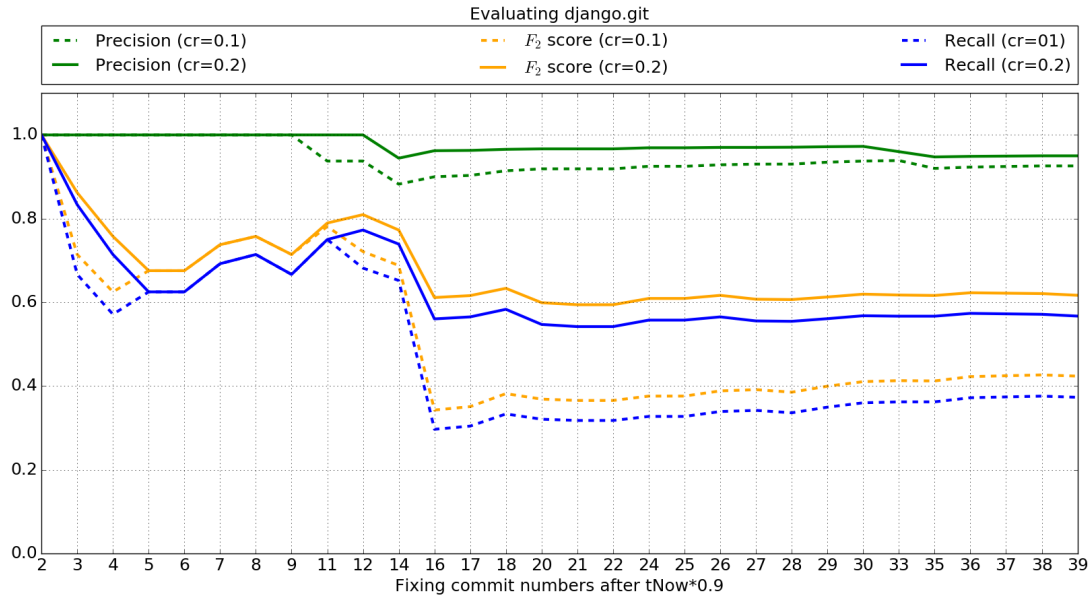

Figure 4.7: Evaluating django.git with precision, recall and F_2 score for the first 30 commits in the horizon after tNow*0.9.

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 several results were found to those of the original paper, however generally they were slightly lower on-average. The parameters 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, it's performance drops.

Chapter 5

Conclusion

5.1 Summary of work

This dissertation presents the work achieved by implementing and analysing FixCache on five open-source Python repositories on GitHub. The success criteria, as outlined in the project proposal was met, and all the components were completed on schedule. It was found, that Python repositories on GitHub perform slightly poorly in terms of FixCache's hit-rate than the repositories outlined by Sunghun 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 How to use FixCache

As it could be seen in section 4.4 in the previous chapter, FixCache works only for the few first commits in the future in terms of precision (django.git is an exception here, where the precision was nearly always 1.0), recall and F_2 score. This means, that FixCache should not be used alone when making long-term bug-predictions, rather it should be re-run after each fixing commit (or even after each commit) to always have an up-to-date prediction file-set.

When used for production, it would be best to use together with some other bug-prediction scheme, in order to maximise the number of correctly identified files. The algorithm itself 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 be only feasible to run it before deploying a huge change. As for the parameters, following the results identified I would recommend using a cache-ratio of 0.2, a pre-fetch-size of 0.1 and a distance-to-fetch of 0.5.

5.3 Further ideas

Further work could be done in getting even higher hit-rate, by doing a more sophisticated per line analysis of lines 'which truly contribute to a bug' as discussed by Sunghun et al.[9].

It would be also good to have an implementation of method-level FixCache for Git, in order to compare the results with what was found in that matter in the original paper.

Finally, some other properties could be investigated: bug-density on a per day basis (ie. more bug prone days); whether there is a connection between the number of contributions and the number of introduced bugs for each user; is there a difference between bug-proneness of Python repositories on GitHub and repositories in other highly used languages (C/C++, Java).

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] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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 *Pro-*

ceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories, MSR '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[9] 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.

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

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