Dissertation notes

April 5, 2016

Overview of FixCache algorithm

FixCache is a bug-prediction algorithm implemented in 2007 by researchers at MIT[**?**]. The original algorithm was implemented with the help of 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. 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 implements FixCache.

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. 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 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. Since the cache size is fixed throughout one run, behind the scenes, the cache implementation handles file removal. It will remove as many files as necessary (using the Least-Recently-Used replace policy) to keep the cache size bounded (by a fix value).

Git

Overview

Git 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 the main repository. This approach makes it easier to develop software when internet connectivity is poor, but introduces other issues, such as merge-conflicts. It was 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. It has a simple design, it is fully distributed and supports non-linear development through branching.

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. 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.

Git File structure

At each revision point in git we can think of data represented by git as a small file structure. Objects in a Git snapshot can either be blobs or trees. Blobs correspond to files whereas trees correspond to directories and subdirectories under the repository. At each point in the history of the 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, which is the root directory of the repository. Further, each tree can have a pointer to many more trees or blobs. Blobs, as they correspond to file, 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).

Branches in Git

Branches are an important feature of Git. 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 you create a branch, what really happens is that Git creates a new pointer to the commit you are viewing at that moment. Git knows which branch you are on at any time by storing a special pointer called "HEAD" which is a pointer to the branch object you are viewing. It is possible to diverge two branches, that is starting from a commit C, make some changes on branch B1, and later switch to branch B2 and make some changes again on top of C. This was the first commits after C on B1 and on B2 will have the same parent, C, and they will be diverged. 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 B1 to the changes made on B2. If B1 and B2 are on the same linear path, we can simply fast forward, that is point the pointer of B1 to the same place when B2 is pointing. Otherwise, we need a real merge, which can be tricky to do automatically. There are two cases here:

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 B1 and B2. 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, but I personally find them quite unintuitive to use.

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.

As each commit can be a parent of one or more commits and similarly each commit can have zero ore commits we can treat the history of a repository as a Directed Acyclic Graph (DAG). This is a big difference between other VCS softwares as in non-distributed VCS branching is harder to achieve, and once we have it it is quite hard to backtrack which branches were used by which user and when[need better explanation+reference here]. The 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.

It is also possible to view each branch as a linear set of commits (even when some branches were merged into this branch). Commits will appear in their order of committed time, regardless on which branch were they made. 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.

Git diff with and without the –stat flag

The command git diff outputs the difference between two snapshots (commits for instance). Without any flags/options set it will output the line-by-line difference with some metadata at the beginning for each file. If the task is only to get the basic information of how files changed it is good to use the –stat flag. With this Git will only outputs which file has had lines deleted and/or added, and how many lines this was. This second is more efficient in the background, due to how diff is implemented (reference here??).

Bibliography

[1] E. Whitehead Jr S. Kim, T. Zimmermann and A. Zeller. Predicting faults from cached history. *Proceedings of the 29th International Conference on Software Engineering*, pages 489 – 498, 2007.