An Empirical Analysis of the FixCache Algorithm

Caitlin Sadowski, Chris Lewis, Zhongpeng Lin, Xiaoyan Zhu*, E. James Whitehead, Jr.

Software Introspection Laboratory
University of California, Santa Cruz
Santa Cruz, CA 95060
{supertri, cflewis, linzhp, ejw}@soe.ucsc.edu

* Department of Computer Science and Technology,
Xi'an Jiaotong
University, Xi'an 710049, China
xyxyzh@gmail.com

ABSTRACT

The FixCache algorithm, introduced in 2007, effectively identifies files or methods which are likely to contain bugs by analyzing source control repository history. However, many open questions remain about the behaviour of this algorithm. What is the variation in the hit rate over time? How long do files stay in the cache? Do buggy files tend to stay buggy, or can they be redeemed? This paper analyzes the behaviour of the FixCache algorithm on four open source projects. FixCache hit rate is found to generally increase over time for three of the four projects; file duration in cache follows a Zipf distribution; and topmost bug-fixed files go through periods of greater and lesser stability over a project's history.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement – Restructuring, reverse engineering, and reengineering, D.2.8 [Software Engineering]: Metrics – Product metrics

General Terms

Algorithms, Measurement, Experimentation

Keywords

Software bug prediction, fixcache

1. INTRODUCTION

Today, software quality assurance is a reactive process. Software testers, developers, beta testers, and users encounter incorrect behavior in software, report it, and then fix the problem. Even though it is well known that nearly all software harbors latent—as yet undiscovered—software bugs, bug fixing activity is only initiated in response to a report of bad behavior. Ideally, we would like to proactively remove bugs from software, before they are encountered by users.

There are two main strategies for bug prediction. Code metrics-based approaches use software quality metrics such as LOC or cyclomatic complexity to predict faulty entities [3, 9]. Metrics-based approaches are easy to implement, but also tend to be relatively static and do not quickly adapt. In contrast, repository-based approaches, like FixCache, analyze the source control repository history to predict future faults [10,7].

Software bug prediction is becoming increasingly effective at identifying those areas of a software system that are highly likely to contain latent software bugs. With software bug prediction, it becomes possible to take action to remove bugs from software before these bugs are encountered by users. Such actions can include software inspections, code refactoring, running static analysis bug finding tools, and increasing code coverage within existing test suites. All of these techniques can uncover or remove bugs early, thereby increasing code quality and potentially reducing software quality assurance costs.

Most bug prediction algorithms described in the research literature are evaluated against a single moment in time of one or more software systems. Even bug prediction algorithms that use information from a project's history of changes typically only report bug prediction results at one specific point in time. This is a problem: for bug prediction to be effectively used in real-world software development settings, software engineers will need to perform software bug predictions repeatedly on the same project. Since the same bug prediction algorithm will be used at different points in time during a project's development, it is important to understand (1) how bug prediction algorithms accommodate the changes a software project experiences over time, and (2) what kind of variation can be expected in bug prediction accuracy.

This paper provides a preliminary exploration of the above questions by taking a detailed look at the performance of the FixCache bug prediction algorithm [8]. Briefly, FixCache is run over the change history of a software project, and provides as output a list of files that are predicted to hold the most latent bugs. The number of files reported is configurable, and is typically 10% of the total code files in a project. The original publication presenting FixCache reports the prediction results at one point in time, at the end of each project's mined software history. Subsequent to this original implementation, a different group of researchers independently implemented FixCache, and provided results on its prediction accuracy at weekly intervals over a two month period [15, 16]. The focus of this work was on leveraging FixCache recommendations to select regression tests. They found that FixCache performance varied over time, with considerable variation in performance from week to week. However, they evaluated only a single project, for a short time period, and reported at weekly intervals. Still, the results are interesting, and suggest that deeper investigation on larger projects and timespans would be useful. Members of the same research group further verified that FixCache can improve test-case recommendations [6]. They found that the most fault-prone module changed over time, and that faults are not always clustered in the same module.

In this paper we explore the following specific research questions:

RQ1. How does the prediction performance of the FixCache algorithm vary over time?

This information is very important for any software team seeking to adopt the FixCache bug prediction algorithm, since it provides in-

MSR 2011, May 21-22, 2011, Honolulu, Hawaii, USA.

Project	Language	Period	Number of Revisions	Number of Hunks	Appx. Number of Code Files*	Number of Buggy Revisions**
Apache httpd	C	1996-01-14 - 2011-01-21	38,230	582,262	1,233	5,848
PostgreSQL	C	1996-07-09 - 2011-01-08	36,848	1,760,057	2,725	13,176
V8	Javascript/C++	2008-06-30 - 2011-03-15	6,178	213,644	1,377	2,630
Voldemort	Java	2009-01-02 - 2011-01-24	2,292	41,540	935	511

Table 1. Analyzed projects.

* Code files are identified with a regular expression matching common source code file extensions, such as .c, .h and .java. The number given is the number of unique file names CVSanalY identified. Files with the same name across the directory structure will only count as one, so the number of files may be marginally higher. This number includes files that were deleted from the repository.

** The number of buggy revisions is a count of the unique revisions that are flagged as bug introducing. This means that there is a commit message which matches one of the following case-insensitive regular expressions: `defect(s)?, patch(ing|es|ed)?, bug(s|fix(es)?)?,(re)?fix(es|ed|ing|age|\s?up(s)?)?, debug(ged)?, \#\d+, back\s?out, revert(ing|ed)?` **or** the case-sensitive `[A-Z]+(-|#)\d+, CVE-\d+-\d+` , and the changes are mapped back to a number of bug introducing changes.

formation on the range of expected performance, as well as a sense of how stable the prediction accuracy is over time.

RQ2. How static is the set of most-buggy files? Can files go from being fault-prone to relatively fault-free?

Since the focus of activity in a software project changes over time as new features are added and others modified, it makes sense that the most buggy files will also vary over time. That is, we expect that the most buggy files will not remain constant over a project's history, but will instead vary over time. One characteristic of the FixCache algorithm is that it is capable of adapting to varying buggy hotspots in a project over time. We would like to characterize this adaptation.

The following sections describe the FixCache algorithm (Section 2) and mining approach (Section 3), followed by a brief description of related work (Section 4). We then present our findings (Section 5), and briefly mention some threats to validity (Section 6).

2. THE FIXCACHE ALGORITHM

The FixCache algorithm maintains a fixed-size "cache" of the entities that are most likely to have bugs. This algorithm works at either the file or method level; here we only focus on identification of fault-prone files. Addition of files to the cache is determined by three heuristics: new or modified files are likely to contain bugs, buggy files are likely to contain more bugs, and files which are changed with buggy files are likely to contain bugs.

The cache is built using the source control history data of a project, and is parameterized by the cache size, cache replacement strategy, "blocksize", and "prefetchsize" (described below). The cache is pre-loaded (filled) with the largest files in the initial revision. FixCache then cycles through commits sequentially ordered by date. Upon each commit, the FixCache algorithm adds up to prefetchsize new or modified files to the cache. We use the SZZ algorithm [14, 8] to identify bug-fixing commits, and trace bug fixing commits to the time when a bug was introduced. Every file which is modified during a bug-fixing commit is identified as containing a bug fix. When a bug fix is identified, the file itself will be put into the cache, plus up to blocksize of the files most frequently changed with the buggy file at the time the fault was introduced (co-changed files).

When a file containing a bug fix is added to the cache, FixCache checks to see if that file is already in the cache. If it is, FixCache counts it as a cache hit. Otherwise, FixCache counts it as a cache miss. The hit rate is equal to:

number of hits / (number of hits + number of misses)

The size of the cache can be set based on available resources, such as time available for bug removal activities. In this paper we set it to 10% of the number of code files in the repository at the end of the period analyzed. To keep the cache at a fixed size, before a file is put into the cache, we have to determine if the cache is full. If it is full, one file has to be moved out of the cache. We implemented four cache replacement policies: least recently used (LRU), the number of times a file has been changed, the number of bugs identified in a file, and the number of distinct authors for a file. All cache replacement policies use LRU to break ties. We only use the LRU cache replacement policy, since it performed better than the others. Cache size, blocksize, and prefetchsize are given in parenthesis in figure captions as "(cachesize/blocksize/prefetchsize)". Our implementation of the FixCache algorithm is freely available[1].

3. MINING APPROACH

We mined two large, long-lifetime projects: Apache httpd [1] and PostgreSQL [11], each of which appeared in the original FixCache paper [8]. We also mined two smaller projects: Voldemort [14] and V8 [17] (see Table 1). Accessing Git repositories significantly speeds up mining in comparison to accessing centralized approaches like CVS or SVN. For example, finding the number of lines of each file at each revision requires a call to the repository for every file at every commit. This process is expensive on a centralized source control management system, taking on the order of weeks for a large project. Rather than making a network request for data to a central server that is busy concurrently communicating with other clients, our mining code queries the local Git repository, and runs as quickly as the CPU, database, and local I/O can handle, reducing expensive multi-week queries to days or hours. Working mostly with projects whose repositories have been mirrored to Git allows us to benefit from this locality, though with the drawback of needing to handle the complexity of branching within Git [5].

[1] https://github.com/SoftwareIntrospectionLab/FixCache

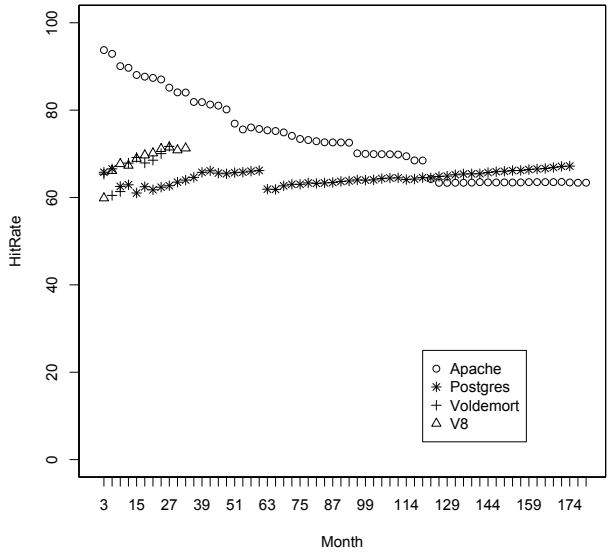

Figure 1. Hit rate vs time for all projects. Apache (55/18/7) PostgreSQL (177/100/20), Voldemort (74/21/15), V8 (123/20/45)

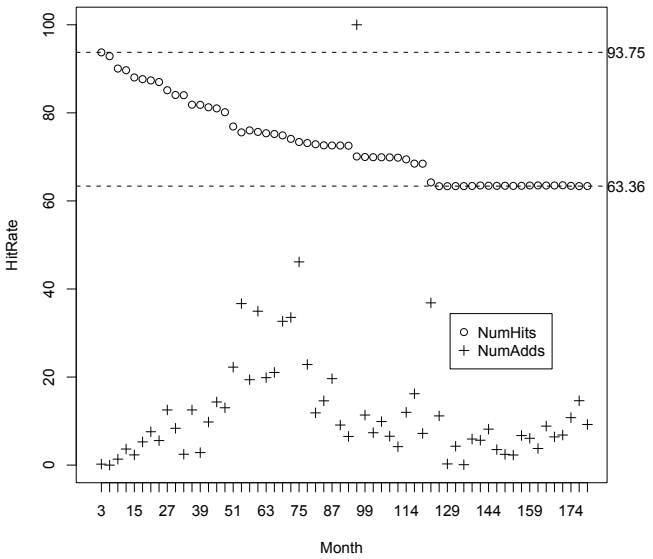

Figure 2. Hit rate vs time overlaid with number of files added to the cache (Apache httpd, 55/18/7).

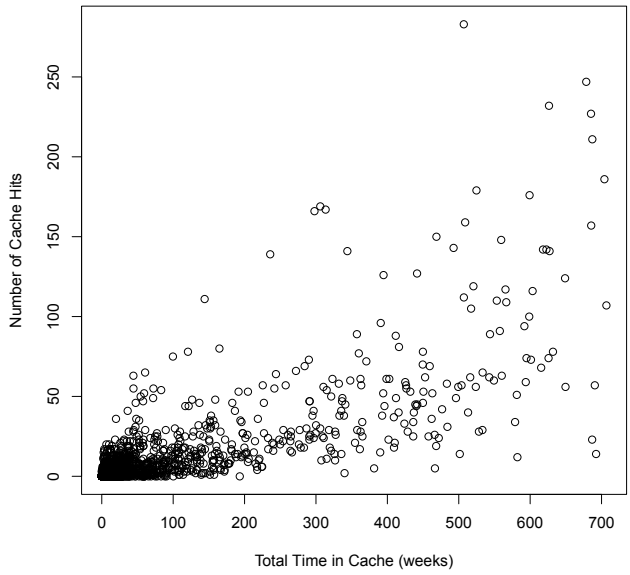

Figure 3. Total number of cache hits for a file vs total amount of time that file spent in the cache (PostgreSQL, 177/100/20).

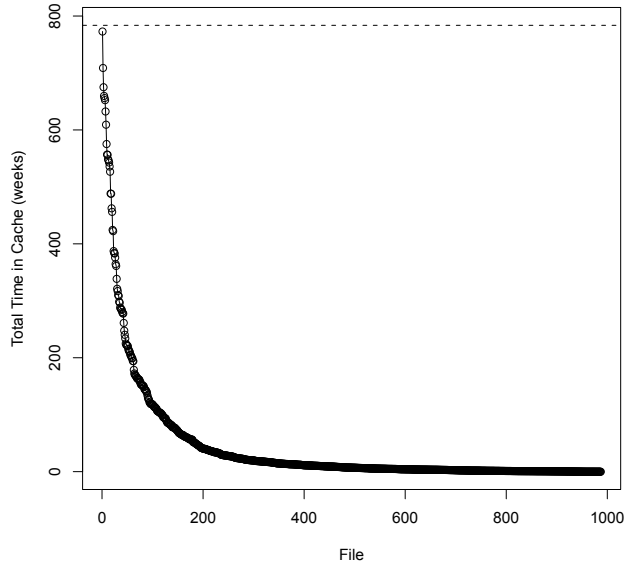

Figure 4. Distribution of the total amount of time each file spent in the cache (Apache htttpd, 55/18/7).

CVSAnalY [12] was used for history mining. We extended this tool[2] to gather the number of lines in each file revision, determine the code hunks modified at each commit, and link bug-fixing hunks back to the original bug-introducing commit using SZZ [13].

4. RESULTS

This section presents empirical results exploring the two research questions. The cache size for each project is 10% of the number of code files in the current project directory at the end of the analyzed period, calculated via a shell script. This worked out to be 75 for Voldemort, 177 for Postgres, 55 for Apache httpd and 123 for V8.

RQ1. How does the prediction performance of the FixCache algorithm vary over time?

[2] https://github.com/SoftwareIntrospectionLab/cvsanaly

Figure 1 presents the hit rate over the entire analyzed history for all four projects. There is a dot on the graph at each three month boundary. The hit rate is relatively stable over time, although each project has hit rate trends. We believe that the initial downward trend in the httpd project is caused by a much smaller amount of files at project start. Once the project ramps up to the current number of source code files, the hit rate stabilizes. Figure 2 is an in-depth look at how the hit rate changes over time for the httpd project. Overlaid with the changing hit rate is the number of files which are added to the cache within each time slice. The discontinuities in the hit rate curve appear to be the result of a sharp increase of additions.

Figure 3 shows the total duration of time spent in the cache for files (in terms of the source control repository timescale) vs. the number of hits for those files. As expected, the trend is generally upwards: staying in the cache for a long time is correlated with having lots

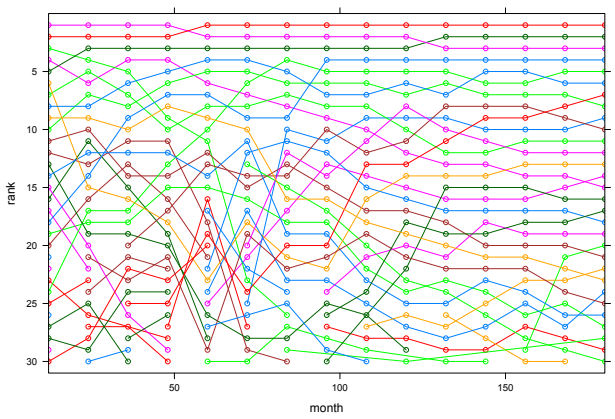

Figure 5. Top 32 most buggy files in Apache httpd (55/18/7)

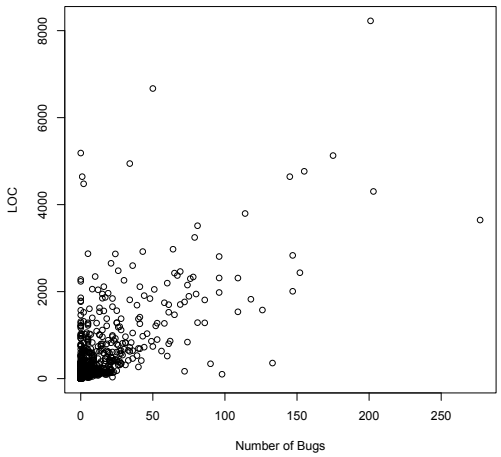

Figure 6. LOC vs number of bugs (Apache httpd, 55/18/7).

of hits. However, there are still several files which stay in the cache for long periods of time, but do not have a high hit count.

Figure 4 displays the distribution of total duration of files in the cache. The vertical line near the top of this graph marks the duration of the entire source control history. It is clear that most files spend a relatively short amount of total time in the cache, hence demonstrating that FixCache is adapting to changing conditions.

RQ2. How static is the set of most-buggy files? Can files go from being fault-prone to relatively fault-free?

Figure 5 shows the top 30 buggy files over time for the Apache project. For a particular time, a dot at position k represents the kth buggiest file. The buggiest files are at the top. As time moves from left to right, lines track how particular files shift in the ranking. It is evident that there is a set of core files which always rank high in terms of bug count. It is also evident that some files change their ranking, with periods of greater and lesser stability of rankings.

Figure 6 plots the maximum LOC for files against the number of bug fixing commits. Although larger files are more likely to contain bugs, it is clear that LOC alone is not a good predictor of fault density. This graph reinforces the dynamic nature of fault density in files—and highlights why FixCache is more effective than code metrics-based approaches.

5. THREATS TO VALIDITY

Bug Tracking. Many bug fixing commits cannot be identified as such from the commit logs, hence our results may be biased based on the type and severity of bugs identified [4], and this may reduce the number of faulty files identified by FixCache [2].

File Identity. Our current approach uses the last path segment in a file's path as a unique identifier for that file. This can cause multiple files with different pathnames but the same final path segment to be treated as the same file. In the projects examined, the set of such duplicate file segments was found to be very small.

6. CONCLUSION

In this short paper, we analyze the behaviour of the FixCache algorithm over time. We find that the FixCache hit rate is stable in the short term, but has definite long-term trends. We also find that the set of most-buggy files is not static, although there is a core set of files which consistently have the highest fault density.

REFERENCES

[1] Apache httpd. https://github.com/apache/httpd.

[2] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The missing links: Bugs and bug-fix commits. *FSE 2010*.

[3] V. Basili, L. Briand, W. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE TSE*, 22(10):751-761, 2002.

[4] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: Bias in bug-fix datasets. *FSE 2009*.

[5] C. Bird, P. Rigby, E. Barr, D. Hamilton, D. German, and P. Devanbu. The promises and perils of mining Git. *MSR 2009*.

[6] E. Engstrm, P. Runeson, and G. Wikstrand. An empirical evaluation of regression testing based on fix-cache recommendations. *Int'l Conf. Software Testing, Verification & Validation (ICST) 2010*.

[7] A.Hassan R.Holt. The top ten list: Dynamic fault prediction. *ICSM 2005*.

[8] S. Kim, T. Zimmermann, E. Whitehead Jr, and A. Zeller. Predicting faults from cached history. *ICSE 2007*.

[9] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. *ICSE 2006*.

[10] T. Ostrand, E. Weyuker, and R. Bell. Predicting the location and number of faults in large software systems. *IEEE TSE*, 31(4):340–355, 2005.

[11] Postgres. http://www.github.com/postgres/postgres.

[12] G. Robles, S. Koch, J. M. Gonzlez-Barahona, and J. Carlos. Remote analysis and measurement of libre software systems by means of the CVSAnalY tool. *RAMSS 2004*.

[13] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *MSR 2005*.

[14] Voldemort. https://github.com/voldemort/voldemort

[15] Z. Wang. Fix Cache Based Regression Test Selection. Master's thesis, Chalmers Univ. of Technology, Univ. of Gothenburg, 2010.

[16] G. Wikstrand, R. Feldt, J. Gorantla, W. Zhe, and C. White. Dynamic regression test selection based on a file cache an industrial evaluation. *ICST 2009*.

[17] V8. https://github.com/v8/v8