BugCache for Inspections : Hit or Miss?

Foyzur Rahman Daryl Posnett Abram Hindle Earl Barr Premkumar Devanbu

Department of Computer Science
University of California Davis, Davis, CA., USA
{mfrahman,dpposnett,ajhindle,etbarr,ptdevanbu}@ucdavis.edu

ABSTRACT

Inspection is a highly effective but costly technique for quality control. Most companies do not have the resources to inspect all the code; thus accurate defect prediction can help focus available inspection resources. BugCache is a simple, elegant, award-winning prediction scheme that "caches" files that are likely to contain defects [12]. In this paper, we evaluate the utility of BugCache as a tool for focusing inspection, we examine the assumptions underlying BugCache with the aim of improving it, and finally we compare it with a simple, standard bug-prediction technique. We find that BugCache is, in fact, useful for focusing inspection effort; but surprisingly, we find that its performance, when used for inspections, is not much better than a naive prediction model — *viz.*, a model that orders files in the system by their count of closed bugs and chooses enough files to capture 20% of the lines in the system.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Code Inspections and Walk-Throughs*

General Terms

Experimentation; Measurement; Reliability; Verification

Keywords

Empirical Software Engineering, Fault Prediction, Inspection

1. INTRODUCTION

The later a bug is discovered, the more expensive and difficult it is to resolve [1]. Deployed bugs are the worst of all, requiring costly, embarrassing, and labor-intensive software updates in the field and/or re-deployments. Thus stakeholders, such as managers and quality assurance people, worry most about faults that escape into released and deployed products. Quality control methods such as inspection and testing aim to detect faults prior to release. Unfortunately, code inspection and testing are costly in terms of time and manpower, so managers seek to optimize their effectiveness. Bug prediction has been suggested as a means to this end. If one could predict which entities — modules, files or lines of code — are bug-prone, quality control teams might focus their effort on inspecting those entities.

In their award-winning paper on *BugCache* [12], Kim *et al.* hypothesized that past knowledge of fault occurrence can optimize testing and inspection effort. Their proposed cache-inspired models depend on the exploitable "locality" of faults. They hypothesized three types of locality. First, faults are more likely to occur in recently added/changed entities (churn locality). Second, if an entity contains a fault, then it is likely to contain more faults (temporal locality). Finally, entities that are logically coupled (by co-changes) with other faulty entities are more likely to contain faults (spatial locality).

To exploit fault locality, Kim *et al.* uses two types of changes, *bug-introducing* changes and *bug-fixing* changes, along with change history, to identify a subset of entities that are highly likely to contain defects. They implemented two types of caches: BugCache, a theoretical model, and FixCache, a practical prediction model. BugCache contains faulty entities and is updated when a bug, or fault, is introduced into a file. In reality, bug-introductions usually occur unnoticed, (and hopefully discovered later!) so BugCache is a theoretical model that can't be used until the full project history is known. FixCache, on the other hand, is a practical realization of BugCache that doesn't need oracular knowledge.

The evaluation of FixCache drew our attention: we wanted to replicate it from a different perspective, with an emphasis on inspection. FixCache was originally evaluated using the measure of "hit rate", which is the cumulative percentage of the successful cache probes through a project history. For *regression testing*, this measure is effective: if FixCache accurately identifies the defect-prone files, one can run tests only on defect-prone, modified files. However, our concern is *inspection efficiency*. Kim *et al.* wrote:

> "A manager then can use the list for quality assurance — for example, she can *test or review* the entities in the bug cache with increased priority" [12, §6] (our italics).

Thus we evaluate FixCache from the point of view of a stakeholder involved in code inspection — "how much effort does FixCache actually save inspectors?"

Code inspectors would prefer to focus their effort on buggiest of entities in order to achieve a greatest possible payoff

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

of bug detection for a given investment in inspection time. Suppose, hypothetically, that FixCache simply loaded the largest files into the cache. In this case, a high hit rate would not necessarily translate into an effective inspection process: a great many lines of code might have to be inspected to find bugs. Likewise, loading smaller files might lead to fewer hits, but might result in efficient inspection that discovers more bugs per unit of work.

To this end, we *first* evaluate FixCache, not against hit rate (cache hits), but against defect density and the cost effectiveness of inspection. *Second*, we address the questions "What makes FixCache work?" and "Of the different cache update policies it uses, which is most effective?" *Third*, we investigate whether certain modifications to these cache update policies would improve FixCache's performance with respect to the inspection cost-effectiveness. *Finally*, we study whether FixCache outperforms, in terms of inspection cost effectiveness, a very simple, standard bug prediction model.

This paper mounts a replicated study of FixCache and makes the following contributions:

- We evaluate FixCache using defect density. *We find that, in fact,* FixCache *does indeed find more bug-dense entities.*
- We study how the different update and replacement policies applicable contribute to FixCache's bug prediction and inspection efficiency, using an inspection cost-effectiveness measure. *We find that there is not much difference in the performance of different update and replacement policies.*
- We compare FixCache with a simple prediction model, using the same inspection measure *and find that there is not much difference.*

2. THEORY

FixCache is easily described. To start, the cache is primed with a set of files, typically the largest files. Then, the FixCache algorithm monitors the commits to the system and updates its cache using a set of policies. A *hit* occurs when a file in the cache undergoes a bug fix; a *miss* occurs when a file outside the cache receives a bug fix. The cache's contents, at any one point in time, contains the target set of buggy files for testing or inspection. FixCache uses 3 update policies.

When a bug-fix is committed, FixCache immediately adds the modified files to its cache; this is the *temporal locality* policy. Next it retroactively finds and adds the file(s) from the commit(s) in which the bug was introduced, as well as the files that were most often committed together with these files at that time. This is the *spatial locality* policy. Finally, FixCache adds recently added/changed files to the cache. We call this the *churn locality* policy. All these files are added to the cache, if they are not already present, when a bug-fix is observed in the anticipatory belief that they will require fixes soon.

Kim *et al.* empirically evaluated FixCache using various cache sizes (most often 10% of the entities in the system) and found it very accurate (73%–95%) at identifying faulty files. They also evaluated FixCache's accuracy at method granularity and found its accuracy to fall within 46%–72%. In this paper we only evaluate FixCache at file granularity. Several papers [7, 19, 22, 24] have reported re-implementations of FixCache, for various applications; our implementation achieved performance levels similar to those reported in these earlier papers, roughly 60%–80% hit rate at file granularity, as we discuss below.

2.1 Cost-Effectiveness

Evidently, if we could inspect just 10% of the files in a project and find most of the project's defects, that would be terrific for the software industry. Unfortunately, unlike a hardware cache that contains fixed size cache lines, FixCache contains source code entities, such as files, that can vary greatly in size. Thus, if the files in the FixCache tend to be large and contain a disproportionately large portion of the project code, then focusing on the files in FixCache may not be very helpful for code inspection. So, first, we ask what proportion of the lines of code are actually contained in the files that reside in the cache.

> **Research Question 1:** What proportion of the total lines of code are in the FixCache?

Even if the entities in FixCache include a very large proportion of the lines of code in the project, they might still be well worth inspecting, if those lines contain proportionately more bugs (*viz.*, higher defect density) than the lines not in the cache. When this happens, prioritizing the cached files increases inspection effectiveness. If, however, cached files turn out to have lower defect density than files that are not cached, then prioritizing files in FixCache for inspection is unwarranted. This leads us to our next research question.

> **Research Question 2:** Do the cached files have higher defect density (defects per lines of code) than files not in the cache?

Answering this question will allow us to measure the efficacy of FixCache. While this gives us a general idea of the possible benefits of using FixCache, it does not tell us why it actually works (if it does). A good understanding of the mechanics of FixCache may enable us to customize or enhance it. Thus arises the imperative to desconstruct the various "locality-based" update policies used by FixCache, and evaluate their merits.

2.2 What Makes FixCache Tick?

The operation of FixCache is predicated upon several update policies. How do these update policies affect the performance of FixCache? There are four sources of cache updates:

1. "Initial pre-fetch": Files added to the cache during cache initialization (following Kim *et al.* [12]).
2. "Temporal locality": Files added to the cache because they were changed to fix bugs.
3. "Spatial locality": Files co-changed with a buggy file, *viz.,* files that co-occur in many transactions where the buggy file was changed.
4. "Churn locality": Files that have recently been changed or added.

Of these four, we consider the first two to reflect core aspects of any cache, including FixCache. First, we must initialize the cache with some files, or else the initial hit rate would be low. Second, temporal locality adds a file to the cache, if it is not already present, when a defect is found in it. The other two sources, spatial locality and churn locality, provide additional policies for including files into the cache; it is these two policies that we evaluate empirically.

Research Question 3: How do temporal and spatial locality policies affect FixCache performance?

With a detailed understanding of the relative influence of different FixCache update policies, future research may explore customizing FixCache to attain a different set of performance goals. For example, while Kim *et al.* used the "hit rate" measure to evaluate performance, one might, as we do, choose other measures, such as density or cost-effectiveness, a measure developed by Arisholm, Briand & Johannessen [3].

Research Question 4: Can we improve FixCache to achieve better defect density?

This question raises additional interesting questions. What is the right way to optimize cache performance? Surely it will depend on a project's requirements and resource availability. Kim *et al.* measured FixCache performance in terms of predictive accuracy. Is this sufficient?

2.3 Evaluating Prediction Performance

Kim *et al.* use the measure "hit rate" to evaluate FixCache. "Hit rate" is the cumulative percentage of successful cache probes at the end of a run through a project's history. At any point in time, the cache may contain both defective and non-defective files. Only files that actually contain defects score hits, so the number of defective files in the cache bears some relation to the hit rate. FixCache considers files in the cache to be defective and those not in the cache to be non-defective. Consequently, the cache contains both *true positives* (TP), files that actually contain defects, and *false positives* (FP). The files not in the cache comprise the *true negatives* (TN), non-defective files, and *false negatives* (FN), defective files that are incorrectly classified. *Recall* measures the ratio of retrieved to relevant files:

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{TP}{\text{Defective}}.$$

No matter how many false positives are in the cache, recall remains unaffected. In fact, we can achieve perfect recall simply by enlarging the cache to hold all files. Recall, however, only tells half the story of a successful prediction algorithm. We are also interested in the ratio of relevant files to those retrieved. If only 10% of the files are defective, then retrieving 100% of the files, which would naively yield perfect recall, is an inefficient way to identify the defective files and demonstrates the limitations of using recall in isolation. Thus, recall is usually paired with another measure that penalizes an algorithm for falsely predicting too many entities as defective. *Precision* is the ratio of correct predictions to the total predicted as defective

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{TP}{\text{InCache}}.$$

When applied in this setting, however, both precision and recall treat all files equally. Effectively, they are indifferent to the number of *lines of code* (LOC) in the files being flagged as defective. When using a prediction tool as a guide for inspection, LOC is critical; smaller, buggy files may well have higher defect density, and consequently may yield a better return on inspection investment. When comparing different cache update policies, or different prediction models that select different numbers of files (and varying amounts of code) for inspection, we adopt a *cost-effectiveness* measure, specifically the area under the cost-effectiveness curve, or AUCEC. *Readers unfamiliar with this rather subtle measure may wish to read Section 2.4 below, before returning here.*

We apply these concepts to the evaluation of FixCache in order to build a more complete picture of its performance in an inspection setting. The extent to which FixCache exceeds the performance of standard models could suggest approaches for improving standard prediction models intended for use in inspection; perhaps an update policy, such as spatial locality, can be fruitfully incorporated into a traditional model. For this reason and to asses its efficacy, we use AUCEC to compare FixCache against a vanilla prediction model.

The FixCache algorithm imposes only a gross ordering of the files to be inspected, *viz.*, those files within the cache are to be inspected prior to those without. The calculation of cost effectiveness requires that we impose an order on the files within the cache so that we do not artificially penalize the FixCache algorithm in our evaluation.

To evaluate FixCache using AUCEC, we impose the same order on the files within the cache and those not in the cache — first by decreasing number of closed bugs, then by increasing size. This means that the smaller files with many closed bugs bubble up to the top.

This approach evaluates the cost-effectiveness of FixCache fairly with our naive algorithms: we found that this ordering provided the best performance for FixCache in every setting when compared with other simple orderings, *e.g.* density, file size, or previous defects.

Research Question 5: Does FixCache outperform standard, simple prediction models, when used for inspection?

2.4 Cost-Effectiveness Curve

Here, we briefly define cost-effectiveness, then the area under the CE curve (AUCEC), measures which we use to investigate RQ5. If you are familiar with these measures, please feel free to skip this section.

Suppose defects are uniformly distributed throughout the source code. Then, if an inspection budget allows inspection of 10% of the source code, we might choose 10% of the lines at random and reasonably expect to find about 10% of the defects. This scheme requires no work or expertise in data gathering and defect modeling. It is therefore reasonable to expect that any useful defect prediction method should be able to improve on this result. This is the basis for the cost effectiveness (CE) metric defined by Arisholm *et al.* [2] to investigate defects in telecommunications software.

Suppose that source code is selected for inspection using a prediction algorithm that orders code entities (*e.g.,* files) in terms of predicted defectiveness. If the algorithm is good and the system has only a few defects, or well-clustered defects,

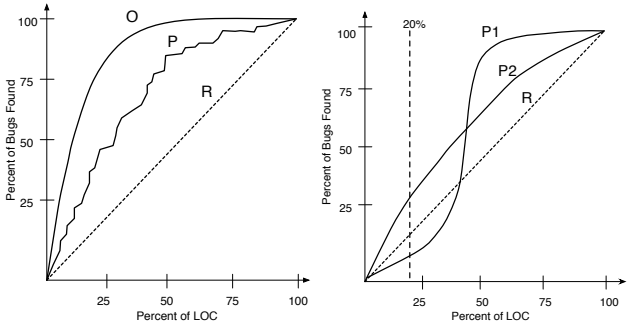

Figure 1: Cost Effectiveness Curve. On the left, O is the optimal, R is random, and P is a possible, *practical*, predictor model. On the right, we have two different models P1 and P2, with the same overall performance, but P2 is better when inspecting 20% of the lines or less.

.

this procedure would allow us to inspect just a few lines of code to find most of the defects. If the algorithm performs poorly and/or the defects are uniformly distributed in the code, we would expect to inspect most of the lines before we find all the defects. The CE curve (see Figure 1, left side) is a harsh but meticulous judge of prediction algorithms; it plots the proportion of identified faults (a number between 0 and 1) found against the proportion of the lines of code (also between 0 and 1) inspected. It is a way to evaluate a proposed inspection ordering. With a random ordering (the curve labeled R on the left side of Figure 1) and/or defects uniformly scattered throughout the code, the CE curve is the diagonal $y = x$ line. At file-granularity, the optimal ordering (labeled O in the Figure 1 left) places the smallest, most defective files first, and the curve climbs rapidly, quickly rising well above the $y = x$ line. A practical algorithm has a CE curve (labeled P in the Figure 1 left) that falls below the optimal ordering, but remains usefully above the $y = x$ line.

The CE curve represents a variety of operating choices: one can choose an operating point along this curve, inspect more or fewer lines of code and find more or fewer defects. Thus, to jointly capture the entire set of choices afforded by a particular prediction algorithm, one typically uses the *area under the CE curve*, or AUCEC, which is also a number between 0 and 1. An imaginary, utterly fantastical, prediction algorithm will have an area very close to 1, *viz.*, by ordering the *lines* so that one can discover all the defects by inspecting a single line; a superb algorithm will have a AUCEC value close to the optimal. Values of AUCEC below 1/2 indicate a poor algorithm. Thus, useful values of AUCEC lie between 1/2 and the optimum.

Consider two different prediction models with nearly identical AUCEC. On the right side of Figure 1, the two curves labeled P1 and P2 have very similar AUCEC values. However, if one were inspecting 20% of the lines or less, P2 offers a better set of operating points. This line budget is indeed quite realistic: inspecting 20% of the LOC in the system is definitely more realistic than inspecting all of it. Thus, the AUCEC, cut off at 20%, is a useful measure of the cost-effectiveness of a prediction model; this is what we use. Arisholm *et al.* [3] refer to a similar notion of AUCEC, conditioned on choosing 20% of the system. As it turns out, the preferred FIXCACHE cache size setting of 10% of the overall number of files in the system, typically results in a cache that contains close to 20% *of the LOC in the system*. So we adopt AUCEC at 20% of the LOC, which we refer to as AUCEC_{20}.

3. EXPERIMENTAL FRAMEWORK

In this section, we define some terminology and present our experimental setup.

3.1 Revision

Source code management (SCM) systems provide a rich version history of software projects. This history includes all commits to every file. These commits have various attributes such as a timestamp, authorship, change content, and commit log message. In our study, a commit, or a revision, consists of an author, a timestamp and a set of files changed. We chose GIT as our version control system, because of its excellent ability to track changes and find the origin of each line of code.

3.2 Bug-fixing Revision

Bugs are discovered and usually recorded into an issue-tracking system such as Bugzilla and subsequently rejected or fixed by the developers. Each bug report records its opening date, the date of the bug fix, a free-form textual bug description, and the final, triaged Bugzilla severity. For this research, we consider any severity other than an `enhancement` to be a bug.

Our study begins with links between Bugzilla bugs and the specific revision that fixes the bug — we call this a *bug-fixing revision*. We employed various heuristics to derive our data. We scan for keywords, such as "bug", "fixed", *etc.*, in the SCM commit log to flag bug-fixing revisions [16]. Also, numerical bug identifiers, mentioned in the commit log, are linked back to the issue tracking system's identifiers [8, 21]. Then these identifiers are crosschecked against the issue tracking system to see whether such an issue identifier exists and whether its status changed after the bug-fixing commit. Finally, we manually inspect the links to remove as many spurious links as possible. Each remaining *linked bug* has a bug-fixing revision. We gratefully acknowledge bug data we obtained from Bachmann *et al.* [4].

3.3 Bug-Introducing Change

We call the lines of code that are associated with the changes that trigger a bug fix "fix-inducing code", following Sliwerski *et al.* [20] (also see [11]), as it is the code that needed repair. For example, if `strcpy (str2,str1);` were changed to `strncpy(str2,str1,n);` then the original line is considered fix-inducing code. New lines may also be added in the bug-fixing revision, but we do not consider these "implicated", since they are part of the treatment, not the symptom. Kim *et al.* uses the term bug-introducing change to describe such fix-inducing code.

We use the popular Sliwerski-Zimmerman-Zeller *SZZ* [20] approach to identify bug-introducing changes. We start with data that links a bug to the revision where that bug was fixed. If a bug fix is linked to revision $n + 1$, then n, the immediately preceding revision, contains the relevant buggy code. The `diff` of revision n and $n + 1$ of each file changed in revision $n + 1$ gives us the potentially buggy code. We call these lines the fix-inducing code. We then use the `git blame` command on the fix-inducing code; `git blame` produces accurate provenance annotations (author, date,

revision number where they were last changed) for each line. With this information, we use the SZZ approach to rule out lines that could not have been involved in the defect, because of their time of introduction. For the rest, this approach identifies the commit that introduced the fix-inducing line.

3.4 Defect Density

We calculate two types of defect density: defect density based on open bugs and defect density based on closed bugs. At time t, we identify number of *closed bugs* of a file up to time t and divide that by the size of the file to calculate *closed bug density*. At time t, we identify *open bugs* as those reported before t and closed after t. *Open bug density* is the count of open bugs divided by file size. Note we use only the bugs that are linked to a file, since we cannot in general identify the culpable file of an unlinked bug. By using this density measure, we are assuming that files with higher *open* bug density are more profitable to inspect.

3.5 FixCache

We faithfully re-implemented FixCache as described in [12]. Following earlier evaluation studies [19, 22, 24] of FixCache, we implemented file-level FixCache, *i.e.* our implementation aims to predict defect-prone files.

Before running FixCache, for efficiency, we first identify all the bug-fixing revisions and chronologically order them. For each of these bug-fixing revisions, we also use the SZZ technique to identify the revision of the bug-introducing change. Now we are ready to run FixCache. We "prime" the cache with the largest files of a project during the *initial pre-fetch*, then scan its history. At every bug-fix revision, we probe the cache to see if the file being fixed is in the cache; if so, we score a hit, otherwise, we score a miss. If a probe results in a cache miss, we "fetch" that buggy file into the cache, and additionally determine all the spatially related files (files that were modified frequently with the buggy file in question). As described by Kim *et al.*, we go back in time to when a bug was introduced using SZZ, and determine *spatially local* files, files that were co-committed with the buggy file being probed at that time. We limit the fetch size of *spatially local* files to the block size parameter [12], prioritized by the frequency of co-commit (more often co-committed files get higher priority).

Regardless of cache hit or miss, we also fetch the recently changed/added files (files that were changed/added between two bug-fix revisions), subject to the pre-fetch size parameter [12].

As new files are added into the cache, we may need to evict old files. Following Kim *et al.*, we used two eviction policies — least recently used (LRU), to retain files recently involved in a bug-fix, and closed bugs, to retain files with the highest number of closed bugs. We did not implement the CHANGE [12] eviction policy as Kim *et al.* reported that it performs poorly.

The original FixCache presented results for 10% cache size. Prior evaluations [19, 22, 24] of FixCache also rely on 10% cache size. To ensure the sensitivity of our analysis, we experimented with several different cache sizes. We worked with 6, 10, 18 and 25% cache size. For these cache sizes, we set the block size to 2, 5, 9 and 13% respectively and pre-fetch size to 1, 1, 2 and 3% respectively.

3.6 Snapshot

As we described earlier, after each bug-fixing revision, we update the cached files. After updating the cache, we determine the state of the system, in particular the cached and uncached files. We also capture other system details such as the cache update source, the last hit time, *etc.* For each cached file, we determine its number of closed bugs, its size, *etc.* We call the overall system state after each bug-fixing revision a *snapshot* of the system.

4. THE DATA SETS

We chose 5 different medium to large-sized open-source projects for our study. All have long development histories, but are otherwise quite different. *Apache Httpd* is a widely used open source web server. *Gimp* is a popular open source image manipulation program. *Nautilus* is the default file manager for the Gnome desktop. *Evolution* is the default email client for the Gnome desktop with support for integrated mail, address book and calender functionality. *Lucene* is the most popular text search engine library. Except Lucene, all of our projects are written in C. Lucene is written in Java. We converted the Apache subversion repository to GIT and used the other projects' GIT repositories directly.

These systems span varied application domains: clients and servers, GUI applications, a file browsers, and libraries for text searching. All are of substantial size and complexity. Table 1 summarizes some descriptive statistics for each project studied. Project size ranges from 191k lines to about 800k lines. The table also presents the study period, the number of buggy files (which represents the number of cache probes made), the average number of source files (.c, .cpp, .h, or .java files) in bug fixing revisions, the average lines of code (as measured by `wc -l` Unix command) in bug fixing revisions, the number of commits made during the study period, the number of bug fixing commit made during this period and the number of bug-introducing commits (commits that introduced/changed some code that was changed/deleted later to fix some bug).

We determine all linked bugs and their associated bug-fixing revisions for all the subject projects. The corresponding bug-fixing change is found by using `diff` between a bug-fixing revision and its immediate preceding revision. After collecting all bug-fixing changes, we use GIT BLAME on each line of the bug-fixing change to identify its *last* modification time. This gives us all of the bug-introducing changes.

5. EVALUATION

We now present our results per each research question presented in Section 2. We only present plots from Apache and Lucene. Data from other projects is discussed in the text.

RQ1: *What is the line coverage of* FixCache*?*

Does FixCache achieve good hit-rates by simply caching a disproportionate fraction of the total LOC in the system? To check this, we plot the cached files' total LOC count in Figure 2 for 10% cache size and LRU based replacement policy. The time-series line plot shows that the FixCache caches a higher proportion of total line count when compared to the proportion of total file count within the cache. We found similar results for all projects. Moreover, we ran a sensitivity analysis by choosing the cache size to be 6, 10, 18

Name	Start Date	End Date	Number of Buggy Files	Avg. Number of Files	Avg. Number of LOC	Number of Commits	Number of Fix Commits	Number of Fix Inducing Commits
Apache	1996-07-03	2009-04-21	610	313	191303	18456	372	1108
Gimp	1997-01-01	2009-08-08	5291	2108	784529	25583	1727	6042
Nautilus	1997-01-02	2009-08-06	1235	312	136736	13462	656	1608
Evolution	1998-01-12	2009-08-08	3980	1146	432498	30528	1363	3733
Lucene	2001-09-11	2010-06-17	2453	1181	236672	5182	683	2369

Table 1: Summary of study subjects

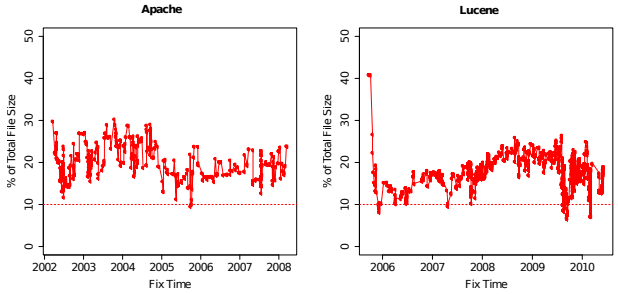

Figure 2: LOC in cached files for Apache and Lucene. If FixCache does not favor any particular size of file, then the dashed flat-line represents the *expected* fraction of total project line count in the cache for a 10% cache size. Time-varying line is the fraction of total project line count in the cache

and 25% of the overall files and using both LRU and Closed Bugs replacement policy. In all cases the FixCache retained a significantly greater (double to triple) proportion of lines, in comparison to the proportion of the total number of files held in the cache. This fact could undermine the efficacy of FixCache, when used to prioritize files for inspection.

These results suggest that the performance of FixCache, when used as an aid for targeting inspections, is not necessarily as good as suggested by the original hit rate measure. For inspection use, we need to consider file size to build a clearer picture of FixCache performance. We do this by comparing the defect density of cached files with that of the un-cached files.

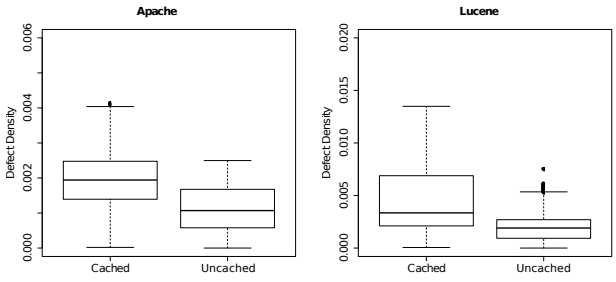

Figure 3: Defect Density boxplot in cached and un-cached files (10% cache size, LRU replacement) for Apache and Lucene.

RQ2a: *Do the cached files have higher defect density than files not in cache?*

Figure 3 shows the boxplot of defect density in cached

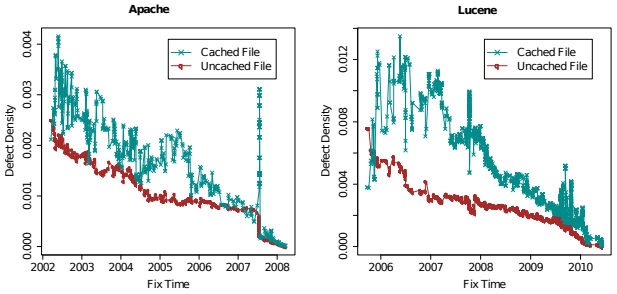

Figure 4: Defect Density in cached and un-cached files (10% cache size, LRU replacement) for Apache and Lucene.

and un-cached files for 10% cache size and LRU replacement policy. Clearly, files in the FixCache have higher defect density. We also ran an one sided paired Wilcoxon signed rank test with a *null hypothesis*: defect density in cached files is no greater than defect density in un-cached files. We found that the FixCache does indeed contain files with greater density (all the p-values were very small ($<< 0.001$)).

We also did a sensitivity analysis for different cache sizes. We found that the above result holds for all projects, for all cache sizes. Moreover, we found that the effect size varies little over all cache sizes for a given project.

Figure 4 shows the time-series line plot of density in cached and un-cached files (again for 10% cache size and LRU replacement policy). This figure highlights the right-censoring [14] effect in our data. We use bug fix information to determine opened/closed bugs for a given sets of files. We use our (*a posteriori*) knowledge of bug fixes to calculate the density of open bugs in the code at any given point in time. As we near the end of our data collection phase, we have fewer fixes; this gradually reduces the number of known, open bugs to zero. The cache, consequently, suffers from a diminishing defect density at the end of the run, which is really an artifact of right-censoring, not an indication that the performance of FixCache is getting worse.

There is a noteworthy, sudden spike in FixCache defect-density and hit rate in July, 2007 for Apache. We did a case study to understand this event. It turned out that on July 20, 2007, 15 distinct files were linked to 7 distinct bugs, resulting in a total of $15 * 7 = 105$ cache probes. This rapid succession of defects in the same files induced a a dramatic boost in the hit rate. Moreover, since each of these files had a large number of opened bugs, we also observed a higher cache defect density at that point. We observed similar (though less pronounced) sudden hit rate spikes for other projects as well. This suggests that FixCache may yield a higher

return in response to certain events, while performing less well at other times.

Though files resident in FixCache have higher density, one might argue that inspecting the entire set of files in the cache during each bug fix commit would induce redundant inspection effort, since some files would be inspected repeatedly. Rather, we could prefer to inspect files that are added or changed (we refer to such files as *churned files*) since the last inspection (the immediate prior bug fix). Thus we can ask, how well does FixCache work if we only inspect the churned files? This immediately raises the question of FixCache efficacy when considering only churned files. We therefore compare defect density of churned cached files against the defect density of churned un-cached files.

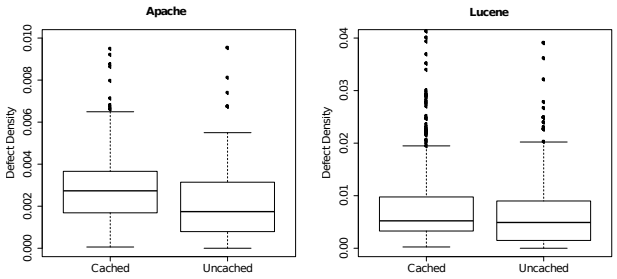

Figure 5: Defect Density boxplot in cached and un-cached *churned* (newly added or changed) files (10% cache size, LRU replacement) for Apache and Lucene.

RQ2b: *Do the churned files in the cache have higher defect density than churned files not in the cache?*

Figure 5 shows the boxplot of the calculated defect density of cached and un-cached churned files at 10% cache size for the LRU replacement policy. Not only the effect size dramatically diminished, it actually flipped for some projects, under some parameter settings. The Evolution project shows a lower defect density of cached files even at the typical 10% cache size. Moreover, Nautilus also shows lower defect density of cached files at all but the 10 and 18% cache size. Effect size is very small across the board; even so, some projects, for some parameter settings, show a statistically significant greater defect density for churned files.

Next, we evaluate the effect of each cache-replacement policy on the defect density achieved in FixCache.

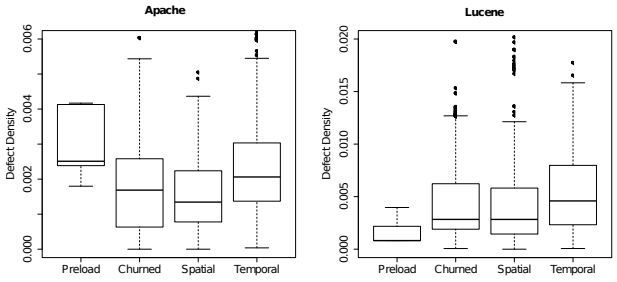

Figure 6: Defect Density by loading reason in cached and un-cached files for Apache and Lucene.

RQ3: *Do different sources of cache updates have different levels of influence on cache performance?*

Figure 6 shows the defect density as contributed by different sources of cache updates.

As is apparent from the figure, temporal locality is the highest contributor of defect density. We disregard the initial preloading of large files as we observe that, across all projects, they are evicted from the cache early in the run. Wilcoxon signed rank tests were used to check whether temporal locality consistently yields greater defect density across all cache sizes. We did find that temporal locality, for various cache sizes, and across all projects, exhibits higher defect density than spatial locality with only a few exceptions: Apache at 18 and 25% and Nautilus at 25% cache sizes. When compared to churn locality, however, temporal locality may not yield better performance.

This immediately raises a question: are temporally local entities are competing for cache space with spatially local or churn-local entities? In terms of defect density, can we do better if we selectively enable/disable the other sources of cache updates? This is a challenging question to answer as enabling and disabling different policies will change both the number of files and the number of lines in the cache. Consequently, we have to take into account the total number of lines to be inspected, as well as the open bug density. We use the aforementioned cost effectiveness measure AUCEC_{20} to consider this question.

To use AUCEC_{20}, we require a prediction model to provide an ordering in which the files should be inspected. As specified, the FixCache algorithm provides only a gross ordering of files, *i.e.*, files in the cache should be inspected prior to files not in the cache. To compare FixCache to other algorithms we tried various orderings to find the one most favorable to FixCache, but, that only used data available to practitioners at the time of prediction. Specifically, we order the files within the cache and those not in the cache independently using the same ordering: order files first by decreasing number of closed bugs, then by increasing size. We found that this ordering provided the best performance for FixCache in every setting. Given this ordering, we can then pick off files from the top of the ordering until we reach just over 20% of the overall line count in the system.

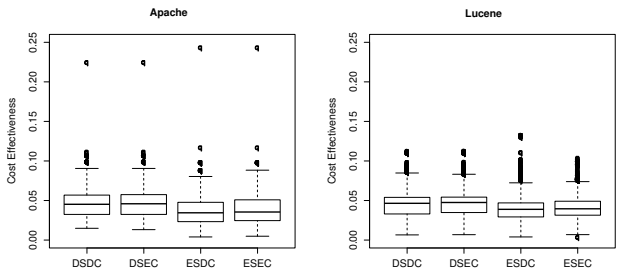

Figure 7: AUCEC_{20} cost effectiveness after enabling/disabling spatial/changed file locality (10% cache size, LRU replacement, 20% of lines inspected) for Apache and Lucene. EC/DC refers to enable/disable churn locality; ES/DS refers to enable/disable spatial locality. DSDC means disable both; all other combinations are labeled similarly

RQ4a: *Do spatial and churn locality update policies improve the cost-effectiveness of* FixCache*?*

Figure 7 shows the AUCEC$_{20}$ measure after enabling and disabling spatial and churn locality.

In Figure 7, we use DS/ES to denote "Disable/Enable Spatial locality" and DC/EC for "Disable/Enable Churn locality". DSDC denotes "both disabled", while DSEC denotes "disable spatial, enable churn". This figure makes it abundantly clear that from the perspective of AUCEC$_{20}$ cost-effectiveness, adding spatial and churn locality to the FixCache update policies provides little benefit.

We want to show that there is little to no practical difference in cost effectiveness across the different locality settings. The Kruskal-Wallis test is a non-parametric test which assess whether n independent samples are drawn from the same population. It is the non-parametric counterpart to a one-way Anova[13]. This test shows that for every project, there is no significant difference in cost-effectiveness observed when enabling or disabling churn and/or spatial locality. Thus, although Figure 6 suggests that churn and spatial locality potentially have some defect density to contribute, and may in fact cache additional files, they do not provide any benefit for inspection when inspecting 20% or fewer lines, (as per AUCEC$_{20}$). This clearly indicates that temporal locality (which caches files when defects are found) underlies most of the predictive power in FixCache.

Finally, we focus on different cache replacement policies in FixCache and compare their performance.

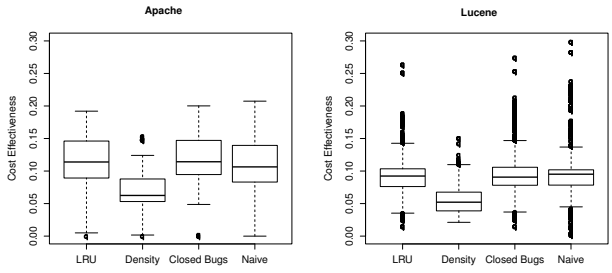

Figure 8: AUCEC$_{20}$ values for Apache and Lucene. The left 3 are FixCache with different replacement policies, and right most is a naive prediction model, which simply order files by closed bug count. Only in Apache (of all) are any of the FixCache version statistically better than the naive model, and even then the effect size is quite small

RQ4b: *Can we improve* FixCache AUCEC$_{20}$ *performance by using a different cache replacement policy?*

The original FixCache used two different policies. The first is an LRU based replacement policy, which keeps frequently hit files in the cache. The principle here is that files that haven't had bug fixes in a while are probably unlikely to have latent bugs. The second was a "closed bugs" policy, which replaced files with the least number of closed bugs first. For our evaluation, we added a third policy based on defect density,which replaced the file with the least density of closed bugs. The latter two policies are based on the theory that files with fewer bugs, or lower defect density, are likely to have fewer latent bugs. All of these policies *prima facie* appear to have merit, so it is reasonable to compare them to see which policies yield better AUCEC$_{20}$ performance.

The results are shown as the left most 3 boxplots in the two plots in Figure 8

For this question, we ran FixCache with cache size set to 6%, 10%, 18% and 25% of the total file count, and consistently obtained results with all projects similar to those seen in Figure 8. The closed bugs policy generally showed the best performance, while the density policy was consistently the worst performer. This finding again, is strongly indicative that the raw count of closed bugs in a file is an excellent indicator of future latent bugs.

Our final question is in some sense the most critical evaluation of FixCache. Does FixCache perform better than a naive prediction model?

RQ5: *When used for inspections, is* FixCache *doing better than a naive model that simply picks previously buggy files?*

For this question we chose a very simple, naive prediction model: *the model simply orders all the files in the system by the number of closed bugs*, and chooses enough files in order to capture 20% of the total line count. The results are shown as the rightmost boxplot in Figure 8.

Testing the hypothesis that the best FixCache replacement policy (*viz* closed bugs) is better than the naive model, we find that, after Benjamini-Hochberg correction, only the Apache project data rejects the null hypothesis. However, as can be seen in Figure 8, even in this case, the effect size is fairly small.

This leads to a rather surprising, conclusion:

> *The naive model is actually about the same utility for inspections, under the* AUCEC$_{20}$ *measure, as the best possible setting for* FixCache*.*

6. THREATS TO VALIDITY

Perry *et al.* [18] identify three forms of validity that must be addressed in research studies. We now examine threats to each form of validity in our study and the methods used to mitigate these threats where possible.

Construct validity refers to the degree to which the measurements are actually related to the concepts they are believed to represent. The main threat to construct validity come from bias in bug reporting. The bugs that are being reported might not be representative of actual bugs encountered in the system. Linking bias is certainly extant in the datasets, and can affect [5] the performance of FixCache, and affect our evaluations of FixCache and the merits of different update policies (RQ1-4). However, it is less likely that the result of comparing FixCache and the simple standard prediction model (RQ5) is affected by bias.

Internal validity is the ability of a study to establish a causal link between independent and dependent variables regardless of what the variables are believed to represent. Our study is focused on examining a link between variables such as closed bugs per file, the FixCache prediction and the actually open bugs in a file. Threats to internal validity come from our choice of evaluation. We attempted to address this threat by using different, specifically tailored measures for different RQs: lines for RQ1, density for RQs 2a, 2b, and 3, and the very demanding AUCEC for the rest. We chose AUCEC at 20% of the lines, because it closely approximates the linecount of the files in FixCache running with a cache size of 10%.

External validity refers to how these results generalize. Our study includes multiple projects, belonging to different software domains, and we find similar results among all projects studied. However, the sample size is only five projects. While this gives evidence of the ability of our results to generalize, further study on more projects will increase our confidence in these findings as answers to the research questions on a broad level. In particular we are concerned about projects that do not follow development processes similar to those of the projects that we studied.

7. RELATED WORK

Kim *et al.* took their inspiration from Hassan *et al.* who proposed the cache metaphor drawing on operating systems research [10]. BugCache principally differs from Hassan *et al.*'s "Top Ten List" in granularity: The top ten list contains modules, while BugCache contains files. BugCache performed better, even at the finer granularity. Our work mirrors this difference; we empirically evaluate the performance of FixCache, at *line granularity*, for prioritizing entities for inspection. In the course of this investigation, we compare FixCache to simple prediction models with an eye toward improving FixCache. Thus, there are two strands of work related to ours — FixCache evaluations and prediction models.

FixCache Evaluations Wikrstrand *et al.* used FixCache to dynamically select regression tests that run over modified, fault-prone (in cache) files. They found that in an embedded industrial system, the FixCache reached a weekly hit rate of 50–80% [24]. The study found that FixCache pre-population did not affect its hit rate and observed that daily updates quickly negate any pre-population effects. Engström *et al.* follow up Wikrstrand *et al.*'s and evaluate the effectiveness of Wikrstrand *et al.*'s algorithm at selecting regression tests. They found the fix-cache approach to be more efficient than its predecessor at selecting regression tests in four case studies [7]. Their work principally differs from ours in their focus on regression testing. Sadowski *et al.* investigated how FixCache's prediction performance varies over time and how much the set of buggy files changes [19]. They report that the hit rate is relatively stable and that, Apache `httpd`, the initial exception, stabilizes as the number of files approached the current project size. They find that most files that remain in the cache are correlated with a high hit rate, but that there is room for improvement. This evaluation differs from ours in the questions posed and our focus on cost-effectiveness, which is line-granular.

Prediction Models A large body of research into prediction models exits. Catal and Diri [6] have published a current survey. The Predictor Models in Software Engineering (PROMISE) series of events at ICSE are a well-known venue in this sub-discipline. The core idea is to build a statistical model, usually some type of linear, logistic, or count model, with either defect-proneness (binary response) or number of defects (count or continuous response) as the dependent variable. Predictors are measures such as size, number of authors [23], churn [17], social network measures [15], complexity [9], and a range of other measures. A full survey of this prolific sub-discipline would consume many pages; for our purposes here, it is sufficient to note that we chose a simple model as a straw-man for comparison, one that simply ordered files by number of closed defects found in that file.

8. CONCLUSION

In this paper, we evaluated the usefulness of FixCache as a tool for focusing code inspection effort. We found that FixCache tends to pull larger files into the cache; however, the bug density of files in FixCache is generally higher. We found, in addition, that of all the update policies in FixCache, temporal locality (recently closed bugs) contributed the most to increase bug density in the cache; indeed, when measured in terms of the demanding cost-effectiveness measure AUCEC_{20}, the other two policies, spatial and churn locality, did not contribute much. Furthermore, we compared three different cache replacement policies, *viz.*, based on LRU, closed bug density, and closed bugs, and found that closed bugs based replacement policy generally performs better in terms of cost-effectiveness.

Perhaps most surprisingly, we found that a very simple scheme — using the number of closed defects in a file to order the files — provides a AUCEC_{20} performance level that is in a statistical dead heat with the best performing configuration of FixCache. While *a priori* this might seem like a discouraging finding, it is in fact quite informative: this suggests that the most important aspect of FixCache, if one were to use it for inspection, is, in fact its temporal locality. Thus, the major conclusion of our paper is this:

> When used for inspection, FixCache gets most of its power by predicting that files that recently experienced a bug fix, do in fact contain additional, latent bugs.

Finally, we emphasize that our evaluation *targeted the utility of* FixCache *for inspections*. While the AUCEC_{20} measures the effectiveness of FixCache or other ordering algorithms when choosing 20% of the lines to inspect, it does not really say much about other possible uses of FixCache, for inspections, static analysis, formal verification, *etc.* We believe these are fruitful directions for further research.

References

[1] The economic impacts of inadequate infrastructure for software testing, www.nist.gov/director/planning/upload/report02-3.pdf.

[2] E. Arisholm, L. Briand, and M. Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. 2007.

[3] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

[4] A. Bachmann and A. Bernstein. Software process data quality and characteristics: a historical view on open and closed source projects. In *IWPSE-Evol '09*, 2009.

[5] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 121–130. ACM, 2009.

[6] C. Catal and B. Diri. A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36(4):7346–7354, 2009.

[7] E. Engström, P. Runeson, and G. Wikstrand. An empirical evaluation of regression testing based on fix-cache recommendations. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 75–78. IEEE, 2010.

[8] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, pages 23+, Washington, DC, USA, 2003. IEEE Computer Society.

[9] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, Washington, DC, USA, 2009. IEEE Computer Society.

[10] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 263–272, Washington, DC, USA, 2005. IEEE Computer Society.

[11] S. Kim, T. Zimmermann, K. Pan, and J. Jr. Automatic identification of bug-introducing changes. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[12] S. Kim, T. Zimmermann, E. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

[13] W. Kruskal and W. Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, 47(260):583–621, 1952.

[14] J. Lawless and J. Lawless. Statistical models and methods for lifetime data. 1982.

[15] A. Meneely and L. Williams. Secure open source collaboration: an empirical study of linus' law. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 453–462. ACM, 2009.

[16] A. Mockus and L. Votta. Identifying reasons for software changes using historic databases. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 120–130. IEEE, 2002.

[17] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 284–292. IEEE, 2005.

[18] D. Perry, A. Porter, and L. Votta. Empirical studies of software engineering: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 345–355. ACM New York, NY, USA, 2000.

[19] C. Sadowski, C. Lewis, Z. Lin, Z. X., and E. J. Whitehead. An empirical analysis of the fixcache algorithm. In *Proceedings of the Eigth MSR Conference (to appear)*, 2011.

[20] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[21] D. Čubranić and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 408–418, Washington, DC, USA, 2003. IEEE Computer Society.

[22] Z. Wang. Fix Cache Based Regression Test Selection. Master's thesis, Computer Science, Chalmers University of Technology, Sweden, 2010.

[23] E. Weyuker, T. Ostrand, and R. Bell. Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559, 2008.

[24] G. Wikstrand, R. Feldt, J. Gorantla, W. Zhe, and C. White. Dynamic regression test selection based on a file cache an industrial evaluation. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 299–302. IEEE, 2009.