Predicting Faults from Cached History

Sunghun Kim [1]
hunkim@csail.mit.edu

Thomas Zimmermann [2]
tz @acm.org

E. James Whitehead, Jr. [3]
ejw@cs.ucsc.edu

Andreas Zeller [2]
zeller@acm.org

[1] *Massachusetts Institute of Technology, USA*

[2] *Saarland University, Saarbrücken, Germany*

[3] *University of California, Santa Cruz, USA*

Abstract

We analyze the version history of 7 software systems to predict the most fault prone entities and files. The basic assumption is that faults do not occur in isolation, but rather in bursts of several related faults. Therefore, we cache *locations that are likely to have faults: starting from the location of a known (fixed) fault, we cache the location itself, any locations changed together with the fault, recently added locations, and recently changed locations. By consulting the cache at the moment a fault is fixed, a developer can detect likely fault-prone locations. This is useful for prioritizing verification and validation resources on the most fault prone files or entities. In our evaluation of seven open source projects with more than 200,000 revisions, the cache selects 10% of the source code files; these files account for 73%-95% of faults—a significant advance beyond the state of the art.*

1. Introduction

Software quality assurance is inherently a resource-constrained activity. In the majority of software projects, the time and people available are not sufficient to eliminate all faults before a release. Any technique that allows software engineers to reliably identify the most fault-prone software functions provides several benefits. It permits available resources to be focused on the functions that have the most faults. Additionally, such a list makes it possible to selectively use time intensive techniques, such as software inspections, formal methods, and various kinds of static code analysis.

Two important qualities of software fault prediction algorithms are *accuracy* and *granularity*. The accuracy is the degree to which the algorithm correctly identifies future faults. The granularity specifies the locality of the prediction. Typical fault prediction granularities are the executable binary [19], a module (often a directory of source code) [11], or a source code file [21]. For example, a directory level of granularity means that predictions indicate a fault will occur somewhere within a directory of source code. The most difficult granularity for prediction is the entity level (or below), where an "entity" is a function or method.

We have developed an algorithm that, in our experimental assessment on seven open source projects, is 73%-95% accurate at predicting future faults at the file level and 46%-72% accurate at the entity level with optimal options. This accuracy is better than or equivalent to other efforts reported in the literature. Moreover, we achieve this accuracy at the entity and file level, which permits a more targeted allocation of available resources because of the greater locality of the predictions.

Our prediction algorithm is executed over the change history of a software project, yielding a small subset (usually 10%) of the project's files or functions/methods that are most fault-prone. The key insight that drives our algorithm is the observation that most faults are local. Put another way, faults do not occur uniformly in time across the history of a function; they appear in bursts. Specifically, we believe bug occurrences have four different kinds of locality:

Changed-entity locality. If an entity was changed recently, it will tend to introduce faults soon.

New-entity locality. If an entity has been added recently, it will tend to introduce faults soon.

Temporal locality. If an entity introduced a fault recently, it will tend to introduce other faults soon.

Spatial locality. If an entity introduced a fault recently, "nearby" entities (in the sense of logical coupling) will also tend to introduce faults soon.

Following Hassan and Holt [11], we borrow the notion of a cache from operating systems research, and apply it for the purpose of fault prediction. We use the cache as a convenient mechanism for holding our current list of the most fault-prone entities, and for aggregating multiple heuristics for maintaining the cache. The switch to a cache involves a subtle but important shift: instead of creating mathematical functions that predict future faults, the cache selects and

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE

removes entities based on specific criteria—in our case the localities specified above. The ideal is to minimize the size of cache while at the same time maximizing its accuracy.

Unlike most existing research on fault prediction that approximates faults simply with fixes, we use *bug-introducing* changes. Following the definitions in [22], a fix is a modification that repairs a software fault; it tells us the *location* where a bug occurred (which lines and files), but not the *time* when the bug was introduced. However, the latter information is crucial for the localities that we defined before; for instance spatial locality requires the time when a fault was introduced to identify nearby entities at that time. We get the time information from bug-introducing changes, the modifications that create faults. Hence, the chronology is bug-introducing change(s), bug report, and finally fix.

BugCache vs. FixCache. This paper describes and evaluates algorithms for maintaining a cache based on fault localities. There are two variants:

BugCache updates the cache at the moment a fault is missed, that is, not found in the cache. We will use BugCache to empirically show the presence of fault localities. Since in practice, a change is not known to be bug-introducing until the corresponding fix, BugCache is a theoretical model.

FixCache shows how to turn localities into a practical fault prediction model. In contrast to BugCache, it has a delayed update: when a fault is fixed, the algorithm traces back to the corresponding bug-introducing change, and only then is the cache updated based on the bug-introducing localities.

This paper makes the following contributions:

Empirical evidence of fault localities. Evaluation of the BugCache algorithm provides empirical evidence that fault localities actually exist.

Very accurate fault prediction. By combining a cache model with different heuristics for fault prediction, the FixCache algorithm has an accuracy of 73%-95% using files and 46%-72% using methods/functions.

Validation of adaptive fault prediction. FixCache is an online learning approach [1], learning from cache hits and misses. Thus it can easily adapt when a system's fault distribution changes. FixCache's high accuracy, equivalent in accuracy to the best approaches in the literature, with smaller granularity, demonstrates the utility of adaptive fault prediction algorithms.

In the remainder of this paper, we discuss fault localities (Section 2) and then proceed to the caching algorithms (Section 3). We also present details on the data collection for our experiments (Section 4). The results of experiments on seven projects at the file and entity level are presented in two sections: one for empirical evidence of localities (BugCache, Section 5) and one for predicting future faults (FixCache, Section 6). We discuss our results and list threats to validity (Section 7), before we close the paper with related work and consequences (Sections 8 and 9).

2. Bug localities

Software engineering does not yet have a widely accepted model for why programmers create software faults. Ko et al. [16] summarizes possible causes for programming errors, using a model of *chains of cognitive breakdowns*. (Note that "breakdowns" comes as a plural; for many errors, there is more than one cause.)

Like Ko et al., we also consider cognitive breakdown as the source for faults. In particular, we assume that faults do not appear individually, but rather in *bursts*: either in the same entity (temporal locality) or nearby entities (spatial locality). Furthermore, we assume any code modification as risky, since the programmer might suffer a cognitive breakdown (changed-entity and new-entity localities). We describe temporal and spatial locality in more detail below.

2.1. Temporal locality

The intuition behind temporal locality is that faults are not introduced individually and uniformly over time. They rather appear in bursts within the same entities. In other words, when a fault is introduced to an entity, another fault will likely be introduced to the same entity soon. An explanation for such bursts that programmers make their changes based on a poor or incorrect understanding, thus injecting multiple faults.

Using temporal locality significantly differs from using cumulative numbers of faults (or changes) to predict future faults. Accumulated numbers result in sluggish predictors that cannot adapt to new fault distributions. In particular, they would miss entities with few, but recent faults. Such entities are more likely exposed to new faults than entities with many old faults.

The weighted time damp model by Graves et al. is similar in spirit to temporal locality [10]. It more heavily weights recent faults to predict future ones and was one of the best models they observed. Compared to the math heavy model in [10], temporal locality has a simpler description and relies on bug-introducing changes rather than on fixes.

Temporal locality also guides cache replacement strategies for our algorithms. If there were no faults for an entity in a long time, it is removed from the cache (see Section 3.5).

2.2. Spatial locality

When programmers make changes based on incorrect or incomplete knowledge, they likely cannot assess the impact of their modifications as well. Thus, when an entity has a fault, there is a good chance of other, nearby entities also having faults. But what are nearby entities? There are several ways to define distance in software. One way is using physical distances among entities. In this case, the entities in the same file or directory would be nearby entities. Another way is using logical coupling among software entities [3, 8]: two entities are close to each other (logically coupled) when they are frequently changed together.

We compute the distance between two entities using logical coupling. If two entities are changed together many times, we give them a short distance, reflecting their logical "closeness". We compute the distance between any two entities e_1 and e_2 as follows:

$$\text{distance}(e_1,e_2) = \begin{cases} \dfrac{1}{\text{count}(\{e_1,e_2\})} & \text{count}(\{e_1,e_2\}) > 0 \\ \infty & \text{otherwise} \end{cases}$$

where $\text{count}(\{e_1,e_2\})$ is the number of times e_1 and e_2 have been changed together.

2.3. Changed-entity and new-entity locality

Research shows that entities that changed recently are more likely to be fault-prone than others. This has been leveraged for fault prediction by using code churn [19] and the "most recently modified/fixed" heuristics [11]. In a similar fashion, new entities are more likely to contain faults than existing ones [10]. We use these results to define additional localities:

- An entity that was changed recently likely contains a fault *(changed-entity locality)*.
- An entity that was added to a system recently likely contains a fault *(new-entity locality)*.

These two localities are used to pre-fetch changed and added entities into the cache on the assumption they will tend to introduce faults soon.

3. Operation of the cache

Our algorithm maintains a list (cache) of what it has chosen as the most fault-prone software entities. The cache size can be adjusted based on the resources that are available for testing or verification. A typical cache size is 10% of the total number of entities, since this provides a reasonable tradeoff between size and accuracy. Larger cache sizes result in higher hit rates (better recall), but with the faults spread out over a greater number of entities (lower precision).

3.1. Basic operation

The basic process of the cache algorithm is as follows:

Initialization:

1. Bug fix changes are extracted by mining a project's version archive and bug database.
2. Bug-introducing changes are identified at the file and entity level, using the approach in [22].
3. Pre-load the cache with the largest entities (LOC) in the initial project revision, creating the initial state of the cache*. (Optional)*

Cache operation:

4. **BugCache:** If revision *n introduces* a fault in an entity, the cache is probed to see if it is present. If yes, count a hit, otherwise a miss.

   FixCache: If revision *n fixes* a fault in an entity, probe the cache to see whether the corresponding entity is present. If yes, count a hit, otherwise a miss.
5. If a fault is missed, determine the bug-introducing change and fetch the entity (temporal locality) as well as nearby entities (spatial locality) into the cache for use in future fault predictions starting at revision *n+1*. The algorithm only uses localities at the time a fault was introduced, i.e., the revision of the bug-introducing change.

   Parameter: Block size (see Section 3.3)
6. Also at revision *n*, pre-fetch entities that have been created (new-entity locality) and modified (changed-entity locality) since revision *n-1*.

   Parameter: Pre-fetch size (see Section 3.4)
7. Since the size of the cache is fixed, we have to remove entities, which are selected using a cache replacement policy such as least recently used.

   Parameter: Replacement policy (see Section 3.5)
8. Iterate over steps 4-7 to cover the existing change and bug history.

Finally, the hit rate is computed by:

$$\textit{hit rate} = \frac{\#\,\textit{of hit}}{\#\,\textit{of hit} + \#\,\textit{of miss}}$$

A hit rate close to 1 means, for BugCache, that the localities described the fault distribution accurately

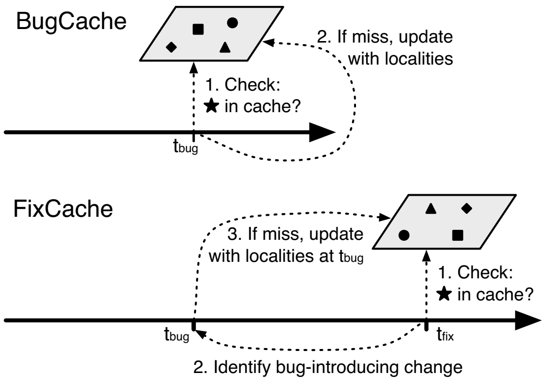
BugCache
2. If miss, update with localities
1. Check: ★ in cache?
t_{bug}
FixCache
3. If miss, update with localities at t_{bug}
1. Check: ★ in cache?
t_{bug}
t_{fix}
2. Identify bug-introducing change

Figure 1. BugCache vs. FixCache

over time and, for FixCache, that it predicted most future faults.

This approach is similar to on-line machine learning algorithms [1] in that our algorithm learns from the fault distributions (hits and misses) and quickly updates the prediction model (cache).

3.2. Bug cache vs. Fix cache

There are two variants of the caching algorithm.

BugCache updates the cache at the moment when a fault, in the form of a bug-introducing change, is missed. However, in practice a change is not known to be bug-introducing until it is fixed. This means that the BugCache needs to know the nature of a change in advance, and hence it is a tool to *empirically show the presence of fault localities* rather than a deployable fault prediction algorithm.

In contrast, ***FixCache*** shows how to turn localities into a practical *fault prediction model* that can be used in any software project. FixCache does not update when a fault (bug-introducing change) is missed—it waits until the fix. In other words it has a *delayed update*: when a fault is fixed, the cache is updated based on the localities that existed when the fault was introduced. The hit rates for FixCache are computed at the time of the fix, the last moment when the fault was still alive.

The difference between BugCache and FixCache is sketched in Figure 1. BugCache computes hit rates and updates the cache when a fault is introduced t_{bug}; FixCache waits until a fault is fixed (t_{fix}). Both use the localities at the time the fault was introduced (t_{bug}).

3.3. Cache update

When we miss a fault in an entity, our cache algorithm loads nearby entities (spatial locality). We adapt the notion of *block size* from cache terminology to describe the upper bounds on how many entities are loaded. A block size of *b* indicates that we load the *b-1* closest entities (i.e., the ones with the shortest distance) along with the faulty entity itself. In our analysis, we investigate the effect of different block sizes.

Table 1. Cache replacement policies.

Id	Last found fault/hit (ago)	Cumulative changes	Cumulative faults
1	1 day	30	**1**
2	**10 days**	20	5
3	9 days	10	7
4	2 days	**5**	4

3.4. Pre-fetches

We use pre-fetching techniques to improve the hit rate of the bug cache. Pre-fetching means that we load entities for which we have not yet encountered a fault. Our motivation is as follows: assume we would load entities only when we encounter a fault (or a fix in case of FixCache). As a consequence, we would have inevitable misses since we start with an empty cache. Additionally, it would be impossible to predict faults for entities that have exactly one fault in their lifetime (this fault is a mandatory miss). In order to reduce the miss count, we pre-fetch potential fault-prone entities in advance by using the algorithms described below.

Initial pre-fetch. Initially the cache is empty, and in the absence of pre-fetching, this would lead to many misses. We avoid this and initialize the cache with entities likely to have faults as predicted by greatest lines of code (LOC). The relation between faults and LOC has been revealed in several studies so far [10, 21].

Per-revision pre-fetch. We pre-fetch entities that were modified or created between two revisions (new-entity and changed-entity locality). We start with the entities that have the highest number of LOC. Additionally, we unload entities that were deleted. The *pre-fetch size* parameter controls the maximum number pre-fetches per revision.

3.5. Cache replacement policies

When the cache is full, our algorithm has to unload entities before it can load new ones. Ideally, we would keep the entities with greatest potential for new faults. A replacement policy describes which entities to unload first. In operating systems a frequently used policy is *least recently used* (LRU), which first replaces the element used the longest time ago. We developed LRU-like policies for our fault-caching algorithms. Specifically, we used the observation that entities with many changes or prior faults are likely to

Table 1. Analyzed open source projects. The period shows the analyzed project timespan. The number of revisions indicates the number of revisions we extracted. The number of entities indicates number of functions or methods in the last revision. The number of bugs indicates the number of bug-introducing changes we extracted by mining the change logs and change histories of each project. For the Eclipse project we use only the core.jdt module due to the large size of the entire project. Similarly, we use only the mozilla/content/ module for the Mozilla project.

					Number of			
Project	Lang.	Software type	SCM	Period	Revisions	Entities	Files	Bugs
Apache HTTP 1.3	C	HTTP server	Subversion	01/1996 ~ 07/2005	7,747	2,113	154	1,954
Subversion	C	SCM software	Subversion	08/2001 ~ 07/2005	6,029	3,693	255	1,566
PostgreSQL	C	DBMS	CVS	04/1996 ~ 08/2005	14,650	8659	598	19,902
Mozilla	C/C++	Web browser	CVS	03/1998 ~ 01/2005	109,636	8203	396	52,265
JEdit	Java	Editor	CVS	09/2001 ~ 06/2005	1,386	5429	420	3,060
Columba	Java	Mail Client	CVS	11/2002 ~ 07/2005	2,848	8428	1428	720
Eclipse	Java	IDE	CVS	04/2001 ~ 01/ 2005	78,948	33214	3330	15,217

have further faults [10, 11, 21] to create weighted LRU algorithms based on previous changes and faults.

Least recently used (LRU). This algorithm unloads the entity that has the least recently found fault (hit). Consider a cache with the entities shown in Table 1. Based on the classical LRU algorithm, Entity 2 would be unloaded, since it is the least recently used entity.

LRU weighted by number of changes (CHANGE). When an entity has changed many times in the past, it is more likely to have faults in the future [10]. We want to keep such entities in the cache as long as possible. Consequently, we unload the entity with the least number of changes. According to this policy, Entity 4 in Table 1 would be unloaded.

LRU weighted by number of previous faults (BUG). This policy is similar to the change-weighted LRU. It removes the entity with the least number of observed faults. The intuition here is that when an entity has had many faults, it will likely continue to have faults. With this policy, Entity 1 in Table 1 would be unloaded.

4. Data collection and fact extraction

Data used in the evaluation of the BugCache and FixCache was collected using the Kenyon infrastructure [4] (Apache 1.3, JEdit, Subversion, and PostgreSQL) and APFEL [6] (Columba, Eclipse, and Mozilla). Analyzed open source projects are shown in **Error! Reference source not found.**. Details of the data collection process are described below.

4.1. Transaction recovery

In order to measure the impact of co-change for spatial locality, we need *transactions* that alter the entire *product* rather than just single files. In Subversion [2], such transactions are directly available. CVS, however, provides only versioning at *file* level, disregarding co-change information between files. To recover per-product transactions from CVS archives, we *group* the individual per-file changes using a *sliding window* approach [25]: two subsequent changes by the same author and with the same log message are part of one transaction if they are at most 200 seconds apart.

4.2. Finding fixes and bug-introducing changes

In order to find bug-introducing changes, bug fixes must first be identified by mining change log messages. We use two approaches: searching for keywords such as "Fixed" or "Bug" [17] and searching for references to bug reports like “#42233” [5, 7, 22]. This allows us to identify whether an entire transaction contains a bug fix. If it does, we then need to identify the specific file change that introduced the bug.

Once we know that a transaction contains a fix, we first list files changed in the transaction and then use the annotation features of CVS and Subversion to identify bug-introducing changes [22]. In the example below revision 1.42 fixes a fault in line 36. This line was introduced in revision 1.23 (when it was line 15). Thus revision 1.23 contains a bug-introducing change.

1.23: Bug-introducing	**1.42: Fix**
…	…
15: `If (foo==null) {`	36: **`if (foo!=null) {`**
16: `    foo.bar();`	37: `    foo.bar();`
…	…

Additionally, bug databases are used (if available) to eliminate false positives. For example, bug-introducing changes that were made after the bug was reported cannot be bug-introducing changes for that particular bug. More details on how to locate bug-introducing changes are presented in previous work, including techniques that reduce the number of false positives [15, 22].

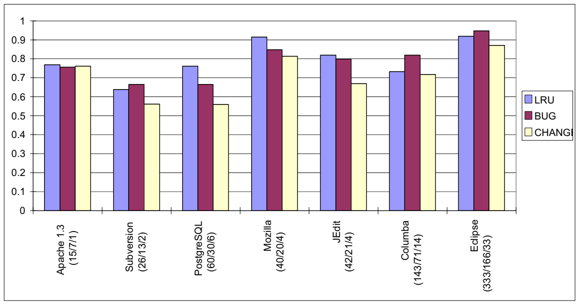

Figure 2. Hit rates, file level. We varied the cache replacement policies. The cache size is 10%, block size is 5%, and pre-fetch size is 1% of the total number of files.

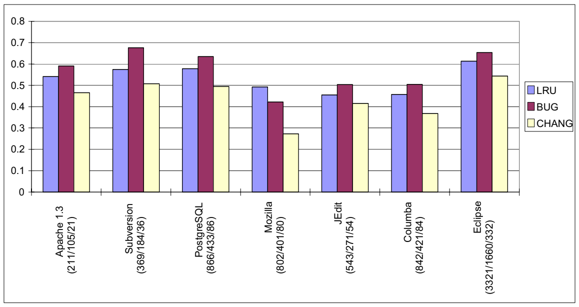

Figure 3. Hit rates, method level. We varied the cache replacement policies. Cache size is 10%, block size is 5%, and pre-fetch size is 1% of the total number of methods.

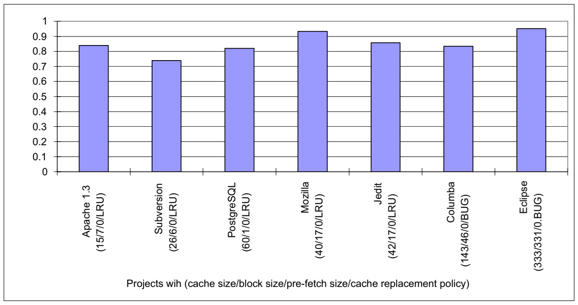

Figure 4. Optimal hit rates, file level. We set the cache size to 10% of the total number of files and determined the optimal parameter combination (block size, pre-fetch size, cache replacement policy) via brute force analysis.

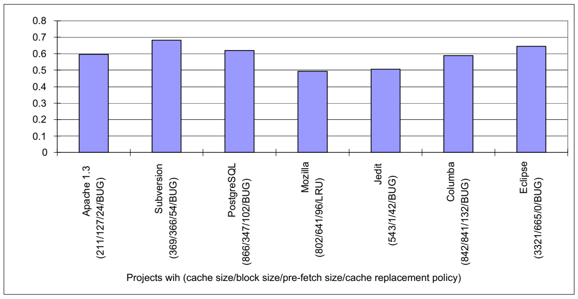

Figure 5. Optimal hit rates, method level. We set the cache size to 10% of the total number of methods and determined the optimal parameter combination (block size, pre-fetch size, cache replacement policy) using brute force.

4.3. Fine-grained changes

In addition to bug-introducing changes, we need a list of co-changed entities for computing spatial locality. From CVS and Subversion we get a list of co-changed files. For method co-changes we perform an additional analysis. First, compute a text diff between revisions. From this diff, determine modified line numbers, which are then mapped to the surrounding methods. This approach is described in detail in [25]. Similarly, we obtain the methods that were added or deleted between revisions (needed for pre-fetching new entities and removing deleted ones).

5. Bug Cache Evaluation

The BugCache algorithm has multiple parameters that can be modified, all of which affect its hit rate. It is possible to modify the cache size, block size, pre-fetch size, and cache replacement policy. To determine which combination of parameters yields the highest hit rate, we literally tried them all. We performed a brute force cache analysis that iterated through multiple option combinations, and compared the results to reason about fault localities (Section 5.1). We also measured the impact of cache replacement policies (Section 5.2) and the relative contributions of each fault locality (Section 5.3).

5.1. Hit rates

The first experiment used constant cache options: a cache size of 10%, block size of 5%, and a pre-fetch size of 1% of the total number of elements (depending on the granularity, either files or entities). For example for Subversion with 3,693 functions, the cache size is 369, block size is 184, and pre-fetch size is 36. Figures 2 and 3 show the hit rates at the file and entity (method/function) level. The file level hit rates are 57%-93%, and entity level hit rates are 28%-68% depending on the cache replacement policy. These results provide initial empirical evidence for the presence of fault localities, especially at the file level.

The BugCache hit rates describe how well it models the fault distribution within a project. In order to obtain the optimal "fit", we identified the options that describe the fault distribution most accurately by running a brute force analysis. The cache size was fixed at 10% of the total number of entities or files. Then we changed the block size, pre-fetch sizes, and cache replacement policy, and observed the resulting hit rate.

Block sizes and pre-fetch sizes were varied from 0 to 100% of the cache size with a step of 5%.

The best option combinations for each project and the resulting hit rates are shown in Figures 4 (file level) and 5 (entity level). At the file level, all projects have a pre-fetch size of 0. This indicates that changed and new-entity localities are not very common at the file level. This changes when the granularity is functions/methods: except for Eclipse, all projects show empirical evidence for changed and new-entity localities.

The projects also differ in the block size: while JEdit has a block size of one method, Columba has a block size of 841 methods. Having a small block size indicates that temporal locality dominates over spatial locality (not many nearby entities have to be loaded and most errors are local). Having a big block size means that some events in the history changed the fault distribution dramatically, causing most of the cache to be replaced in one operation. However, such events are the exception; typically only a small part of the cache is replaced. Recall that block size is the maximum number of elements to be replaced, not the average number.

An important implication of these results is that fault distributions vary across projects and thus *fault prediction algorithms need to be adapted to a specific project* [20].

5.2. Cache Replacement Policy

We implemented three cache replacement policies (LRU, BUG, and CHANGE) to unload elements from the cache. To see which algorithm works best for a given set of cache parameters, we performed an experiment using the same values for the cache size, block size, and pre-fetch size, varying only the cache replacement policy.

Figures 2 and 3 show the resulting hit rates. At the file level, the LRU policy has the best results for 4 out of the 7 projects, with BUG having the best results for the remaining 3. At the function/method level, BUG has the best results for all projects, except for Mozilla (LRU). Interestingly, the CHANGE policy works poorly at both granularities. This is somewhat contrary to the results of Hassan and Holt [11], where the *most-frequently-modified* heuristic was one of the best fault predictors.

5.3. Bug Localities

BugCache combines four fault localities for its model. But are the contributions of the localities the same? To measure the relative predictive strength of each locality, each entity was marked with the reason (initial prefetch, or kind of locality) that caused it to be loaded. Figure 6 shows for the Apache 1.3 project the ratio of reasons why hit entities were loaded into the cache. The results show that faults have strong temporal (59%) and spatial (18%) locality, and weak changed entity (4%) and new entity (1%) locality. The initial pre-fetch is surprisingly effective, accounting for 18% of the total hits.

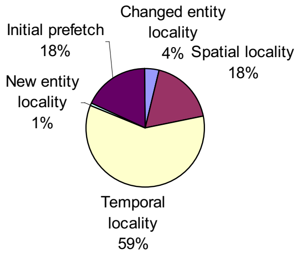

Figure 6. Contribution of initial pre-fetch and fault localities on method level for Apache 1.3. Cache size is 211, block size is 127, pre-fetch size is 24, replacement policy is BUG. The hit rate is 59.6%.

One possible explanation for these results is that faults indeed occur in bursts, in most cases locally within one single entity. However, there are enough cases where errors affect multiple entities, and hence spatial locality succeeds in predicting them. When no data is available, code complexity (as represented by LOC) acts as a strong predictor of faults. Changed and new-entity locality predicted only small portions of faults.

6. Fix Cache Evaluation

The previous section provided empirical evidence for the presence of fault localities in software projects. But how can we leverage fault localities for prediction?

A typical application of the FixCache prediction algorithm is as follows: Whenever a fault is found and fixed, our algorithm automatically identifies the change to the original code that introduced the fault. Then it updates the cache using the localities from the moment this bug-introducing change was applied. 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. Developers can also directly benefit from FixCache. If a developer is working on entities in the cache, he can be made aware that he is working on a potentially instable or fault-prone part of the software.

While faults can only become part of the cache as soon as they are fixed, the cache still contains suspicious locations based on recent changes. In particular, the cache would also direct resources to newly added or changed locations. All in all, we expect that a cache will help directing efforts to those entities, which are most likely to contain errors—thus FixCache can assist in increasing quality and reducing effort.

6.1. Evaluation

We performed the FixCache analysis over the same set of seven projects and again selected the best cache parameters for each project with brute force. The cache size was set to 10% of files or entities respectively.

Figure 7 compares the results of FixCache to the ones of BugCache for files. For most projects there seems to be a small drop in accuracy (2-4%). Figure 8 shows the comparison for entities. Except for Subversion, the results stay the same or improve. These results indicate that fault localities and the FixCache algorithm can predict future faults.

In summary, the hit rates (predictive accuracy) are 73-95% at the file level, with typical performance in the low to mid 80s. The most directly comparable work is by Hassan and Holt [11], which also uses a caching approach, but at the module level. For a cache size of 10% of all modules, their hit rates vary from 45%-82%. The hit rates we observed for FixCache are better and more fine-grained, which is typically harder to predict. Ostrand et al. [21] predicted fault density of files using negative binomial linear regression. Using this method and they selected 20% of all files, which predicted 71-93% of future faults. FixCache achieves a comparable accuracy, but with only 10% of files, twice the precision.

On entity level we used again a cache size of 10, with the cache holding 10% of all project entities. For FixCache the best hit rates range from 46-72% (see Figure 8). As expected, predicting bugs at the fine-grained entity level is more difficult than predicting bugs at coarser granularity.

6.2. Discussion

Why does the cache model have better predictive accuracy than previous prediction models? Most models found in the literature use fault correlated factors and develop a model to predict future faults. Once developed, the model is static, and incorporates all previous history and factors. In contrast, the cache model is *dynamic* and is able to adapt more quickly to new fault distributions, since fault occurrences directly affect the model. This approach is similar to on-line machine learning algorithms [1] in that the cache learns from the fault distributions of each project. Even though projects have different fault distributions, the cache model adaptively learns from hits and misses to update its prediction model. This adaptation approach results in better predictive power.

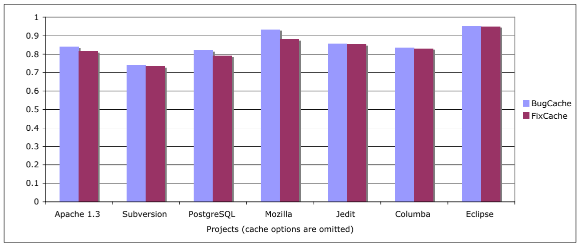

Figure 7. Optimal hit rates, file level, for BugCache and FixCache.

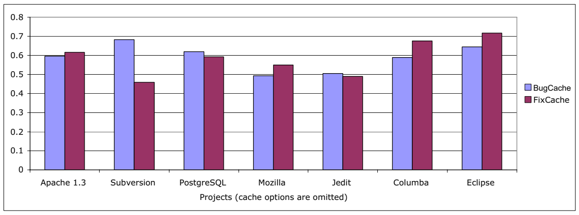

Figure 8. Optimal hit rates, entity level, for BugCache and FixCache.

The selection of cache options and replacement policies affects the hit rate. The options vary across projects due to differing fault and change distributions. We observed the following rules of thumb: 7-15% of the total number of files/entities is a good cache size. For entities, we suggest a block size of 30-50% and a pre-fetch size of 10-30% of the cache size. The BUG cache replacement policy works for most cases. However, cache options should be periodically optimized by brute force analysis on past predictions. We are currently working on building such self-configuring caches.

7. Threats to Validity

We identify the following threats to validity.

Systems examined might not be representative. Seven systems were examined in this paper, more than any other work reported in the literature. In spite of this, it is still possible that we accidentally chose systems that have better (or worse) than average cache hit rates. Since we intentionally chose systems for which we could identify fixes based on the change description log (required for determination of bug-introducing changes), we might have a project selection bias.

Systems are all open source. All systems examined in this paper are developed as open source. Hence they might not be representative of closed-source development since different development processes could lead to different fault localities. Despite being open source, several of the analyzed projects have substantial industrial participation.

Fault and fix data is incomplete. Even though we selected projects with a high quality of historic data, we still can only extract a subset of all faults (typically 40%-60% of those reported in bug tracking systems). However, we are confident that the hit rate improves with the quality of the dataset.

Entities change their names. Entities are identified by file name, function name, and signature. As a consequence an entity's history is lost when it is renamed. To some extent, this effect is weakened by the new-entity pre-fetch since renaming entities is captured as simultaneous deletion and addition. Origin analysis can recognize when elements change their names [9, 14, 24]. In future work, we will investigate whether adopting origin analysis increases the hit rate.

8. Related Work

Previous work on fault prediction falls into one of the following categories: identifying problematic entities, usually modules, with software quality metrics [11, 12, 13, 21] and predicting fault density of entities using software change history [10, 19].

8.1. Identifying problematic entities

Hassan and Holt proposed a caching algorithm for fault-prone modules, called the top-ten list [11]. They used four factors separately: modules that were most frequently modified, most recently modified, most frequently fixed, and most recently fixed. Like our cache, their top-ten list is dynamically maintained, i.e., changes over time. However, our approach combines all four factors to derive synergy. Additionally, we use spatial locality (logical coupling) as a predictor, which boosts the performance of our approach. Furthermore Hassan and Holt predicted at the module level of granularity, where a module is a collection of files. In contrast, we predict for individual files and methods, which is of greater benefit for developers and testers.

Ostrand et al. predicted fault density of files with a negative binomial linear regression model [21]. With their model, they selected 20% of all files as the most problematic ones in a project. This list predicted 71-93% of future faults. This compares most directly to Figure 7, where we predict 73-95% of future faults, but with greater precision (10% vs. 20% of all files).

Khoshgoftaar and Allen proposed stepwise multiple regression on software complexity metrics such as LOC and cyclomatic complexity to predict future fault density [12, 13]. Their top 10% of modules identified 64% and the top 20% identified 82% of all faults. Since they rely on complexity metrics (and fixing a fault does not change them much), their predictions tend to be static over time and do not easily adapt to new fault densities.

8.2. Predicting fault density

Graves et al. assumed that modules *that were changed recently* are more fault-prone than modules that were changed a long time ago [10]. They built a weighted time damp model to predict faults from changes over where recent changes are weighted over older ones. This model improved predictive accuracy substantially, which provides additional empirical evidence for the locality of faults.

Mockus et al. identified *properties of changes*, such as number of changed subsystems, number of changed lines, whether the change is a fix [18]. They used these properties to predict the risk of changes with logistic regression. The most significant factor was whether the change is a fix, meaning that fixes are more risky than other changes. To some extent this is similar to our temporal fault locality.

Śliwerski et al. computed the *risk of code locations* by the percentage of bug-introducing changes [23]. However, they did not evaluate whether past risk predicts future risk. Additionally, their risk concept is static and does not adapt to new change information.

Nagappan et al. observed that relative *code churn* measures such as changed-LOC/LOC predict future faults better than absolute code churn measures such as changed-LOC [19]. Nagappan et al. studied Windows binaries, i.e., components. Hence it is unclear how well their approach works at more fine-grained levels. Our cache algorithms use absolute measures. However, relative measures are intriguing, and we will explore their application to caching in the future.

9. Conclusions and future work

If we know that a fault has occurred, it is useful to search its vicinity for further faults. Our FixCache model predicts these further faults with high accuracy: At the file level, it can cover about 73-95% of future faults; at the function/method level, it covers 46-72% of future faults—with a cache size of only 10%. This is a significantly better accuracy and lower granularity than found in the previous state of the art. The cache can serve as a priority list to test and inspect software whenever resources are limited (i.e., always).

The FixCache is able to adapt more quickly to recent software change history data, since the fault occurrences directly affect the model. This is another significant advantage over static models, which constitute the state of the art. We are the first to use spatial locality as a bug predictor, and the combination of four locality concepts again shows significant advantages.

Even so, we still see room for improvement. Our future work will concentrate on the following topics.

- In our study, option combinations for each project vary due to the various fault or change distributions of different projects. We are currently investigating self-adaptive cache algorithms that will learn from hits/misses and change cache options for the next prediction.

- We showed that different levels of software granularity result in different hit rates. We can design *hierarchical caches* that simultaneously fetch entities at different granularities such as modules, files, and methods.

- Finally, we are currently working on integrating FixCache into history-aware programming tools such as eROSE [26]. This way, whenever a fault is fixed, the tool can automatically suggest further locations to be examined for related faults.

Overall, we expect that future approaches will see software history not only as a series of revisions and changes, but also as a series of successes and failures—and as a source for continuous awareness and improvement. The FixCache is a first step in this direction.

References

[1] E. Alpaydin, *Introduction to Machine Learning*: The MIT Press, 2004.

[2] B. Behlendorf, C. M. Pilato, G. Stein, K. Fogel, K. Hancock, and B. Collins-Sussman, "Subversion Project Homepage," 2005.

[3] J. Bevan and E. J. Whitehead, Jr., "Identification of Software Instabilities," *Proc. of 2003 Working Conference on Reverse Engineering (WCRE 2003)*, Victoria, Canada, 2003.

[4] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey, "Facilitating Software Evolution with Kenyon," *Proc. of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005)*, Lisbon, Portugal, 2005.

[5] D. Cubranic and G. C. Murphy, "Hipikat: Recommending pertinent software development artifacts," *Proc. of 25th International Conference on Software Engineering (ICSE)*, Portland, Oregon, 2003, pp. 408-418.

[6] V. Dallmeier, P. Weißgerber, and T. Zimmermann, "APFEL: A Preprocessing Framework For Eclipse," http://www.st.cs.uni-sb.de/softevo/apfel/, 2005.

[7] M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems," *Proc. of 2003 Int'l Conference on Software Maintenance (ICSM'03)*, 2003, pp. 23-32.

[8] H. Gall, M. Jazayeri, and J. Krajewski, "CVS Release History Data for Detecting Logical Couplings," *Proc. of Sixth International Workshop on Principles of Software Evolution* (IW-PSE'03), Helsinki, Finland, 2003, pp. 13-23.

[9] M. W. Godfrey and L. Zou, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities," *IEEE Trans. on Software Engineering*, vol. 31, pp. 166- 181, 2005.

[10] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Transactions on Software Engineering*, vol. 26, pp. 653-661, 2000.

[11] A. E. Hassan and R. C. Holt, "The Top Ten List: Dynamic Fault Prediction," *Proc. of International Conference on Software Maintenance (ICSM 2005)*, Budapest, Hungary, 2005, pp. 263-272.

[12] T. M. Khoshgoftaar and E. B. Allen, "Ordering Fault-Prone Software Modules," *Software Quality Journal*, vol. 11, pp. 19-37, 2003.

[13] T. M. Khoshgoftaar and E. B. Allen, "Predicting the Order of Fault-Prone Modules in Legacy Software," *Proc. of The Ninth International Symposium on Software Reliability Engineering*, Paderborn, Germany, 1998, pp. 344-353.

[14] S. Kim, K. Pan, and E. J. Whitehead, Jr., "When Functions Change Their Names: Automatic Detection of Origin Relationships," *Proc. of 12th Working Conference on Reverse Engineering (WCRE 2005)*, Pittsburgh, PA, USA, 2005, pp. 143-152.

[15] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead, Jr., "Automatic Identification of Bug Introducing Changes," *Proc. of International Conference on Automated Software Engineering (ASE 2006)*, Tokyo, Japan, 2006.

[16] A. J. Ko and B. A. Myers, "A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems," *Journal of Visual Languages and Computing*, vol. 16, pp. 41-84, 2005.

[17] A. Mockus and L. G. Votta, "Identifying Reasons for Software Changes Using Historic Databases," *Proc. of International Conference on Software Maintenance (ICSM 2000)*, San Jose, California, USA, 2000, pp. 120-130.

[18] A. Mockus and D. M. Weiss, "Predicting Risk of Software Changes," *Bell Labs Technical Journal*, vol. 5, pp. 169-180, 2002.

[19] N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," *Proc. of 2005 Int'l Conference on Software Engineering (ICSE 2005)*, Saint Louis, Missouri, USA, 2005, pp. 284-292.

[20] N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures," *Proc. of 2006 Int'l Conference on Software Engineering (ICSE 2006)*, Shanghai, China, 2006, pp. 452-461.

[21] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the Location and Number of Faults in Large Software Systems," *IEEE Transactions on Software Engineering*, vol. 31, pp. 340-355, 2005.

[22] J. Śliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?," *Proc. of Int'l Workshop on Mining Software Repositories (MSR 2005)*, Saint Louis, Missouri, USA, 2005.

[23] J. Śliwerski, T. Zimmermann, and A. Zeller, "HATARI: Raising Risk Awareness. Research Demonstration," *Proc. of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005)*, Lisbon, Portugal, 2005, pp. 107-110.

[24] P. Weißgerber and S. Diehl, "Identifying Refactorings from Source-Code Changes," *Proc. of International Conference on Automated Software Engineering (ASE 2006)*, Tokyo, Japan, 2006, pp. 231-240.

[25] T. Zimmermann and P. Weißgerber, "Preprocessing CVS Data for Fine-Grained Analysis," *Proc. of Proc. Intl. Workshop on Mining Software Repositories (MSR)*, Edinburgh, Scotland, 2004.

[26] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," *IEEE Trans. Software Eng.*, vol. 31, pp. 429-445, 2005.