Predicting Concurrency Bugs: How Many, What Kind and Where Are They?

Bo Zhou Iulian Neamtiu Rajiv Gupta
Department of Computer Science and Engineering
University of California Riverside, CA, USA
{bzhou003, neamtiu, gupta}@cs.ucr.edu

ABSTRACT

Concurrency bugs are difficult to find and fix. To help with finding and fixing concurrency bugs, prior research has mostly focused on static or dynamic analyses for finding specific classes of bugs. We present an approach whose focus is understanding the differences between concurrency and non-concurrency bugs, the differences among various concurrency bug classes, and predicting bug quantity, type, and location, from patches, bug reports and bug-fix metrics. First, we show that bug characteristics and bug-fixing processes vary significantly among different kinds of concurrency bugs and compared to non-concurrency bugs. Next, we build a quantitative predictor model to estimate concurrency bugs appearance in future releases. Then, we build a qualitative predictor that can predict the type of concurrency bug for a newly-filed bug report. Finally, we build a bug location predictor to indicate the likely source code location for newly-reported bugs. We validate the effectiveness of our approach on three popular projects, Mozilla, KDE, and Apache.

1. INTRODUCTION

Concurrent programming is challenging, and concurrency bugs are particularly hard to diagnose and fix for several reasons, e.g., thread interleavings and shared data complicate reasoning about program state [17], and bugs are difficult to reproduce due to non-determinism and platform-specific behavior. As a result, we show that fixing concurrency bugs takes longer, requires more developers, and involves more patches, compared to fixing non-concurrency bugs.

Many recent efforts have focused on concurrency bugs, with various goals. On one side, there are empirical studies about the characteristics and effects of concurrency bugs [8, 15, 25], but they do not offer a way to predict bug quantity, type and location. On the other side, static [7] or dynamic analyses [16] aim to detect particular types of concurrency bugs. Such analyses help with precise identification of bugs in the current source code version, but their focus is different—finding specific type of bugs in the current version, rather than using evolution data to predict the future number, kind, and location of bugs; in addition, program analysis is subject to scalability constraints which are particularly acute in large projects. While other prior efforts have introduced models for predicting bug quantity [11, 21] and bug location [19, 23] without regard to a specific bug category, we are specifically interested in isolating concurrency bugs and reporting prediction strategies that work well for them. Hence in this paper we study the nature of concurrency bugs, how they differ from non-concurrency bugs, and how to effectively predict them; we use statistics and machine learning as our main tools.

Our study analyzes the source code evolution and bug repositories of three large, popular open-source projects: Mozilla, KDE and Apache. Each project has had a history of more than 10 years, and their size has varied from 110 KLOC to 14,330 KLOC. Such projects benefit from our approach for several reasons: (1) large code bases pose scalability, coverage and reproducibility problems to static and dynamic analyses; (2) large collaborative projects where bug reporters differ from bug fixers benefit from predictors that help fixers narrow down the likely cause and location of a bug reported by someone else; (3) a quantitative predictor for estimating the incidence of concurrency bugs in next releases can help with release planning and resource allocation.

We now summarize our methodology and findings. By analyzing bug reports, commit logs and source code, we found that concurrency bugs fall into four categories: atomicity violations, order violations, races, and deadlocks (Section 2). Next, for each bug type, we analyzed multiple facets (e.g., patches, files, comments, and developers involved) to characterize the process involved in, and differences between, fixing concurrency and non-concurrency bugs. We found that *compared to non-concurrency bugs, concurrency bugs take twice as long to fix, bug-fixes affect 46% more files, require patches that are 4 times larger, involve 17% more developers, involve 72% more patches for a successful fix, and have 17% higher severity.* Within concurrency bugs, we found that *atomicity violations are the most complicated bugs, taking highest amounts of time, developers and patches to fix, while deadlocks are the easiest and fastest to fix* (Section 4).

Using the historic values of these bug characteristics, we construct two models to predict the number of extant concurrency bugs that will have to be fixed in future releases: a model based on generalized linear regression (Section 5.1), and one based on time series forecasting (Section 5.2). Our predicted number of concurrency bugs differed very little from the actual number: *depending on the bug type, our*

EASE '15, April 27 - 29, Nanjing, China

Thread 1	Thread 2

`apr_atomic_dec(&obj->refcount);`
 	
`if (!obj->refcount) {`
`  cleanup_cache_object(obj);`
`}`
	`apr_atomic_dec(&obj->refcount);`
	`if (!obj->refcount) {`
	`  cleanup_cache_object(obj);`
	`}`

Figure 1: Atomicity violation bug #21287 in Apache (`mod_mem_cache.c`).

Thread 1	Thread 2

`nsThread::Init(...){`
`  ...`
 	
`  mThread = PR_CreateThread(Main,...);`
`  ...`
	`nsThread::Main(...){`
	`  ...`
	`  mState = mThread->GetState(...);`
	`  ...`

Figure 2: Order violation bug #61369 in Mozilla (`nsthread.cpp`).

prediction was off by just 0.17–0.54 bugs.

While these quantitative predictors provide *managers* an estimate of the number of upcoming concurrency bugs, to help *developers* we constructed two additional, qualitative predictors. First, a *bug type* predictor that can predict the likely type of a newly-filed bug concurrency bug, e.g., atomicity violation or deadlock *with at least 63% accuracy* (Section 6). Second, a *bug location* predictor that predicts the likely bug location from a new bug report with *at least 22% Top-1 accuracy and 55% Top-20% accuracy* (Section 7).

2. CONCURRENCY BUG TYPES

We now briefly review the four main types of concurrency bugs, as introduced by previous research [7, 15, 17].

Atomicity violations result from a lack of constraints on the interleaving of operations in a program. Atomicity violation bugs are introduced when programmers assume some code regions to be atomic, but fail to guarantee the atomicity in their implementation. In Figure 1 we present an example of an atomicity violation, bug #21287 in Apache: accesses to variable obj in function decrement_refcount are not protected by a lock, which causes the obj to be freed twice.

Order violations involve two or more memory accesses from multiple threads that happen in an unexpected order, due to absent or incorrect synchronization. An order violation example, bug #61369 in Mozilla, is shown in Figure 2: nsThread::Main() in Thread 2 can access mThread's state before it is initialized (before PR_CreateThread in Thread 1 returns).

Data races occur when two different threads access the same memory location, at least one access is a write, and the accesses are not ordered properly by synchronization.

Deadlocks occur when two or more operations circularly wait for each other to release acquired resources (e.g., locks).

3. METHODOLOGY

We now present an overview of the three projects we examined, as well as the methodology we used for identifying and analyzing concurrency bugs and their bug-fix process.

3.1 Projects Examined

We selected three large, popular, open source projects for our study: Mozilla, KDE and Apache. The Mozilla suite is an open-source web client system implementing a web browser, an email client, an HTML editor, newsreader, etc. Mozilla contains many different sub-projects, e.g., the Firefox web browser, and the Thunderbird mail client. In this paper, we mainly focus on the core libraries, and the products related to the Firefox web browser. KDE is a development platform, a graphical desktop, and a set of applications in diverse categories. Apache is the most widely-used web server; we analyzed the HTTP server and its supporting library, APR, which provides a set of APIs that map to the underlying operating system. The evolution time span and source code size are presented in Table 1.

We focus on these three projects for several reasons. First, their long evolution (more than 10 years), allow us to observe the effect of longer or shorter histories on prediction accuracy. Second, they are highly concurrent applications with rich semantics, and have large code bases, hence predicting bug type and location is particularly helpful for finding and fixing bugs. Finally, given the popularity of their core components that constitute the object of our study, finding and fixing concurrency bugs is a key priority for these projects. We believe that these characteristics make our chosen projects representative of some of the biggest challenges that the software development community faces as complex applications become more and more concurrent.

3.2 Identifying Concurrency Bugs

We now describe the process for collecting concurrency bugs and computing the attributes of their bug-fixing process. All three projects offer public access to their bug trackers [1, 10, 20]. We first selected the fixed bugs; then we split the fixed bugs into concurrency and non-concurrency bugs using a set of keywords and cross-information from the commit logs; finally we categorized the concurrency bugs into the four types. Our process is similar to previous research [8, 15]. Potential threats to the validity of our process will be discussed in Section 8. The results, explained next, are given in Table 1.

Identifying "true" bugs. To identify the viable bug report candidates, we only considered bugs that have been confirmed and fixed; that is, we only selected bug reports marked with resolution `FIXED` and status `CLOSED`. We did not consider bugs with other statuses, e.g., `UNCONFIRMED` and other resolutions (e.g., `INVALID`, or `WONTFIX`)—the reason is that for bugs other than `FIXED` and `CLOSED`, the bug reports did not have detailed information and discussions also in general, they will not contain correct patches. Without reasonably complete bug reports it would be impossible for us to completely understand the root cause of the bugs.

We limited the searching process to those parts of the project with a long history and large source code base. For Mozilla, we only selected 8 products which are directly related to the core and browser parts: *Core, Firefox, Directory, JSS, NSPR, NSS, Plugins*, and *Rhino* [4]. For Apache, we only chose C/C++ products: the *Apache HTTP Server* and the *Apache Portable Runtime*. For KDE, we considered all products in the KDE Bugtracking System. The 5th column of Table 1 shows the number of bugs that have been fixed and closed.

Candidate concurrency bug reports. As the 5th column of Table 1 shows, there were more than 250,000 bugs left after the previous step. To make our study feasible, we automatically filtered bugs that were not likely to be relevant to concurrency by performing a search process on the bug report database. We retained reports that contained a keyword from our list of relevant concurrency terms; the

Table 1: Bug reports and concurrency bug counts.

Project	Time span	SLOC		Bug reports		Concurrency Bugs				
		First release	Last release	Fixed & closed	Matching concurrency keywords	Atomicity violation	Order violation	Data race	Deadlock	*Total*
Mozilla	1998-2012	1,459k	4,557k	114,367	1,149	60	34	10	30	*134*
KDE	1999-2012	956k	14,330k	118,868	887	14	13	29	19	*75*
Apache	2000-2012	110k	223k	22,370	423	18	8	10	5	*41*
				Total	2,459	*92*	*55*	*49*	*54*	*250*

list included terms such as "thread", "synchronization", "concurrency", "mutex", "atomic", "race", "deadlock". Note that similar keywords were used in the previous research [8, 15]. In Table 1 (6th column), we show the number of bug reports that matched these keywords. Many bugs, however, are mislabeled, as explained shortly. Our keyword-based search for bug reports could have false negatives, i.e., missing some of the real concurrency bugs (which we identify as a threat to validity in Section 8). However, we believe that a concurrency bug report that did not contain any of the aforementioned keywords is more likely to be incomplete and much more difficult to analyze its root cause.

Determining the concurrency bug type. We then manually analyzed the 2,459 bug reports obtained in the previous step to determine (1) whether they describe an actual concurrency bug and if yes, (2) what is the concurrency bug type. In addition to the bug description, some reports also contain execution traces, steps to reproduce, discussion between the developers about how the bug was triggered, fix strategies, and links to patches. We used all these pieces of information to determine whether the bug was a concurrency bug and its type.

For all bugs we identified as concurrency bugs, we analyzed their root cause and fix strategy, and binned the bug into one of the four types described in Section 2. In the end, we found 250 concurrency bugs: 134 in Mozilla, 75 in KDE, and 41 in Apache; the numbers for each category are presented in the last set of grouped columns of Table 1; the Mozilla and Apache numbers are in line with prior findings by other researchers, though their study has analyzed bugs up to 2008 [15]. We found that, in Mozilla and Apache, atomicity violations were the most common, while in KDE, data races were the most frequent.

Note that searching bug reports alone is prone to false positives (incorrectly identifying a non-concurrency bug as a concurrency bug) and false negatives (missing actual concurrency bugs) [8]. To reduce the incidence of such errors, we also keyword-searched the commit logs (e.g., CVS and Mercurial for Mozilla) and then cross-referenced the information obtained from the bug tracker with the information obtained from the commit logs. For instance, Mozilla bug report #47021, did not contain any of the keywords but we found the keyword *race* in the commit log associated with the bug, so based on this information we added it to our set of concurrency bugs to be categorized.

Concurrency bug types and keywords. We found many bug reports that contained keywords pertaining to other types of bugs. The following table shows the percentage of bug reports in each category (computed after our manual categorization) containing each of the four keywords that one would naturally associate with the corresponding bug type.

Keyword	Percentage of bug reports containing the keyword			
	Atomicity bug reports	Order bug reports	Race bug reports	Deadlock bug reports
"atomic"	29.35	14.54	8.16	1.85
"order"	21.74	40.00	16.33	14.81
"race"	70.65	63.64	51.02	7.41
"deadlock"	16.30	16.36	12.24	94.44

Note how 70.65% of the atomicity violation reports contain the keyword "race", while only 29.35% contain the keyword "atomic". In fact, a higher percentage of atomicity violation and order violation bug reports contain the term "race" (70.65% and 63.64%, respectively) compared to the race bug reports (51.02%). These findings suggest that: (1) while searching for concurrency bug reports using an exhaustive keyword list followed by manual analysis has increased our effort, it was essential for accurate characterization, as many bugs contain misleading keywords, and (2) using approaches that can assign weights to, and learn associations between, keywords, are likely to be promising in automatically classifying bug types—we do exactly that in Section 6.

3.3 Collecting Bug-fix Process Data

To understand the nature of, and differences in, bug-fixing processes associated with each concurrency bug type, we gathered data on bug features—the time, patches, developers, files changed, etc., that are involved in fixing the bugs. We now provide details and definitions of these features.

Patches represents the number of patches required to fix the bug; we extract it from the bug report. *Days* represents the time required to fix the bug, computed as the difference between the date the bug was opened and the date the bug was closed. *Files* is the number of files changed in the last successful patch. We extracted the number of files changed by analyzing the bug report and the commit information from the version control system. *Total patch size* indicates the combined size of all patches associated with a bug fix, in KB. *Developers* represents the number of people who have submitted patches. *Comments* is the number of comments in the bug report. *Severity:* to capture the impact that the bug has on the successful execution of the applications, our examined projects use a numerical scale for bug severity. Mozilla uses a 7-point scale[1], while KDE and Apache use 8-point scales. To have a uniform scale, we mapped KDE[2] and Apache[3] severity levels onto Mozilla's.

4. QUANTITATIVE ANALYSIS OF DIFFERENCES IN BUG-FIXING PROCESSES

Prior work has found that fixing strategies (that is, code changes) differ widely among different classes of concurrency bugs [15]; however, their findings were qualitative, rather than quantitative. In particular, we would like to be able to answer questions such as: *Which concurrency*

[1] 0=Enhancement, 1=Trivial, 2=Minor, 3=Normal, 4=Major, 5=Critical, 6=Blocker.

[2] 0=Task, 1=Wishlist, 2=Minor, 3=Normal, 4=Crash, 4=Major, 5=Grave, 6=Critical.

[3] 0=Enhancement, 1=Trivial, 2=Minor, 3=Normal, 4=Major, 4=Regression, 5=Critical, 6=Blocker.

fixes require changes across multiple files? Do atomicity violation fixes require more patches to "get it right" compared to deadlock fixes? Which concurrency bug types take the longest to fix? Are concurrency bugs more severe than non-concurrency bugs? Do concurrency bugs take longer to fix than non-concurrency bugs?

Therefore, in this section we perform a quantitative assessment of the bug-fix process for each concurrency bug type, as well as compare concurrency and non-concurrency bugs, along several dimensions (features). While bug-fixing effort is difficult to measure, the features we have chosen provide a substantive indication of the developer involvement associated with each type of bug. Moreover, this assessment is essential for making inroads into predicting the number of concurrency bugs in the code that are yet to be discovered.

To compare concurrency bugs and non-concurrency bugs, we randomly selected 250 non-concurrency bugs found and fixed in the same product, component, software version and milestone with the 250 concurrency bugs we found. The reason why we used the same product/component/version/milestone for concurrency and non-concurrency bugs was to reduce potential confounding factors. We manually validated each non-concurrency bug and bug report for validity, as we did with concurrency bugs.

4.1 Feature Distributions

We now present our findings. For each feature, in Figure 3 we show a boxplot indicating the distribution of its values for each concurrency bug type. Each boxplot represents the minimum, first quartile, third quartile and maximum values. The black horizontal bar is the median and the red diamond point is the mean. The second-from-right boxplot shows the distribution across all concurrency bugs. The rightmost boxplot shows the distribution for non-concurrency bugs. For legibility and to eliminate outliers, we have excluded the top 5% and bottom 5% when computing and plotting the statistical values. We now discuss each feature.

Patches are one of the most important characteristics of bug fixing. Intuitively, the number of patches could be used to evaluate how difficult the bugs are—the more patches required to "get it right," the more difficult it was to fix that bug. We found (Figure 3) that atomicity violations take the highest number of patches (usually 2–5, on average 3.5), while order violations take on average 2.4 patches, followed by races at 1.7 patches and deadlocks at 1.6 patches. Non-concurrency bugs require on average 1.4 patches.

Days. Predicting the bug-fix time is useful for both concurrency bugs and non-concurrency bugs, as it helps managers plan the next releases. We found (Figure 3) that the average bug-fix time is longer than 33 days for all 4 types of concurrency bugs, which means that usually the time cost associated with concurrency bugs is high. Similar to the number of patches, atomicity violation bugs took the longest to fix (123 days on average), order violations and races took less (66 days and 44 days, respectively), while deadlocks were fixed the fastest (33.5 days on average). Non-concurrency bugs take on average 34 days.

Files. This characteristic can be used to estimate the extent of changes and also the risk associated with making changes in order to fix a bug—the higher the number of affected files, the more developers and inter-module communications are affected. We found (Figure 3) that bug fixes affect on average 2.8 files for atomicity, 2.4 files for order violations, 1.9 files for races and 1.8 files for deadlocks.

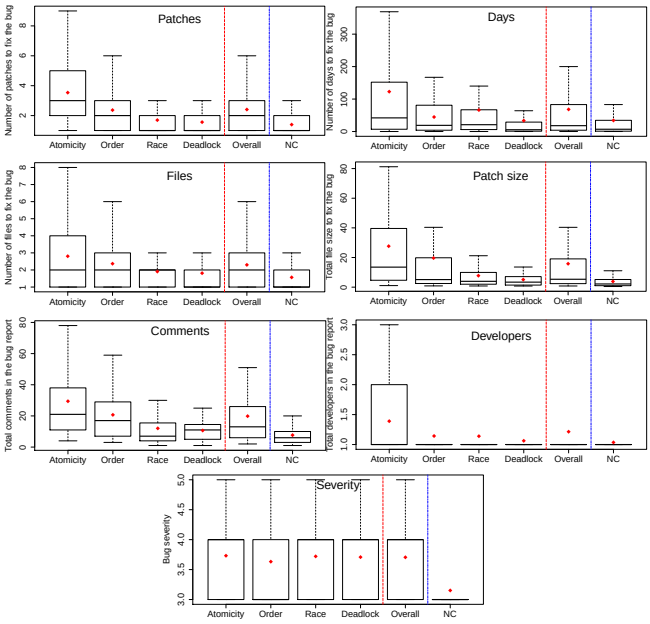

Figure 3: Feature distributions for each class of concurrency bugs (Atomicity, Order, Race, Deadlock), all concurrency bugs combined (Overall) and non-concurrency bugs (NC).

Non-concurrency bugs affect on average 1.6 files.

Total patch size. The total size of all patches, just like the number of files, can be used to indicate the risk associated with introducing the bug-fixing changes: if the size of the patches is large, many modifications have been made to the source code (e.g., pervasive changes, large-scale restructuring). We found that average concurrency patch sizes tend to be large, with atomicity (27.6KB) and order violations (19.7KB) far ahead of races (7.7KB) and deadlocks (5.1KB). Non-concurrency patches are smaller, 3.8KB on average.

Comments. The number of comments in the report can indicate hard-to-find/hard-to-fix bugs that developers solicit a lot of help with. Examples of such bugs that are hard to reproduce and fix include Mozilla bugs #549767, #153815, #556194, where even after removing the "mark as duplicate" comments, there are more than 100 comments dedicated to reproducing and fixing the bug. We found that the average number of comments is again highest for atomicity violations (29.4) followed by order violations (20.7), races (12.0) and deadlocks (10.6). The number is much smaller for non-concurrency bugs (7.6).

Developers. The more developers are involved into submitting patches for a bug, the more difficult it was to find and fix that bug. We found that atomicity fixes involve on average 1.39 developers while the other bugs involve fewer developers (1.21). Non-concurrency bugs involve, on average, 1.03 developers.

Severity. Bug severity is important as developers are more concerned with higher severity bugs which inhibit functionality and use. We found that all types of concurrency bugs have average severity between 3.6 and 3.7. Since severity level 3 is Normal and level 4 is Major, we can infer that concurrency bugs are higher-priority bugs. Non-concurrency bugs tend to be lower severity (mean 3.1), which underlines the importance of focusing on concurrency bugs.

4.2 Differences Among Concurrency Bugs

We now set out to answer another one of our initial questions: *Are there significant differences in the bug-fix process*

Table 2: Wilcoxon Rank Sum and Signed Rank Tests results; p-values were adjusted using the FDR procedure; ** indicates significance at $p = 0.01$ while * at $p = 0.05$.

Features	Category	Order	Race	Deadlock
Patches	Atomicity	0.0116*	<0.0001**	<0.0001**
	Order		0.0168*	0.0084**
	Race			0.8427
Days	Atomicity	0.0790	0.1728	0.0002**
	Order		0.6302	0.0790
	Race			0.0270*
Files	Atomicity	0.4654	0.1215	0.0868
	Order		0.6378	0.5034
	Race			0.7363
Patch size	Atomicity	0.0079**	<0.0001**	<0.0001**
	Order		0.2666	0.0412*
	Race			0.2814
Comments	Atomicity	0.0367*	<0.0001**	<0.0001**
	Order		0.0034**	0.0063**
	Race			0.5370
Developers	Atomicity	0.0110*	0.0110*	0.0005**
	Order		0.9690	0.2695
	Race			0.2695
Severity	Atomicity	0.7933	0.7933	0.9228
	Order		0.7933	0.7933
	Race			0.7933

among different categories of concurrency bugs?

To answer this question we performed a pairwise comparison across all pairs of concurrency bug types. For generality and to avoid normality assumptions, we performed the comparison via a non-parametric test, the Wilcoxon signed-rank test. To avoid type I errors, we performed a Wilcoxon signed-rank test by applying *false discovery rate* (FDR) procedures [3]. We present the results, obtained after the correction, in Table 2. The starred values indicate significance at $p = 0.01$ (**) and $p = 0.05$ (*), respectively. We found that atomicity and deadlock tend to be significantly different from the other categories, while for order and race, it depends on the feature. We also found that bug severity does not differ significantly among concurrency bug types.

4.3 Discussion

Concurrency bugs v. non-concurrency bugs. We found significant differences *for all these features* between concurrency and non-concurrency bugs. In Figure 3, the last two boxplots in each graph show the distribution of values for that feature for all concurrency bugs (Overall) and non-concurrency bugs (NC). We found that, compared to non-concurrency bugs, concurrency bugs involve 72% more patches for a successful fix, take twice as long to fix, bug-fixes affect 46% more files, require patches that are four times larger, generate 2.5 times as many comments, involve 17% more developers, and have a 17% higher severity.

For each feature, the differences between concurrency and non-concurrency bugs are significant ($p < 2e^{-16)}$); we used Cliff's delta to compute the effect size measure; the results indicated significance (effect size *Large* for all features except severity, where they were *Medium*). For brevity, we omit presenting the individual results.

Differences among types. Based on our findings, we infer that (1) concurrency bugs are usually difficult to find the root cause of and get the correct fix for, and (2) there are significant differences between different types of concurrency bugs hence these types should be considered separately. These two points provide the impetus for the work presented in the remainder of the paper.

5. PREDICTING THE NUMBER OF CONCURRENCY BUGS

Costs associated with software evolution are high, an estimated 50%–90% of total software production costs [12, 24]. Predicting the number of extant bugs, that will have to be fixed in upcoming releases, helps managers with release planning and resource allocation, and in turn can reduce software evolution costs. Therefore, in this section we focus on predicting the future number of concurrency bugs.

In Section 4, we observed relationships between the number of concurrency bugs and the features we analyzed. Hence, to estimate the likelihood of concurrency bugs in the project, we naturally turn to using the features as inputs. In this section we focus on (1) understanding the effect of each feature on each type of concurrency bug, as well as its prediction power for the number of those concurrency bugs, (2) using the features to build predictor models and evaluating the accuracy of the models, and (3) understanding the effect of time and autocorrelation on prediction accuracy. In particular, we explore two predictors models—one based on multiple linear regression and one based on time series.

Time granularity. There is an accuracy–timeliness tradeoff in how long a window we use for bug prediction. A time frame too short can be susceptible to wild short-term variations or lack of observations, while too long a time frame will base predictions on stale data. Therefore, we built several models, with varying time spans, for computing past values of independent variables and bug numbers. In the first model, named *Monthly*, we counted the dependent variable (number of bugs) and independent variables (patches, days, files, patch size, etc.) at a monthly granularity based on the open date of the bugs, e.g., one observation corresponds to May 2010, the next observation corresponds to June 2010, and so on. We also tried coarser granularities, 3-months, 6-months, and 12-months, but the predictions were less accurate (albeit slightly). Therefore, in the remainder of this section, monthly granularity is assumed.

5.1 Generalized Linear Regression

To analyze the relationship between the number of concurrency bugs and each feature, we built a *generalized linear regression* model to avoid the normality assumption. We choose the number of bugs as dependent variable and the features, i.e., *patches*, *days*, *files*, *patch size*, *comments*, *developers*, and *severity*, as independent variables.

In Table 3 we present the regression results for each type of concurrency bug and across all concurrency bugs (again, this is using the monthly granularity). For each independent variable, we show the regression coefficient and the p-value, that is, the significance of that variable. We found that not all independent variables contribute meaningfully to the model. For example, for data race bugs, *files*, *patch size*, *developers* and *severity* are good predictors (low p-value), but the other features are not; moreover, the regression coefficients for files, developers and severity are positive. Intuitively, these results indicate that past changes to files, high number of developers and high bug severity are correlated with a high incidence of data race bugs later on; since the coefficient for patch size is negative, it means that past patches will actually *reduce* the incidence of data races in upcoming releases. For atomicity violations and order violations we have similar results. When predicting the number of all concurrency bugs (last two columns in Table 3), we found that three variables contribute to the model: *files*, *developers* and *severity*.

Table 3: Results of the generalized regression model; ** indicates significance at $p = 0.01$; * indicates significance at $p = 0.05$.

Features	Atomicity violation		Order violation		Data race		Deadlock		All conc.	
	coefficient	p-value	coefficient	p-value	coefficient	p-value	coefficient	p-value	coefficient	p-value
Patches	0.0196	0.014*	0.0197	0.094	0.0171	0.173	-0.0090	0.607	0.0032	0.709
Days	-0.0001	0.004**	<0.0001	0.912	<0.0001	0.360	<0.0001	0.950	-0.0001	0.084
Files	0.0147	0.001**	0.0321	<0.001**	0.0161	0.002**	0.0313	<0.001**	0.0136	0.003**
Patch size	-0.0011	<0.001**	-0.0011	<0.001**	-0.0038	<0.001**	0.0017	0.141	-0.0004	0.122
Comments	-0.0006	0.176	-0.0017	0.003**	0.0008	0.126	0.0017	0.049*	-0.0004	0.354
Developers	0.1517	<0.001**	0.2035	<0.001**	0.1847	<0.001**	0.2086	<0.001**	0.1757	<0.001**
Severity	0.1257	<0.001**	0.1055	<0.001**	0.1173	<0.001**	0.1035	<0.001**	0.1331	<0.001**
Pseudo R^2	*0.9388*		*0.9572*		*0.9701*		*0.9653*		*0.9165*	

Table 4: Summary of stepwise regression model.

Bug category \ Independent variables	Patches	Days	Files	Patch size	Comments	Developers	Severity
Atomicity	✓	✓	✓	✓	✓	✓	✓
Order	✓		✓	✓	✓	✓	✓
Race	✓		✓	✓	✓	✓	✓
Deadlock	✓		✓	✓	✓	✓	✓
All concur.		✓	✓	✓	✓	✓	✓

Finally, we used the Cox & Snell pseudo R^2 to measure how well the model fits the actual data—the bigger the R^2, the larger the portion of the total variance in the dependent variable that is explained by the regression model and the better the dependent variable is explained by the independent variables. We show the results in the last row of Table 3; the results indicate high goodness of fit, 0.91–0.97, which confirms the suitability of using the model to predict the number of concurrency bugs based on feature values.

Finding parsimonious yet effective predictors. To balance prediction accuracy with the cost of the approximation and avoid overfitting, we looked for more parsimonious models that use fewer independent variables. We applied backward stepwise regression, a semi-automated process of building a model by successively adding or removing independent variables based on their statistical significance, then computing the Akaike Information Criterion (AIC) for finding the important variables. Table 4 shows the result of stepwise regression; we use '✓' to mark the independent variables that should be used when constructing predictor models. For example, for races, we can still get a good prediction when eliminating the *days* feature.

5.2 Times Series-based Prediction

Since our data set is based on time series, and prior work has found bug autocorrelation (temporal bug locality [11]), we decided to investigate the applicability of time series forecasting techniques for predicting concurrency bugs. In particular, we used ARIMA (Autoregressive integrated moving average), a widely-used technique in predicting future points in time series data, to build a concurrency bug prediction model. In a nutshell, given a time series with t observations $X_1, \ldots, X_t$ and error terms $\varepsilon_1, \ldots, \varepsilon_t$, an ARIMA model predicts the value of an output variable $\hat{X}_{t+1}$ at time step $t+1$; that is, $\hat{X}_{t+1} = f(X_1, \ldots, X_t, \varepsilon_1, \ldots, \varepsilon_t)$. Note that prior values for X, i.e., $X_1, \ldots, X_t$ are part of the model, hence the term "autocorrelation". The quality of the model is measured in terms of goodness of fit (adjusted R^2) and other metrics such as RMSE—the root mean squared error between the predicted ($\hat{X}_t$) and actual (X_t) values.

Concretely, we constructed ARIMA predictor models for each bug class. In each case $X_1, \ldots, X_t$ were the number of bugs; $\varepsilon_1, \ldots, \varepsilon_t$ were the values of independent variables (patches, days, files, etc.); and $\hat{X}_1 \ldots, \hat{X}_t$ were the predicted values; the differences between X_i and $\hat{X}_i$ were used when computing the prediction accuracy. For example, if $X_{May\ 2010}$ was the actual number of atomicity bugs in May 2010, then the time series model used $X_{April\ 2010}$, $X_{March\ 2010}$, ..., as lagged (true) values; $Patches_{April\ 2010}$, $Patches_{March\ 2010}$, ..., $Days_{April\ 2010}$, $Days_{March\ 2010}$, ..., $Files_{April\ 2010}$, $Files_{March\ 2010}$, ..., etc. as error terms; and $\hat{X}_{May\ 2010}$, $\hat{X}_{April\ 2010}$, ... as predicted values.

Table 5: Time series based prediction model result.

Bug category	RMSE	Adjusted R^2	ARIMA parameter
Atomicity	0.3485	0.8626	ARIMA(1,1,1)
Order	0.1947	0.9149	ARIMA(0,0,0)s
Race	0.1832	0.9285	ARIMA(1,0,0)
Deadlock	0.1728	0.9244	ARIMA(2,0,0)s
All conc.	0.5400	0.8965	ARIMA(0,0,0)

Table 5 shows the ARIMA results for each concurrency bug type and each time granularity; we performed this analysis using the R toolkit. The first column shows the concurrency bug type, the second column shows the root mean square error (the lower, the better); the third column shows the goodness of fit R^2; the last column shows the ARIMA parameter that was automatically chosen by R as best-performing model. ARIMA has three parameters: (p, d, q) where p is the autoregressive (AR) parameter, d is the number of differencing passes and q is the moving average (MA) parameter; put simply, p and q indicate the number of past samples involved in the prediction. For example, for Atomicity, the best model was ARIMA(1,1,1) meaning it got best results when $\hat{X}_t$ was computed using just the prior observation X_{t-1} and the prior error term ε_{t-1}, and differencing once. The time series analysis has also found *seasonal* patterns in the bug time series. Such entries are marked with a trailing 's', e.g., for Order and Deadlock bugs. In all cases, the season length was determined to be 12 months—this is not surprising, given that certain projects follow a fixed release cycle; we leave further investigation of seasonal patterns to future work. We observed that the RMSE is low for our data sets—the typical difference between the predicted and actual bug numbers was 0.17–0.54, depending on the bug type. To illustrate the accuracy of time series-based prediction, in Figure 4 we show the predicted (blue, triangle marks) and actual (red, round marks) time series for the total number of concurrency bugs each month.

Discussion. We now discuss why multiple models are needed. ARIMA is based on autocorrelation, that is, it works well when the current value X_t and the lagged value X_{t-l} are *not* independent. While accurate in helping managers forecast the number of bugs, the autoregressive nature of ARIMA models in a sense espouses the time locality of concurrency bugs: if the prior release was buggy, the next release is likely to be buggy, too—in that case the managers can delay the next release to allow time for finding and fixing the bugs.

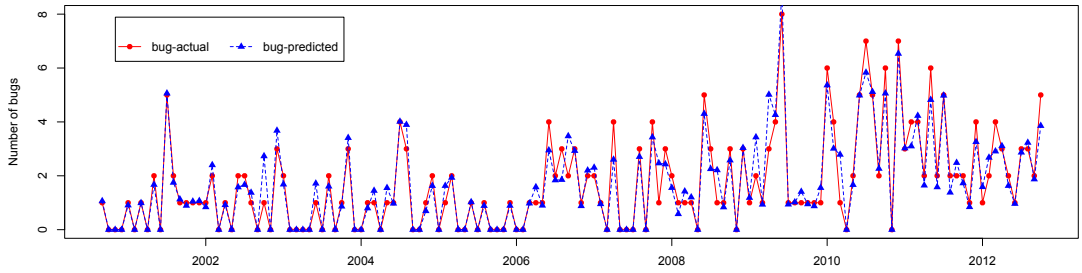

Figure 4: Time series of predicted and actual numbers of concurrency bugs each month.

However, at the risk of stating the obvious, the managers cannot control the *past* number of bugs, but by examining the model and the non-autoregressive features (number of patches, files, developers, etc.) they can adjust the software process so that future values of the non-autoregressive features will permit the number of bugs to decrease. It is also up to the project managers to decide whether a one-month horizon is enough for release planning, or longer horizons (e.g., 3- or 6-months) would be more suitable.

6. PREDICTING THE TYPE OF CONCURRENCY BUGS

When a new bug report has been filed and is examined, determining the nature of the bug is essential for a wide range of software engineering tasks, from bug triaging to knowing whom to toss a bug to [9], to finding the root cause and eventually fixing the bug.

In particular for concurrency bugs, the root causes and fixing strategies can vary widely among bug categories (Section 2). Therefore, when a concurrency bug report is filed, it is essential that the developers determine its category in order to speed up the fixing process. To support this task, using the categorized bugs reports described in Section 4, we built a predictor model that, given a newly-filed bug report, predicts which type it is: atomicity violation, order violation, deadlock or race. We next describe the approach and then the results.

6.1 Approach

The approach is based on machine learning, i.e., classifiers that use relevant keywords extracted from bug reports as input features and learn the association between keywords and specific concurrency bug types based on a Training Data Set (TDS). Next, we verify the prediction accuracy by presenting the classifier with validation inputs and comparing the classifier output with the true output; the set of bug reports used for validation is called a Validation Data Set (VDS). We now describe the predictor construction process.

Textual data preparation. We applied standard information retrieval techniques to extract relevant keywords from bug reports: we used Weka to transform bug reports from the textual description available in the bug tracker into a set of keywords usable by the classifier[4] and build our TDS.

Classifier choice. After preparing the TDS, the next step was to train the classifier and validate the learned model. We use Weka's built-in Naïve Bayes, Bayesian Network, C4.5 and Decision Table classifiers in our approach.

Training and validation. As shown in our prior work [5], using a larger dataset for training bug classifiers does not necessarily yield better results; in fact, training a classifier with old samples can *decrease* prediction accuracy as the classifier is trained with stale input-output pairs that do not match the current project state.

[4]To extract keywords from bug reports, we employed TF-IDF, stemming, stop-word and non-alphabetic word removal [18], using Weka's `StringtoWordVector` class.

Table 6: Accuracy of bug category prediction, in percents; highest accuracy indicated in bold.

Training set	Validation set	Classifier			
		Naïve Bayes	Bayesian Net	C4.5	Decision Table
All	Mozilla	39.39	**63.64**	45.45	60.61
	KDE	37.50	**56.25**	**56.25**	31.25
	All projects	38.00	**60.00**	48.00	50.00
2004+	Mozilla	50.00	60.00	40.00	**63.33**
	KDE	63.64	**72.73**	36.36	**72.73**
	All projects	52.38	61.90	38.10	**66.67**
Mozilla	Mozilla	60.00	**65.00**	55.00	50.00
KDE	KDE	46.67	**60.00**	**60.00**	53.33
Apache	Apache	57.14	**71.43**	**71.43**	**71.43**

Table 7: Detailed result of the Bayesian Net classifier.

Training /Validation set	Bug category	Evaluation measure		
		precision	recall	F-measure
All /All projects	Atomicity	0.647	0.524	0.579
	Order	0.600	0.429	0.500
	Race	0.429	0.750	0.545
	Deadlock	0.778	1.000	0.875
2004+ /All projects	Atomicity	0.625	0.625	0.625
	Order	0.571	0.333	0.421
	Race	0.545	0.750	0.632
	Deadlock	0.750	1.000	0.857

To quantify the effect of recent vs. old training samples, we constructed two bug training/validation sets: one set, referred to as *All*, contained all the concurrency bug reports since project inception (that is 1998–2012 for Mozilla, 1999–2012 for KDE, and 2000–2012 for Apache); the other set, referred to as *2004+*, contained only more recent samples, i.e., bug reports from 2004–2012 for each project. We chose 2004 as a threshold as a trade-off between still having a significant history yet eliminate the initial evolution period.

In both cases, the TDS/VDS split was 80%/20%, as follows. To construct the VDS, we sorted the bug reports in the *2004+* dataset chronologically. For the *2004+* scenario, we set aside the most recent 20% as the VDS. For the *All* scenario, to preserve the 80/20 proportion, we kept the same VDS but from the TDS we discarded a random set so that the TDS size for *All* was the same as for *2004+*.

6.2 Results

Table 6 shows each classifier's accuracy. The first column indicates the training set we used, while the second column indicates the validation set. The rest of the columns show the prediction accuracy, in percents, using different classifiers. We highlight the best results in bold; in a nutshell, Bayesian Net performs best (as it is usable across the board).

The first set of rows shows the results when the *All* train-

Table 8: Strongest prediction keywords.

Mozilla	KDE	Apache	All projects
deadlock	deadlock	between	deadlock
moztrap	cef	mac	warhammer
structure	synchron	crash	concur
race	concur	import	first
network	hang	got	atom
spin	order	id	network
xpcom	callback	call	race
semaphore	backport	determine	spin
gclevel	manage	intern	lock
runtime	cur	subsequ	backgroundparser

ing set was used, that is bug reports selected from all projects across the entire time span. In the second column we show the VDS used: bugs from Mozilla, KDE, or from all three projects (we did not perform this validation for Apache due to its low representation in the VDS). We found that the best classifier was Bayesian Net, which attained 56.25%–63.64% prediction accuracy in identifying the concurrency bug type.

The second set of rows shows the results when the *2004+* (recent history) training set was used. We found that the best classifier was Decision Table, which attained 63.33%–72.73% prediction accuracy in identifying the concurrency bug type. We consider a predictor with this level of accuracy to be potentially very useful to developers. Also, the deleterious effect of "stale" training samples is readily apparent, as all classifiers except C4.5 perform better on this more recent data set, *2004+*, than on the *All* data set.

In the last three rows we show the results obtained by using project-specific TDS/VDS sets. We used the complete-history data set for Mozilla and Apache; in KDE we could not find any concurrency bugs prior to 2004. We found that Bayesian Net performs best for Mozilla (65%), Bayesian Net and C4.5 perform best for KDE (60%), while for Apache, Bayesian Net, C4.5 and Decision Table are tied, with 71.43%.

Overall the Bayesian Net classifier had the best performance in most cases (7 out of 9). Hence in Table 7 we show the precision, recall and F-measure attained with this classifier. We found that deadlock bugs have the highest precision, recall and F-measure value since they are quite different from the other three classes. Order violation has the lowest precision, recall and F-measure value. Upon manual inspection, we found that in several cases order bugs were classified as data races (the nature of order and race bugs makes them difficult to distinguish in certain cases). For instance, KDE bug #301166 was classified as data race due to the keywords "thread" and "asynchronously", but it could be considered both an order violation and data race bug.

Observations. These results reveal several aspects. First, for a new project we recommend that project managers choose Bayesian Net as classifier, since it has performed best in most cases. Second, recent training sets achieve the highest accuracy (compare *2004+* with *All*) when using the right classifier—Decision Table in our case. Third, a large, cross-project training dataset can yield better results than per-project training sets—compare KDE trained on *2004+* with KDE trained on its own data sets; this might be due to lower bug report quality in KDE. This might be promising for predicting bugs in a new project for which we have no large concurrency bug sets; we leave this to future work.

What do classifiers learn? To gain insight into how classifiers learn to distinguish among bug types, we extracted the 10 "strongest" nodes, i.e., with the highest conditional probability in the trained Bayesian Nets, on the data sets used in the last four rows of Table 6 (that is, each project trained on its own bug reports and an all-projects VDS trained on the *2004+* TDS). Table 8 lists the keywords in these nodes, in the order of strength. We found that, in addition to textual keywords (e.g., "deadlock", "race", "spin", "hang"), the network has learned to use names of program classes, variables and functions (e.g., "gclevel", "cef", "cur", "backgroundparser"). We believe that the high prediction power of these program identifiers could be a useful starting points for static analysis, an investigation we leave to future work. Interestingly, another high-probability node was the developer ID of a frequent Mozilla contributor ("warhammer" in the last column).

Table 9: Source path prediction results.

Project	Accuracy (%)			Classifier
	Top-1	Top-10%	Top-20%	
Mozilla	31.82	50.00	59.09	Decision Table
KDE	25.00	50.00	56.25	Naïve Bayes
Apache	22.22	44.44	55.56	Bayesian Net
All projects	26.09	47.83	52.17	Naïve Bayes

Note that we have built our classifier assuming the input is a concurrency bug report. However, as future bug reports will not be subject to our manual analysis to decide whether they are concurrency or non-concurrency (our goal is to avoid manual intervention) we will not have ground truth on whether they represent a concurrency or a non-concurrency bug. To solve this, we have built a high-accuracy (90%) "pre-classifier" that triages bug reports into concurrency and non-concurrency. For brevity, we leave out details of this classifier. Since the proportion of concurrency bugs is small compared to other bugs, we would still have some false positives to manually eliminate among the classified bug reports, but the workload is reduced greatly thanks to the pre-classifier.

7. PREDICTING CONCURRENCY BUGS' LOCATION

The previous section showed one useful step for finding and fixing a bug: predicting its type. It is also useful to figure out *where*, in the source code, the new bug is likely to be located; hence in this section we present our approach for predicting the likely location of a concurrency bug.

7.1 Approach

We used a classifier that takes a bug report as input and produces a set of source code paths as output. More specifically, the classifier's output is a vector of binary values, and each position in this vector corresponds to a source code path. For example, consider three bugs #1, #2, and #3, such that #1's location was code path /foo/, #2's location was path /bar/, and #3 has affected both code paths /bar/ and /baz/. Suppose the order in the output vector is (/foo/, /bar/, /baz/). Then the correct classifier output for bug #1 would be (1,0,0); for bug #2 it would be (0,1,0); and for bug #3 it would be (0,1,1).

7.2 Results

The training process is similar to the one used in the prior section, that is, we used 80% of the bug set for training and 20% for validation. Measuring prediction accuracy is slightly more convoluted, because a bug can affect multiple files, and we are interested in predicting a Top-k of most likely locations, rather than a single location.

We now explain how we compute Top-k accuracy when a single bug spans multiple code paths. For each bug i in the

VDS, assuming the bug has affected j paths, we have a list of true source code paths $\{tpath_{i1}, \ldots, tpath_{ij}\}$ (each unique path is called a "class"; we employed Mulan [26] for this part). We present the bug report i to our classifier, which returns a list of m likely output paths $\{opath_{i1}, \ldots, opath_{im}\}$. More specifically, for each validation input, Mulan returns as output a vector of real numbers indicating the probability of the sample belonging to each class; probabilities under a threshold are replaced with 0. Next, from the set $\{opath_{i1}, \ldots, opath_{im}\}$ we select the highest-ranked k output paths, in order of probability, i.e., a subset $\{opath_{i1}, \ldots, opath_{ik}\}$. Then, we check if the set $\{tpath_{i1}, \ldots, tpath_{ij}\}$ is a subset of $\{opath_{i1}, \ldots, opath_{ik}\}$. Finally, we compute Top-k accuracy: for Top-1 accuracy, we count a hit if the probability value assigned to the true path class is the highest-ranked in output vector; for bugs affecting multiple files, say 2, if the 2 highest probabilities correspond to the true path classes, and so on. To compute Top-10% accuracy, we check whether the bug location(s) is(are) in the Top-10% highest output class probabilities; similarly for Top-20%.

In Table 9 we present the results. In the first column we show the project, and in the columns 2–4 we present the attained accuracy for each of the three metrics. The last column shows the classifier used to achieve that accuracy (we only present the best results across the four classifiers).

We achieve 22.22%–31.82% Top-1 accuracy, depending on the project (column 2). We consider this to be potentially very useful for locating bugs, because it means the developer is presented with the *exact bug location* in 22.22%–31.82% of the cases, depending on the project (we had 45 path locations for Mozilla, 47 for KDE and 19 for Apache). When measuring Top-10% accuracy, the accuracy increases to 44.44%–50%. When measuring Top-20% accuracy, we obtained higher values, 55.26%–59.09%, which is expected. That is, in more than half the cases, the bug location is in the Top-20% results returned by the classifier. Since all our projects have large code bases, narrowing down the possible bug location can considerably reduce bug-fixing time and effort.

On a qualitative note, we found that certain locations are more prone to concurrency bugs: Mozilla had 6 bugs in files under /mozilla/netwerk/cache/src and 4 bugs files under /mozilla/xpcom/base, whereas KDE had 6 bugs in /KDE/extragear/graphics/digikam/libs. Our method can guide developers to these likely bug-prone locations after receiving a bug report to help speed up bug finding and fixing.

8. THREATS TO VALIDITY

Selection bias. We have chosen three projects for our study. These projects are mostly written in C/C++ and are, we believe, representative for browsers, desktop GUI, and server programs that use concurrency. However, other projects, e.g., operating systems, database applications or applications developed in other programming language (Java), might have different concurrency bug characteristics. For example, prior efforts [8,15] have found that deadlocks in MySQL represent 40% of the total number of concurrency bugs, whereas for our projects, deadlocks account for 22% (Mozilla), 25% (KDE), and 12% (Apache) of concurrency bugs. Nevertheless, for atomicity violation and order violation, our results are similar to prior findings [15].

Data processing. Our keyword-based search for bug reports could have missed some concurrency bugs—a weakness we share with other prior studies [8, 15]. However, a concurrency bug report that did not contain any keywords on our list is more likely to be incomplete and more difficult to analyze its root cause. To reduce this threat, we used an extensive list of concurrency-related keywords, and searched both the bug tracker and the commit logs. Completely eliminating this threat is impractical, as it would involve manual analysis (which itself is prone to errors) for more than 250,000 bug reports.

Unfixed and unreported bugs. Some concurrency bugs might go unfixed or unreported because they strike infrequently, on certain platforms/software configurations only, and are hard to reproduce. It would be interesting to consider these kinds of bugs, but they are not likely to have detailed discussions and they will not have patches. As a result, these bugs are not considered as important as the reported and fixed concurrency bugs that are used in our study.

Short histories. When relying solely on machine learning and statistics for training, our approach works better for projects with larger training data sets—this could be problematic for projects with short histories or low incidence of concurrency bugs, though cross-project prediction could be useful in that case, as we have shown.

Bug classification. We used four categories and manual categorization for concurrency bugs. We excluded bugs which did not have enough information to be categorized. This can lead to missing some concurrency bugs, as discussed previously. As a matter of fact, some concurrency bugs may belong to multiple categories, e.g., an order violation could also be considered a data race.

9. RELATED WORK

Bug characteristic studies. Bug characteristic studies have been performed on other large software systems [25], though the objectives of those studies were different, e.g., understanding OS [6] errors. In contrast, our study focuses specifically on understanding and predicting concurrency bugs. Many other efforts [2, 11, 13, 14, 22] have mined bug and source code repositories to study and analyze the behavior and contributions of developers and their effects on software quality.Some of the efforts used machine learning for analysis and prediction. In contrast, we do in-depth analysis and prediction for concurrency bugs.

Studies on concurrent programs. Lu et al. [15] analyzed 105 concurrency bugs collected from four open source projects (Mozilla, Apache, OpenOffice, MySQL). Their study focused on understanding concurrency bug causes and fixing strategies. Fonseca et al. [8] studied internal and external effects of concurrency bugs in MySQL. They provide a complementary angle by studying the effects of concurrency bugs (e.g., whether concurrency bugs are latent or not, or what type of failures they cause). We use a similar methodology for deciding which bugs to analyze, but with different objectives and methods: characterizing bug features, a quantitative analysis of the bug-fixing process and constructing prediction models for bug number, type and location.

Predicting bug location. Ostrand et al. [21] used multivariate negative binomial regression model and revealed that variables such as file size, the number of prior faults, newly-introduced and changed files can be used to predict faults in upcoming releases. Based on the model they predict the number of faults in each file, and fault density. They found that their Top-20% files predicted to be buggy contained 71%–92% of the detected bugs. Their study, like ours, has

revealed that bug numbers are autocorrelated. However, we use different variables to construct the predictor model, and instead of predicting the number of bugs per file and bug density per file, we predict the number of concurrency bugs in the system, and type/location for newly-filed concurrency bug reports. Kim et al. [11] proposed bug cache algorithms to predict future bugs at the function/method and file level by observing that bugs exhibit locality (temporal, spatial) and the fact the entities that have been introduced or changed recently tend to introduce bugs. Their study was performed on 7 large open source projects (including Apache). Their accuracy was 73%–95% for files and 46%–72% for functions/methods. Their study, like ours, has revealed that bug numbers are autocorrelated. We do not investigate localities beyond temporal; however, they might help improve our prediction accuracy.

Wu et al. [27] used time series for bug prediction but did not consider the impact of independent variables on the time series, as we do. Rao et al. [23] compared five information retrieval models for the purpose of locating bugs. Their work mainly focused on comparing models (concluding that Unigram and Vector Space work best) and calculating the likelihood that one file would be buggy based on its similarity with known buggy files. We use a different model; we focused on finding the exact location one concurrency bug would affect; and we had to solve the multi-label classification problem. Moin et al. [19] used commit logs and bug reports to locate bugs in the source file hierarchy. However, their method is coarser-grained than ours, e.g., if two bugs are in mozilla/security/nss/lib/certdb and mozilla/security/nss/lib/pki respectively, they considered the bugs to be in the same location, mozilla/security/nss/lib/, but our prediction model can distinguish the difference between these two locations.

10. CONCLUSIONS

We have performed a study of concurrency bugs in three large, long-lived open source projects. We have found that concurrency bugs are significantly more complicated, taking more time and resources to fix, than non-concurrency bugs. We have also found that concurrency bugs fall into four main categories (atomicity violations, order violations, races, and deadlocks) and that among these categories, deadlocks are easiest, while atomicity violations are hardest to fix. We have shown that effective forecast methods can be constructed to help managers and developers predict the number of concurrency bugs in upcoming releases, as well as the likely type and location for newly-filed concurrency bug reports.

11. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. This work was supported in part by National Science Foundation grant CCF-1149632.

12. REFERENCES

[1] Apache Software Foundation Bugzilla. `https://issues.apache.org/bugzilla/`.

[2] K. Ayari, P. Meshkinfam, G. Antoniol, and M. Di Penta. Threats on building models from cvs and bugzilla repositories: the mozilla case study. In *CASCON'07*, pages 215–228, 2007.

[3] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *J. R. Stat. Soc. Series B (Methodological)*, pages 289–300, 1995.

[4] M. L. Bernardi, C. Sementa, Q. Zagarese, D. Distante, and M. Di Penta. What topics do firefox and chrome contributors discuss? In *MSR'11*.

[5] P. Bhattacharya, I. Neamtiu, and C. R. Shelton. Automated, highly-accurate, bug assignment using machine learning and tossing graphs. *Journal of Systems and Software*, 85(10):2275–2292, 2012.

[6] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP'01*, pages 73–88, 2001.

[7] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP'03*, pages 237–252, 2003.

[8] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues. A study of the internal and external effects of concurrency bugs. In *DSN'10*, pages 221–230, 2010.

[9] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In *FSE'09*.

[10] KDE Bugtracking System. `https://bugs.kde.org/`.

[11] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *ICSE'07*, pages 489–498, 2007.

[12] J. Koskinen. Software maintenance costs, Sept 2003. `http://users.jyu.fi/~koskinen/smcosts.htm`.

[13] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. Predicting the severity of a reported bug. In *MSR'10*.

[14] M. Linares-Vasquez, K. Hossen, H. Dang, H. H. Kagdi, M. Gethers, and D. Poshyvanyk. Triaging incoming change requests: Bug or commit history, or code authorship? In *ICSM'12*, pages 451–460, 2012.

[15] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS'08*.

[16] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS XII*, pages 37–48, 2006.

[17] B. Lucia, B. P. Wood, and L. Ceze. Isolating and understanding concurrency errors using reconstructed execution fragments. In *PLDI'11*.

[18] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[19] A. Moin and M. Khansari. Bug localization using revision log analysis and open bug repository text categorization. In *Open Source Software: New Horizons*, IFIP AICT, pages 188–199. 2010.

[20] Mozilla Bugzilla. `https://bugzilla.mozilla.org/`.

[21] T. Ostrand, E. Weyuker, and R. Bell. Predicting the location and number of faults in large software systems. *IEEE TSE'05*, pages 340 – 355.

[22] J. Park, M. Kim, B. Ray, and D.-H. Bae. An empirical study of supplementary bug fixes. In *MSR'12*.

[23] S. Rao and A. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *MSR'11*.

[24] I. Sommerville. *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004.

[25] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai. Bug characteristics in open source software. *Empir Software Eng*, pages 1–41, 2013.

[26] G. Tsoumakas, E. Spyromitros-Xioufis, J. Vilcek, and I. Vlahavas. Mulan: A java library for multi-label learning. *Journal of Machine Learning Research*, 12:2411–2414, 2011.

[27] W. Wu, W. Zhang, Y. Yang, and Q. Wang. Time series analysis for bug number prediction. In *SEDM'10*.