Abbaspour Asadollah *et al. Journal of Internet Services and Applications* (2017) 8:4
DOI 10.1186/s13174-017-0055-2

Journal of Internet Services and Applications

RESEARCH Open Access

Concurrency bugs in open source software: a case study

Sara Abbaspour Asadollah[1*], Daniel Sundmark[1], Sigrid Eldh[2] and Hans Hansson[1]

Abstract

Concurrent programming puts demands on software debugging and testing, as concurrent software may exhibit problems not present in sequential software, e.g., deadlocks and race conditions. In aiming to increase efficiency and effectiveness of debugging and bug-fixing for concurrent software, a deep understanding of concurrency bugs, their frequency and fixing-times would be helpful. Similarly, to design effective tools and techniques for testing and debugging concurrent software, understanding the differences between non-concurrency and concurrency bugs in real-word software would be useful.

This paper presents an empirical study focusing on understanding the differences and similarities between concurrency bugs and other bugs, as well as the differences among various concurrency bug types in terms of their severity and their fixing time, and reproducibility. Our basis is a comprehensive analysis of bug reports covering several generations of five open source software projects. The analysis involves a total of 11860 bug reports from the last decade, including 351 reports related to concurrency bugs. We found that concurrency bugs are different from other bugs in terms of their fixing time and severity while they are similar in terms of reproducibility. Our findings shed light on concurrency bugs and could thereby influence future design and development of concurrent software, their debugging and testing, as well as related tools.

Keywords: Concurrency bugs, Bug severity, Fixing time, Open source software, Apache Hadoop, Apache ZooKeeper, Apache Oozie, Apache Accumulo, Apache spark, Case study

1 Introduction

With the introduction of multicore and other parallel architectures, there is an increased need for efficient and effective handling of software executing on such architectures. An important aspect in this context is to understand the bugs that occur due to parallel and concurrent execution of software. In this paper, we look into how the increase of such executions have impacted a number of issues, including the occurrence of related bugs, the difficulty to fix these bugs compared to fixing non-concurrent ones, and the distribution of unreproducible concurrency and non-concurrency bugs.

Testing and debugging concurrent software are faced with a variety of challenges [1]. These challenges concern different aspects of software testing and debugging, such as parallel programming [2], performance testing, error detection [3] and more. Since concurrent software exhibit more non-deterministic behavior and non-deterministic bugs are generally viewed to be more challenging than other types of bugs [4–6], testing and debugging concurrent software are also considered to be more challenging compared to testing and debugging of sequential software.

Developing concurrent software requires developers to keep track of all the possible communication patterns that evolve from the large number of possible interleavings or concurrently overlapping executions that can occur between different execution threads through utilizing the shared memory.

Handling the many execution scenarios that this results in is a notoriously difficult task in debugging and makes it equally hard to create test cases [7].

In the presented study, we are particularly interested in isolating concurrency bugs from other types of bugs (non-concurrency bugs) and analyzing the distinguishing features in their respective fixing processes. Hence, the main emphasis of this research is on concurrency bugs,

*Correspondence: sara.abbaspour@mdh.se
[1]Mälardalen University, Västerås, Sweden
Full list of author information is available at the end of the article

Springer Open

© The Author(s). 2017 **Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

and on exploring the nature and extent of concurrency bugs in real-world software. This exploration of bugs can be helpful to understand how we should address concurrency bugs, estimate the most time-consuming and difficult to reproduce ones, and prioritize them to speed up the debugging and bug-fixing processes. Also it could be helpful for designers to avoid the errors that are more likely to occur during the early phases of the software life-cycle.

In our study, we address the following research questions:

- **RQ1:** How common are different types of concurrency bugs compared to non-concurrency bugs?
- **RQ2:** What is the fraction of unreproducible bugs that are also concurrency bugs?
- **RQ3:** How long time is required to fix concurrency bugs compared to fixing non-concurrency bugs?
- **RQ4:** Are concurrency bugs severer than non-concurrency bugs?
- **RQ5:** How long time is required to fix the severest concurrency bugs as compared to fix the severest non-concurrency bugs?

In this study, we investigate the bug reports from five open source software projects, i.e., Apache Hadoop project, Apache ZooKeeper project, Oozie project, Accumulo project and Apache Spark project. We classify the reported bugs into three distinct categories, i.e., fixed and closed concurrency bugs, fixed and closed non-concurrency bugs and unreproducible bugs. We further classify the concurrency bugs based on bug type. For concurrency and non-concurrency we additionally consider severity and fixing time. We compare the non-concurrency, concurrency and unreproducible bugs in terms of their reporting frequency. Our results indicate that a relatively small share of bugs is related to concurrency issues, while the vast majority are non-concurrency bugs. Fixing time for concurrency and non-concurrency bugs is different but this difference is not big. However, the fixing time for unreproducible concurrency and unreproducible non-concurrency bugs is similar. In addition, concurrency bugs are considered to be slightly severer than non-concurrency bugs.

The remainder of this paper is organized as follows. We describe our research methodology in Section 2. In Section 3, we present the classification schemes of the study. We provide a comprehensive set of results, quantitative analysis of bug reports and answer the research questions in Section 4. The discussion on obtained results and the threats to validity are presented in Section 5. We survey related work in Section 6. Finally, we conclude the study and highlight the direction of future work in Section 7.

2 Methodology

This study is carried out by following the workflow outlined in Fig. 1. First, we start with the *Bug source software selection* in order to select a proper open source software for our study. Second, in the *Bug report selection* we identify the set of concurrency bug reports in the issue tracking database of the selected project through a keyword search. Then we manually analyze the full set of identified bug reports in order to exclude those which are not concurrency-related. Finally, in the *bug reports classification* stage, we collect data for the concurrency bugs, and classify the bug reports based using the classification scheme described in Section 3. The following subsections describe the steps of this research workflow in further detail.

2.1 Bug-source software selection

We considered five open source applications viz., Apache Hadoop project, Apache ZooKeeper project, Apache Oozie project, Apache Accumulo project and Apache Spark project. The projects coordinate distributed processes with significant number of releases and an issue management platform for managing, configuring and testing. The projects have a web interface for managing, configuring and testing its services and components while the detailed information on bugs and bug fixes are openly available.

Apache Hadoop project[1] is a Java based distributed computing framework built for applications implemented using the MapReduce[2] programming model [8]. Hadoop has changed constantly and considerably in 59 releases over six years of development. The Hadoop framework has been widely adopted by both the industry and research communities [9]. Due to Hadoop's key concept of parallel and distributed abstraction, it is recently adopted by several big companies such as Facebook, Ebay, Yahoo, Amazon, Adobe and more.

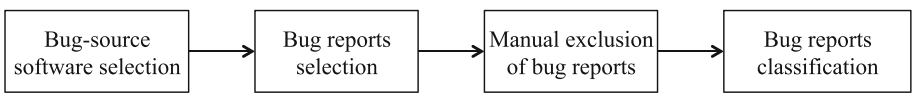


Fig. 1 Research methodology workflow

Apache ZooKeeper project[3] is a fault-tolerant and a distributed coordination service for other distributed applications such as cloud computing applications [10]. This tool provides fundamental services by encapsulating distributed coordination algorithms and maintaining a simple database. An application can use the ZooKeeper client to build higher-level coordination functions [11] such as leader election, barriers, queues, and read/write revocable locks.

Apache Oozie project[4] is a coordination system and server-based workflow scheduling to manage data processing jobs for Hadoop. This tool is a mature solution for defining and executing workflows (that can be triggered by a user, time event or data arrival) [12]. It supports useful functionalities such as the persistence of an execution history.

Apache Accumulo project[5] is a distributed, column oriented and multidimensional data storage [11] that built on top of Apache Hadoop, Zookeeper and Thrift. Accumulo's design is based on the Google Bigtable implementation and is very similar to HBase (the fundamental data storage capabilities are between Accumulo and HBase are clearly similar).

Apache Spark project[6] is a cluster computing system from Apache with incubator status. This tool is pretty fast at running programs and writing data [13]. Spark supports in-memory computing, that enables it to query data much faster compared to diskbased engines such as Hadoop, and supports a rich set of other tools such as Spark SQL (for SQL and structured data processing), MLLib (for machine learning) and Spark Streaming. It also provides an optimized engine that supports general computation graphs (GraphX). The aim of the project is to build a faster distributed parallel processing framework [11] that can perform better for certain tasks than Hadoop.

These five projects track both enhancement requests and bugs using JIRA[7]. JIRA is an issue management platform, which allows users to manage their issues throughout their entire life cycle. It is mainly used in software development and allows users to track any kind of unit of work, such as project task, issue, story and bug to manage and track development efforts.

2.2 Bug reports selection

In this stage, we selected the concurrency bugs from the bug report database including bug reports from the period 2006-2015, i.e., the last decade. In total, the projects bug report databases contain 11860 issues in this period that are tagged as "Bug".

We automatically filtered reports that are not likely to be relevant by performing a search query on bug report databases. Our search query filtered bugs based on (1) "Bug" as report type, (2) the status of the report, and (3) keywords relevant to concurrency. Figure 2 summarizes the bug report selection process.

In filtering based on *"Bug" as report type* step, we practically searched in the projects report databases for the reports with issue type "Bug"[8] according to our main objective and bug definition.

In filtering based on *the status of the report* step, we searched for bugs with "Closed" (i.e., this report considered finished, the resolution is correct) and "Fixed" resolution status (i.e., fix for this issue has been implemented). We only selected "Fixed" and "Closed" reports since unfixed and open bug reports might be invalid and root causes described in the reports could be incorrect. It would then be impossible for us to completely understand the details on these bugs and determine their types. It would have been interesting to also consider bugs with other statuses or resolutions (such as duplicate bugs, or bugs that were coded with "won't fix") but these bug

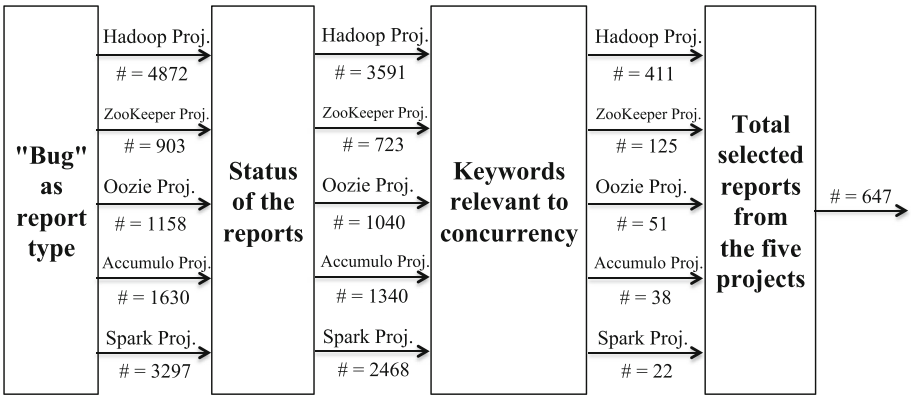


Fig. 2 Concurrency bug report selection workflow

reports are not likely to be as complete and reliable as bugs that are labeled closed and fixed. Thus without reasonably complete bug reports it would not be possible to recognize the reported bugs.

In filtering based on *the keywords relevant to concurrency* step, we decided to use the keywords that could help us to include the bug reports were compatible with the scope of this study. In identifying such keywords, we reviewed the keywords utilized in similar previous studies [1, 14]. The keywords included in the search, i.e. the terms, were:

thread, blocked, locked, race, dead-lock, deadlock, concurrent, concurrency, atomic, synchronize, synchronous, synchronization, starvation, suspension, "order violation", "atomicity violation", "single variable atomicity violation", "multi variable atomicity violation", livelock, live-lock, multi-threaded, multithreading, and multi-thread.

After filtering, we obtained a final set with 647 reports. The detailed information is presented in Table 1. These 647 reports were reported bugs with at least one concurrency keywords (from the above list) in their description, which was already fixed and closed by developers.

Some concurrency bugs might go unfixed or unreported because they are difficult or impossible to reproduce. In order to investigate the distribution of these kind of bugs (unreproducible bugs) and obtain the results, we perform three steps. Figure 3 summarizes these steps. The only difference between these steps and the ones we applied for concurrency bug selection workflow (Fig. 2) is, how we implement the "*the status of the report*" step.

We automatically filtered reports that are not likely to be relevant by performing a search query on the bug report database. Our search query filtered bugs based on (1) "Bug" as report type, (2) the status of the report, and (3) keywords relevant to concurrency.

In filtering based on *"Bug" as report type* step, we filtered the projects' reports with the same issue type as we applied for filtering the reports for concurrency reports. We included the bugs from the period 2006-2015 (last decade). Similarly, after this step we had 11860 reported bugs in our database.

Table 1 The selected bugs from each project after *the keywords relevant to concurrency* step

Project	Concurrecny bugs
Hadoop	221
ZooKeeper	83
Oozie	18
Accumulo	21
Spark	8
Total	351

In filtering based on *the status of the report* step, we searched for bugs with "Cannot Reproduce"[9] resolution status. After accomplishing this filter, we have obtained 513 reported bugs.

Finally, in filtering based on *the keywords relevant to concurrency* step, we used the same keywords as we have used for excluding concurrency bugs. We obtained a final set with 203 reports by applying these steps.

2.3 Manual exclusion of bug reports and sampling of non-concurrency bugs

In this stage, we manually analyzed the 647 bug reports obtained in the previous step. The manual inspection revealed that some of the bugs that matched the search query were not concurrency bugs. Thus, we excluded them. More specifically, we determined the relevance of the bugs by checking (1) if they describe a concurrency bug, and if they do, (2) what type of concurrency bug it is. The latter is done, by comparing their descriptions (or explanations) with our concurrency bug classification (given in Section 3.1). If we could not map a report with any class, we excluded that report from our set. We also excluded reports with very little information, since we could not analyze them properly. After filtering, we obtained a final set with 351 concurrency bugs.

As explained in Section 1, our main objective is understanding the differences between non-concurrency and concurrency bugs. For comparison purposes, we randomly sampled an equally sized subset of non-concurrency bugs (351) that were reported during 2006-2015 and were "Fixed" and "Closed". These bugs were used for comparative analysis between the concurrency and non-concurrency bug sets.

Table 2 shows the bug counts across the different steps of the bug report selection process. Note that this selection process may have some limitations, as discussed in more detail in Section 5.1.

2.4 Bug reports classification

We analyzed the issues and information contained in the reports using them to map to the concurrency bug classification manually. Each bug report contains several types of information, which were valuable in recognizing and filtering the concurrency bugs with other types of bugs to aid us understand the characteristics of bugs. The bug reports contained for example the description of the bug with some discussions among the developers on how to detect, where to detect (bug localization) and how to fix the bugs. Typically most of the reports include a description of the correction of the bug, and a link to the version of the software where the bug has been corrected, and even the scenario of reproducing the reported bug. The reports also contain additional fields such as

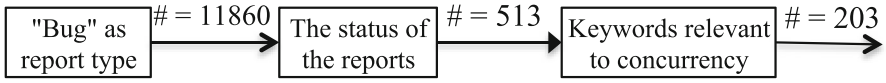


Fig. 3 Unreproducible concurrency bug report selection workflow

perceived priority, created date, resolved date, version affected and more.

We used different types of fields in order to explore the concurrency bug issues in the five selected projects. We used the *priority* field to estimate the severity of the bug. The interval between the *Created date* and *Resolved date* fields was used to calculate the amount of (calendar) time required to fix the bug (fixing time).

3 Study classification schemes

In order to perform the bug classification process, we defined three main classifiers and grouped the reports based on these classifiers. The classifiers were **type of concurrency bug**, **fixing time** and **severity**. For *Concurrency bug classification*, we used the classification scheme for concurrency bugs described in Section 3.1. For *fixing time*, we calculated the fixing time for each bug report (Section 3.2) and for *severity*, we used the scheme proposed by the JIRA in [15] (Section 3.3). The details of these three classification schemes are described as follows:

3.1 Concurrency bug classification

We classified and mapped the relevant bug reports related to the types of concurrency bugs using a classification of concurrency bug types based on observable properties. In [14], one of our main contribution is a taxonomy for concurrency bugs by classifying the bugs in a common structure considering relevant observable properties. This classification is based on an assumption that a concurrency bug has occurred, i.e., the properties of each bug may not be sufficient to identify a bug, but once a concurrency bug has occurred the properties can be used to uniquely identify which type of bug it is [16]. It categorizes concurrency bugs into seven disjoint classes as follows:

- ***Data race*** occurs when at least two threads access the same data and at least one of them write the data [17]. It occurs when concurrent threads perform conflicting accesses by trying to update the same memory location or shared variable [18, 19].
- ***Deadlock*** is "a condition in a system where a process cannot proceed because it needs to obtain a resource held by another process but it itself is holding a resource that the other process needs" [20]. More generally, it occurs when two or more threads attempts to access shared resources held by other threads, and none of the threads can give them up [18, 21]. During deadlock, all involved threads are in a waiting state.
- ***Livelock*** happens when a thread is waiting for a resource that will never become available while the CPU is busy releasing and acquiring the shared resource. It is similar to deadlock except that the state of the process involved in the livelock constantly changes and is frequently executing without making progress [22].
- ***Starvation*** is "a condition in which a process indefinitely delayed because other processes are always given preference" [23]. Starvation typically occurs when high priority threads are monopolising the CPU resources. During starvation, at least one of the involved threads remains in the ready queue.
- ***Suspension-based locking or Blocking suspension*** occurs when a calling thread waits for an unacceptably long time in a queue to acquire a lock for accessing a shared resource [24].
- ***Order violation*** is defined as the violation of the desired order between at least two memory accesses [25]. It occurs when the expected order of interleavings does not appear [26]. If a program fails to enforce the programmer's intended order of

Table 2 Report counts from different step of bug report selection process

Filter	Selected reports	# of reports
2006-2015 & Bug & Fixed & Closed	Total bug reports (from five projects)	11860
	Keywords match related bug reports	647
	Concurrency bug reports analyzed	351
2006-2015 & Bug & Fixed & Closed	Sample of non-concurrency bug reports	351
2006-2015 & Bug & (Cannot reproduce)	Total bug reports (from five projects)	513
	Keywords match related bug reports	203

execution then an order violation bug could happen [1].

- ***Atomicity violation*** refers to the situation when the execution of two code blocks (sequences of statements protected by lock, transaction) in one thread is concurrently overlapping with the execution of one or more code blocks of other threads such a way that the result is inconsistent with any execution where the blocks of the first thread are executed without being overlapping with any other code block.

3.2 Fixing time calculation

Fixing is the time duration (days) which developers/debuggers spend to fix a reported bug. Fixing time, calculated by subtracting the *Created date* and *Resolved date* fields of the bug, i.e. it refers to the calender time rather than the actual time spent fixing the bug.

3.3 Bug report severity classification

In order to define priority for each issue based on developers' perspective we used a classification scheme similar to the classification defined in [15].

- ***Blocker*** shows the highest priority. It indicates that this issue takes precedence over all others.
- ***Critical*** indicates that this issue is causing a problem and requires urgent attention.
- ***Major*** shows that this issue has a significant impact.
- ***Minor*** indicates that this issue has a relatively minor impact.
- ***Trivial*** is the lowest priority.

4 Results and quantitative analysis

This section provides the analysis of the data collected for bugs obtained from the projects bug database. We used 702 bugs (i.e., 351 are concurrency bugs while the rest (351) are non-concurrency bugs sampled for our analysis) reported between 2006 and 2015. The bug selection process is described in Section 2. We provide the raw data of this study at https://goo.gl/wcdD16.

RQ1: How common are different types of concurrency bugs compared to non-concurrency bugs?

As seen in Fig. 4, out of the 9169 reported bugs in the projects' databases, 351 (i.e., ∼4%) bugs are related to concurrency issues and are causing a certain type of concurrency bug while the rest (i.e., ∼96%) are identified as non-concurrency bugs.

The 351 concurrency bugs were further categorized according to the concurrency bug classification in [14]. As mentioned already in Section 3.1, this taxonomy defines seven types of concurrency bugs: (1) *Data race*, (2) *Deadlock*, (3) *Livelock*, (4) *Starvation*, (5) *Suspension*, (6) *Order violation* and (7) *Atomicity violation*. For the sake of this study, we have added one more category to the taxonomy: (8) *Not clear*. The *Not clear* category includes reports that cover bugs related to concurrency and parallelism, but are not classified according to the concurrency bugs taxonomy. For these bugs, the summary and description of the report indicate that it is a concurrency bug, but further classification of bug type is impossible by a very project implementation-specific explanation of the bug details and solution.

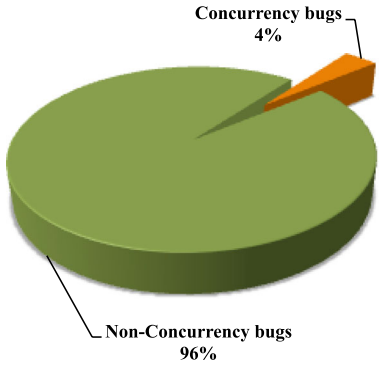


Fig. 4 Distribution of non-concurrency and concurrency bug types

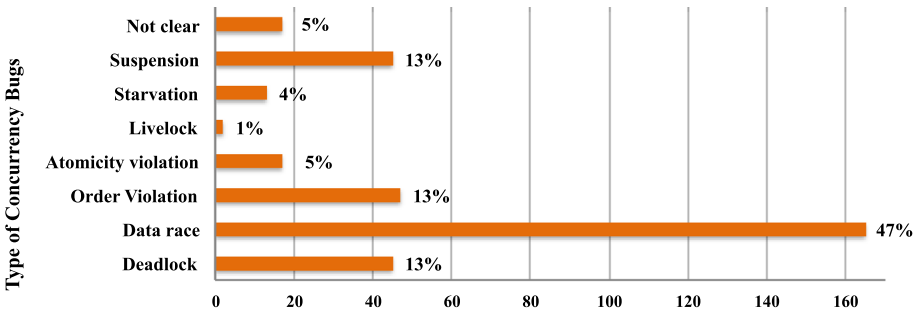
In addition, we investigated the frequency with which each type of concurrency bug appears, in order to get insights of bug prioritization.

Figure 5 summarizes the number of concurrency bugs according to the category. From a total of 351 bug reports, almost half of them (i.e., 47%) concern data races (or race conditions), a well-known and common concurrent bug [27], while only two bug reports (< 1% of the reports) were identified as *Livelock*.

Answer RQ1: Only 4% of the total set of bugs are related to concurrency issues, while the majority of bugs (i.e., 96%) are of non-concurrency type.

RQ2: What is the fraction of unreproducible bugs that are also concurrency bugs?

In order to compare the difficulty in reproducing the concurrency bugs to the difficulty in reproducing non-concurrency bugs, we analyzed the distribution of unreproducible bugs from the all five projects' repository. Figure 6 provides a view of the distribution of concurrency bugs in terms of reproducibility[10]. In Fig. 6, a bug which reported during 2006 to 2015 (last decade) is categorized as *All*, a fixed and closed bug is categorized as *Fixed & Closed*. If a report tagged as "Cannot reproduce" then it is categorized as *unreproducible*. A bug with at least one keyword related to concurrency issues is categorized as *Concurrency keywords matched*, some of these bugs are fixed and closed and others are unreproducible. We are not considering the bugs that are on the investigation and



	Deadlock	Data race	Order Violation	Atomicity violation	Livelock	Starvation	Suspension	Not clear
# of reported bugs	45	165	47	17	2	13	45	17

Fig. 5 Distribution of concurrency bugs

the report related to these are excluded from our study although they are included in *All* category. Finally if a bug falls into one of the concurrency classification types then it is categorized as a *Concurrency bug*. This Venn chart illustrates the comparisons between these categories.

Based on our investigation, the total number of bugs within this time span is 11860. We identified 9162 (∼77%) of them as *fixed & closed*, whereas only 513 of them (∼4%) could not be reproduce by developers. Out of 513 unreproducible bugs, 203 bugs are categorized in *concurrency keywords matched* group.

Answer RQ2: The fraction of unreproducible bugs from the total set of bugs is only 4% while 2% of the total set are unreproducible and related to concurrency issues.

RQ3: How long time is required to fix concurrency bugs compared to fixing non-concurrency bugs?

In order to gain better understanding on how difficult is to fix concurrency bugs in comparison with non-concurrency bugs, we conducted a quantitative analysis of the effort required to fix both concurrency and non-concurrency bugs. This effort was measured by calculating the calendar time between the *Created date* and *Resolved date* fields from projects' databases. We used this time as an indicator for the complexity involved in fixing these bugs.

Table 3 lists the detailed statistics on the obtained results for fixing time of concurrency and non-concurrency bugs. Figure 7 summarizes the results of comparing the fixing time for concurrency and non-concurrency bugs in the form of box-plots (the vertical

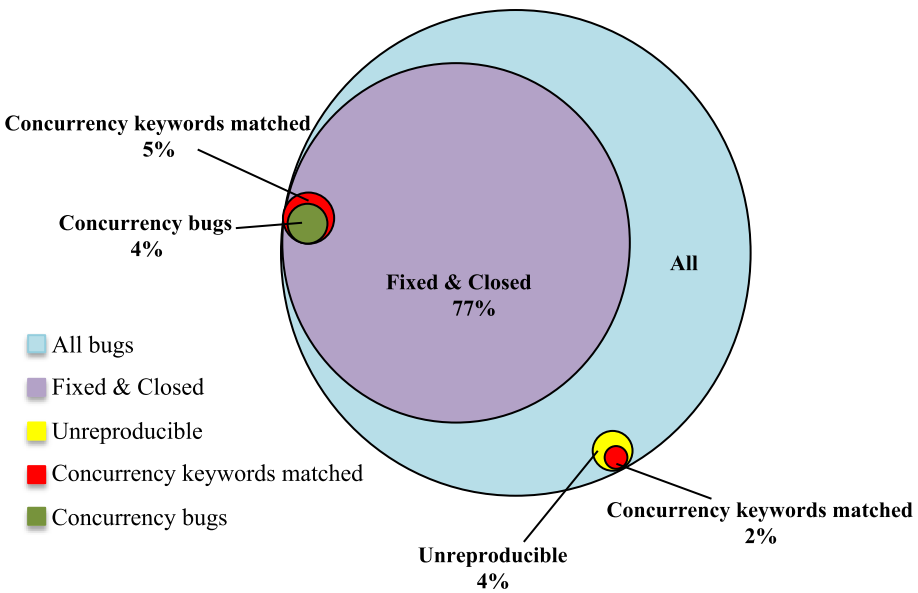


Fig. 6 Distribution of reproducible and unreproducible concurrency bugs

Table 3 Descriptive statistics results for concurrency and non-concurrency bug sets in terms of fixing time. We report here: average, minimum, maximum, median and standard deviation values

	Fixing time				
Bug	Average	Minimum	Maximum	Median	Standard deviation
Concurrency	82.5	0.1	2025.0	18.2	207.0
Non-concurrency	66.2	0.1	998.1	13.0	145.1

axis scale of the plot is logarithmic). The average of fixing time for concurrency and non-concurrency bugs is 82 days and 66 days for fixing concurrency and non-concurrency bugs, respectively.

To evaluate if there is any statistical difference between concurrency and non-concurrency bugs fixing time, we use a *Wilcoxon Signed Rank* test, a non-parametric hypothesis test for determining if there is any statistical difference among the two data sets, with the assumption that the data is drawn from an unknown distribution. We use 0.05 as the significance level.

In addition, we calculate the *Vargha-Delaney A-statistic* as a measure of effect size [28] for analyzing significance. This statistic is independent of the sample size and has a range between 0 and 1. The choice of what constitutes a significant effect size can depend on context. Vargha and Delaney [28] suggest that A-statistic of greater than 0.64 (or less than 0.36) is indicative of "medium" effect size, and of greater than 0.71 (or less than 0.29) can be indicative of a "large" effect size.

We are interested in determining if the fixing time for concurrency bugs is similar to the one for non-concurrency bugs. We begin by formulating the statistical hypotheses as follows: the null hypothesis is that fixing time of the concurrency and non-concurrency bugs sets have identical distributions (H_0) and the alternative hypothesis is that the distributions are different (H_1). Based on the *p*-value of *0.006*, which is less than 0.05, we reject the null hypothesis. That is, the fixing time of concurrency bugs and non-concurrency bugs are statistically different. When calculating the Vargha-Delaney A-statistic, we obtained a value of *0.559* which indicates a "small" standardized effect size [28]. From our results, we can see that the fixing time for concurrency bugs is different from the fixing time for non-concurrency bugs, but that this difference corresponds to a "small" standardized effect size.

We were also interested in understanding the differences between fixing time for each type of concurrency bugs. Figure 8 summarizes our results in the form of box plots. It is obvious that *Order violation*, *Data races* and *suspension* took almost same time to fix in average (i.e., 99, 98, 97 days on average, respectively). *Atomicity violation* and *Deadlock* type took less time (49 and 42 on average, respectively) while *Livelock* and *Starvation* type took

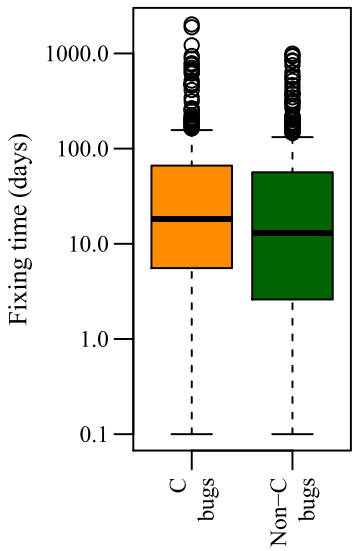


Fig. 7 Fixing time comparison for concurrency and non-concurrency bugs; boxes span from 1st to 3rd quartile, black middle lines are marking the median and the whiskers extend up to 1.5× the inter-quartile range while the circles represent the outliers

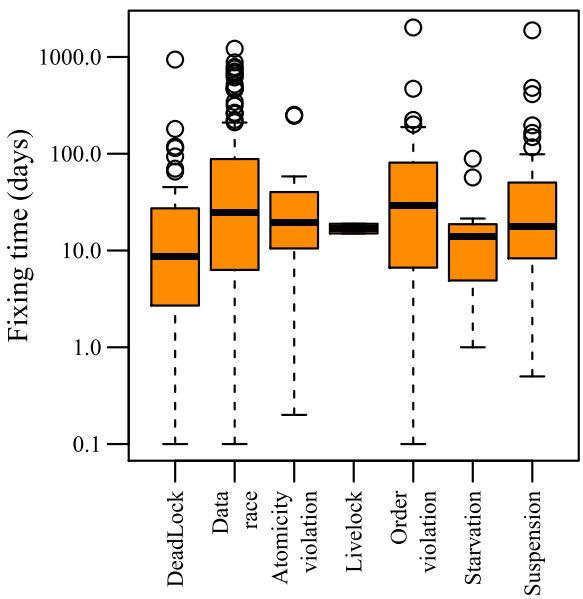


Fig. 8 Effort required to fix each type of concurrency bugs

shorter fixing time (17 and 21 days on average, respectively). Table 4 lists the detailed statistics on the obtained results for each type of concurrency bugs.

To evaluate if there is any significant statistical difference between the different types of concurrency bugs, we use a *Wilcoxon Signed Rank test* and calculate the A-statistic effect size. To this end, we report in Table 5 the *p*-values and the effect size for each type of concurrency bugs. The tested hypotheses are formulated as follows: the null hypothesis is that fixing time results between two different bug types are drawn from the same distribution and the alternative hypothesis is that the fixing time results are drawn from different distributions. We use a traditional statistical significance limit of 0.05 and Vargha and Delaney's suggestion [28] for statistical significance.

Examining Table 5, we can conclude that the null hypothesis is accepted with *p*-values above the traditional statistical significance limit of 0.05 for the majority of bug types except for *"Deadlock-Data race"*, *"Deadlock-Order violation"* and *"Deadlock-Suspension"* pairs where the null hypothesis is rejected. This shows that the bug fixing time is not different except between "Deadlock-Data race", "Deadlock-Order violation" and "deadlock-Suspension". For example, in Table 5, we show the obtained *p*-value of *0.003* for testing the pair "Deadlock-Data race", which is less than 0.05, and therefore we can reject the null hypothesis: the fixing time for *Deadlock* and *Data Race* bug types are different. In addition, the A-statistic for the same pair of bug types is about *0.824* (or 0.175 in the second row), which is greater than the significance level of 0.71. We can say that in this case the effect size is "large". We can conclude that fixing time for *deadlock* and *data race* bug types is different with a "large" effect size.

It should however be noted that the likelihood of statistical errors vastly increases when doing multiple tests using the same dataset. The results from the inter-bug-type comparisons are thus less reliable than the results from the comparison between concurrency and non-concurrency bugs.

Answer RQ3: Concurrency bugs require longer fixing time than non-concurrency bugs, but the difference is not large.

RQ4: Are concurrency bugs severer than non-concurrency bugs?

We analyze the difference between concurrency and non-concurrency bug severity in order to understand if the severity of bugs is differently distributed. Figure 9 shows the severity distributions.

In order to statistically compare the severity between concurrency and non-concurrency bugs, we apply a *Two-Sample Kolmogorov-Smirnov* test (also known as two-sample K-S test) to find if the frequency between these two types of bugs is significantly different. Our null hypothesis can be formulated as follows: are the severity level results of concurrency bugs and non-concurrency bugs drawn from the same distribution. In this test, if the *D-value* is larger than the *critical-D-value*, the observed frequency is distributed differently.

Table 6 shows that the *D*-value is *0.122*, which is larger than the Critical-*D*-value of *0.07*. Based on this fact, statistically we have enough evidence to conclude that there is a difference between the concurrency and non-concurrency bug severity distribution. In other words, the concurrency and non-concurrency severity types are distributed differently.

Finally, in order to identify the severity distribution between concurrency bugs types, we analyze the obtained results, which are shown in Fig. 10. The results indicate the distribution of concurrency bugs together with the severity distribution of each concurrency bug type. We expected that most of the "Blocker" bugs to be of *Deadlock* type. In reality, as shown in Fig. 10, most of the "Blocker" bugs are of *Data race* type. We can interpret this fact in the following way: the *Data race* type might represent the most problematic bug type in terms of severity in the selected projects. On the other hand, we sorted concurrency bugs severity based on their distributions. Table 7 summarizes the obtained results. The rank of severity in the table is based on the definition given in Section 3.3.

Table 4 Fixing time comparison for concurrency bug types

	Fixing time				
Concurrency bugs	Average	Minimum	Maximum	Median	Standard deviation
Deadlock	42.2	0.1	943.2	8.7	142.6
Data race	98.2	0.1	1221.0	24.8	189.5
Order violation	99.3	0.1	2025.0	29.1	298.8
Atomicity violation	48.7	0.2	253.7	19.7	77.8
Livelock	16.9	15.0	18.9	16.9	2.7
Starvation	20.7	1.0	89.2	13.9	25.1
Suspension	96.8	0.5	1890.8	17.9	290.3

Table 5 Wilcoxon test for concurrency bugs fixing time comparison

$H_0 \backslash H_A$	Hypothesis test result	Deadlock	Data race	Order violation	Atomicity violation	Livelock	Starvation	Suspension
Deadlock	*P*-value	–	0.003596	0.01648	0.1041	0.4447	0.5632	0.01825
	A-statistic	–	0.824977	0.2347107	0.08517906	0.009917355	0.06688705	0.224775
Data race	*P*-value	0.003596	–	0.9828	0.7515	0.7185	0.1312	0.8194
	A-statistic	0.175023	–	0.142112	0.05393939	0.00697888	0.04932966	0.1394123
Order violation	*P*-value	0.01648	0.9828	–	0.6191	0.5816	0.1164	0.7458
	A-statistic	0.2451791	0.857888	–	0.08958678	0.01083563	0.0707989	0.235427
Atomicity violation	*P*-value	0.1041	0.7515	0.6191	–	0.9474	0.3627	0.9183
	A-statistic	0.2624793	0.9460606	0.268944	–	0.01153352	0.07555556	0.2584206
Livelock	*P*-value	0.4447	0.7185	0.5816	0.9474	–	0.5714	0.8744
	A-statistic	0.2716253	0.9930211	0.2826814	0.1023691	–	0.0784573	0.2709458
Starvation	*P*-value	0.5632	0.1312	0.1164	0.3627	0.5714	–	0.2291
	A-statistic	0.2631405	0.9506703	0.2703949	0.09814509	0.01149679	–	0.259596
Suspension	*P*-value	0.01825	0.8194	0.7458	0.9183	0.8744	0.2291	–
	A-statistic	0.2462994	0.8605877	0.2444628	0.08923783	0.01059688	0.07043159	–

After comparing the different type of concurrency bugs, we found that most of the bugs categorized as being part of the *Data race* type in terms of severity belongs to the Minor class; the highest population of *Deadlock* bugs belong to Critical class; the highest population of bugs categorized in the *Suspension* type belongs to Critical class; the highest population of bugs corresponding to *Atomicity violation* type belongs to Major and Minor class; the highest population of *Order violation* bugs belongs to Minor class and the highest population of *Starvation* bugs belong to Major class. We can interpret this as follows: *Deadlock* and *Suspension* bugs have higher severity.

Answer RQ4: Concurrency bugs are considered to be severer than non-concurrency bugs, but the difference is relatively small.

RQ5: How long time is required to fix the severest concurrency bugs as compared to fix the severest non-concurrency bugs?

We analyze the fixing time of the severest concurrency bugs and the severest non-concurrency bugs to investigate how fast the severest bugs get fixed. Based on the severity classification given in Section 3.3, "Blocker" group defined for categorizing the bugs with highest priority, thus we extract the bugs tagged as "Blocker" from our concurrency and non-concurrency bugs' data sets. We also calculate the fixing time for each bug as explained in Section 3.2. We analyze the fixing time of the bug with "Blocker" severity in order to compare the fixing time for the severest concurrency and non-concurrency bugs.

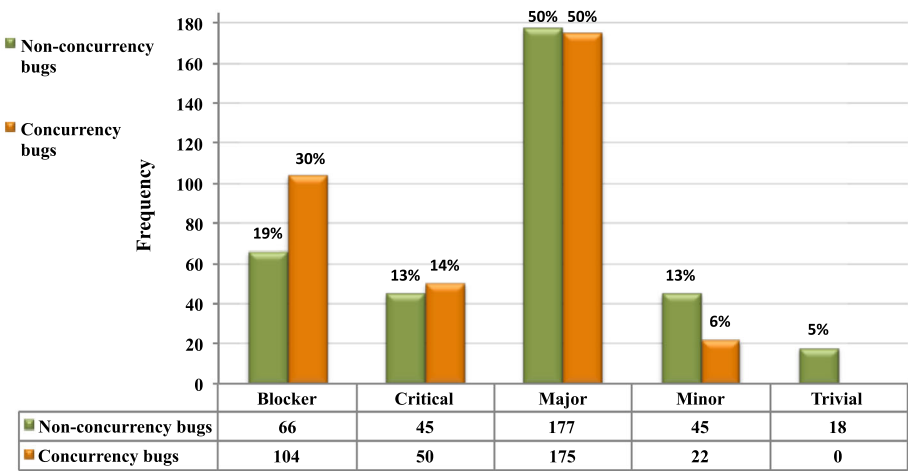


Fig. 9 Concurrency and non-concurrency bugs severity

Table 6 Kolmogorov-Smirnov test for concurrency and non-concurrency bugs severity comparison

Shade	Non-concurrency bugs			Concurrency bugs			$\lvert f(x) - g(x)\rvert$
	Observed frequency	Observed proportion	Observed cumulative proportion f(x)	Observed frequency	Observed proportion	Observed cumulative proportion g(x)	
Blocker	66	0.188034188	0.188034188	104	0.296296296	0.296296296	0.108262108
Critical	45	0.128205128	0.316239316	50	0.142450142	0.438746439	0.122507123
Major	177	0.504273504	0.820512821	175	0.498575499	0.937321937	0.116809117
Minor	45	0.128205128	0.948717949	22	0.062678063	1	0.051282051
Trivial	18	0.051282051	1	0	0	1	0
Critical D-value = $D_{351,0.05}$ = 1.36 /$\sqrt{}$351 = 0.072				D-value = Sup $\lvert f(x) - g(x)\rvert$ = 0.122507123			

Table 8 lists the detailed statistics on the obtained results for fixing time of concurrency and non-concurrency bugs with "Blocker" severity. As it shows the fixing time for concurrency and non-concurrency bugs with blocker severity are very similar, with an average of ∼57 and ∼54 days for fixing concurrency and non-concurrency bugs, respectively. Figure 11 also summarizes the results of comparing the fixing time for concurrency and non-concurrency bugs in the form of box-plots.

In addition, to evaluate if there is any statistical difference among two data sets (blocker concurrency and non-concurrency bugs fixing time), we use *Wilcoxon Signed Rank* test. Also, we calculate the *Vargha-Delaney A-statistic* as a measure of effect size [28] for analyzing significance. The tested hypotheses are formulated as follows: the null hypothesis (H_0) is that fixing time of the blocker concurrency and blocker non-concurrency bugs sets have identical distributions. The alternative hypothesis (H_1) is that the distributions are different. Based on the p-value of *0,308*, which is larger than 0.05, we accept the (H_0). That is, the fixing time of concurrency bugs and non-concurrency bugs have statistically identical distributions. We calculate the Vargha-Delaney A-statistic and obtain a value of *0.52* which indicates a "small" standardized effect size [28]. From our results, we can see that the

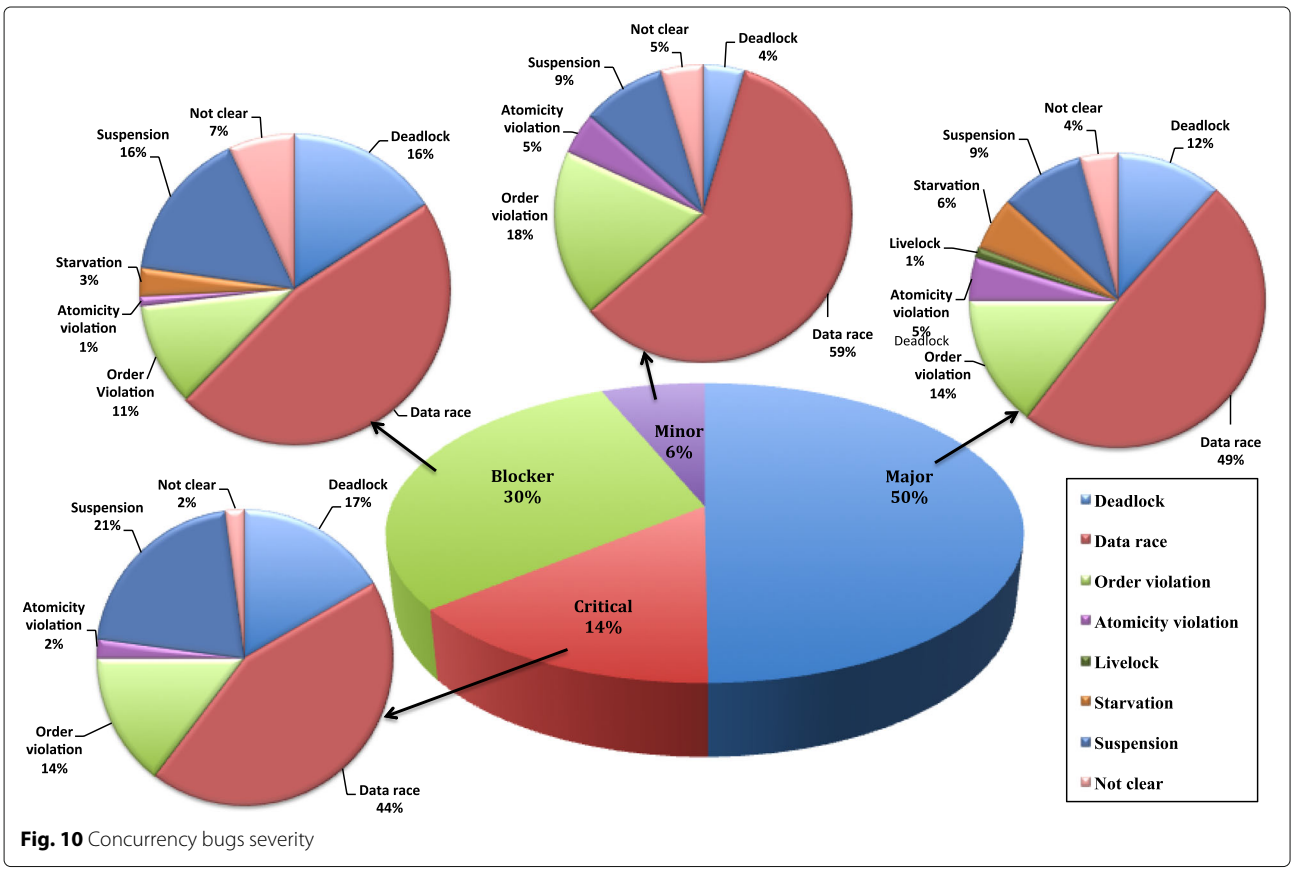


Fig. 10 Concurrency bugs severity

Table 7 Sorted concurrency bugs severity

Rank	Severity	Most frequent concurrency bug types
1	Blocker	–
2	Critical	Deadlock (17%)
		Suspension (21%)
3	Major	Atomicity violation (5%)
		Starvation (6%)
		Livelock (1%)
4	Minor	Data race (59%)
		Order violation (18%)
		Atomicity violation (5%)
5	Trival	–

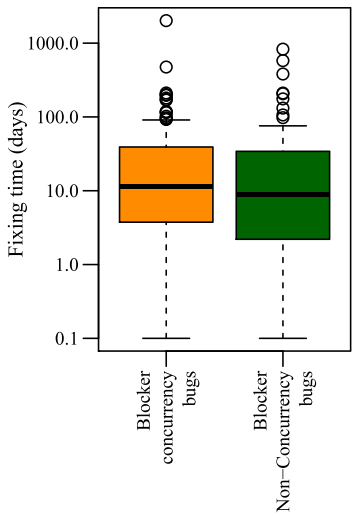


Fig. 11 Fixing time comparison for blocker concurrency and blocker non-concurrency bugs

fixing time for blocker concurrency bugs is not different from the fixing time for blocker non-concurrency bugs and this similarity corresponds to a "small" standardized effect size.

Answer RQ5: Fixing time for the severest concurrency bugs is statistically similar to fixing time for the severest non-concurrency bugs.

5 Discussion

In our study, we find a much smaller share of concurrency bugs than the one found by other similar studies. This could possibly be due to the different bug-source software and case studies of our study and other similar studies. An interesting observation is that 52% of 351 concurrency bugs were reported in the five-year interval of 2006–2010, and the remaining 48% were reported in the five-year interval of 2011–2015.

Similarly, the fixing time found by prior studies is much larger for concurrency bugs than for non-concurrency bugs. We find a difference, but it is relatively small. The large portion of fixing time in other studies could be due to reproducing the bugs and using the bug scenarios in the bug descriptions. However, in our study as shown in Fig. 6, we find few reports stating difficulties in reproducing the bug (4% in compare to 77%).

The involvement of more than one thread cause a concurrency bug. For this reason, we predict to find that concurrency bugs were severer than non-concurrency bugs. However, we expected most of the "Blocker" bugs belong to *deadlock* type due to its characteristic and properties but this was not the case. In our study *Data race* was the biggest portion of "Blocker". Our interpretation is here that *Data races* are the most problematic bugs to fix in these projects.

Moreover, our investigation shows that about half of the concurrency bugs are of *Data race* type. The reason could be that there are not many approaches to avoid Data race bug in programming or the approaches are not well developed and explained for developers. About 13% of the bugs belongs to the *Suspension* type. By investigating the bug reports' descriptions and comments, we noticed that most of the *Suspension* bugs occurred when the developer put the code in waiting mode for an unnecessary long time (by using sleep() or wait() method), thereby causing a *Suspension* bug. We also noticed that, developers (or testers) in confronting with *Order violation* bugs were confusing. They typically were not being able to easily find the correct order of thread execution (lots of description in the bug report between the developers to figure out the real cause of the bug). In addition, as we expected *Livelock* has the lowest frequency of concurrency bug types. By reading the reports' descriptions and comments, we understand that most of developers were aware of *Livelock* and during the implementation they took care of this type of bug.

Table 8 Fixing time for the bugs with *Blocker* severity

	Fixing time				
Bug	Average	Minimum	Maximum	Median	Standard deviation
Concurrency	57.4	0.1	2025.0	11.3	206.1
Non-concurrency	54.3	0.1	831.4	8.9	134.1

Our analysis on unreproducible concurrency bugs illustrates that only 513 of reported bugs (∼4%) were unreproducible. Out of 513 bugs 203 bugs (∼40%) have at least one concurrency keyword on their descriptions. While 647 bugs out of 11860 (∼5%) contain concurrency keywords. Our interpretation is here that the number of unreproducible concurrency bugs are similar to the number of unreproducible non-concurrency bugs.

Furthermore, by comparing the number of two sets of reproduced and categorized bugs i.e., closed and fixed concurrency keywords matched and unreproducible concurrency keywords matched, our interpretation is here that the number of closed and fixed concurrency keywords matched bugs is two times bigger than the number of unreproducible concurrency keywords matched bugs.

5.1 Validity threats

In the design and execution of this study, there are several considerations that need to be taken into account as they can potentially limit the validity of the results obtained. These considerations are listed below:

- We have chosen the five projects for our study. Based on the their characteristics (given in Section 2.1), we consider the projects likely to be representative for browsers, desktop GUI, and server programs that use concurrency. However, other projects (e.g., implemented in other programming languages) might show different concurrency bug characteristics. For instance, prior studies [1, 7] shows *deadlocks* in MySQL represent 40% of the total number of concurrency bugs while in our project, *deadlocks* account for a mere 13% of the concurrency bugs. Nevertheless, for *atomicity violation* and *order violation*, our results are similar to prior findings [1].
- The reports with other status such as "In Progress" (i.e., this issue is being actively worked on at the moment by the assignee) or "Open" (i.e., This issue is in the initial "Open" state, ready for the assignee to start work on it) were not considered in this study. However, there is a chance that we did not include the relevant reports from these groups.
- It is possible - or even likely - that the search query may have failed to identify some actual concurrency bugs. It should however be noted that we used more keywords compared to previous similar studies (see e.g., [1, 7]) and we argue that a concurrency bug report that does not contain any of the keywords is likely to be incomplete.
- Some concurrency bugs might go unfixed because they occur infrequently or only on certain platforms/software configurations. It would be interesting to consider these kinds of bugs, but they are not likely to have detailed discussions. As a result, these bugs are not considered as important as the reported and fixed concurrency bugs that are used in our study.
- We used Seven categories (type) and manual categorization for concurrency bugs. We excluded bugs which did not have sufficiently detailed information to be categorized. This procedure can lead to erroneously discarding some concurrency bugs.
- It should be noted that the fixing time measure used is elapsed calendar time and does hence not necessarily reflect the actual effort spent. It could for instance be the case that less severe bugs are put on hold for some time at times when efforts need to be devoted to severer bugs or other tasks. However, it seems likely that there at least at an aggregated level is a reasonable positive correlation between calendar time and effort.
- Since all unreproducible reported bugs did not have sufficient and detailed discussions thus we could not continue further and map their descriptions (or explanation) with our concurrency bug classification. However, we believe that the amount of extracted bugs can provide an overview of unreproducible bugs in general.
- Even if the obtained results (for all research questions RQ1-RQ5) are based on data samples from the five projects, these results might apply to other software as well. More analysis is required to confirm whether this is in fact the case.

6 Related work

A series of related studies on debugging, predicting and fixing concurrent software have been conducted. In particular, there is a large body of studies on prediction [29–32] and propagation [33, 34] of bugs in source code. Most of these studies strive to identify the components or source code files, that are most prone to contain bugs. Fault prediction partially focuses on understanding the behavior of programmers and its effects on software reliability. This work is complementary to the study conducted in this research, which is concentrated on a specific type of bugs (i.e., concurrency bugs) and on understanding their consequences.

In [35], Vandiver et al. analyzed the consequences of bugs for three database systems. This work is focused on presenting a replication architecture, instead of on studying bugs. The authors did not distinguish between concurrency and non-concurrency bugs, and only evaluated whether they caused crash or Byzantine faults.

Three open-source applications bug databases (Apache web server, GNOME desktop environment and MySQL database) are investigated by Chandra and Chen [36], again in a study with a slightly different focus than ours.

The authors analyzed all types of bugs (only 12 of them were concurrency bugs) to determine the effectiveness of generic recovery techniques in tolerating the bugs. Concurrency bugs are only one possible type of bug that affects their results. In contrast, based on our main objective we focus on a more narrow type of bugs by limiting our study scope to concurrency bugs, but provide a broader analysis (comparing concurrency and non-concurrency bugs) taking into consideration several types of these bugs.

Farchi et al. [37] analyzed concurrency bugs by creating such bugs artificially. They asked programmers to write codes which have concurrency bugs. We believe that artificially creating bugs may not lead to bugs that are representative of the real-world software bugs. We, on the other hand, analyze the bug database of an open-source software, which is well maintained, and widely used software.

Lu et al. examined concurrency bug patterns, manifestation, and fix strategies of 105 randomly selected real-world concurrency bugs from four open-source applications (MySQL, Apache, Mozilla and OpenOffice) bug databases [1]. Their study focused on several aspects of the causes of concurrency bugs, and the study of their effects was limited to determining whether they caused deadlocks or not. We use the similar study methodology to find relevant bug reports for our analysis but we provide a complementary angle by studying the effects of recent concurrency bugs with a more fine-grained classification than mapping bugs into deadlock and not-deadlock bug classes.

7 Conclusion and future work

This paper provides a comprehensive study of 11860 fixed bug reports from a widely used open source storage designed for big-data applications. The study covers the fixed bug reports from the last ten years, with the purpose of understanding the differences between concurrency and non-concurrency bugs. Two aspects of these reports are examined: fixing time and severity. Based on a structured selection process, we ended up with 351 concurrency bugs and 351 non-concurrency bugs (sampled). By analyzing these reports, we have identified the frequencies of concurrency, non-concurrency and unreproducible bugs. The study also helped us to recognize the most common type of concurrency bugs in terms of severity and fixing time. The main results of this study are: (1) Only a small share of bugs is related to concurrency while the vast majority are non-concurrency bugs. (2) Similarly, only a small share of bugs are unreproducible and related to concurrency issues (2%). (3) Fixing time for concurrency and non-concurrency bugs is different but this difference is relatively small. (4) Concurrency and non-concurrency bugs are different in terms of severity, while concurrency bugs are severer than non-concurrency bugs. (5) Fixing time for the severest concurrency bugs and the severest non-concurrency bugs is similar. These findings could help software designers and developers to understand how to address concurrency bugs, estimate the most time-consuming ones, and prioritize them to speed up the debugging and bug-fixing processes.

Endnotes

[1] https://issues.apache.org/jira/browse/HADOOP

[2] MapReduce is a programming model and an associated implementation for processing and generating large datasets that is amenable to a broad variety of real-world tasks [38].

[3] https://issues.apache.org/jira/browse/ZOOKEEPER

[4] https://issues.apache.org/jira/browse/OOZIE

[5] https://issues.apache.org/jira/browse/ACCUMULO

[6] https://issues.apache.org/jira/browse/SPARK/

[7] https://www.atlassian.com/software/jira

[8] Bug is "a problem which impairs or prevents the functions of the product" [15].

[9] This issue (bug) could not be reproduced at this time, or not enough information was available to reproduce the issue [15].

[10] Bug reproducibility indicates the reproduction of software failure(s) caused by bug(s).

Acknowledgements
This research is supported by Swedish Research Council (VR) through the EXACT project, and the Swedish Knowledge Foundation (KKS) through grant 20130258. Additionally, we would like to thank Eduard Paul Enoiu for interesting discussions and valuable feedback.

Authors' contributions
SAA is the main driver and author of all parts. DS, SE and HH have contributed in ideas, discussion, review and revision of all sections of the paper. All authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Author details
[1] Mälardalen University, Västerås, Sweden. [2] Ericsson AB, Stockholm, Sweden.

Received: 24 August 2016 Accepted: 13 March 2017
Published online: 04 April 2017

References

1. Lu S, Park S, Seo E, Zhou Y. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ACM Sigplan Notices. Washington: ACM; 2008. p. 329–39.
2. Peri R. Software development tools for multi-core/parallel programming. In: 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging. Seattle: ACM; 2008. p. 9.

3. Zhang W, Sun C, Lim J, Lu S, Reps T. Conmem: Detecting crash-triggering concurrency bugs through an effect-oriented approach. ACM Trans Softw Eng Methodol (TOSEM). 2013;22(2):10.
4. Desouza J, Kuhn B, De Supinski BR, Samofalov V, Zheltov S, Bratanov S. Automated, scalable debugging of MPI programs with Intel® Message Checker. In: Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications. St. Louis: ACM; 2005. p. 78–82.
5. Godefroid P, Nagappan N. Concurrency at Microsoft: An exploratory survey. In: CAV Workshop on Exploiting Concurrency Efficiently And Correctly. Princeton; 2008.
6. Süß M, Leopold C. Common mistakes in OpenMP and how to avoid them. In: OpenMP Shared Memory Parallel Programming. Academic Press Ltd.; 2008. p. 312–23.
7. Fonseca P, Li C, Singhal V, Rodrigues R. A study of the internal and external effects of concurrency bugs. In: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on. Chicago: IEEE; 2010. p. 221–30.
8. Joshi SB. Apache Hadoop Performance-tuning Methodologies and Best Practices. In: Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering. ICPE '12. New York: ACM; 2012. p. 241–2.
9. Polato I, Ré R, Goldman A, Kon F. A comprehensive view of Hadoop research? A systematic literature review. J Netw Comput Appl. 2014;46: 1–25.
10. Medeiros A. ZooKeeper's atomic broadcast protocol: Theory and practice. Technical report. 2012.
11. Kamburugamuve S, Fox G, Leake D, Qiu J. Survey of apache big data stack. Indiana University, Tech. Rep. 2013.
12. Dendek PJ, Czeczko A, Fedoryszak M, Kawa A, Wendykier P, Bolikowski Ł. Content analysis of scientific articles in apache hadoop ecosystem. In: Intelligent Tools for Building a Scientific Information Platform: From Research to Implementation. Springer; 2014. p. 157–72.
13. Shoro AG, Soomro TR. Big data analysis: Apache spark perspective. Glob J Comput Sci Technol. 2015;15(1):.
14. Abbaspour Asadollah S, Hansson H, Sundmark D, Eldh S. Towards classification of concurrency bugs based on Observable properties. In: 1st International Workshop on Complex Faults and Failures in Large Software Systems. Italy: IEEE Press; 2015.
15. What is an Issue - Atlassian Documentation. 2015. https://confluence.atlassian.com/jira063/what-is-an-issue-683542485.html. Accessed 12 June 2016.
16. Abbaspour Asadollah S, Sundmark D, Eldh S, Hansson H, Afzal W. 10 years of research on debugging concurrent and multicore software: a systematic mapping study. Softw Qual J. 2016;25:1–34.
17. Yoshiura N, Wei W. Static data race detection for java programs with dynamic class loading. In: Internet and Distributed Computing Systems. Springer; 2014. p. 161–73.
18. Henningsson K, Wohlin C. Assuring fault classification agreement - an empirical evaluation. In: International Symposium on Empirical Software Engineering, ISESE'04. Redondo beach CA: IEEE Computer Society; 2004. p. 95–104.
19. Akhter S, Roberts J, Vol. 33. Multi-core Programming: Richard Bowles; 2006.
20. Bhatia Y, Verma S. Deadlocks in distributed systems. Int J Res. 2014;1(9): 1249–52.
21. Gove D. Multicore Application Programming: For Windows, Linux, and Oracle Solaris: Addison-Wesley Professional; 2010.
22. Chapman B, Jost G, Van Der Pas R, Vol. 10. Using OpenMP: Portable Shared Memory Parallel Programming: MIT press; 2008.
23. Stallings W, Vol. 7th. Operating Systems- Internals and Design Principles: Prentice Hall Englewood Cliffs; 2012.
24. Lin S, Wellings A, Burns A. Supporting lock-based multiprocessor resource sharing protocols in real-time programming languages. Concurr Comput Pract Experience. 2013;25(16):2227–51.
25. Jayasinghe D, Xiong P. CORE: Visualization tool for fault localization in concurrent programs. Citeseer; 2010.
26. Park S, Vuduc RW, Harrold MJ. Falcon: Fault Localization in Concurrent Programs. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1. New York: ACM; 2010. p. 245–54.
27. Qi S, Otsuki N, Nogueira LO, Muzahid A, Torrellas J. Pacman: Tolerating asymmetric data races with unintrusive hardware. In: High Performance Computer Architecture, 2012 IEEE 18th International Symposium on. USA: IEEE; 2012. p. 1–12.
28. Vargha A, Delaney HD. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J Educ Behav Stat. 2000;25(2):101–32.
29. Neuhaus S, Zimmermann T, Holler C, Zeller A. Predicting vulnerable software components. In: Proceedings of the 14th ACM Conference on Computer and Communications Security. Alexandria: ACM; 2007. p. 529–40.
30. Nagappan N, Ball T. Static analysis tools as early indicators of pre-release defect density. In: 27th International Conference on Software Engineering. Saint Louis: ACM; 2005. p. 580–6.
31. Rahman F, Khatri S, Barr ET, Devanbu P. Comparing static bug finders and statistical prediction. In: Proceedings of the 36th International Conference on Software Engineering. Hyderabad: ACM; 2014. p. 424–34.
32. Lewis C, Lin Z, Sadowski C, Zhu X, Ou R, Whitehead EJ. Does bug prediction support human developers? findings from a google case study. In: Software Engineering (ICSE), 2013 35th International Conference on. San Francisco: IEEE; 2013. p. 372–81.
33. Voinea L, Telea A. How do changes in buggy Mozilla files propagate? In: Proceedings of the 2006 ACM Symposium on Software Visualization. Brighton: ACM; 2006. p. 147–8.
34. Pan WF, Li B, Ma YT, Qin YY, Zhou XY. Measuring structural quality of object-oriented softwares via bug propagation analysis on weighted software networks. J Comput Sci Technol. 2010;25(6):1202–13.
35. Vandiver B, Balakrishnan H, Liskov B, Madden S. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. ACM SIGOPS Oper Syst Rev. 2007;41(6):59–72.
36. Chandra S, Chen PM. Whither generic recovery from application faults? A fault study using open-source software. In: Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on. New York: IEEE; 2000. p. 97–106.
37. Farchi E, Nir Y, Ur S. Concurrent bug patterns and how to test them. In: Parallel and Distributed Processing Symposium. Washington: IEEE; 2003. p. 7.
38. Dean J, Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. Commun ACM. 2008;51(1):107–13.

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Immediate publication on acceptance
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com