Preserving Reactiveness: Understanding and Improving the Debugging Practice of Blocking-Call Bugs

Arooba Shahoor
Kyungpook National University
Daegu, South Korea
arooba@knu.ac.kr

Jooyong Yi
UNIST
Ulsan, South Korea
jooyong@unist.ac.kr

Dongsun Kim*
Korea University
Seoul, South Korea
darkrsw@korea.ac.kr

Abstract

Reactive programming reacts to data items as they occur, rather than waiting for them to complete. This programming paradigm is widely used in asynchronous and event-driven scenarios, such as web applications, microservices, real-time data processing, IoT, interactive UIs, and big data. When done right, it can offer greater responsiveness without extra resource usage. However, this also requires a thorough understanding of asynchronous and non-blocking coding, posing a learning curve for developers new to this style of programming. In this work, we analyze issues reported in reactive applications and explore their corresponding fixes. Our investigation results reveal that (1) developers often do not fix or ignore reactiveness bugs as compared to other bug types, and (2) this tendency is most pronounced for blocking-call bugs – bugs that block the execution of the program to wait for the operations (typically I/O operations) to finish, wasting CPU and memory resources. To improve the debugging practice of such blocking bugs, we develop a pattern-based proactive program repair technique and obtain 30 patches, which we submit to the developers. In addition, we hypothesize that the low patch acceptance rate for reactiveness bugs is due to the difficulty of assessing the patches. This is in contrast to functionality bugs, where the correctness of the patches can be assessed by running test cases. To assess our hypothesis, we split our patches into two groups: one with performance improvement evidence and the other without. It turns out that the patches are more likely to be accepted when submitted with performance improvement evidence.

CCS Concepts

• **Software and its engineering → Software maintenance tools**; **Software evolution**; **Maintaining software**; **Error handling and recovery**.

Keywords

proactive debugging, program repair, reactive programming, blocking calls, non-intrusive patches

*Corresponding author.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680319

ACM Reference Format:
Arooba Shahoor, Jooyong Yi, and Dongsun Kim. 2024. Preserving Reactiveness: Understanding and Improving the Debugging Practice of Blocking-Call Bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680319

1 Introduction

Reactive programming has been gaining traction in recent years. Many software vendors, including Netflix and Uber, have adopted reactive programming for their applications, which require high responsiveness and scalability. The data-oriented concept of reactive programming deviates from the classical control-oriented programming paradigm and hence poses different challenges not observed in traditional programs.

In this work, we study the common errors and their fixes observed in reactive programs. In particular, we focus on blocking-call bugs, a type of error that is very frequently encountered in this programming paradigm [14]. Typically, reactive programs process streams of data using chains of functions. If a function in the chain is blocked, for example, to wait for a network response, the entire chain is blocked. While reactive libraries such as Reactor [24] and RxJava [25] make it easy to construct a chain of functions, they alone do not prevent functions from being blocked. In this paper, we develop a technique to automatically fix blocking-call bugs.

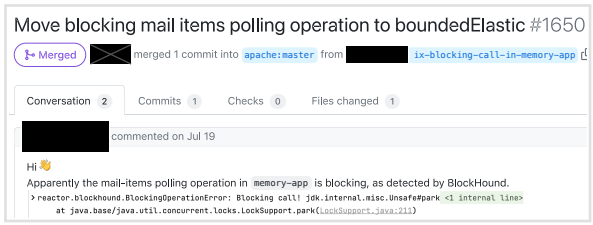
To the best of our knowledge, this is the first study that investigates the blocking-call bugs and their fixes in reactive programs and proposes a repair technique for them. Our study consists of the following three stages.

- **Stage 1:** We first examine the bug types observed in 29 open-source reactive projects built with reactive libraries such as Reactor [24] and RxJava [25]. We count the number of reactiveness bugs fixed and unfixed by developers against those of non-reactiveness bugs. It turns out that reactiveness bugs are less likely to be fixed, and among them, blocking-call bugs are the most common type.
- **Stage 2:** We develop an automated program repair (APR) technique to fix blocking-call bugs. The pattern-based program repair [66] has shown to be effective for fixing specific types of bugs [77, 84]. We apply this technique to blocking call bugs. We extract five fix patterns from `GitHub` and `StackOverflow`. All the curated patterns are non-intrusive, meaning they do not change the functionality of the program. We applied our technique to 30 blocking-call bugs for which we could identify the presence of the bug. Our technique successfully fixed all 30 bugs without causing regression.

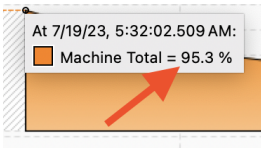
- **Stage 3:** Developers often ignore bug reports even when the bug is reproducible [12] or decline to accept a patch [6]. A previous study [64] noted that pull requests associated with performance enhancements are often given less precedence than those addressing functionality issues. We hypothesize that developers would be more likely to accept the patch if it is accompanied by evidence demonstrating the bug has been fixed. For blocking-call bugs, performance analysis results before and after fixing the bug could serve as compelling evidence. Figure 1 shows an example of performance analysis. To validate this hypothesis, we conduct a comparative study. We create pull requests (PRs) with the patches for 30 blocking-call bugs. Half of these PRs are supplemented with evidence of the bug fix, while the other half are without it. The result of this study reveals that PRs with the evidence are more acceptable.

In summary, this paper contributes the following:

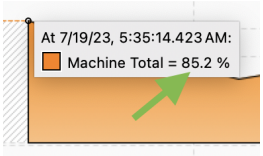
- Investigation results of the fix ratio between reactiveness bugs and other types of bugs.
- Pattern-based repair approach to generating patches for blocking-call bugs.
- Comparative study results of the fix ratio between patches submitted together with and without the evidence of fixing.



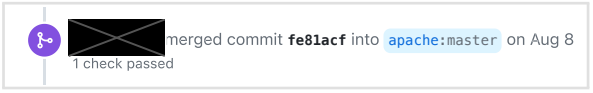
(a) Reported blocking-call issue with stack traces of the symptom in the `apache/james` project. The corresponding patch has been attached to this pull request.



(b) Performance measurement results before applying the patch.



(c) Performance measurement results after applying the patch.



(d) Developers of the project reviewed the patch and performance analysis results. This pull request has been merged after the review with multiple positive comments.

Figure 1: Pull request reporting and fixing a blocking call bug in the `apache/james` [20] project. A patch is proactively submitted and performance improvement evidence have been submitted together. This pull request has been merged.

2 Background

2.1 Reactive Programs

The Reactive programming paradigm is widely applied in web development, cloud computing, the Internet of Things (IoT), and real-time data processing, among other areas. Such applications employ reactive libraries (such as Reactor [24], RxJava [25], and Vert.x [13]) to handle a large number of concurrent requests and process real-time data streams from multiple sources (as in microservices or IoT apps) asynchronously, keeping the application responsive and resource-efficient.

When used correctly, reactive programming can help applications handle increasingly more requests with fewer number of threads. Based on the publish-subscribe protocol, reactive APIs react to data items (events) as they occur leveraging the underlying event loop model [22]. The event loop handles multiple operations on a single thread, using callbacks from any operation that might take a long time to complete. However, this approach of programming comes with its own set of challenges, the biggest of which is the paradigm shift from the imperative programming model.

Reactive programming deals with asynchronous data streams, event-driven programming, and complex data flow patterns. This requires, from developers, a thorough understanding of the pub-sub design pattern and appropriate usage of reactive operators to ensure the application remains reactive end to end, and the main execution thread never blocks; being fully reactive is key. To this end, this paper focuses on the bugs found in reactive programs that hamper their reactivity, and the corresponding fixes.

```
1  public ReactorRabbitMQChannelPool(Mono<Connection> connectionMono,
       Configuration configuration) {
2      ...
3      newPool = PoolBuilder.from(connectionMono.flatMap(this::openChannel))
4          .sizeBetween(1, configuration.maxChannel)
5          .destroyHandler(
6              channel -> Mono.fromRunnable(
7                  Throwing.runnable(() -> {
8                      if (channel.isOpen()) channel.close();
9                  }))
10             + .then().subscribeOn(Schedulers.boundedElastic())
11         ).buildPool();
12 }
```

Figure 2: Example of a reactive Java program [16].

Figure 2 shows an example of a reactive program, where a popular reactive library, Reactor, is used. The usage of the other reactive libraries is similar. This example is excerpted from a real-world software project, James[1]. Lines 3–11 build a pool of channels. Specifically, the `from` method call at Line 3 specifies that each channel should be created by invoking the `openChannel` method. Then, `sizeBetween` at Line 4 specifies the number of channels to be created. At Line 5, the `destroyHandler` method call assigns the callback method to be invoked when the channel is destroyed. Let us disregard Line 10 for the moment. Finally, `buildPool` at Line 11 constructs the pool of channels as specified.

2.2 Blocking-call Bugs

While reactive libraries make it easy to write a reactive program in a declarative style, care must be taken to actually gain the efficiency of reactive programming. The code in Figure 2, without Line 10, does not run efficiently. When a pool of channels is destroyed, the function defined in Line 7–9 is called for each channel. The problem is that, without Line 10, each channel is handled sequentially,

[1]Java Apache Mail Enterprise Server. https://github.com/apache/james-project

even though their outputs are identical to those of the version with Line 10. Thus, until the handling of the current channel is completed, the next channel cannot be executed. In other words, the handling of the current channel blocks the handling of the next channel. Contrary to imperative programming, in the reactive pipeline, blocking even a single request impacts the latency of all other requests since it has limited threads to fall back on. If an executing thread is not released, the pool will be exhausted very soon, resulting in the freezing of the entire event loop [61].

Line 10 fixes this blocking-call bug by delegating the handling of channels to multiple different threads. Thus, multiple channels can be handled in parallel, and the efficiency of reactive programming can be achieved. In fact, this patch is obtained by running our tool presented in Section 4.2 (the FP1 pattern is used[2]) and our pull request has been merged into the project.

The blocking-call bugs are less likely to be fixed than other types of bugs. First, these bugs are hard to detect as the functional properties of the buggy program are identical to a non-buggy program; non-functional properties, such as execution time or memory footprints, are different in general. Secondly, a blocking bug or its performance impact is hard to reproduce. Finally because performance bugs often require a large refactoring [6], the developers may switch such bug reports to feature requests [12].

2.3 Proactive Program Repair

Unlike many automated program repair (APR) techniques, which use test suites to localize a patch location and validate patch candidates, our pattern-based program repair approach does not require test cases. This is a *proactive* approach to program repair, that can be applied to the target program *before* a bug-revealing test case is found.

Reproducing the symptoms with test cases may not always be possible as test cases are not available for some types of bugs, such as memory bloating and performance issues. Traditional APR pipelines cannot work for such bugs. This is where proactive approaches can prove advantageous given that the fix is non-intrusive (i.e., functionality-preserving). The reason is that if the fix is simple and non-intrusive, replication of the bug becomes supplementary rather than a necessity.

Such proactive program repair approaches have been successfully applied to fix non-functional bugs [63, 77, 84]. In this work, we extend the idea to fix blocking-call bugs in reactive programs. As the patches do not change the functionality of the target program, this approach serves to fix as many (potential) non-functional bugs as possible.

3 Study Design

3.1 Overview

Our empirical study has three stages as shown in Figure 3: (1) inspecting the debugging practice of bugs in open-source projects using reactive libraries, (2) generating non-intrusive patches for fixing blocking-call bugs, and (3) submitting the patches as pull requests and investigating how they are accepted. The purpose of the study is to understand how reactiveness bugs (as a subset of *hard-to-reproduce* and *often-postponed* issues) are addressed in practice, and how we can improve the debugging practice.

Our study investigates the following research questions:

(1) **RQ1**: How different is the fix ratio between reactiveness bugs and non-reactiveness bugs?
(2) **RQ2**: Does our proactive repair technique generate non-intrusive patches fixing blocking-call bugs?
(3) **RQ3**: Are the patches more acceptable if they are submitted with improvement evidence?

RQ1: It is intuitive that developers would immediately resolve bugs that are easy to reproduce and impact the program functionality. However, non-functional bugs do not break functionality and are often difficult to replicate. Reactiveness is one such non-functional feature. This research question aims to garner numerical statistics on the proportion of reactiveness bugs addressed when compared to other bugs in the same application.
RQ2: After confirming the lack of attention given to reactiveness bugs, we aim to improve their fix ratio, without relying on bug detection techniques. We hypothesize that if the repair is non-intrusive (easy, simple, and not disrupting functionality), it will be more readily accepted by developers. To this end, we devise a pattern-based method for fixing blocking-call bugs in reactive apps using past bug/fix patterns. This research question evaluates how effective is our proactive technique in resolving such bugs non-intrusively.
RQ3: While RQ2 stems from the assumption that non-intrusive patches might enhance the chances of addressing non-functional bugs, this RQ seeks to explore whether showcasing performance enhancements would prompt developers to willingly embrace their fixes.

3.2 Stage 1: Measuring the Fix Ratio

We hypothesize that reactiveness bugs are not given much precedence in comparison to other bug types. Therefore, the objective of the first stage is to measure and compare the fix ratio between bugs relevant to reactiveness and other types. Note that since the focus of this study is the preservation of the application's reactiveness and exploring bugs that violate it, we made two generic classifications for this comparative analysis: reactiveness bugs and other bugs (not pertaining to reactiveness). Further exploration of other bug types is not conducted.

As the first step in this stage, we collect open-source projects utilizing reactive libraries. We then count the reported bugs and classify them into bugs relevant to reactiveness issues and bugs related to other issues. We then compute the number of fixed and unfixed bugs in each class.

To curate projects for our study, we conduct a search on `GitHub` using the following criteria:

- **Domain.** We consider projects using the following popular reactive libraries: Reactor [24], RxJava [25], and Vert.x [13].
- **Maintained.** We choose projects that are still being maintained and have been updated within the year leading up to Jan 19, 2022. Archived projects are not considered.
- **Number of contributors.** Projects with at least 10 contributors are selected. Personal projects are not taken into account. Any

[2] `then()` transforms `Mono<Object>` to `Mono<Void>` which is required by `destroyHandler`.

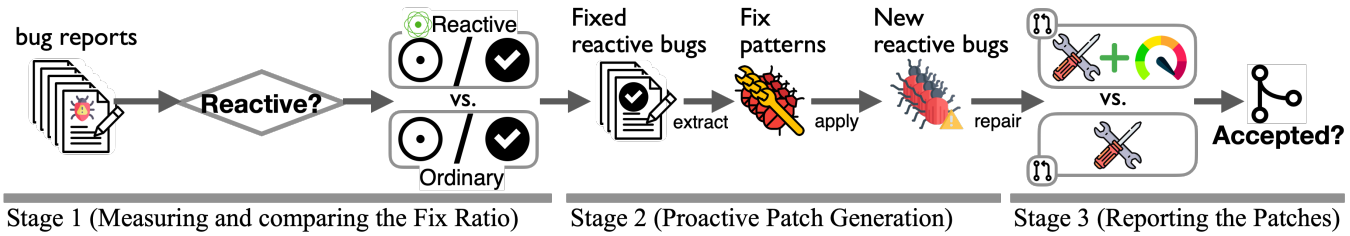


Figure 3: Overview of our study.

project with 10+ contributors not only reflects the scale of the project but also ensures that the response towards a reported issue (addressing or ignoring) is not a mere personal preference of a few [81, 84].

- **Number of commits.** The selected projects have at least 100 commits on their GitHub repository. This is to ensure that the result of our study is applicable to projects of non-trivial scale since a greater number of commits intuitively reflect a larger code base [84].
- **Popularity.** Projects with at least 10 stargazers, watchers, or forks were selected. A number of studies used this threshold for ensuring a decent outreach and impact of the study subjects [81, 84, 88, 89].

Two authors are involved in this process. The search and filtering are performed manually, without the use of any API. As a result, 29 projects are curated. For each selected project, we examine *issues* to collect bug reports. In the *issues*, we search for keywords such as "*reactive*", "*rxjava*", and "*reactor*". We then manually go through each of the reported issues, reviewing the content of the post itself, the comments, and any linked PR, to verify that the issue indeed pertains to reactiveness. We group the bugs into three categories: `fixed`, `unfixed` and `won't fix` based on the issue status (Open or Closed) as well as default and custom labels such as `resolved`, `not planned`, `wontfix`, `invalid` etc. Note that for *unfixed* issues, we further ensured their reporting time to be at least a year old to validate our comparison of the ratio of fixed vs. unfixed bugs.

3.3 Stage 2: Patch Generation

Based on the observations from the first stage (Section 3.2), we design an automated program repair (APR) technique to help debug blocking-call bugs. Although reactiveness bugs are less likely to be fixed by developers, there are still some fixed cases with patches from which we extract common fix patterns. Note that proactive approaches with fix patterns [77, 84] are effective in fixing non-functional bugs.

In order to collect the recurring fix patterns for blocking calls, we first collect all the blocking call cases in reactive Java projects. Two of the authors are assigned to this task. We manually search issues on GitHub (GH) and StackOverflow (SO) with specific keywords. Although this search and pattern extraction is a manual process, it has been employed by several studies [65, 66, 69, 72, 73, 77, 79] and has proven to be effective, as once the patterns have been identified, they can be used many times to address relevant blocking calls. We extract common patterns per the following procedure:

(1) For `StackOverflow`, we search for keywords such as 'blocking', 'blocking call', and 'blocking in non-blocking method', along with 'reactive', 'reactive Java', or simply 'Java'. Then, from the results, we select the top 1,000 results ranked by 'best match', whose posts, comments, or answers contain the specified keywords. The actual count of search results was a little greater than 1000; however, we chose to review the first 1000 as it suffices to extract common patterns [84].
(2) In the case of `GitHub`, 1,000 PRs, issues, commits, and discussions are searched, using the same keywords and ranking criteria. Note that '1000' does not imply 1000 PRs and an equal number of commits, discussions, etc. Instead, the count of search results for each category (commits, discussions, etc.) averaged around 300+, 200+, and so forth. We aggregated the search results across all categories to reach a cumulative count of 1000 reviewed results, aligning with that on `StackOverflow`.

After assessing the search results, we curate fix patterns that are accepted as answers in at least two SO posts or two merged commits in GH, and which we are able to apply and verify locally as valid, i.e., they indeed remove the blocking call exceptions. As mentioned earlier, blocking call bugs are generally paid less attention to, hence we did not have a large data source to extract patterns from. Therefore, as an additional measure, we examine the fix patterns by applying profilers (JFR [3] and VisualVM [37]). Based on profiling results, only those patterns are considered that show performance improvement. The results of fix pattern mining are described in Section 4.2.

Based on the fix patterns, our proactive approach follows the common steps of existing APR approaches to fixing non-functional bugs [77, 84]: (1) parsing the source code, (2) identifying locations to apply patches, and (3) creating patches. Basically, the approach scans the target program and applies the steps to all source code files in the program as follows:

- **Parsing the Source Code**: We make use of `JavaParser` [17] for accessing and modifying Java source code via the Abstract Syntax Tree (AST). With more than 5000 stars and 1000+ forks of their GitHub repository, JavaParser is the most reliable choice for automating the AST modifications based on our curated fix patterns. Since our prototype tool is implemented as a CLI tool, accepting project path as an argument, we make use of `JavaParser`'s `SymbolSolverCollectionStrategy` to locate the project root and the containing source root(s). Then, leveraging the `CompilationUnit` [9] of each source root, the child nodes are traversed until it detects the node type of interest (e.g., a blocking method call).
- **Applying Patches**: If a blocking call pattern is detected, its corresponding fix pattern is implemented directly in the AST. The modification might require creating new nodes or removing some. Once the fix(es) have been applied to all the compilation

units within a source root, the parsed files are saved implicitly by `JavaParser` upon saving the source root. `LexicalPreservingPrinter` [18] is chosen as the printer. It ensures that the code that is not modified by our transformations remains completely unchanged with the original formatting. For the new AST nodes (not parsed from source code), since there is no original format to preserve, it defaults to the pretty-printed form, which follows the standard Java formatting.

- **Repeating and Reporting**: Our prototype tool repeats the above-mentioned process for all source roots with the identified project root. Upon identification of the blocking call pattern, it tracks and prints the containing class name so that the user can identify which class files to review for the modifications made.

3.4 Stage 3: Reporting the Patches

After examining 103 reported blocking-call issues across 29 projects, we categorize reasons for fixing and not fixing the issues (see Section 4.1.1). We observe that many reactive bugs, despite having working patches, may not be accepted by developers. This motivates us to supplement the patches with evidence of performance improvement. As a blocking-call bug affects the execution time, CPU usage, and thread status, we measure and visualize these values before and after the fix, and submit these results with our PRs, as illustrated by the James Project PR in Figure 1.

Our intuition here is that the likelihood of PR acceptance is higher if submitted with performance improvement evidence, along with the patches. We formulate the following null and alternative hypotheses to statistically test our intuition:

If the difference in acceptability is statistically significant, we can reject the null hypothesis, H_0. the evidence is not sufficient to persuade developers to accept the patches generated by our proactive approach.

- H_0: Patches submitted with improvement evidence have no acceptability difference from those without the evidence.
- H_a: Patches submitted with improvement evidence are more acceptable than those without the evidence.

When submitting PRs, we attach improvement (e.g., performance) analysis results to test the above hypotheses. Specifically, half of the PRs are submitted with the generated patches and the analysis results together, while another half of the PRs are submitted only with the patches. Then, we count the number of accepted and rejected PRs, and we compute the time-to-accept if a PR is accepted.

Note that all submitted PRs contain genuine patches, which were also manually validated. As mentioned, performance improvement evidence was not provided in half of the PRs; this is to mimic the common practice of submitting patches without such evidence. Moreover, if the developer does not approve of the patch via their comments, we update the PR with performance improvement evidence.

4 Results

4.1 Fix Ratio

Following the steps specified in Section 3.2 (Stage 1) led to the acquisition of 29 open-source projects employing reactive libraries. From these projects, we identified 33,210 issues in total (the average is 1,145 issues per project). The list of projects and their corresponding issues are available in our replication package [1].

Table 1: Outcomes of reactiveness and non-reactiveness bugs (33,210 in total) collected from 29 open-source projects, following the procedures described in Section 3.2.

Group	Outcome		
	Fixed	Unfixed	Total
Non-Reactiveness Bugs	29,244 (88.6%)	3775 (11.4 %)	33,019 (100.0%)
Reactiveness Bugs	115 (60.9%)	74 (39.1%)	189 (100.0%)

∗ The difference of the fix ratio between two groups (`Fixed` vs. `Unfixed` + `Won't Fix`) is statistically significant (p-value < 0.01).

∗ The list of the selected 29 projects and the outcomes of their bugs are available in our replication package [1].

Reactiveness bugs are less likely to be fixed than non-reactiveness bugs as shown in Table 1. Based on the procedure described in Stage 1 of Section 3.2, we classified 33,210 issues into 33,019 non-reactiveness and 189 reactiveness bugs. While 11.4% (3,775/33,019) of non-reactiveness issues remained unfixed, the ratio of unfixed reactiveness bugs is **39.1%** (74/189). The fix ratio of non-reactiveness and reactiveness bugs are 88.6% (29,244/33,019) and 60.9% (115/189), respectively. To figure out whether the difference between the fix ratio of the two groups (`Fixed` vs. `Unfixed`) is significant, we applied Barnard's exact test [58] to the number of bugs in Table 1. It turned out that the difference is statistically significant (p-value < 0.01).

Although the total number of bugs between the two groups is substantially different, the fix ratio of the groups effectively shows the difference in debugging practices. The results imply that developers tend to put less effort into bugs if they cannot be easily reproduced, for example, bugs relevant to reactiveness and performance.

To assess the number of reactiveness bugs (189) in terms of their prevalence, we conducted a small empirical study for another, most commonly occurring bug in Java: the Null Pointer Exception (NPE). After applying the search method the same as for reactiveness bugs, it turns out that even NPE accounts for only 667 issues. Furthermore, note that GitHub issues are used for multiple purposes: other than bug reports, Github is also a medium for feature requests, questions, discussions, and more.

4.1.1 *Common reactiveness bugs: Blocking-call.*

We identified that *blocking-call* (see Section 2.2) is one of the most common types of reactiveness bugs and they are even less likely to be fixed than other reactiveness bugs. There are 103 blocking-call bugs out of 189 reactiveness bugs across the 29 projects. **40.8%** (42/103) of the blocking-call bugs remained unfixed while 59.2% (61/103) of them are fixed[3]. To find out why blocking-call bugs are fixed or unfixed, we manually inspected all reports:

We review the content of the post, including any stack traces attached. The usernames of the reporter and issue closer are verified to identify self-reported issues. Additionally, we inspect any *linked* issues and pull requests, as well as the tags assigned by the project correspondent (e.g., `Invalid` , `question` , `waiting-for-triage` ,

[3]The types of reactiveness bugs other than blocking-call bugs are available in our replication package [1].

`feature-request`, etc.). Finally, we go through all comments on the post, if present.

As a result, we were able to categorize the rationales for fixing/not fixing such bug types. First, the following are the categories of the most *common reasons for developers to fix blocking call bugs:*

- Reactive libraries obviously threw a "blocking-call" exception: 36.1% (22/61).
- Self-reported and completed by a developer due to performance/scalability issues: 32.8% (20/61).
- Reported issues were successfully reproduced and deemed critical by developers: 16.3 % (10/61).
- Reasons were unclear: 8.2% (5/61).
- Issues were known and fixed, but the causes were somehow forgotten: 4.9% (3/61).
- Bug reporters showed how the bug could degrade the performances: 1.7% (1/61).

The above categorization results imply that developers tend to fix reactiveness bugs if the symptoms are obvious and easy to reproduce.

We were also able to classify *common reasons developers ignored or did not fix reactiveness bugs*:

- Developers assumed that the reporters used the program incorrectly, and they suggested a workaround instead. However, the reporters disagreed and the bugs remained unfixed: 23.8% (10/42).
- Developers assumed that the reported blocking-call operations were inevitable, and they asked to whitelist the operations: 23.8% (10/42).
- Ignored without any discussion: 16.7% (7/42).
- Developers refused to fix it due to the huge effort required, and they asked to whitelist the blocking-call operations: 14.2% (6/42).
- Developers assumed that the reported bugs were false positives, and the bugs were ignored: 7.1% (3/42).
- Marked as a feature request without any discussion: 4.8% (2/42).
- Discussions were made but no concrete solution was drawn: 4.8% (2/42).
- Issues were unable to be reproduced and no further discussions: 2.4% (1/42).
- Issues were deemed as a very unusual case: 2.4% (1/42).

Our inspection results may imply that developers put some effort into understanding the symptoms, but the resolutions were often workarounds or whitelisting because they failed to understand or refused to put more effort into fixing them. Moreover, even in some cases when the reporters suggested potential patches, the developers decided not to fix the bugs; our conjecture here is that the developers could not verify whether the suggested patch would actually fix the bug immediately, due to the nature of reproduction and assessment of blocking-call bugs. The developers might need to see the evidence of fixing the bugs by the suggested patches.

4.1.2 *Key factors contributing to blocking bugs in reactive programs.*

Based on our inspection to categorize blocking-call bugs, we identified the 3 most recurring causes for such bugs in reactive applications as follows:

Factor 1. Misuse of blocking operators. Figure 4 illustrates the use of `block()` [19] operator from the Reactor library in the `eventuate-common` project [7]. This operator is seemingly called on

Figure 4: Blocking call due to a `block()` operator [7].

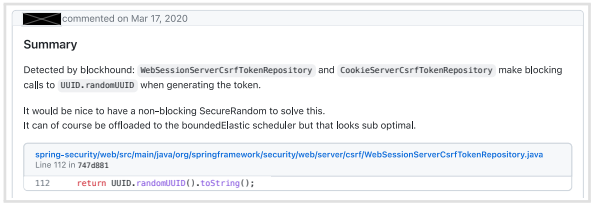


Figure 5: Blocking call due to an improper use of a Java utility [8].

the main thread as there are no schedulers that offload the operation to a blocking-compatible thread anywhere in the reactive code chain, hence triggering the blocking call error.

Factor 2. Improper use of utilities and third-party libraries. With an inadequate understanding of a library or utility and its use case, there is a high chance of employing its properties or methods in a way that may produce unexpected results. Figure 5 demonstrates an example scenario that uses Java's UUID utility within a reactive pipeline in the Spring-security project [8]. This utility performs some file read operation from the OS to create entropy [30] and hence blocks the thread during the process.

Factor 3. I/O bound operations in a non-blocking context. As mentioned above, in the reactive pipeline, blocking calls such as I/O operations are only allowed in specific contexts, such as I/O threads or blocking-compatible Scheduler threads. Consider the scenario in Figure 6, where a blocking I/O method is used in a non-blocking context, and though it is converted to `toCompletableFuture()` [10], it is then appended with the `join()` [11] operator. `join()`, just like Reactor's `block()`, blocks the thread until a result is returned.

Note that counting bugs within these three factors separately is not feasible as these cases are not exactly mutually exclusive. For example, at times, the developer performs I/O bound operations in a non-blocking context (Factor 3), by misusing blocking operators provided by the reactive library (Factor 1). The purpose of this section was to briefly highlight what we found as some of the main and recurring reasons for blocking call bugs in reactive projects. To avoid these scenarios, developers must become cognizant of writing asynchronous and non-blocking code, scheduling blocking operations on blocking-compatible threads, and the proper use and chaining of reactive operators. We will discuss these solutions (i.e., common fix patterns) in detail in Section 4.2. It is also important to note that Rx libraries' *blocking* methods (e.g., RxJava's `blockingFirst`, Reactor's `block` etc.) are permissible when the application is *partially* reactive, and the user wants to use the result of the reactive pipeline in the imperative part of the code.

Figure 6: Blocking I/O in non-blocking context [23].

RQ1: *How different is the fix ratio between reactiveness bugs and non-reactiveness bugs?*
Answer to RQ1: *The fix ratio of reactiveness bugs is significantly lower than that of other types of bugs. This implies that developers often ignore or postpone certain type of bugs (reactiveness bugs in this case). Blocking-call bugs are one of the most common unfixed and ignored bugs.*

4.2 Patch Generation

Following the procedure described in Section 3.3, we curated the fix patterns and applied them to the open-source projects containing the unfixed blocking-call bugs mentioned in Table 1 and Section 4.1. We will first provide brief descriptions for each of the recurring fix patterns we curated that address the common blocking call bugs:

4.2.1 Fix Patterns. We extracted 5 common fix patterns from the fixed blocking-call bugs according to the procedure described in Section 3.3. All fix patterns preserve the functionality of the original program. They only make the program more reactive. Due to space constraints, we only show the patterns for Reactor. Other reactive libraries (i.e., RxJava and Vert.x) can be handled similarly.

FP1. Offloading to Separate Worker Threads The following describes the schema of this fix pattern.

FP1. Offloading to Separate Worker Threads

```
// 1. τ(E) <: Mono | Flux
// 2. E involves a blocking operation
// 3. In the reactive pipeline to which E belongs, subscribeOn is not invoked.
-   E
+   E.subscribeOn(Schedulers.boundedElastic())

// Example
-   return Mono.fromCallable(() -> extractContent(inputStream, contentType));
+   return Mono.fromCallable(() -> extractContent(inputStream, contentType))
+       .subscribeOn(Schedulers.boundedElastic());
```

In the schema, E represents an expression of type Mono or Flux – Reactor's types to denote a sequence of at most one or $N > 1$ items, respectively. Suppose E involves a blocking operation such as an I/O operation. FP1 substitutes E.subscribeOn(Schedulers.boundedElastic()) for E. Please see the accompanying example. Note that in the example, Mono.fromCallable returns a value of type Mono, and hence FP1 can be applied. After this transformation, the execution of E – e.g., extractContent(...) – is offloaded to a separate worker thread, making the execution of E asynchronous.

FP2. Lazy Method Call Consider the example shown in the following box.

FP2. Lazy Method Call

```
// E involves a blocking operation
-   Mono.just(E) // eager evaluation of E
+   Mono.fromCallable(() -> E) // lazy evaluation of E

// Example
-   Mono.just(getSomething()).subscribe(e -> doSomething(e));
+   Mono.fromCallable(() -> getSomething()).subscribe(e -> doSomething(e));
```

In Line 6, Mono.just(getSomething()) executes getSomething() immediately *before* subscribe is called. In comparison, Mono.fromCallable(() -> getSomething()) executes getSomething() *only after* subscribe is called. What happens when getSomething() involves a blocking operation and takes a long time to complete? In the former, the thread running Mono.just is blocked until getSomething() completes. However, in the latter, the execution of getSomething() is delayed until subscribe is called. By combining FP1 and FP2, we obtain the following code where getSomething() is executed on a separate worker thread.

```
Mono.fromCallable(() -> getSomething())
    .subscribeOn(Schedulers.boundedElastic()).subscribe(e -> doSomething(e));
```

FP3. Reactive Filtering The Reactor library provides a method, filter, taking as input a predicate (e.g., isUserOnline in Line 5).

FP3. Reactive Filtering

```
- E1.filter(E2 -> E3)
+ E1.filterWhen(E2 -> Mono.fromCallable(() -> E3))

// Example
- userIDs.filter(id -> isUserOnline(id))
+ userIDs.filterWhen(id -> Mono.fromCallable(() -> isUserOnline(id)))
```

What if the predicate involves a blocking operation? FP3 replaces filter with filterWhen which executes the predicate asynchronously, possibly in a separate thread.

FP4. Non-blocking Chaining The following describes the two schemas of this fix pattern.

FP4. Non-blocking Chaining

```
// FP4.1
// 1. τ(E) <: Mono
// 2. τ(S) <: Mono | Flux
- E.block(); S
+ E.then(Mono.fromRunnable(() -> S))

// FP4.2
// 1. τ(E) <: Flux
// 2. τ(S) <: Mono | Flux
// 3. blk ∈ {blockFirst, blockLast}
- E.blk(); S
+ E.then(Mono.fromRunnable(() -> S))
```

FP4.1 is applied to E.block(); S. The thread running this code is blocked at block() (i.e., S is not executed) until E is completed. FP4.1 removes this blocking behavior by connecting E and S using the then method. The obtained code forms a single non-blocking reactive chain. FP4.2 is similar to FP4.1, but it considers the case where E is a Flux.

FP5. Non-blocking Subscription Unlike in FP4, what if a block method call is not followed by any other code, as shown in the following?

FP5. Non-blocking Subscription

```
// 1. τ(E) <: Mono | Flux
// 2. blk ∈ {block, blockFirst, blockLast}
void M(...) {
   ...
  - E.blk();
  + E.subscribe();
}
```

In the above code, any variant of blk subscribes to the publisher E, while ensuring that calling method M is blocked until E is finished emitting. FP5 removes this blocking behavior by replacing block with subscribe. Invoking subscribe triggers the execution of E but does not block the thread running M.

The complete list of fix patterns (including the corresponding fixes for RxJava and Vert.x) is provided in the Figshare repository [15]. The effort for extrapolating the fixes mentioned here to those for RxJava and Vert.x is minimal given the similarity of the operators; for example, `Mono.fromCallable()` used in Reactor for lazy evaluation becomes `Single.fromCallable()` in RxJava.

4.2.2 Applying fix patterns. We applied our fix patterns to the unfixed blocking-call bugs in the 29 open-source projects curated in Stage 1. To detect new blocking-call bugs, we used BlockHound [5], a dynamic analysis tool that detects blocking calls in non-blocking threads. This is necessary as the blocking calls do not follow a specific pattern, and code that might appear benign might be triggering some blocking operation under the hood.

The following list describes the distribution of the fix patterns applied to the new bugs detected. The corresponding pattern numbers are mentioned with their sub-parts (a/b/c) matching the full version of the pattern list available in our replication package [15] :

- Offloading blocking code to worker threads via Schedulers (FP1a): 46.7% (14/30).
- Offloading blocking code to Java's executor thread (FP1b): 33.3% (10/30).
- Offloading blocking code to Vertx's executeBlocking() method (FP1c): 6.7% (2/30)
- Lazily executing blocking operation by wrapping with fromCallable() (FP2a): 3.3% (1/30)
- Non-blocking chaining of operations using then() operator (FP4a): 3.3% (1/30).
- Non-blocking chaining of operations using Completablefuture.thenRun() (FP4b): 3.3% (1/30).
- Non-blocking subscription (FP5): 3.3% (1/30).

4.2.3 Impact of Patch Application.

Performance improvement. One of the main goals of this study is to show the criticality of blocking-call bugs in reactive applications. To test the hypotheses (H_0 and H_a) described in Section 3.4, we measured and compared the performance of the application before and after the fix. The performance was measured in terms of latency (how long the main thread had been in a blocked, waiting, or sleeping state), CPU usage, heap usage, as well as usage of the physical memory.

The main impetus for adopting reactive programming in any project is scalability and resource efficiency. Developers expect to see the benefit of adopting the reactive approach as the application scales up, handling more load with the same resources. In this regard, CPU and memory are deemed to be the most critical resources. These metrics are also common in empirical studies that compare the performance of different programming paradigms [59, 67].

It is important to note that the actual impact of having or eliminating blocking calls can be observed as the application scales, with more processing on the limited number of threads available in reactive applications. Nevertheless, Table 2 provides the performance evaluation results of 10 of the 29 target programs before and after applying fixing blocking bugs. The statistical significance of the differences is denoted as ∗:p-value<0.05 and ∗∗:p-value<0.01. Note that here we chose the 10 projects with significant reductions due to space limitations. The complete version of the performance analysis results is available in the replication package [15].

To address variations during performance profiling, we performed 10 executions for the measurement of each metric. Figure 7 compares CPU, heap, memory, and latency values before and after the fix, across 10 iterations for the Apache James Project[16]. We can notice that the CPU and heap usage, as well as latency, generally stayed lower after fixing blocking calls, apart from the CPU usage between iterations 5-7. The case of physical usage memory is interesting. Despite an apparent rise in the final iteration, the average across all iterations revealed a slight reduction in the physical memory usage post-bug fix (0.2% decrease, as shown in Table 2).

Here it is worth mentioning that the four metrics are not necessarily directly proportional; their relationship depends on the nature of the subject. Hence, some metrics may show slight increases post-bug fix; however, these increases are marginal when compared to the reductions seen in other metrics. All in all, for the 29 reactive subjects, we saw a 6.44% decrease in heap usage, 3.5% reduction in CPU usage, 3.5% lower latency, and 0.4% less physical memory usage after fixing blocking calls. Again, the metric values for each iteration, along with the standard deviations, are provided in the complete version of the performance analysis results.

Table 2: Performance improvement after blocking call fixes.

Subject	CPU (%) ↓	Heap (Mib) ↓	Latency (s) ↓	Memory (%) ↓
Vert.x Kafka Client [31]	12.3 (22%)	139.8 (33.3%)∗	0 (0%)	(3.2%)∗∗
Mercury [46]	0 (0%)	62.1 (14.9%)	0 (0%)	0.4 (0.4%) ∗∗
Spring Reactive Sample [53]	0 (0%)	24 (15.4%)	0 (0%)	1.2 (1.2%)∗∗
Vert.x Micrometer Metrics [54]	1.6 (3.6%)	31.5 (10.1%)∗∗	0 (0%)	0.2 (0.2%)
Apache James Project [16]	0 (0%)	20.5 (6%)	0.1 (0.3%)	0 (0%)
Spring Framework [29]	0.8 (0.8%)	10.6 (3.4%)	0 (0%)	0 (0%)
AWS SDK for Java [4]	3.7 (14.01%)	0 (0%)	1 (28%)	0.3 (0.3%)∗∗
Vert.x Redis Client [33]	0 (0%)	139.8 (62%)∗	0 (0%)	0 (0%)
Spring Data Examples [28]	11 (16.1%)	0 (0%) ∗	0 (0%)	0 (0%)
Nettosphere [21]	0 (0%)	0 (0%)	0.2 (1.4%)	0.1 (0.12%)

Mib: Mebibytes, ∗: p-value < 0.05, ∗∗: p-value < 0.01.

Non-intrusive fixes. All 30 patches preserve the original behavior of the reactive applications (i.e., functionality-preserving). It is important to note that all our patch patterns are non-intrusive; that is, they preserve the original functionality; no regression error was detected upon running the subjects' regression test suite. Moreover, the fix patterns were chosen considering the preservation of the developer's original intention. For instance, if a synchronous operation is performed in a reactive context (such as executing an I/O task or waiting for the result from the previous operation), the applied patch fixes the blocking call in a way that achieves the desired behaviour (waiting for the result, etc.) while ensuring that the

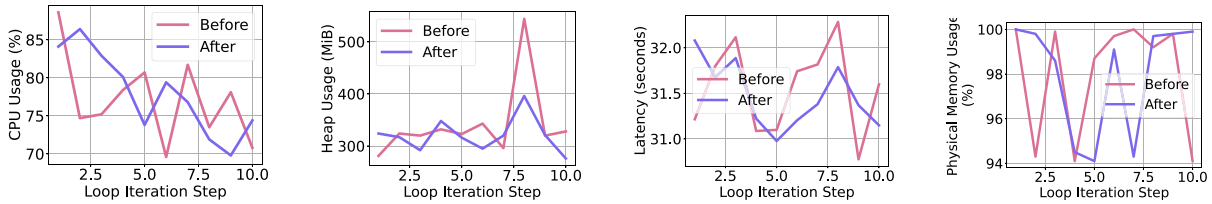


Figure 7: Performance measurement over 10 loops after fixing blocking calls in `Apache James Project` [16].

blocking was done in the compatible schedulers [26] provided by the reactive libraries. This was further validated by the responses of the developers to the submitted patches.

Patch Assessment. Our repair approach preserves the original functionality of the program since we use only non-intrusive fix patterns. Indeed, all 30 patches obtained pass the regression test suites of the subjects. Our manual investigation cross-checked by two authors also confirms their non-intrusiveness. As these patches were generated using only a few fix patterns described in Section 4.2, their assessment was straightforward. We also consulted the performance metrics we measured before and after applying the patches. In all 30 patches, we observed improvement in the performance metrics, which further confirms the usefulness of the patches. Nevertheless, whether a patch is correct and useful in practice, is ultimately determined by the developers who maintain the project. Instead of claiming that our patches are correct, we submitted them to the developers of the projects. In the next subsection, we show how those developers respond to our patches.

> **RQ2**: *Does our proactive repair technique generate non-intrusive patches fixing blocking-call bugs?*
> **Answer to RQ2**: *Our proactive repair approach generates 30 patches that resolve blocking-call bugs (verified by performance improvement) while being non-intrusive (i.e. preserving the original functionality).*

4.3 Acceptance Ratio of the Patches

Following the procedures explained in Section 3.4, we submitted PRs reporting new blocking-call bugs and corresponding patches generated by our proactive repair approach (Section 4.2). We created 30 patches to the unfixed issues in the 29 projects curated earlier.

Among 30 PRs, we submitted 15 PRs with performance analysis results and another 15 PRs were submitted without the results. We randomly selected the PRs to be submitted with the results. We then observed the outcomes of the PRs.

PRs with performance analysis results are more likely to be accepted by developers, as shown in Table 3. Eight out of 15 PRs (53%) with performance analysis results are accepted by developers, while 3 out of 15 (20%) PRs are accepted without the results. In addition, two of the PRs with results are explicitly rejected, while 4 PRs without results are rejected. To figure out whether the difference between the fix ratio of two groups (`Accepted` vs. `Rejected` + `Ignored`) is significant, we applied Barnard's exact test [58] to the number of PRs shown in Table 3. It turned out that the difference is statistically significant. (p-value < 0.01) and we can reject the null hypothesis, H_0, defined in Section 3.4.

Another observation made from this live study is that developers tend to accept PRs containing performance analysis results without much discussion. However, if the PRs do not include the results, they will put more effort into reproducing and understanding the symptoms reported in the PRs. Figure 8 shows a comment by a developer who reviewed one of our PRs without the results. The developer had to inspect the source code where a potential blocking call was invoked. This inspection confirmed that the blocking call can block a reactive thread. Thus, the developer improved our PR to fix the bug.

Table 3: Outcomes of the pull requests (PRs) submitted for RQ3, following the procedures described in Section 3.4. The group `w/ perf. results` stands for PRs submitted with performance analysis results. The group `w/o perf. results` are PRs without the results.

Group	Outcome			Total
	Accepted	Rejected	Ignored	
w/ perf. results	8 (53.3%)	2 (13.3%)	5 (33.3%)	15 (100.0%)
w/o perf. results	3 (20.0%)	4 (26.7%)	8 (53.3%)	15 (100.0%)

∗ The difference of the fix ratio (`Accepted` vs. `Rejected` + `Ignored`) between two groups is statistically significant (p-value < 0.01).

∗ The list of 25 subjects considered for our live study is available in our replication package [1].

∗ PR links will be disclosed later due to the double-blind policy, however, their contents (with the identity removed) can be found in our replication package.

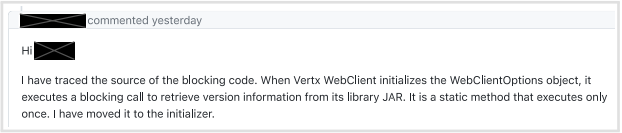


Figure 8: Comment by a developer who reviewed our PR without performance analysis results.

This may imply that our proactive approach is effective in fixing blocking bugs with non-intrusive patches. In addition, submitting the patches with improvement evidence makes the PRs more likely to be accepted by the developers.

> **RQ3**: *Are the patches more acceptable if they are submitted with improvement evidence?*
> **Answer to RQ3**: *Non-intrusive patches generated by our proactive approach are more likely to be accepted by developers if they are submitted with performance improvement evidence.*

4.4 Reviewing the Declined Patches

As specified in Table 3, six out of 30 pull requests (PRs) made were rejected. One PR to *vertx-web* [36] was turned down based on the missing evidence of the impact on concurrency. Another one to *vertx-tracing* [35] with performance improvement evidence was rejected on the grounds of a missing correlation between the

blocking call and the area with performance degradation. Two PRs, both without performance analysis results, one to *vertx-micrometer-metrics* [32] and another to *vertx-redis-client* [34], were rejected by the same developer without any concrete explanation; we were instead suggested to whitelist (allow) the blocking calls. One patch submitted to *spring-data-elasticsearch* [27] without performance-enhancement evidence was rejected due to bug irreproducibility (despite providing replication steps). A final patch that was rejected was submitted to *aws-sdk-java-v2* [27] with performance improvement evidence. The developer mentioned that they would like to avoid this kind of patch without providing further explanation. We were asked to report the issue as a *feature request* instead.

Due to the double-blind policy, we have not provided the reference links to the actual pull requests, however the contents of the PRs, with the removed identity, are available in our replication package [15].

5 Discussions

5.1 Applicability to Other Programming Languages

While Java stands out in its widespread adoption of reactive libraries [13, 25, 49–51], many other languages such as Python, Rust, and Kotlin also offer support for asynchronous programming through mechanisms such as async/await, coroutines, and asynchronous tools, and libraries. Similar to reactive Java programs, 'blocking' scenarios can also occur in the programs written in these languages – the blocking call problem is language-agnostic.

Consider the following Kotlin code snippet extracted from Plees Tracker [48] project. The shown `suspend` function performs a blocking I/O operation caused by the invocation of the `openInputStream` method; a `suspend` function can be paused without blocking the thread on which it is executing and can be resumed later [40].

```
suspend fun importData(context: Context, cr: ContentResolver, uri: Uri) {
     val inputStream = cr.openInputStream(uri) // blocking I/O operation
     ...
  }
```

The following shows how this blocking bug was fixed by the developer [44]:

```
suspend fun importData(context: Context, cr: ContentResolver, uri: Uri) {
+  withContext(Dispatchers.IO) {
     val inputStream = cr.openInputStream(uri) // blocking I/O operation
     ...
+  }
}
```

This fix uses a `withContext` block to offload the problematic blocking I/O operation to a separate pool of threads associated with `Dispatchers.IO`. This fix pattern is often observed elsewhere as well [38, 39, 41–43, 45, 47, 52, 55]. Note that the concept used in this fix pattern is similar to FP1 (Offloading to Separate Worker Thread) described in Section 4.2.1. This is not coincidental. Reactive programming concepts are common across many programming languages. Thus, programs using these concepts are likely to share similar issues and solutions. As concrete evidence for that, our replication package includes a list of similar blocking call bugs and their fixes in programs written in Kotlin, Python and Rust.

5.2 Implications for Researchers and Practitioners

5.2.1 Implications for researchers. The results of RQ2 highlight the effectiveness of proactive bug fixes in resolving performance issues without extensive refactoring. Further research into the scalability of proactive bug repair across diverse systems and languages is encouraged. The results of RQ3 emphasize the importance of bug-fix evidence for performance-related PRs, encouraging exploration of additional factors influencing PR outcomes. Finally, the introduction of Virtual Threads (see Section 6.5), which are capable of suspending and resuming at 'blocking call' encounters [76], prompts an investigation into how their integration affects reactive programming performance, particularly in case of unintentional blocking calls in a reactive pipeline.

5.2.2 Implications for practitioners. The results of RQ1 highlight blocking-call bugs as a prevalent issue in Java reactive programming, indicating a need for improved documentation on threading and blocking operations in reactive frameworks. Developer training programs should address the learning curve of transitioning to the asynchronous and non-blocking nature of the reactive paradigm. Lastly, the findings for RQ3 suggest that performance bug PRs are more likely to be accepted when accompanied by evidence of performance enhancement. We, therefore, recommend code contributors keep pull requests concise while including proof of performance improvement, especially for irreproducible, non-functional bugs.

5.3 Threats to Validity

Threats to external validity may lie in the target subjects (projects, bugs, pull requests, etc.) that our study investigates as they are open-source projects; thus, the results may not apply to other types of subjects, such as those using closed-source techniques. In addition, our study focuses only on Java subjects, while other languages that implement reactive programming were not considered. There are other libraries (e.g., Akka and Mutiny) for writing reactive programs even for Java; however, this threat might be mitigated as our target libraries (i.e., RxJava, Reactor, and Vert.x) are some of the most popular and representative in the Java community.
Threats to internal validity may include the manual extraction of the fix patterns by the authors. To address this threat, each fix pattern is supported by real patches that fix a blocking call in reactive programs implemented using RxJava, Reactor, and Vert.x.
Threats to construct validity may relate to the performance analysis results used in the third stage of our empirical study. To show the effectiveness of the patches generated by our proactive repair approach, our experiment measures and compares the performance metrics (CPU, memory etc.) before/after applying the patches. Although the improvement in these metrics may not prove the correctness of the patches, it might be enough to signify the impact of blocking calls in reactive applications.

6 Related Work

Reactive Programming (RP): Reactive apps offer flexibility and scalability, but only with the correct usage of reactive tools and libraries. Dobslaw et al. [60] studied the frequency, causes, and fixes of blocking-call violations in Java projects. The goal of the

findings is to raise awareness of improper blocking calls in reactive pipelines and promote the correct usage of reactive patterns. Alabor and Stolze [56] studied software engineers debugging RxJS-based apps, identifying prevalent challenges and solutions.

There have also been comparative studies on RP tools. Zimmerle et al. [90] mined GitHub and StackOverflow to study the reactive operators used in the 3 mainstream reactive libraries RxJava, RxJS, and RxSwift. The goal is to find the most common problems RP users are facing. Ponge et al. [83] compared the performance of RxJava, Reactor and Mutiny, the 3 mainstream reactive programming libraries in Java. Likewise, Komolov et al. [68], compared two multi-threading paradigms (reactive programming and continuation-passing style) in terms of their maintainability, performance, and testability.

In the context of automation support, Banken et al. [57] created RxFiddle, an online interactive debugger and visualizer to assist in the debugging and understanding of data flows in reactive programs. Köhler and Salvaneschi [68] present 2RX, an automated refactoring eclipse plugin that converts asynchronous logic (SwingWorker and Future) into reactive code.

Pattern-Based Program Repair: Pattern-based program repair has been widely studied since its first introduction [66]. Patches generated from such a technique have shown to be more acceptable by developers than those from heuristic-based approaches such as GenProg [87]. Recently, there have been automatic approaches to fix pattern mining [69, 70] to mitigate the manual effort.

Several studies have improved the idea of pattern-based program repair; there are studies extracting patterns for different targets such as JavaScript faults [79] and performance bugs [77]. Other studies explore diverse sources of fix patterns, such as Q&A posts [73], similar snippets [65], fault localization results [69], and static analysis warnings [71]. In addition, TBar [72] incorporated common fix patterns from other existing studies and showed that fix patterns are effective when fixing bugs.

Fix patterns are also useful to generate non-intrusive patches. Nistor et al. [77] examined patches for performance bugs written in the C and Java languages. The patches were short and simple and did not impact the functionality of the program.

Concurrency Bugs: Concurrency bugs in Java have been the focus of research for a long time; recently researchers are striving to develop effective tools to minimise their occurrence. Rehype [80] is a dynamic analyzer that performs automatic performance analysis on runtime execution traces in Java to detect inefficient concurrency patterns and suggest source code improvements, with an estimated effect on performance. ThreadRadar [75] creates glyph-based visualizations representing actively running thread information such as runtime consumption, types, etc. It integrates these visualizations within the source code, to better understand symptoms of concurrency bugs occurring at runtime. ARS (Adaptive Randomized Scheduling) [86], inspired by adaptive random testing, is an algorithm proposed by Zan Wang et al. to detect non-deadlock concurrency bugs.

Asynchronous Code Optimization: DrAsync [85] identifies and visualizes 8 common promise-based JavaScript anti-patterns through static and dynamic analysis, finding 2,600 instances in 20 repositories. It enhances performance by refactoring these issues and provides visual tools like timelines to link promises to source code. While DrAsync focuses on promise anti-patterns, our study addresses bugs in reactive applications and how performance evidence impacts patch acceptance.

Desynchronizer [62] uses static analysis to recommend converting synchronous API calls to asynchronous ones, identifying 256 suitable refactorings from 316 calls in 12 applications. While 244 conversions were successful with minimal behavioral changes, the tool's recommendations need programmer validation. Unlike our focus on performance issues from blocking-call bugs in reactive applications, Desynchronizer offers a broad solution for improving asynchronous performance by migrating synchronous APIs.

Virtual Threads: The Java community is currently working on techniques for addressing the blocking bugs by redesigning the language structure itself. Specifically, JDK 19 introduced a preview version of *Virtual threads* (VT), also known as *Project Loom*, which are managed by the JVM itself and are not bound to the OS. VT can suspend their execution, store their progress in memory, and resume later, wherever the OS encounters a 'blocking call' [76]. In between suspension and resumption, a virtual thread is no longer using the CPU, enabling many virtual threads to run on the same OS thread, without impacting the JVM's performance/memory consumption [2].

Virtual threads sparked speculations about reduced blocking call costs [59]. However, they only partially address reactive programming challenges. VT lack support for backpressure, change propagation, and composability, some of the key features of reactive programming [74, 78]. In addition, code blocks using `synchronized` keywords will still pin/block the carrier (OS) threads, limiting concurrent virtual thread execution. As it turns out, the use of `synchronized` blocks is pervasive across the Java codebase [82].

7 Conclusions

This study has been motivated by the observation that developers tend to put more or less effort into different types of bugs. The results of our study show that (1) reactiveness bugs are less likely to be fixed than other types of bugs; (2) blocking-call bugs are one of the most common reactiveness bugs; (3) there are common fix patterns for blocking-call bugs; (4) our proactive repair approach successfully generates patches for the blocking-call bugs; and (5) the patches are more likely to be accepted by developers if submitted with evidence of fixing. We hope the results of this study shed light on the debugging practice of non-functional bugs and increase the ratio of their resolution at large.

8 Data Availability

We make the replication package publicly available, which includes all the code and datasets to reproduce our experiments at https://figshare.com/s/0b7e78405b0b86f6cbfb [1].

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1A5A1021944 and 2021R1I1A3048013) and the Institute for Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2021-0-01001 and RS-2023-00222830).

References

[1] [n. d.]. Preserving Reactiveness: Understanding and Improving the Debugging Practice of Blocking-call Bugs — Replication Package. https://figshare.com/s/0b7e78405b0b86f6cbfb [Online; accessed 31. Jul. 2023].

[2] 2023. https://www.diva-portal.org/smash/get/diva2:1763111/FULLTEXT01.pdf [Online; accessed 30. Jul. 2023].

[3] 2023. About Java Flight Recorder. https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/about.htm#JFRUH170 [Online; accessed 30. Jul. 2023].

[4] 2023. aws-sdk-java-v2. https://github.com/aws/aws-sdk-java-v2 [Online; accessed 23. Nov. 2023].

[5] 2023. BlockHound. https://github.com/reactor/BlockHound [Online; accessed 31. Jul. 2023].

[6] 2023. Blocking call detected in io.lettuce.core.metrics.DefaultCommandLatencyCollector. https://github.com/lettuce-io/lettuce-core/issues/1513 [Online; accessed 27. Jul. 2023].

[7] 2023. Blocking code in reactive package · Issue #124 · eventuate-foundation/eventuate-common. https://github.com/eventuate-foundation/eventuate-common/issues/124 [Online; accessed 27. Jul. 2023].

[8] 2023. Blocking in WebSessionServerCsrfTokenRepository · Issue #8128 · spring-projects/spring-security. https://github.com/spring-projects/spring-security/issues/8128 [Online; accessed 27. Jul. 2023].

[9] 2023. CompilationUnit (javaparser-core 3.2.5 API). https://www.javadoc.io/doc/com.github.javaparser/javaparser-core/3.2.5/com/github/javaparser/ast/CompilationUnit.html [Online; accessed 19. Nov. 2023].

[10] 2023. CompletableFuture (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#toCompletableFuture-- [Online; accessed 1. Aug. 2023].

[11] 2023. CompletableFuture (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#join-- [Online; accessed 1. Aug. 2023].

[12] 2023. Doing blocking calls from DynamoDbEnhancedClientExtension methods. https://github.com/aws/aws-sdk-java-v2/issues/2996 [Online; accessed 27. Jul. 2023].

[13] 2023. Eclipse-Vertx/Vert.x. Eclipse Vert.x.

[14] 2023. How to Avoid Blocking Calls in Reactive Programming. https://bhavinshah7.medium.com/how-to-avoid-blocking-calls-in-reactive-programming-92a85c18866c [Online; accessed 15. Dec. 2023].

[15] 2023. "It's a bug, not a feature": A Proactive Approach to Repairing Blocking-call Bugs by Non-intrusive Patches. https://figshare.com/s/0b7e78405b0b86f6cbfb [Online; accessed 31. Jul. 2023].

[16] 2023. james-project. https://github.com/apache/james-project [Online; accessed 3. Dec. 2023].

[17] 2023. javaparser. https://github.com/javaparser/javaparser [Online; accessed 19. Nov. 2023].

[18] 2023. LexicalPreservingPrinter (javaparser-core 3.5.0 API). https://www.javadoc.io/doc/com.github.javaparser/javaparser-core/3.5.0/com/github/javaparser/printer/lexicalpreservation/LexicalPreservingPrinter.html [Online; accessed 19. Nov. 2023].

[19] 2023. Mono (reactor-core 3.5.8). https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html#block-- [Online; accessed 1. Aug. 2023].

[20] 2023. Move blocking mail items polling operation to boundedElastic. https://github.com/apache/james-project/pull/1650 [Online; accessed 15. Dec. 2023].

[21] 2023. nettosphere. https://github.com/Atmosphere/nettosphere [Online; accessed 12. Dec. 2023].

[22] 2023. Overview :: Spring Framework. https://docs.spring.io/spring-framework/reference/web/webflux/new-framework.html [Online; accessed 16. Dec. 2023].

[23] 2023. processor-health sample in chapter 13 blocks the I/O thread. https://github.com/cescoffier/reactive-systems-in-java/issues/230 [Online; accessed 27. Jul. 2023].

[24] 2023. Project Reactor. https://projectreactor.io/

[25] 2023. RxJava: Reactive Extensions for the JVM. https://github.com/ReactiveX/RxJava

[26] 2023. Schedulers (reactor-core 3.6.0). https://projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html [Online; accessed 18. Nov. 2023].

[27] 2023. spring-data-elasticsearch. https://github.com/spring-projects/spring-data-elasticsearch [Online; accessed 23. Nov. 2023].

[28] 2023. spring-data-examples. https://github.com/spring-projects/spring-data-examples [Online; accessed 12. Dec. 2023].

[29] 2023. spring-framework. https://github.com/spring-projects/spring-framework [Online; accessed 3. Dec. 2023].

[30] 2023. UUID.randomUUID() is detected as blocking, any replacement? · Issue #157 · reactor/BlockHound. https://github.com/reactor/BlockHound/issues/157 [Online; accessed 27. Jul. 2023].

[31] 2023. vertx-kafka-client. https://github.com/vert-x3/vertx-kafka-client [Online; accessed 3. Dec. 2023].

[32] 2023. vertx-micrometer-metrics. https://github.com/vert-x3/vertx-micrometer-metrics [Online; accessed 3. Dec. 2023].

[33] 2023. vertx-redis-client. https://github.com/vert-x3/vertx-redis-client [Online; accessed 3. Dec. 2023].

[34] 2023. vertx-redis-client. https://github.com/vert-x3/vertx-redis-client [Online; accessed 23. Nov. 2023].

[35] 2023. vertx-tracing. https://github.com/eclipse-vertx/vertx-tracing [Online; accessed 3. Dec. 2023].

[36] 2023. vertx-web. https://github.com/vert-x3/vertx-web [Online; accessed 23. Nov. 2023].

[37] 2023. VisualVM: Home. https://visualvm.github.io [Online; accessed 30. Jul. 2023].

[38] 2024. Android - what is Inappropriate blocking method call in Coroutine. https://stackoverflow.com/questions/73452001/android-what-is-inappropriate-blocking-method-call-in-coroutine [Online; accessed 4. Apr. 2024].

[39] 2024. Coroutine suspend function and blocking calls. https://stackoverflow.com/questions/60709511/coroutine-suspend-function-and-blocking-calls [Online; accessed 4. Apr. 2024].

[40] 2024. Coroutines basics | Kotlin. https://kotlinlang.org/docs/coroutines-basics.html [Online; accessed 4. Apr. 2024].

[41] 2024. How to make "inappropriate blocking method call" appropriate? https://stackoverflow.com/questions/58680028/how-to-make-inappropriate-blocking-method-call-appropriate [Online; accessed 4. Apr. 2024].

[42] 2024. How to replace blocking code for reading bytes in Kotlin. https://stackoverflow.com/questions/56533497/how-to-replace-blocking-code-for-reading-bytes-in-kotlin [Online; accessed 4. Apr. 2024].

[43] 2024. How to use a Kotlin coroutine to call a blocking function? https://stackoverflow.com/questions/64306381/how-to-use-a-kotlin-coroutine-to-call-a-blocking-function [Online; accessed 4. Apr. 2024].

[44] 2024. import file: handle when the storage is in fact a network one by vmiklos · Pull Request #420 · vmiklos/plees-tracker. https://github.com/vmiklos/plees-tracker/pull/420 [Online; accessed 4. Apr. 2024].

[45] 2024. Inappropriate blocking method call · Issue #10 · aseemsavio/IntelligentPiCamera. https://github.com/aseemsavio/IntelligentPiCamera/issues/10 [Online; accessed 4. Apr. 2024].

[46] 2024. mercury. https://github.com/Accenture/mercury [Online; accessed 24. Mar. 2024].

[47] 2024. Moshi: how to fix "Inappropriate blocking method call" warning in the coroutine? https://stackoverflow.com/questions/73343117/moshi-how-to-fix-inappropriate-blocking-method-call-warning-in-the-coroutine [Online; accessed 4. Apr. 2024].

[48] 2024. plees-tracker. https://github.com/vmiklos/plees-tracker [Online; accessed 4. Apr. 2024].

[49] 2024. reactive-streams-jvm. https://github.com/reactive-streams/reactive-streams-jvm [Online; accessed 4. Apr. 2024].

[50] 2024. reactor-core. https://github.com/reactor/reactor-core [Online; accessed 4. Apr. 2024].

[51] 2024. smallrye-mutiny. https://github.com/smallrye/smallrye-mutiny [Online; accessed 4. Apr. 2024].

[52] 2024. Some feedback · Issue #1 · srrollin/weather. https://github.com/srrollin/weather/issues/1 [Online; accessed 4. Apr. 2024].

[53] 2024. spring-reactive-sample. https://github.com/hantsy/spring-reactive-sample [Online; accessed 24. Mar. 2024].

[54] 2024. vertx-micrometer-metrics. https://github.com/vert-x3/vertx-micrometer-metrics [Online; accessed 24. Mar. 2024].

[55] 2024. what reason show message that "Possibly blocking call in non-blocking context could lead to thread starvation"? https://stackoverflow.com/questions/73252926/what-reason-show-message-that-possibly-blocking-call-in-non-blocking-context-co [Online; accessed 4. Apr. 2024].

[56] Manuel Alabor and Markus Stolze. 2020. Debugging of RxJS-based applications. In *REBLS 2020: Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. Association for Computing Machinery, New York, NY, USA, 15–24. https://doi.org/10.1145/3427763.3428313

[57] Herman Banken, Erik Meijer, and Georgios Gousios. 2018. Debugging data flows in reactive programs. In *ICSE '18: Proceedings of the 40th International Conference on Software Engineering*. Association for Computing Machinery, New York, NY, USA, 752–763. https://doi.org/10.1145/3180155.3180156

[58] G. A. BARNARD. 1947. SIGNIFICANCE TESTS FOR 2×2 TABLES. *Biometrika* 34, 1-2 (01 1947), 123–138. https://doi.org/10.1093/biomet/34.1-2.123

[59] D. Beronić, P. Pufek, B. Mihaljević, and A. Radovan. 2021. On Analyzing Virtual Threads – a Structured Concurrency Model for Scalable Applications on the JVM. In *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE, 1684–1689. https://doi.org/10.23919/MIPRO52101.2021.9596855

[60] Felix Dobslaw, Morgan Vallin, and Robin Sundström. 2020. Free the Bugs: Disclosing Blocking Violations in Reactive Programming. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 177–186. https://doi.org/10.1109/SCAM51674.2020.00025

[61] Vladimir Filipchenko. 2023. An Intro to Spring WebFlux Threading Model. https://hackernoon.com/an-intro-to-spring-webflux-threading-model [Online; accessed 27. Jul. 2023].

[62] Satyajit Gokhale, Alexi Turcotte, and Frank Tip. 2021. Automatic migration from synchronous to asynchronous JavaScript APIs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 160 (oct 2021), 27 pages. https://doi.org/10.1145/3485537

[63] David Harel, Guy Katz, Assaf Marron, and Gera Weiss. 2012. Non-Intrusive Repair of Reactive Programs. In *2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*. 3–12. https://doi.org/10.1109/ICECCS20050.2012.6299199

[64] Vincent J. Hellendoorn, Premkumar T. Devanbu, and Alberto Bacchelli. [n. d.]. Will They Like This? Evaluating Code Contributions with Language Models. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 16–17. https://doi.org/10.1109/MSR.2015.22

[65] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *ISSTA 2018: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 298–309. https://doi.org/10.1145/3213846.3213871

[66] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, San Francisco, CA, USA, 802–811. https://doi.org/10.1109/ICSE.2013.6606626

[67] Sirojiddin Komolov, Nursultan Askarbekuly, and Manuel Mazzara. 2020. An empirical study of multi-threading paradigms Reactive programming vs continuation-passing style. In *ICCBD '20: 2020 the 3rd International Conference on Computing and Big Data*. Association for Computing Machinery, New York, NY, USA, 37–41. https://doi.org/10.1145/3418688.3418695

[68] Mirko Köhler and Guido Salvaneschi. 2019. Automated Refactoring to Reactive Programming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 835–846. https://doi.org/10.1109/ASE.2019.00082

[69] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining Relevant Fix Patterns for Automated Program Repair. *Empirical Softw. Engg.* 25, 3 (may 2020), 1980–2024. https://doi.org/10.1007/s10664-019-09780-z

[70] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon. 2021. Mining Fix Patterns for FindBugs Violations. *IEEE Transactions on Software Engineering* 47, 1 (Jan. 2021), 165–188. https://doi.org/10.1109/TSE.2018.2884955

[71] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyande. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–12. https://doi.org/10.1109/SANER.2019.8667970

[72] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *ISSTA 2019: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/3293882.3330577

[73] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Campobasso, Italy, 118–129. https://doi.org/10.1109/SANER.2018.8330202

[74] Mark. 2020. Experimenting with Project Loom EAP and Spring WebMVC. *Paluch* (Sept. 2020). https://paluch.biz/blog/182-experimenting-with-project-loom-eap-and-spring-webmvc.html

[75] Oliver Moseler, Lucas Kreber, and Stephan Diehl. 2022. The ThreadRadar visualization for debugging concurrent Java programs. *J. Visualization* 25, 6 (Dec. 2022), 1267–1289. https://doi.org/10.1007/s12650-022-00843-w

[76] Arthur Navarro, Julien Ponge, Frédéric Le Mouël, and Clément Escoffier. 2023. Considerations for integrating virtual threads in a Java framework: a Quarkus example in a resource-constrained environment. https://doi.org/10.1145/3583678.3596895 [Online; accessed 30. Jul. 2023].

[77] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and Fixing Performance Problems That Have Non-intrusive Fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 902–912.

[78] Tomasz Nurkiewicz. 2022. Project Loom: Revolution in Java Concurrency or Obscure Implementation Detail? *InfoQ* (Oct. 2022). https://www.infoq.com/presentations/loom-java-concurrency

[79] Frolin S. Ocariza, Jr., Karthik Pattabiraman, and Ali Mesbah. 2014. Vejovis: Suggesting Fixes for JavaScript Faults. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 837–847. https://doi.org/10.1145/2568225.2568257

[80] Indigo Orton and Alan Mycroft. 2021. Refactoring traces to identify concurrency improvements. In *FTfJP '21: Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs*. Association for Computing Machinery, New York, NY, USA, 16–23. https://doi.org/10.1145/3464971.3468420

[81] Jevgenija Pantiuchina, Fiorella Zampetti, Simone Scalabrino, Valentina Piantadosi, Rocco Oliveto, Gabriele Bavota, and Massimiliano Di Penta. 2020. Why Developers Refactor Source Code: A Mining-based Study. *ACM Trans. Software Eng. Method.* 29, 4 (Sept. 2020), 1–30. https://doi.org/10.1145/3408302

[82] Gaetano Piazzolla. 2023. Java Virtual Threads - Dev Genius. *Medium* (Jan. 2023). https://blog.devgenius.io/java-virtual-threads-715c162c6c39

[83] Julien Ponge, Arthur Navarro, Clément Escoffier, and Frédéric Le Mouël. 2021. Analysing the performance and costs of reactive programming libraries in Java. In *REBLS 2021: Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. Association for Computing Machinery, New York, NY, USA, 51–60. https://doi.org/10.1145/3486605.3486788

[84] Arooba Shahoor, Askar Khamit, Jooyong Yi, and Dongsun Kim. 2023. LeakPair: Proactive Repairing of Memory Leaks in Single Page Web Applications. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*.

[85] Alexi Turcotte, Michael D. Shah, Mark W. Aldrich, and Frank Tip. 2022. DrAsync: Identifying and Visualizing Anti-Patterns in Asynchronous JavaScript. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 774–785. https://doi.org/10.1145/3510003.3510097

[86] Zan Wang, Dongdi Zhang, Shuang Liu, Jun Sun, and Yingquan Zhao. [n. d.]. Adaptive Randomized Scheduling for Concurrency Bug Detection. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 10–13. https://doi.org/10.1109/ICECCS.2019.00021

[87] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[88] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/3180155.3180233

[89] Carlos Zimmerle, Kiev Gama, Fernando Castor, and José Murilo Mota Filho. [n. d.]. Mining the Usage of Reactive Programming APIs: A Study on GitHub and Stack Overflow. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE, 23–24. https://doi.org/10.1145/3524842.3527966

[90] Carlos Zimmerle, Kiev Gama, Fernando Castor, and José Murilo Mota Filho. 2022. Mining the usage of reactive programming APIs: a study on GitHub and stack overflow. In *MSR '22: Proceedings of the 19th International Conference on Mining Software Repositories*. Association for Computing Machinery, New York, NY, USA, 203–214. https://doi.org/10.1145/3524842.3527966

Received 2024-04-12; accepted 2024-07-03