Inferring Performance Bug Patterns from Developer Commits

Yiqun Chen, Stefan Winter, Neeraj Suri
DEEDS Group, Dept. of Computer Science
Technische Universität Darmstadt
Darmstadt, Germany
{ychen, sw, suri}@cs.tu-darmstadt.de

***Abstract*—Performance bugs, i.e., program source code that is unnecessarily inefficient, have received significant attention by the research community in recent years. A number of empirical studies have investigated how these bugs differ from "ordinary" bugs that cause functional deviations and several approaches to aid their detection, localization, and removal have been proposed. Many of these approaches focus on certain subclasses of performance bugs, e.g., those resulting from redundant computations or unnecessary synchronization, and the evaluation of their effectiveness is usually limited to a small number of known instances of these bugs. To provide researchers working on performance bug detection and localization techniques with a larger corpus of performance bugs to evaluate against, we conduct a study of more than 700 performance bug fixing commits across 13 popular open source projects written in C and C++ and investigate the relative frequency of bug types as well as their complexity. Our results show that many of these fixes follow a small set of bug patterns, that they are contributed by experienced developers, and that the number of lines needed to fix performance bugs is highly project dependent.**

***Index Terms*—Software Engineering, Performance, Testing**

I. Introduction

Performance is among the most important non-functional properties of programs [1], [2]. Unfortunately, performance bugs accompany the software development process like functional bugs and software quality is equally deteriorated by those, e.g., in the form of decreased end-user interaction, wasted computing resources, or even DoS attacks [3]. A variety of performance diagnosis tools and approaches have emerged over the past decade to assist developers with the identification and localization of a wide range of performance bugs in different scenarios [4]–[7]. While it is expected that these tools/approaches cannot eradicate performance bugs entirely, it remains unclear which performance bugs are how well addressed by these tools or to which degree these tools are being used in practice. Therefore, it is important to investigate which types of performance bugs get fixed (and how) to guide future research in this area.

For this purpose we have conducted a large scale study on 733 performance bug fixing commits across 13 popular open source projects. To assess how well existing tools and approaches support developers with the detection and removal of performance bugs, we investigate the duration between the introduction and removal of performance bugs as well as the expertise of the bug fixing developer and the bug complexity in terms of lines modified by the fix. If performance bug detection and removal are well addressed by existing approaches or tools, they must be expected to have a short performance bug fix time. Similarly, with proper tool support, even inexperienced developers should be able to identify and remove performance bugs. Besides these measures, the number of the changed lines in bug fix commits also indicates how complicated performance bug fixes are.

Since tools and approaches for performance bug detection and localization usually specialize for certain classes of performance bugs, we perform a classification of the performance bug fixing commits to assess which classes dominate the bugs being fixed.

Besides assessing the alignment of existing tools and approaches for performance bug detection and localization with characteristics of performance bug fixing commits, we hope that our study can serve as a benchmark that future tools and approaches can be evaluated against, similar to CoREBench [8] or BugSwarm [9] for correctness bugs and we make our entire data set publicly available for this purpose[1]. Despite the relatively large number of subjects in our data set, there are applications that require even larger numbers of defects to obtain statistically significant evaluation results. For these scenarios we envision the creation of *performance mutants*, along the lines proposed in recent work [10] and proposals [11], and provide insights to support the creation of performance mutants that resemble performance bugs fixed in real world projects.

Summing up the above discussion, our study makes the following contributions:

- We present results from a large scale study of 13 open source projects from different domains with a total of 733 manually analyzed performance bug fixing commits.
- Using data from this study we assess the alignment of the current state of the art in performance bug detection and localization with performance bugs that get fixed in practice.
- The result of our study provides a database of performance bug fixes to serve as a benchmark for the further development and improvement of performance bug detection and localization techniques.

[1] https://yqchen.gitlab.io/perf-bugs/

- We discuss how the presented work can serve as the basis for performance mutation operators, but also why this basis is not (yet) sufficient for a practical performance mutation approach.

The remainder of the paper is structured as follows. After reviewing related work in Section II, we discuss the methodology adopted in this study in Section III. We then discuss the categorization of performance bug code patterns in Section IV, followed by an analysis of other performance bug fixes (fix duration, developer experience, changed lines) in Section V, a discussion of the threats to validity in Section VI, and a conclusion in Section VII.

II. Related Work

A. Performance Bugs for Evaluating and Training Detection and Localization Approaches

Performance bugs have been studied extensively in recent years and a large number of detection and localization approaches have been proposed. Interestingly, their efficacy has often been evaluated on applications with previously unknown (or disregarded) rather than known performance bugs [6], [12]–[14]. A great advantage of such an evaluation is the potential discovery of previously undiscovered bugs. While the detection of every single formerly unknown performance bug is a great achievement, only few bugs are found by each individual approach and bugs from many approaches would need to be combined to form a suitable performance bug data set to evaluate future approaches against. Unfortunately, newly found bugs are commonly described in insufficient detail in (space constrained) research articles to enable their reliable extraction as a reusable performance bug data set, which is what we target with our study.

A number of articles on approaches to detect or localize performance bugs use actual performance bugs or performance bug simulations to demonstrate the aproaches' efficacy [15]–[17]. These evaluations commonly focus on few bugs (between 15 and 70 in the cited articles) and do not report the relative occurrence of the targeted bug types compared to other bug types or the complexity of the targeted bugs.

Other articles explicitly state certain performance bug patterns they attempt to detect or localize [18]–[22]. Some of these patterns are highly application dependent (e.g., [19]). We attempt to map the patterns in our study to existing patterns based on the information provided in the respective articles.

PerfLearner successfully uses 300 randomly sampled performance bug reports from a total of 1383 reports across three projects to generate performance test frames that are evaluated against 10 other reported performance bugs [23]. The authors of PerfLearner report that extracting and reproducing these 10 performance bugs took approximately 400 work hours, which illustrates the complexity of reproducing performance bugs and relating them with performance bug reports. We avoid these problems in our study by directly targeting performance bug fixing commits. A downside of this decision is that we cannot quantify the performance impact of the bugs we study. We deem this tolerable, as such quantifications are highly sensitive to changes in the studied programs' operational environment, such as hardware and software configurations.

B. Empirical Studies of Performance Bugs

A number of studies target the assessment and characterization of performance bugs [4], [24]–[27] similar to our study.

Zaman et al. compare quantitative characteristics of performance and security bugs in Firefox [24]. Their automated analysis of more than 180 000 bug reports, out of which 4293 are performance bugs, reveals that performance bugs take more time to fix and are tackled by more experienced developers. Our study targets a smaller sample of performance bugs, but from a larger variety of projects, which are manually analyzed to confirm they are indeed performance bugs and characterize them according to how they hamper performance. Zaman et al. also present a smaller scale qualitative study of 400 randomly sampled performance and non-performance bug reports from Firefox and Chrome, which reveals that the performance bug reports tend to suffer from poor reproducibility [25].

Jin et al. present an empirical study of 109 randomly sampled performance bug reports from five applications. Their categorization according to how these bugs are fixed resembles the categorization we present in this study with the main difference that our categorization focuses on the intended semantics of performance bug fixes instead of their syntactical appearance, as we elaborate in Section III-B.

Nistor et al.'s comparison of 210 performance bugs against non-performance bugs in three software projects shows that performance bug fixes are equally likely to introduce new functional bugs as non-performance bugs and that performance bugs are more difficult to fix than non-performance bugs [26].

Han and Yu conclude from a study of 193 manually inspected performance bug reports and related changelogs across three projects that performance bug observability is highly configuration dependent, while fixing performance bugs does require source changes [27]. This supports our decision to focus our study of performance bugs on source code changes.

C. Bug Collections

We expect our data set of fixed performance bugs to serve as a basis for the evaluation of future performance bug detection and localization approaches, similar to existing data sets for correctness bugs. CoREBench [8] is a collection of 70 regression errors from four open source projects that have been extracted with the goal to serve as a more realistic alternative to mainly hand-seeded bugs in the Siemens test suite [28] and the SIR [29]. BugSwarm [9] is a collection of Python and Java correctness bug fixes mined from Travis-CI logs of GitHub projects and reproduced in isolated environments. Contrary to our data set, these two projects ensure that each of their bugs are reproducible. The reason why we cannot guarantee reproducibility is that the magnitude of latencies induced by performance bugs heavily depends on (a) configuration parameters of the software project [27] and (b) the complexity of inputs used to trigger the bugs [3]. It is likely for the

same reasons that performance bugs are reported to be more commonly detected and fixed via code reasoning than dynamic tests and that they sometimes "magically" disappear [26]. We have, therefore, decided to focus on bug fixing commits rather than bug reports in our study. Neither CoREBench nor BugSwarm cover performance bugs.

III. Methodology

We investigate performance bugs in real-world projects to assess how well performance bugs targeted by detection and localization approaches are aligned with the bugs that get fixed in practice. Due to the difficulties that performance bug reproducibility poses (see Section II), we identify performance bug fixing commits by manual inspection. To limit the corresponding overhead we pre-filter and sample commits according to criteria discussed in Section III-A. Section III-B details how the identified performance bugs are classified and how this entails deviations of the derived taxonomy from existing ones. We then introduce the performance bug complexity metrics we use in this study: a measure of performance bug fix duration in Section III-C, a measure of experience for performance bug fixing developers in Section III-D, and a bug fix complexity measure in Section III-E.

A. Selection of Projects and Commits

TABLE I: Total commit counts for each project

Project	Total
`NetworkManager`	209
`pulseaudio`	106
`grep`	123
`rsyslog`	136
`lvm2`	123
`llvm`	4567
`git`	1107
`clang`	860
`gecko-dev`[2]	4329
`openssl`	169
`systemd`	327
`libgcrypt`	145
`linux`	18975

We start the choice of projects we target in our study from the top 100 popular projects from the Debian repository[3] that are written in the C programming language. We base our selection on the "vote" data of Debian's popularity contest, which reflects the regular usage of the projects. Our focus on C is motivated by the observation that performance critical code is commonly written in languages that are "close" to the underlying hardware platform and that compile to native machine code. Moreover, as a language that dominates operating systems and other important parts of virtually every software stack, C has a high practical relevance. This is also reflected by the observation that the top 100 C projects in Debian's popularity contest are among the 133 top projects when no restriction is made on the programming language. In addition to these projects, `clang`, `llvm`, and `linux` are also selected as survey targets, because they are widely used large and complex projects with numerous commits and contributors. Thus, the performance of these three projects is expected to be of relevance for a large user base.

We identify performance bug fixing commits in the selected projects by searching for a number of keywords in the commit messages, as listed below along with matching text examples.

- **performance** "This patch improves the **performance** by ...%"
- **speed up** "These changes **speed up** the processing of"
- **accelerate** "This patch **accelerate**s"
- **fast** "After the patch it is ... times **fast**er"
- **slow** "Before the patch it is **slow** in function"
- **latenc** "The **latenc**y of ... is reduced "
- **contention** "This patch reduces the **contention** of"
- **optimiz** "The **optimiz**ation of the function"
- **efficient** "The patch makes function ... more **efficient**"

We exclude those projects from our candidate list, (1) for which we cannot easily access commit messages because we cannot unambiguously identify or access the official development repository (21 projects), (2) that are Debian specific and not used on other distributions in order to avoid a corresponding bias (4 projects), and (3) that have less than 100 commits that match our keywords (65 projects).

The last of these criteria has been added to exclude projects that do not have a particular performance relevance. For instance, `libcap2` implements operations to get and set POSIX capability states, which are not particularly performance critical. Accordingly, the project does not have a single commit message matching the aforementioned keywords and is, hence, excluded from our study. The 13 projects meeting all our criteria are listed in Table I along with the total matching commit counts. The complete table with matching counts for each keyword can be found on the our data set website provided in Section I.

The keyword-based detection mechanism for performance bug fixing commits is susceptive to false positives, e.g., **performance** could also match a feature commit stating "This patch does not introduce a **performance** regression". Hence, all matched commits need manual investigation. As some the projects like `clang`, `linux`, or `gecko-dev` have thousands of matching commits, we limit our manual assessment to a random sample of 200 commits for those projects.

B. Taxonomy

Performance bugs can be categorized by various criteria, e.g. the work by Jin et al. [4] studies syntactical representations of performance bugs. This study, instead, classifies performance bug fixes by the semantics behind these code changes. For instance, the code in Listing 2 shows the introduction of a

[2]The project name for the next Firefox version in development
[3]https://popcon.debian.org/by_vote

new API from the syntactical perspective. The goal of the taxonomy developed in this paper, in contrast, focuses on *how the newly introduced function and its usage affect the performance of the implementation*. In Listing 2, the newly introduced `try_fgrep_pattern()` function introduces a more efficient matcher, which only applies for certain scenarios. If such a scenario is encountered, this light weight matcher provides a faster execution path that speeds up the character pattern matching process. Therefore, this commit is tagged "fast-path" as the code change introduces a shortcut to speed up execution for certain scenarios (a detailed discussion is provided in Section IV-A).

We give preference to a manual semantic classification of performance bug fixing commits over a purely syntactical taxonomy to obtain comparability of bug fixes that transcend project or developer specific preferences, such as coding styles, to which purely syntactical taxonomies are sensitive.

C. Bug Fix Time

To determine the need for better tool support, we extract a number of metrics in our analysis of the identified commits. The first metric captures the latency to fix performance bugs. Intuitively, the more time developers spend on fixing a performance bug, the more difficult the performance bug is. However, this metric may be misleading in a cross-project comparison, as projects evolve at different speeds. For example, if project A has hundreds of commits every day while project B has only a handful of commits every week, the performance bugs in project B are likely to take longer than in project A, although the performance bugs in project B may not be any easier to detect and fix than those in project A. Consequently, we also take the number of commits between the introduction and the fixing of performance bugs in a project into consideration, to determine whether the project is actively maintained. We define the metric *fix time commit frequency* (FTCF) to present the accordingly normalized fix time as:

$$FTCF = n_{cmt}(\,(t_{intro}, t_{fix}]\,) \tag{1}$$

where t_{fix} is the time stamp of the fix commit, t_{intro} is the time stamp of the commit which introduces the fixed performance bug, and n_{cmt} denotes the number of commits in the specified interval. The larger the FTCF value is, the longer is the fix time if projects have the same level of activity in terms of commit rates. If a project has a commit rate that is twice as high as that of another project, then its FTCF is also twice as high for an identical time interval, indicating that the corresponding bug fixing time is actually slower. The reasoning behind this is that in intensively maintained projects, bugs should also be detected and fixed faster. To obtain the FTCF, we need to identify the bug introducing commit that corresponds to an identified fix. We basically follow the widely used approach to infer bug inducing commits outlined in [30], but assume that each modification in the bug fixing commit is *a necessary modification* to fix the bug. Consequently, we search for each modified line in the bug fixing commit the commit that last changed that line before. To be conservative and rather under- than over-estimate the actual fix time interval, we select the commit that is temporally closest to the fixing commit from this set as the bug introducing commit, because that is the last commit that made a change that then required a fix.

D. Seniority of Fixers

We also consider the expertise of the developers, who are fixing performance bugs, as an indicator of the fixing effort. If performance bugs are mostly fixed by more experienced developers, this indicates that better tool support for the detection and localization of performance bugs may be required or that the existing tools require a high expertise to be used effectively. To quantify the developer expertise, we measure the time Δ_{dev} between the bug fix at time t_{fix} and the first commit of the fixing developer dev at time $t_{dev(1)}$.

$$\Delta_{dev} = t_{fix} - t_{dev(1)} \tag{2}$$

By comparing the experience Δ_{dev} among the developers in the project, it is clear whether the fixer of performance bugs are relatively more experienced in the project or not. The comparison should only cover those developers who are actively contributing to the project at the time of the fix. Thus, the set of developers, who are *candidates* for contributing the fix, is:

$$C_{dev} = \{dev | t_{dev(1)} \leq t_{fix} \leq t_{dev(n)}\} \tag{3}$$

where $t_{dev(n)}$ is the time stamp of the last commit contributed to the project by developer dev. Using the expertise measured as Δ_{dev} across all developer candidates $dev \in C_{dev}$, we can judge the relative expertise of a developer contributing a bug fix within a project. However, using such project wise ranks may not be accurate in reflecting the skill of bug fixers in a cross project comparison, because the difference in the total number of developers varies significantly among the projects. For instance, assuming fixer A is ranked 10 out of 20 developers in a project, while fixer B is ranked 500 out of 1000 developers in another project, fixer A may not be more skilled than fixer B despite the higher absolute rank number. Moreover, the absolute value of the time difference is obviously misleading as projects are started at different times and, thus, have different lifetimes, which can introduce significant offsets in the Δ_{dev} values that are more strongly affected by the project than the actual developer experience. To quantify the skill of bug fixing developers and make them comparable across projects, we introduce a project *seniority* metric to evaluate the skill. The first concern of the seniority is the time when the performance bug is fixed and when is the project started. The time difference between the project initialization and the bug fix is noted as Δ_{base} and defined as:

$$\Delta_{base} = t_{fix} - t_{init} \tag{4}$$

where t_{init} is the time stamp of the first commit in the project. The seniority of dev at t_{fix} is defined as:

$$S_{dev} = \frac{\Delta_{dev}}{\Delta_{base}} = \frac{t_{fix} - t_{dev(1)}}{t_{fix} - t_{init}} \tag{5}$$

Given a bug fix commit, the skill of the fixer can be represented as the seniority. To have a relative comparison across the project, the seniority of the fixer denoted as S_{fix} is compared to a mean value of the seniority of all developers, given a performance bug fixing commit. In our study we select the median to aggregate the seniority vector of a project and the seniority difference is defined as:

$$\Delta S = S_{fix} - median(\{S_{dev} | dev \in C_{dev}\}) \quad (6)$$

This seniority measure reflects the experience of the bug fixing developer relative to the first commit of the project and relative to other active developers in the project and is suitable for a cross-project comparison.

E. Number of Changed Lines

The number of changed lines directly indicates how complicated a bug fixing commit is. Usually simple changes with a small number of lines modified are more likely to be diagnosed by tools. If a pattern of performance bugs involves a lot of small sized commits, the current tool support probably has not covered this form of performance bugs yet and new tools on such problems are needed. The number of changed lines is also relevant for our longer term goal to create performance mutants to test performance bug detection and localization approaches against, because they indicate to which degree traditional mutation operators (and their implementation in mutation tools) that are commonly applied to individual statements of a program are sufficient to simulate realistic performance bugs. Similarly, the type of change is giving an indication about the nature of performance mutation operators. If performance bug fixes tend to add rather than remove lines, the corresponding mutation operators would need to remove parts of the source code in order to introduce performance bugs. Otherwise, mutation operators would have to insert buggy code to create performance mutants. Therefore, we provide detailed data for the studied bug fixing commits, i.e., the total number of affected lines along with ratios of added, removed, and modified lines.

IV. The Shape and Variety of Fixed Performance Bugs

During the manual investigation of the commits identified in our keyword-based search, we found that many performance bug fixing commits follow certain patterns. Based on this observation, 7 common patterns have been identified. Among these patterns, two (*asm* and *async*) are project specific and require detailed knowledge of the respective subsystems in the project. These patterns are not likely to be found in other projects and introduce a certain project-based bias to the presented results. Therefore, the discussion of these two categories is kept brief and combined with the discussion of the generic *misc* class of bug fixing commits that do not match any of the larger pattern classes in Section IV-F.

A. Fast-path

A *fast-path* is a construct to avoid repeated or slow computation when possible. We limit the fast-path notion in this study to control flow based fast-paths and classify other avoidance techniques as other patterns. The control flow based fast-path pattern can have different syntactic representations. A simple form of skipping heavy computation is demonstrated in Listing 1. The `if` statement is a typical fast-path avoiding heavy computations when they are not needed. Real-world

Listing 1: Simple fast-path example

```
int foo(int bar) {
  if (some_cond(bar))
    return fast_path();
  return very_heavy_computation(bar);
}
```

occurrences of this pattern are usually much more complicated and obfuscated. For instance, fast paths may be needed in loops, where programs tend to spend most of their time [31, p. 655]. If heavy computations are encapsulated in functions that are called within a loop body, existing profilers cannot identify the inefficient code inside loops, as profilers rank functions with aggregated execution time. Hence, tools to analyze loops are helpful to identify such cases. Nistor et al. studied memory access patterns and proposed Toddler to detect inefficient loops [32] while Song et al. detect inefficient loops more effectively by combining both static and dynamic analysis on root causes [33]. Tsakiltsidis et al. [34] listed a string of python anti-patterns, which include a couple of examples that we classify as fast-path, e.g. using `if` branches to circumvent heavy computations imposed by logging. To avoid slow-path execution, developers usually have to manually implement the fast-path and ensure that both paths are functionally identical. A fast-path implementation may, for instance, apply a different algorithm to achieve the same result as the slow-path. An example is commit `290ca116c917`[4] in

Listing 2: Example in `grep`

```
+ static int
+ try_fgrep_pattern(int matcher,
+                    char *keys,
+                    size_t *len_p) {
+   /* Implementation */
+ }
  int main(int argc, char **argv) {
    ...
+   else if ((matcher == G_MATCHER_INDEX ||
+             matcher == E_MATCHER_INDEX)
+            && 1 < n_patterns)
+     matcher =
+       try_fgrep_pattern(matcher,
+                         keys, &keycc);
    execute = matchers[matcher].execute;
    ...
  }
```

the `grep` project. Listing 2 shows a simplified diff of this

[4]http://git.savannah.gnu.org/cgit/grep.git/commit/?id=290ca116c9172d97b2b026951fac722d3bd3ced9

commit. The commit fixes a performance regression when multiple regular expression patterns are provided to the program. The generic matcher instance matches slowly, which is why the function `try_fgrep_pattern()` is implemented to "peek" if provided regular expressions can be matched by a light-weight matcher. If they can, the code simply uses the light-weight matcher and else falls back to the generic one.

B. Arguments

Some commits change the values of arguments passed to a function so that the control flow can take an existing fast-path in that function. In a more generic sense, this pattern represents those optimization that bypass heavy or redundant computations by controlling the input value. For instance, when the input value for `bar` in Listing 1 is chosen so that `some_cond(bar)` is more likely to return a non-zero value, the performance of function `foo()` would improve.

As we will see in Section V, this pattern occurs in all targeted projects. We observed that many operations are controlled by flag arguments, i.e., bit fields passed to the processing function to control its behavior. The flag passed to the `do_fork()` function in the Linux kernel, for example, specifies whether the processing fork should copy the page table or file descriptors etc. By setting or clearing bits in the flag, the function execution may behave differently, including the execution of a fast-path instead of a slow-path. In a broader sense, any global state of the program can also be regarded as arguments passed to every function in the program. Thus, like fast-path pattern in Section IV-A, arguments related performance bugs also have a wide range of syntactical representations and often require complex reasoning to set arguments or global variables in a way that improves performance, but does not entail functional deviations.

C. Cache memoization

The result of a computation should be stored if it is needed onward to avoid redundant re-computations of the same result. A trivial loop iterating over a string like in the following example is unnecessarily slow when the code is compiled without compiler optimizations.

```
for (char *c = str;
     c - str < strlen(str);
     c++) { /*...*/ }
```

The loop keeps calculating the (unchanged) length of the dynamically allocated `str` to test whether the iterator has reached the end of the string. Although most modern compilers can move the call to `strlen()` outside the loop body by performing a loop invariant analysis [31, p. 641], the optimization does not cover all cases. If the duplicated calls to `strlen()` are hidden in a wrapper function, the success of the compiler optimization depends on the wrapper returning an invariant value and the ability of the optimization to infer that. We refer to performance bug fixes that "cache" the result of such computations for future usages as *cache memoization*. Intuitively, cache memoization effectively solves the redundancy in the previous duplicated string length computation example. The pattern name is coined in the work by Toffola et al. [35], which lists opportunities to cache computation results in JavaScript.

In C/C++ projects, cache memoization optimizations are also frequently observed. The commit `3548068c22f8`[5] in `clang` exemplifies a typical cache memoization situation. All modifications are applied to the `Sema` class (semantics) of the Objective C frontend. The patch adds a selector variable named `RespondsToSelectorSel` in `Sema` to cache a selector (not shown in Listing 3) and modifies a callback function (`Sema::ActOnInstanceMessage()`) in Listing 3. The callback tests if the event related selector `Sel` equals the contextual unary selector and removes the selector from the warning pool in the case of equality. Instead of fetching the contextual unary selector every time the relevant event is fired, the optimized version tests the equality of `Sel` and the cached contextual selector, and fetches the contextual selector only when it is not yet cached.

Listing 3: A simplified cache memoization example in `clang`

```
- IdentifierInfo *SelectorId =
-   &Context.Idents.get("resp");
- if (Sel ==
-   Context.Selectors.
-   getUnarySelector(SelectorId))
+ if (RespondsToSelectorSel.isNull()) {
+   IdentifierInfo *SelectorId =
+     &Context.Idents.get("resp");
+   RespondsToSelectorSel =
+     Context.Selectors.
+     getUnarySelector(SelectorId);
+ }
+ if (Sel == RespondsToSelectorSel))})
    // remove selector
```

D. Data Access

Different data structures yield different data access overheads, e.g., retrieving unsorted data without an index in a vector or list requires a linear search, while accessing data from hashed maps only introduces the overhead of hash functions. To speed up data accesses, many projects provide developers with a set of predefined data structures optimized for project specific usages. However, depending on the complexity of the project and the variety of data structures, it may not be easy for developers to anticipate how these data structures are best used during development. In `llvm` for instance, `llvm::DenseMap` pre-allocates a large bulk of memory for faster iteration on small key value pairs[6]. As a substitute of `std::map<KeyT, ValT>` from the standard library, `DenseMap` yields better performance for the case in commit `f28cb39e4ca0`[7], as shown in Listing 4. The effect of such a change is usually difficult to predict upfront without intimate knowledge of the data structure and the context in which it is used. We assume that it is for the same reasons that this category of performance bug fixes is less covered in existing work.

[5]git commit id: 3548068c22f809e5bc64b83d2c3622018469256c
[6]http://llvm.org/docs/ProgrammersManual.html#llvm-adt-densemap-h
[7]git commit: f28cb39e4ca07c387dd270ce123753f898a75d5c

Listing 4: A simplified `DenseMap` example in `llvm`

```
  class GlobalsModRef : public ModulePass,
    public AliasAnalysis {
    // ...
-   std::map<const Value *,
-            const GlobalValue *>
-     AllocsForIndirectGlobals;
+   DenseMap<const Value *,
+            const GlobalValue *>
+     AllocsForIndirectGlobals;
```

E. Synchronization

Multicore processors have brought significant performance boosts for parallel and parallelizable programs. Despite the processing speedup, multiple processors accessing the shared memory simultaneously have raised the problem of possible race conditions. To surmount the problem, memory accesses are synchronized by *synchronization primitives* to guard *critical sections*, in which accesses to shared memory are serialized. As serialization diminishes the performance gains from parallel processing, improperly serialized program parts can become performance bottlenecks, e.g., when a critical section protects thread private data (e.g., stack objects) or the critical section is protected by inefficient synchronization primitives, as these primitives themselves yield differing overheads.

In C/C++ projects, mutex-like locks are the most commonly used synchronization primitives, while *spinlocks* are widely used in operating system kernels. Since a mutex may block the execution of the program, low CPU utilization could be regarded as a rudimentary indicator of possible performance problems [5]. Besides CPU utilization, developers nowadays also profile the waiting time of each thread on a lock [6], [7] to localize where locks are mostly contended.

In the projects we assess in this study (as we will see in Section V-A), synchronization related performance problems are relatively infrequent compared to other performance bug fixes (but also take more effort to fix). Our investigation in related projects shows that developers nowadays tend to minimize the amount of shared data to avoid race conditions. Linux kernel developers also tend to use the lockless RCU (Read-Copy-Update) mechanism [36] to prevent heavy weight synchronization.

F. Miscellaneous

Some of the targeted projects introduce low level assembly implementation of algorithms. We refer to such optimization as the *asm* pattern. In particular, cryptographic libraries rely on the fine tuned inline assembly implementations to utilize CPU specific hardware features for speeding up en- and decryption operations. In the two cryptographic libraries we investigate in this paper (textttlibgcrypt and `openssl`) there are profound optimization patches using new CPU features in these two projects[8] as demonstrated in Figure 2. Apart from cryptographic libraries, the Linux kernel also features optimizations of inlined assembly in the sampled commits.

[8] `openssl` uses perl for inlined assembly, so 139 commits involving inlined assembly are not counted in our statistics

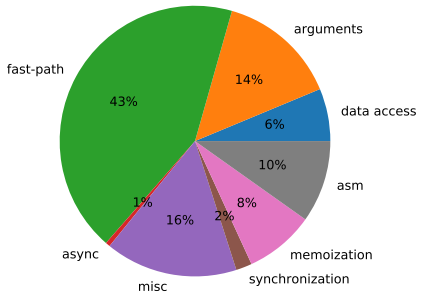

Fig. 1: Distribution of performance bug patterns across all investigated commits

Another less common pattern is the optimization for I/O heavy scenarios. To avoid the time spent on blocking I/O, the optimization uses non-blocking counterparts of the I/O operations and waits for the operation in an asynchronous handler. This pattern is thus labeled *async* and is observed in `systemd` and `NetworkManager`.

Some commits apply fundamental changes to a project to improve performance. Such commits involve changes of the software architecture and highly rely on specific contextual knowledge of the project. Therefore, this category is of limited relevance for the goal of our study and we do not discuss this category in further detail.

V. Performance Bugs Characteristics

In the following we discuss the results of our empirical study of 733 manually investigated performance bug fixing commits from 13 open source projects. The distribution of performance bug patterns is discussed in Section V-A, followed by a discussion of the effort for fixing performance bugs in Section V-B, Section V-C and Section V-D.

A. Bug Pattern Distribution

To assess the relative frequency of different performance bug fixes, we categorize all fixes according to the patterns described in Section IV. Figure 1 shows the result of the classification across all investigated performance bug fixing commits and Figure 2 shows the pattern distribution for each project. The number on each bar is the number of performance bug fixing commits for the respective project.

From Figure 1 we observe that the most dominant form of fixed performance bugs is the “fast-path”, which accounts for $43\,\%$ of all sampled commits in our survey. As discussed in Section IV-A, this pattern corresponds to a wide range of syntactical representations and, intuitively, fast-path is a straightforward way to circumvent slow operations. The second most frequent category is composed of the idiosyncratic performance bugs that do not match any common pattern. The *argument* pattern is the third most frequent pattern contributing $14\,\%$ of all performance bug fixes. As discussed in

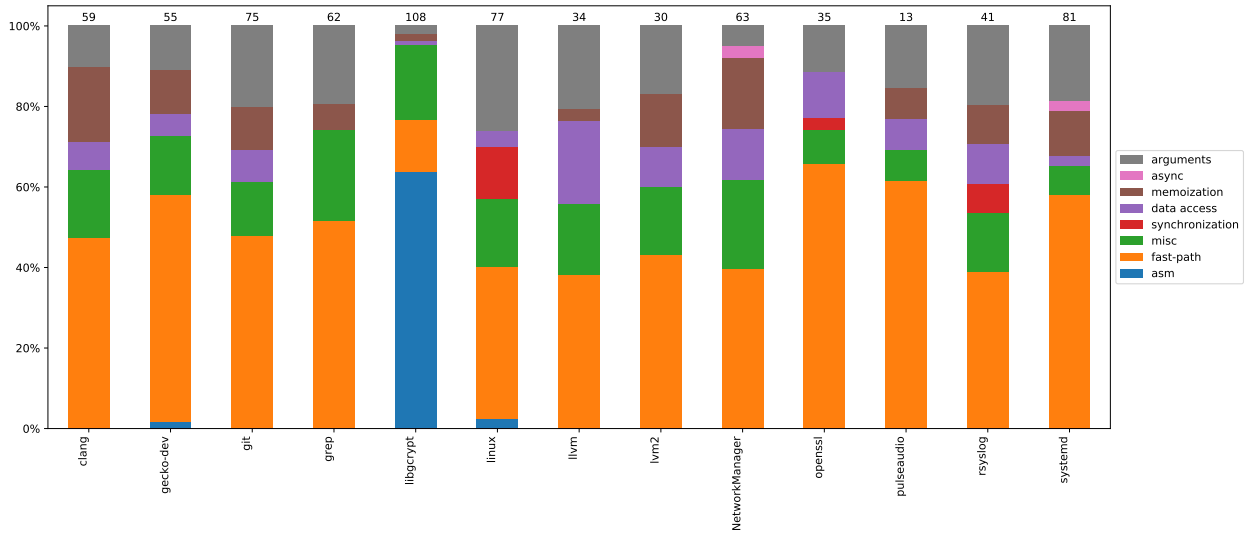

Fig. 2: Distribution of the identified performance bug patterns relative to the number of investigated commits (stated on top of the bars) for each project.

Section IV-B, C and C++ developers often use flags to control dynamic behavior and, thus, tweaking flag arguments passed functions can also optimize the performance.

Surprisingly, performance bug fixes involving inline assembly language account for 10 % of all investigated bug fixing commits. However, as Figure 2 shows, the assembly pattern fixes only occur in three projects, with a strong majority in a single project, i.e., `libgcrypt`. In `libgcrypt` the most frequent performance optimizations are gained by utilizing new CPU features to boost various cryptographic algorithms. The few assembly optimizations in `linux` involve subtle fixes of the crucial procedures written in assembly. From these observations we conclude that assembly based performance bug fixes are strictly limited to very specific application scenarios.

The cache memoization and data access take the most of the remaining code pattern shares, accounting for 8 % and 6 % of the patterns. Although these numbers are not particularly high, it is important to note that the patterns occur across almost all projects in our investigation.

Synchronization problems are the least common performance bug pattern fixed in the investigated commits. Synchronization related performance bugs, in particular those related to lock contention, have been addressed by previous research [5]–[7]. This work, however, is relatively new and we do not expect the developed techniques to be already part of the standard tool set of open source developers. From the commits we investigated, our impression is that synchronization related performance bugs either are a relatively rare occurrence or that they just do not get detected and fixed. Firefox, for instance, was once a project suffering synchronization related performance bugs [5] while in our investigation no synchronization problems have been sampled. Another example is `linux`, where the synchronization fixes in the sampled commits are not related to lock contentions, but substitute locks with lockless RCUs [36].

The least frequent performance bug fix pattern we observed is to make tasks asynchronous. Such optimizations only apply for very specific scenarios, where either some procedures are I/O heavy or the result of the procedure is not needed for some time.

B. Performance Bug Fix Duration

The effort devoted to fixing performance bugs is a significant indicator to prioritize performance problems to be addressed in future research. As discussed in Section III-C, we use the number of commits between the bug introducing and fixing commits to indicate the performance bug fix duration. Figure 3 shows this number (FTCF) across various patterns, where the y-axis is scaled logarithmically due to some outliers with high values.

The boxes in Figure 3 show that the median fix time of most patterns lies between 10 and 100 commits, with the exception of asm and cache memoization. As discussed in Section IV-F, asm basically utilizes new CPU features. Based on our observation that most commits in the asm category replace less efficient assembly instructions by more efficient ones, this indicates that applying new CPU instructions to improve assembly code performance needs less time than optimizing inefficient code written in a higher order language. Another quickly fixed bug pattern is cache memoization. Therefore, redundant computations of invariant results seem to be easy to identify and straight-forward to repair.

Synchronization related bugs have largest discrepancy in FTCF and a 75 % quartile that is two orders of magnitude higher than the median. In Section IV-E we observed that

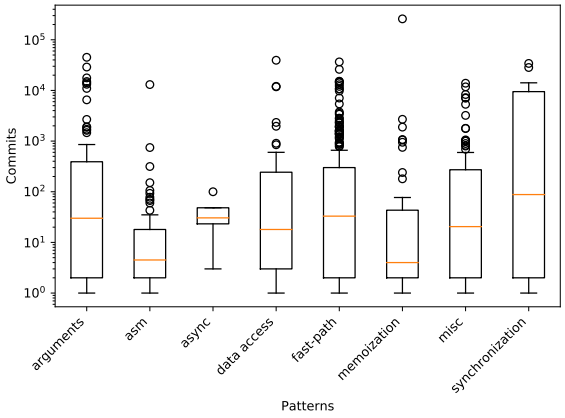

Fig. 3: Performance bug fix duration for different bug patterns measured by FTCF (see Section III-C)

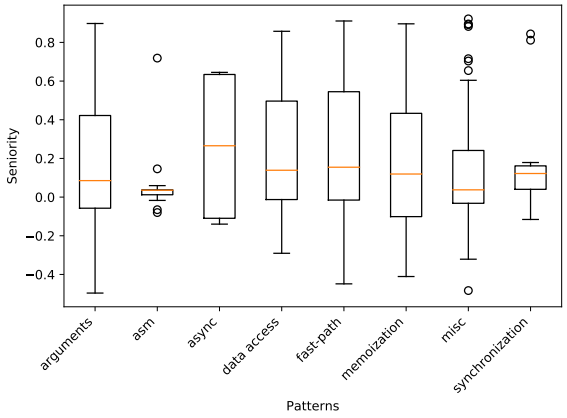

Fig. 4: Seniority of bug fixing developers across performance bug patterns. The metric captures the distance of a project-local seniority metric from the median seniority of all candidate developers for the fix on the same project. A seniority of 0 indicates experience matching the median, positive seniority higher experience, and negative seniority lower experience.

modern synchronization optimizations often either adapt the RCU mechanism or alter the synchronization across concurrent threads. Substituting existing reader-writer locks with RCU is likely to require little effort, while reasoning about and fixing an inefficient synchronization without introducing a race is likely to require more time.

Another observation is that both the fast-path and arguments patterns have a high number of outliers. On the one hand this shows that the majority of performance bugs can be fixed very fast, on the other hand a small amount of such bugs need significantly more time to fix. This observation indicates that our data set comprises few difficult cases of these classes that appear to be challenging in addition to the larger group of simpler cases.

C. Performance Bug Fixing Developer Experience

The second metric indicating performance bug difficulty (*seniority* in Section III-D) reflects the experience of developers who fix performance bugs. The baseline of the seniority metric is 0, when time between the first commit of the bug fixing developer and the bug fixing commit is the median across the respective time differences for all candidate developers who could have fixed the bug at that time. Figure 4 shows the seniority of fixers for each of the patterns. All boxes in Figure 4 have the median greater than 0, which means that performance bug fixing developers usually fall into the group of more experienced developers. Few boxes cross the 0 mark, indicating that for most projects less than 25 % of the performance bug fixes are contributed by the 50 % of the developers who have most recently joined the project. This indicates that fixing performance bugs is likely to require a certain degree of familiarity with the project code and that existing performance diagnosis tools may be difficult to use for less experienced developers.

D. Performance Bug Fix Size

The number of lines that a bug fixing commit consists of is another measure how complex the performance bug fix is. Figure 5 shows the number of modified lines across the patterns[9]. Although different patterns yield different complexity in terms of the code changed, the most frequent patterns have relatively low numbers of changed code lines. Complex code changes comprising hundreds of lines, such as for asm and async, are either project specific or less common. This essentially means that performance bugs can be generally fixed by touching a relatively small amount of source code.

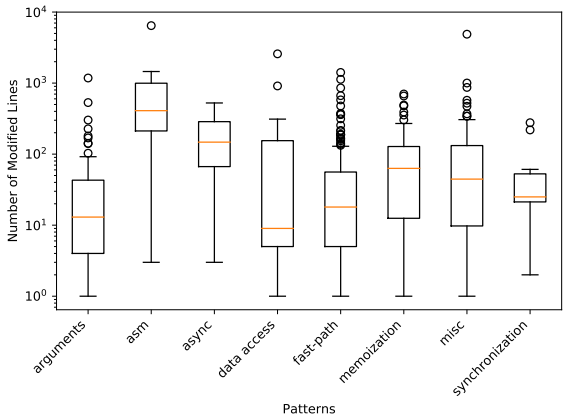

Fig. 5: Modified source lines of code per bug fixing commit across performance bug patterns

Since we envision performance mutations to support future work on performance bug detection, localization, and repair, it is also meaningful to study in which form performance bugs are fixed to guide the creation of corresponding mutation operators. Figure 6 shows the type of changes across performance bug fixes. The most dominant code change type to fix performance bugs is the addition of source code lines. Consequently, mutation operators resembling the identified performance bugs should mostly focus on code removal. This finding is not surprising, as fast-path implementations usually entail the

[9]Estimated by `diffstat -m`

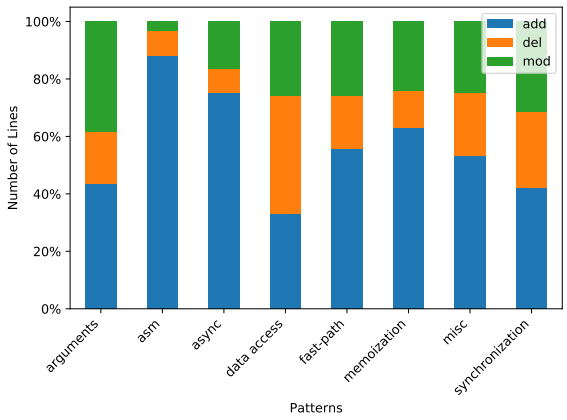

Fig. 6: Relative distribution of source line of code change types (addition, deletion, modification) in performance bug fixes by bug pattern

addition of logic to identify the condition under which the fast-path can be executed and the actual implementation of the faster operation. Unfortunately, generating and adding semantic-preserving code must be expected to be simpler than ensuring that code removals are semantic-preserving, which we consider the main challenge for the realization of realistic performance mutations given the presented observations.

VI. Threats to Validity

There are a number of threats to the validity of the conclusions presented in this paper.

First, we apply a number of heuristics to infer various performance bug characteristics. Our FTCF metric to approximate fix duration in a project-agnostic way requires knowledge of the bug introducing commit and this is based on the unverified assumption that in general each line of a bug fixing commit is necessary for the fix. This is a conservative assumption and, as a consequence, the exact fix times underlying Section V-B could in fact be longer and performance bugs more difficult to fix.Similarly, we approximate developer experience by the duration a developer has been active on the project before contributing the bug fixing commit. This may not accurately reflect developer experience if the developer has been actively developing other projects before. However, our manual investigation of the bug fixing commits show that fixing performance bugs often requires detailed project knowledge, which is less likely to be transferable from other prior projects.

Another threat lies in the criteria of the presented taxonomy, as the borders among semantic categories are fuzzy. In order to limit the impact of this threat we make our entire data set publicly available[10] for reuse and cross-validation.

The third threat comes from the fact that this study focuses on C projects with the exception of `llvm` and `clang`, because we assume that C is mostly used for "low level" programming for which performance is of a higher concern. In industry, C++ is also used for low latency applications where performance matters. While it is possible that different language features result in different syntactic manifestations of performance bugs, we do not expect this to significantly affect the presented results, as the presented taxonomy explicitly abstracts from syntactic details. Accordingly, the performance bug distribution of `llvm` and `clang` is similar to the C projects according to Figure 2. Therefore, the project coverage bias is unlikely to defy the results in Section V.

The last threat is attributed to the selection of popular Debian packages ranked by votes, because there is no guarantee that the votes really reflect the popularity of these packages. Nevertheless, the number of projects investigated is large enough to compensate the drawbacks of possibly selecting less popular projects.

[10]https://yqchen.gitlab.io/perf-bugs/

VII. Conclusion

In this study, we assess the alignment of current research and tool development in the area of performance bug detection and localization with actual performance bugs derived from 733 performance bug fixing commits across 13 open source projects written in C and C++. We manually investigated these commits to confirm they actually constitute performance bug fixes and to group them in semantic categories according to how they intend to achieve a speed-up of the modified code. In summary, we found that more than half of the studied performance bug fixes introduce fast-paths in the control flow or tweak arguments to trigger the execution of existing fast-paths.We define a set of three complexity metrics suited for cross-project comparison, which are related to bug fix duration, developer experience of bug fixing developers, and the amount of code changed by the fix. The empirical assessment of these metrics shows that usually 10 to 100 commits lie between the introduction and removal of performance bugs, that performance bugs tend to be fixed by more experienced developers, and that the lines of code that performance bug fixes comprise greatly vary across different bug categories. From these observations we conclude that performance bugs are fixed in a relatively short time for active projects, but that existing tool support does not effectively target less experienced developers, which is a strong motivation to develop more effective and intuitively usable tools. We also found that performance bug fixes usually entail 10 to 100 changed lines of code, most of which are code additions. Finally, our results provide important insights about the distribution of performance bugs in software projects and about the complexity of these bugs. Besides guiding work on performance bug detection and localization approaches, this data is valuable for the generation of realistic performance mutants to support a fault-based assessment of these approaches.

Acknowledgment

This research is supported in part by H2020-SU-ICT-2018-2 CONCORDIA GA #830927 and BMBF-Hessen TUD CRISP.

REFERENCES

[1] C. H. Kim, J. Rhee, K. H. Lee, X. Zhang, and D. Xu, "Perfguard: Binary-centric application performance monitoring in production environments," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, (New York, NY, USA), pp. 595–606, ACM, 2016.

[2] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *2007 Future of Software Engineering*, FOSE '07, (Washington, DC, USA), pp. 171–187, IEEE Computer Society, 2007.

[3] C. Lemieux, R. Padhye, K. Sen, and D. Song, "Perffuzz: Automatically generating pathological inputs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, (New York, NY, USA), pp. 254–265, ACM, 2018.

[4] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, (New York, NY, USA), pp. 77–88, ACM, 2012.

[5] T. Yu and M. Pradel, "Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, (New York, NY, USA), pp. 389–400, ACM, 2016.

[6] M. M. u. Alam, T. Liu, G. Zeng, and A. Muzahid, "Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs," in *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, (New York, NY, USA), pp. 298–313, ACM, 2017.

[7] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield, "Analyzing lock contention in multithreaded applications," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, (New York, NY, USA), pp. 269–280, ACM, 2010.

[8] M. Böhme and A. Roychoudhury, "CoREBench: Studying Complexity of Regression Errors," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, (New York, NY, USA), pp. 105–115, ACM, 2014.

[9] N. Dmeiri, D. A. Tomassi, Y. Wang, A. Bhowmick, Y.-C. Liu, P. Devanbu, B. Vasilescu, and C. Rubio-González, "BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes," in *Proceedings of the 41st International Conference on Software Engineering (ICSE) 2019 (to appear)*, 2019.

[10] B. Cody-Kenny, M. O'Neill, and S. Barrett, "Performance Localisation," in *Proceedings of the 4th International Workshop on Genetic Improvement Workshop*, GI '18, (New York, NY, USA), pp. 27–34, ACM, 2018.

[11] A. B. Sánchez, P. Delgado-Pérez, I. Medina-Bulo, and S. Segura, "Search-based Mutation Testing to Improve Performance Tests," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '18, (New York, NY, USA), pp. 316–317, ACM, 2018.

[12] C. Killian, K. Nagaraj, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala, "Finding Latent Performance Bugs in Systems Implementations," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, (New York, NY, USA), pp. 17–26, ACM, 2010. event-place: Santa Fe, New Mexico, USA.

[13] M. Jovic, A. Adamoli, and M. Hauswirth, "Catch me if you can: performance bug detection in the wild," in *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pp. 155–170, 2011.

[14] O. Olivo, I. Dillig, and C. Lin, "Static Detection of Asymptotic Performance Bugs in Collection Traversals," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, (New York, NY, USA), pp. 369–378, ACM, 2015. event-place: Portland, OR, USA.

[15] L. Song and S. Lu, "Statistical Debugging for Real-world Performance Problems," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, (New York, NY, USA), pp. 561–578, ACM, 2014. event-place: Portland, Oregon, USA.

[16] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 1013–1024, ACM, 2014.

[17] M. Alam, J. Gottschlich, and A. Muzahid, "AutoPerf: A Generalized Zero-Positive Learning System to Detect Software Performance Anomalies," *arXiv:1709.07536 [cs]*, Sept. 2017. arXiv: 1709.07536.

[18] C. U. Smith and L. G. Williams, "Software performance antipatterns," in *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, (New York, NY, USA), pp. 127–136, ACM, 2000.

[19] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting Performance Anti-patterns for Applications Developed Using Object-relational Mapping," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 1001–1012, ACM, 2014. event-place: Hyderabad, India.

[20] S. Tsakiltsidis, A. Miranskyy, and E. Mazzawi, "On Automatic Detection of Performance Bugs," in *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 132–139, Oct. 2016.

[21] M. Sujon, M. Shafiuzzaman, M. M. Rahman, and R. Rahman, "Characterization and localization of performance-bugs using Naive Bayes approach," in *2016 5th International Conference on Informatics, Electronics and Vision (ICIEV)*, pp. 791–796, May 2016.

[22] C. Trubiani, A. Bran, A. van Hoorn, A. Avritzer, and H. Knoche, "Exploiting load testing and profiling for Performance Antipattern Detection," *Information and Software Technology*, vol. 95, pp. 329–345, Mar. 2018.

[23] X. Han, T. Yu, and D. Lo, "PerfLearner: Learning from Bug Reports to Understand and Generate Performance Test Frames," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, (New York, NY, USA), pp. 17–28, ACM, 2018. event-place: Montpellier, France.

[24] S. Zaman, B. Adams, and A. E. Hassan, "Security Versus Performance Bugs: A Case Study on Firefox," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, (New York, NY, USA), pp. 93–102, ACM, 2011.

[25] S. Zaman, B. Adams, and A. E. Hassan, "A qualitative study on performance bugs," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, (Piscataway, NJ, USA), pp. 199–208, IEEE Press, 2012.

[26] A. Nistor, T. Jiang, and L. Tan, "Discovering, reporting, and fixing performance bugs," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, pp. 237–246, May 2013.

[27] X. Han and T. Yu, "An empirical study on performance bugs for highly configurable software systems," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16, (New York, NY, USA), pp. 23:1–23:10, ACM, 2016.

[28] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Softw. Engg.*, vol. 10, pp. 405–435, Oct. 2005.

[29] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Proceedings of 16th International Conference on Software Engineering*, pp. 191–200, May 1994.

[30] J. Śliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?," in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, (New York, NY, USA), pp. 1–5, ACM, 2005.

[31] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[32] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: Detecting performance problems via similar memory-access patterns," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 562–571, IEEE Press, 2013.

[33] L. Song and S. Lu, "Performance diagnosis for inefficient loops," in *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, (Piscataway, NJ, USA), pp. 370–380, IEEE Press, 2017.

[34] S. Tsakiltsidis, A. V. Miranskyy, and E. Mazzawi, "Towards automated performance bug identification in python," *CoRR*, vol. abs/1607.08506, 2016.

[35] L. Della Toffola, M. Pradel, and T. R. Gross, "Performance problems you can fix: A dynamic analysis of memoization opportunities," *SIGPLAN Not.*, vol. 50, pp. 607–622, Oct. 2015.
[36] P. E. McKenney, "Is parallel programming hard, and, if so, what can you do about it? (v2017.01.02a)," *CoRR*, vol. abs/1701.00854, 2017.