12-2018

PerfLearner: learning from bug reports to understand and generate performance test frames

Xue HAN
University of Kentucky

Tingting YU
University of Kentucky

David LO
Singapore Management University, davidlo@smu.edu.sg

DOI: https://doi.org/10.1145/3238147.3238204

PerfLearner: Learning from Bug Reports to Understand and Generate Performance Test Frames

Xue Han
Department of Computer Science
University of Kentucky
Lexington, KY, USA
xha225@g.uky.edu

Tingting Yu
Department of Computer Science
University of Kentucky
Lexington, KY, USA
tyu@cs.uky.edu

David Lo
School of Information Systems
Singapore Management University
Singapore
davidlo@smu.edu.sg

ABSTRACT

Software performance is important for ensuring the quality of software products. Performance bugs, defined as programming errors that cause significant performance degradation, can lead to slow systems and poor user experience. While there has been some research on automated performance testing such as test case generation, the main idea is to select workload values to increase the program execution times. These techniques often assume the initial test cases have the right combination of input parameters and focus on evolving values of certain input parameters. However, such an assumption may not hold for highly configurable real-word applications, in which the combinations of input parameters can be very large. In this paper, we manually analyze 300 bug reports from three large open source projects - Apache HTTP Server, MySQL, and Mozilla Firefox. We found that 1) exposing performance bugs often requires combinations of multiple input parameters, and 2) certain input parameters are frequently involved in exposing performance bugs. Guided by these findings, we designed and evaluated an automated approach, PerfLearner, to extract execution commands and input parameters from descriptions of performance bug reports and use them to generate test frames for guiding actual performance test case generation.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Software Testing, Performance Bugs, Software Mining

ACM Reference Format:
Xue Han, Tingting Yu, and David Lo. 2018. PerfLearner: Learning from Bug Reports to Understand and Generate Performance Test Frames. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238204

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238204

1 INTRODUCTION

Software performance is critical to the quality of a deployed system. A performance bug can cause significant performance degradation [3], leading to problems such as poor user experience, long response time, and low system throughput [6, 21, 28, 33, 48]. Compared to functional bugs that typically cause system crashes or incorrect results, performance bugs are substantially more difficult to handle [3, 10] because they often manifest themselves by special inputs and in specific execution environments [33, 36]. Over the past decade, numerous research efforts have been made to analyze, detect, and fix performance bugs [7, 22, 28, 29, 34, 36]. For example, many profiling techniques [28] have been proposed to dynamically determine what program entities (e.g., methods) are responsible for the excessive execution time and resource consumption given an input.

Profiling methods depend on the chosen set of input values, which is a known weakness [46] for successfully detecting performance bugs in the subject under test. To address this problem, several test case generation techniques have been proposed to generate large workload test inputs for increasing the chance of exposing performance bugs [7, 41]. However, there are several limitations in existing performance test generation techniques – many techniques focus on evolving the values of certain input parameters while keeping the other parameters as default. For example, Burnim et al. [7] focus on increasing the workload values of data inputs while keeping the values of configuration options as default. These techniques may be ineffective at detecting performance bugs due to combinatorial effects of different input parameters. For example, in Apache bug#52914, the performance bug is exposed only when the configuration options `KeepAlive` and `RequestReadTimeout` are specified. Otherwise, by using the default configuration, this performance bug cannot be triggered even if the workload (e.g., the number of requests) is increased.

While a full performance testing with all combinations of input parameters can address the above problem, it is infeasible due to the enormous combination space. For example, the latest version of Apache HTTP Server has 618 input parameters (610 configuration options and 8 types of data inputs). It is impractical to try all combinations of values for these input parameters. To reduce the cost of performance testing, Shen et al. [46] use a genetic algorithm (GA) as a search heuristic for obtaining combinations of input parameter values that maximize the execution time. However, this technique evolves all input parameters, which can be inefficient because many parameters may not provide contributions to the application's performance.

The goal of our research is twofold. First, we want to understand to what extent performance bugs are related to the combinations of input parameters. A study on performance bug reports from bug tracking systems, such as Bugzilla, can help us understand the characteristics of input parameters and their contributions to performance bugs. Second, we aim to develop a framework to automatically generate combinations of input parameters, also called *test frames* (discussed in Section 2), for guiding the generation of actual performance test cases[1]. To the best of our knowledge, no existing research achieves the same goal.

Our main idea is to mine information from the application's bug reports to identify commands (i.e., commands for executing the program) and input parameters (i.e., configuration options and data inputs) that have caused performance bugs and use them to generate test frames for testing newer versions of the application. PerfLearner is used during software maintenance and evolution, where the projects' issue tracking systems have been established. Specifically, we extract and rank commands and input parameters from each bug report. We then generate test frames (a combination of the commands and input parameters) for each bug report and prioritize the most frequently generated test frames among all bug reports. Our hypothesis includes: 1) bug reports contain a specific set of vocabulary related to commands and input parameters that can make the automated text extraction possible; 2) commands and input parameters appearing frequently in performance bug reports may be more likely to trigger performance bugs than the infrequent ones. PerfLearner is applicable software projects with established issue tracking systems.

In this research, we manually identified and analyzed 300 performance bug reports from three popular open source projects. We discovered that it is possible to leverage information retrieval and natural language processing techniques to extract commands and input parameters from bug reports. We found that some input parameters are more likely to cause performance bugs and should be used with higher priority in performance testing. Based on our findings, we develop PerfLearner, an approach that combines natural language processing and information retrieval to automatically extract relevant commands and input parameters from bug reports and use them to generate performance test frames for guiding performance testing.

In summary, our paper makes the following contributions:

- We develop a tool, PerfLearner, that can automatically extract performance-related commands and input parameters, and generate performance test frames from the bug reports. To the best of our knowledge, this is the first work that automatically generates test frames from bug reports written in natural language.
- We implement PerfLearner and conduct an empirical study to demonstrate its effectiveness and efficiency in generating performance test frames and detecting real performance bugs.

We envision the approach to be applied to at least two scenarios. First, given a performance bug report, a developer who wants to know the commands and input parameters that have caused this bug, may analyze the bug report with PerfLearner. Second, a testing engineer can use PerfLearner to generate and prioritize performance test frames from the historical performance bug reports. The test frames can be converted into actual test cases by giving input parameters with concrete values. Note that PerfLearner is orthogonal to existing performance testing tools. Existing tools focus on increasing the values of certain workload-sensitive input parameters while assuming the test frames (i.e., the combination of input parameters) exist. Therefore, PerfLearner can be used to enhance the effectiveness and efficiency of existing performance testing tools.

To evaluate the approach, we apply PerfLearner to 300 bug reports collected from Apache HTTP Server, MySQL, and Firefox bug tracking systems. Our results show that PerfLearner is able to extract commands and input parameters from performance bug reports with a high accuracy. When using PerfLearner to generate test frames, compared to a state-of-the-art combinatorial testing (CT) technique, it generates significantly less (59.5%) test frames on average to get the ground truth test frame. When combining PerfLearner with an existing performance test input generation tool [46] to detect 10 randomly selected performance bugs, PerfLearner detects 7 out of 10 bugs within a reasonable time whereas when using the test input generation tool alone failed to detect all 10 bugs.

2 BACKGROUND

The concept of *test frame* was first introduced in the category-partition method with test specification language (TSL) [37]. TSL was created to define combinations of program input parameters and environment factors. Each combination is a test frame that can be converted into actual test cases. A performance test frame consists of three input categories: command, configuration, and data input. A test frame can have one command in the command category, zero or more configuration options in the configuration category, and zero or more data inputs in the data input category. Each command, configuration option, and data input in a test frame is generally referred to as a *test frame element or frame element*.

We define a *command* as an action to execute a functional unit [37] of the program. For example, the MySQL server has several data manipulation commands, including `SELECT`, `UPDATE`, and `INSERT`. These commands correspond to three different functional units: retrieve, modify, and add data records. We define *input parameters* as explicit input points along with the command. An input parameter can be a *configuration option* or a *data input*. Configuration options refer to a set of predefined options, e.g., command-line options or directives in a configuration file. Data inputs refer to the user-supplied data that is processed by the command. For example, the data input associated with the command `UPDATE` is the name of a table `COLUMN`.

Figure 1 shows a performance bug report snippet with the associated test frame and a test case. The test frame for manifesting this performance bug involves three frame elements: a command `UPDATE`, a configuration option `innodb_fill_factor`, and a data input `COLUMN`. A frame element can be *workload-sensitive*. In this example, the `UPDATE` command is workload-sensitive because a large number of `UPDATE` queries is required to trigger the performance

[1] An actual test case is built from a test frame by specifying a concrete value for each input parameter [37].

Bug Description
Updates to indexed column much slower in 5.7.5. Repeating the test done for Heap engine on InnoDB shows a big regression for updates to an indexed column. InnoDB is more than 2X slower than 5.6.21 and 4X slower than 5.0.85. . . . innodb_fill_factor whose default value is 100 . . .
Test Frame
<update>[workload] <innodb_fill_factor> <column>
Test Case
update foo i = i + 2 where i = 100 innodb_fill_factor=100
mysqlslap –number-of-queries=100

Figure 1: MySQL bug #74325

bug. In MySQL, a workload can be simulated by benchmark tools[2] such as `mysqlslap`. Since many performance test generation techniques have been focusing on identifying the workload-sensitive inputs [18, 50], pinpointing the workload from a bug report may speed up this process for performance test case generation techniques. The actual test case is created by assigning concrete values to frame elements.

3 PERFORMANCE BUG STUDY

Before designing our approach, we wish to understand to what extent performance bugs are related to certain commands and input parameters.

3.1 Data Collection

We chose three large open source software projects: Apache HTTP Server, MySQL Database Server, and Mozilla Firefox Browser. With publicly accessible source code and well-maintained bug tracking systems, these projects have been widely used as subject programs by existing bug characteristic studies [28, 54, 55].

We collected performance bugs from bug tracking systems of Apache, MySQL, and Firefox. We searched these systems using a set of commonly used general keywords and phrases to describe the symptoms of performance bugs, such as "slow", "latency", and "low throughput" [23]. We also searched terms that attribute to a specific aspect of the performance problems such as "CPU spikes", "cache hit", and "memory leak" to identify performance bugs. Next, we selected reports with the bug status field marked as either "RESOLVED", "VERIFIED", or "CLOSED" and the resolution field marked as "FIXED".

The whole process yielded a total of 1383 bugs. With a large amount of the returned bug reports, we calculate the needed sample size is 300, given a confidence level of 95% and a confidence interval of 5. This sampling strategy has been commonly used by existing work [2, 20].

We manually examined 300 bugs in a random order, and during the manual inspection, we follow those reports that have sufficient bug description details and discussions posted by commentators. For each bug report, we try to identify commands, configuration options, data inputs, and workload that cause the performance bug.

To ensure the correctness of our results, the manual inspection was performed independently by two inspectors – graduate students who have 2-4 years of industrial web development experience

[2] A benchmark tool is used to measure the performance of the program under test with synthesized workload.

Table 1: Subjects and Their Characteristics

Application	*Searched Bugs*	*Sampled Bugs*	*# of CMD*	*# of CO*	*# of DI*
Apache	428	100	10	610	8
MySQL	455	100	11	1240	5
Firefox	500	100	24	563	17
Total	1383	300	45	2413	30

with Apache, MySQL, and Firefox. We hold two training sessions of 30 minutes each to explain to inspectors the test frame elements to be extracted from the bug report. Each inspector is given the same set of bugs each week to write down what they consider to be the command, configuration options, data inputs, and workload that trigger the bug in the report. Inspectors met twice a week to compare and consolidate their findings. A bug report is selected only when both inspectors agree on the outcome of the manual inspection. We refer to the consensus outcome as ground truth frame elements for the bug reports. This process terminates for each subject after 100 bug reports have been included in the sample dataset.

The number of bugs sampled is similar to recent works on performance bug study [10, 23, 28, 56]. While a larger number of bug reports may yield a better evaluation, the cost of the manual process is high — our data collection process took a total of 320 to 400 hours spanning across more than 10 weeks. Columns 1-3 of Table 1 list the subject programs, the number of bugs returned by the keyword search, and the number of performance bugs sampled after manual inspection. Columns 4-7 list the number of commands, configuration options, and data inputs available in all three subjects. The full lists of the three categories are saved in separate *frame element databases*, including command database, configuration database, and data input database. We collected such information by studying all artifacts that are publicly available to users, including documents (e.g., user manuals and online help pages), configuration files, and source code. Each database can be updated separately to accommodate changes in different application versions.

3.2 Results Analysis

After manually analyzing 300 bug reports, we summarize the following findings:

- A majority (89% to 92%) of studied performance bugs involves more than one input parameters (i.e., configuration options and data inputs): 91% in Apache, 92% in MySQL, and 89% in Firefox. These results imply that combinatorial effects among input parameters should be considered in performance testing.
- A significant number (41%) of performance bugs are related to configurations: 58% in Apache, 41% in MySQL, and 25% in Firefox. These results are consistent with a recent performance bug study [23].
- Only 23% of bugs require specific workload values to manifest: 21% in Apache, 29% in MySQL, and 19% in Firefox. These results imply that workload is only part of the requirement for exposing performance bugs; other factors, such as configuration options, should also be considered for performance testing.

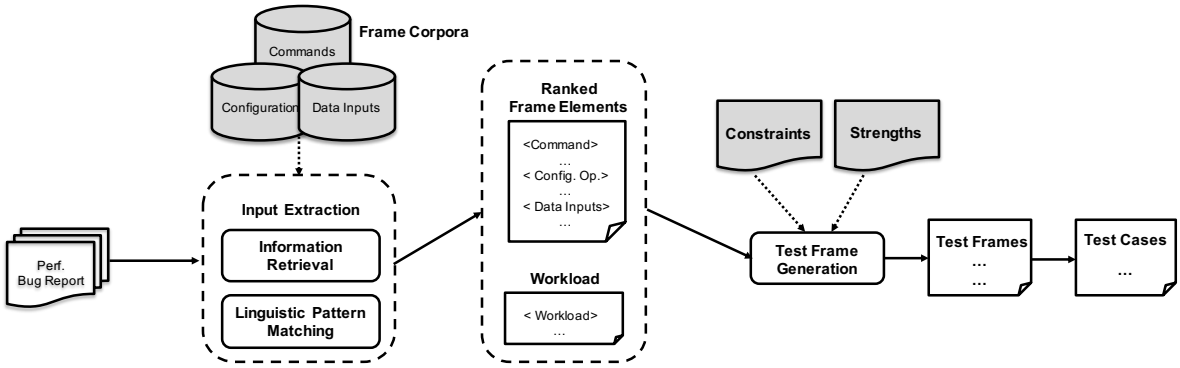

Figure 2: Overview of the PerfLearner framework

- Among all 45 commands and 30 inputs for the three software projects, 55% of them appear more than three times in the studied bug reports. Among all 2413 configuration options for the three software projects, only 5% of them are related to the studied performance bug reports and 57% of them appear more than one time. These results suggest that a small subset of configuration options tend to affect application's performance, so performance testing might focus mainly on such options to improve the efficiency of testing. In addition, test frame elements that appear multiple times in performance bug reports might be more likely to cause performance bugs than the others and should be given higher priority in performance testing.

4 PERFLEARNER APPROACH

Guided by the findings in Section 3, we design and develop PerfLearner, an automated approach for extracting performance test frames from bug reports. Figure 2 shows an overview of the PerfLearner framework. PerfLearner consists of three steps: frame element extraction, test frame generation, and performance test case generation. The shaded boxes indicate the information supplied by users.

Frame Element Extraction. Given a performance bug report, PerfLearner automatically extracts frame elements and their associated workload from the report. PerfLearner assumes that a bug report has already been labeled as "performance bug", although existing techniques on classifying bug reports [17, 47, 49] can be adopted to automatically classify performance bugs. The list of frame elements are application and domain-specific, e.g., each application is associated with a list of different configuration options. The bug corpora for each application is built from sources described in Section 3.1. The output of this step is a list of ranked frame elements and their associated workload (if any) under each input category for each bug report.

Performance Test Frame Generation. PerfLearner utilizes ranked frame elements, a strength file, and a constraint file to generate performance test frames. The strength file, which is used to restrict the number of test frames, specifies the strength[3] of interaction among elements within each input category. The constraint file specifies the constraints among frame elements to ensure their combinations are valid. Both files are defined once by developers for each application, and generic to all bug reports in the same application. Next, PerfLearner generates a set of test frames for each performance bug report by combining the selected commands and input parameters with respect to the strength and constraint files. These test frames are closely related to the performance bug described in the report. Finally, PerfLearner counts the frequency of test frames generated from all bug reports and ranks them in a descending order. The top-ranked test frames are used first to generate performance test cases.

Performance Test Case Generation. PerfLearner iteratively selects a test frame from the ranked test frames and converts it into actual performance test cases by assigning frame elements with concrete values. PerfLearner can be combined with existing performance testing tools, such as profiling and test generation tools. The current version of PerfLearner is combined with a performance test input generation tool [46] that uses a search-based algorithm to automatically generate input values to expose performance bugs.

4.1 Test Frame Element Extraction

For each bug report that is labeled as a performance bug report, PerfLearner extracts commands, configuration options, data inputs, and the associated workload. A straightforward approach is to match frame element databases against each bug report using a "grep"-like method. The matched elements can then be ranked by counting their occurrences – the element with the highest count is more likely to be the ground truth frame element for the performance bug. However, in a bug report written in natural language, many words can be ambiguous in their meaning – the same word can refer to a command or a configuration option depending on the context. For example, in the Apache bug #52914, the word token `timeout` can be matched as either a command or a configuration option. In addition, simply counting the occurrence of a token may result in false positives. In the Apache bug #52914, both `start` and `request` appear in the bug report, so both tokens would be matched as commands of this bug report. Incidentally, the count of `start` is actually higher than the count of `request`, although the ground truth command is `request`.

PerfLearner employs two strategies to address the above problems. First, PerfLearner uses natural language processing and information retrieval, together with user manuals to address the

[3]Combinatorial testing of strength t ($t \geq 2$) requires that each t-wise tuple of values of the different system input parameters is covered by at least one test case [8].

Table 2: Number of Patterns to Detect Frame Elements

Application	*# of Matched Patterns*		
	Commands (8)	Data Inputs (4)	Workload (6)
Apache	104	77	168
MySQL	228	159	453
Firefox	203	146	270

Definitions: [CMD]∈{drop, create, ...}, [SYMP]∈{slow, long, ...}
Pattern:[CMD]+[SYMP]
Description: Command verbs appear in the same sentence that symptoms exist.
Example: $[\text{DROP}]_{CMD}$ TABLE on very large tables can be very $[\text{slow}]_{SYMP}$.

Figure 3: A common pattern to identify command

Definitions: <ADP>∈{to, on, ...}, <NOUN>∈{[DATA INPUT]}, [CMD]∈{update, insert, ...}
Pattern: [CMD]+<ADP>+<NOUN>
Description: Data input is identified as the subject of the command.
Example: $[\text{update}]_{CMD}$ $[\text{to}]_{ADP}$ indexed $[\text{column}]_{NOUN}$ much slower in 5.7.5

Figure 4: A common pattern to identify data inputs

Definitions: [INPUT]∈{file, html, ...}, <VERB>∈{contain, has, ...}, <ADJ>∈{long, large, ...}
Pattern: [INPUT]+<VERB>+<ADJ>+<NOUN>
Description: Workload details the content of data inputs.
Example: a text $[\text{file}]_{INPUT}$ $[\text{containing}]_{VERB}$ a very $[\text{long}]_{ADJ}$ $[\text{line}]_{NOUN}$

Figure 5: A common pattern to determine a workload

mismatch problem between the frame elements (query) and bug reports (documents). Second, we summarize 18 linguistic patterns that are commonly used to describe commands (eight patterns), input parameters (four patterns), and workload in bug reports (six patterns). While the frame elements are application-specific, the linguistic patterns are generic and hence can be reused for different applications.

To avoid overfitting, the first author summarized the linguistic patterns from the 1083 bug reports (excluding the 300 sampled bug reports in the dataset). In the experiment, these patterns are applied to the 300 bug reports. We can automatically detect the presence of these patterns to locate sentences describing a particular input category and identify the frame element under that category more accurately. Table 2 shows the number of patterns we identified in all sentences from the 300 bug reports. While there has been some research on using linguistic patterns in other software activities, such as analyzing developer intention [11, 30] and detecting missing information [9], little work is known on using linguistic patterns to identify commands and input parameters.

4.1.1 Commands. We observe that a command often appears with the bug symptom in one sentence. For example, the sentence describing the symptom of Apache bug #52914 is "I could reproduce the 100% CPU with POST requests", where the symptom is "100% CPU" and the command is `request`. If we identify sentences containing bug symptoms, it narrows down the search and improves the accuracy of finding the performance bug-triggering commands.

We have defined six linguistic patterns using the part-of-speech tag for detecting (one or more) sentences containing symptoms. If such sentences are detected, PerfLearner matches the command against these sentences and counts their occurrences. The identified k commands are ranked at the top k position in a descending order with respect to their occurrences.

Our patterns can precisely identify commands in 91% performance bug reports (i.e., ranked at the top-1), compared to the 78% precision rate by the "grep"-like method. The most frequently used pattern, as seen in Figure 3, illustrates a pattern that uses a verb and a phrase, where the verb refers to the command element and the phrase refers to the predefined list of phrases indicating performance bug symptoms. If any of the symptoms appear in the sentence, the verb is identified as a candidate of the bug-triggering command. If no symptom sentences are detected, PerfLearner prioritizes sentences that appeared in the bug report title as well as the first post, and uses the 'grep"-like method to count the occurrences of commands. If no command sentences are detected, the same approach is applied to the entire bug corpus.

4.1.2 Data Inputs. PerfLearner ranks data inputs in a similar way as commands because simply matching a bug report against the elements in data input is imprecise. PerfLearner defines four linguistic patterns to detect sentences that contain data inputs and rank data inputs within these sentences. Figure 4 shows one of the commonly used patterns. This pattern indicates that data inputs coexist with commands in the same sentence. Specifically, the sentence starts with a command (i.e. update), followed by a preposition (i.e. to, on) and the data input (i.e. column).

4.1.3 Configuration Options. Unlike commands and data inputs, we observe that many configuration options cannot be directly searched from bug reports. One solution is to leverage information retrieval (IR) algorithms such as TF-IDF [40] and cosine similarity [16] based on the vector space model (VSM) to rank configuration options in terms of their relevance to the bug report. A straightforward method is to split the configuration name into tokens to calculate its cosine similarity to the bug report. However, we observe that many configuration options share with the same tokens. Since a configuration option name is often short, this approach may result in many equally ranked configuration options. For example, in Figure 1, `innodb_buffer_pool_instances` and `innodb_buffer_pool_size` would be ranked equally if "innodb", "buffer", and "pool" are the three word tokens appearing in the report.

To improve the accuracy of ranking, we leverage manuals that describe configuration options to bridge the lexical gap between configuration option names and bug reports. In the example of Figure 1, the manual description of `innodb_fill_factor` (Figure 7) contains words such as "b-tree", "index", and "space", which also appear in the bug report, can be used to link the configuration option to the bug report effectively.

To compute the similarity between a configuration option o and a bug report br, we first concatenate o with its textual description, where $o = o \cup o.desc$. PerfLearner then processes o by standard NLP pre-processing steps: *word tokenization* and *stop word removal*. The tokenization converts bug reports into a "bag of words" using white spaces. We then remove punctuation, numbers, and standard stop

words. Compound words, such as the configuration option name `Browser.chrome.image_icons.max_size` can be split by camel case, dots, or underlines into tokens.

Next, all words are reduced to their base form using lemmatization. Unlike stemming that simply chops off the ending of a word, lemmatization involves a complex word analysis and generally provides better results. Finally, we also remove repeated text sections, as quotations of the previous commentator in the bug report happen very frequently and the repeated text would affect the accuracy of text token distribution. Each bug tracking system may have their own mechanism to mark quotations. For example, Bugzilla based bug tracking systems, quotation starts with the greater sign (">") symbol on each new line and the quotation block has a CSS class of "quote". Developers can design their own match patterns for removing quotations and plug it into PerfLearner.

After processing o (the combined configuration option and its description), let V be the vocabulary of all text tokens from both the bug report br and o. Let r = $[w_{t,br}|t \in V]$ and o = $[w_{t,o}|t \in V]$ be the VSM representations of the bug report br and the configuration option o. The term weights $w_{t,br}$ and $w_{t,o}$ are computed using the classical TF-IDF method described in existing literature [16]. After the vector space representations are computed, the textual similarity score between o and br can be calculated using the standard *cosine similarity* between their corresponding vectors:

$$sim(br, o) = cos(br, o) = \frac{r \times o}{|r| \times |o|}$$

The score is computed by the inner product of the two vectors, divided by their Euclidean distance. For MySQL bug #74325 (Figure 1), by utilizing the configuration API description (Figure 7), PerfLearner ranks `innodb_fill_factor` at the top.

4.1.4 Identifying Workload. In performance testing, we need to know which frame elements are workload-sensitive, so testing can focus on generating workload values for these elements. We have defined six linguistic patterns to identify such frame elements. The most frequently used pattern is to locate sentences containing benchmark tool names. Benchmark tools are often used to simulate workload in performance bug reports. For instance, MySQL bug report #74325 uses benchmark tool `mysqlslap` to generate a large number of database updates. Therefore, by searching for the benchmark name `mysqlslap`, we can detect that `update` is workload-sensitive. This pattern applies to 44.2% of performance bug reports involving specific workload.

The second commonly used linguistic pattern detects sentences describing workload information of data inputs (Figure 5). In this pattern, the data input (i.e. a text file) is followed by a verb (i.e. containing) that details the content of input data (i.e. a very long line). Once this pattern is detected, the corresponding data input is considered to be workload-sensitive.

4.2 Performance Test Frame Generation

PerfLearner generates performance test frames from the ranked frame elements, the workload specification, a strength file, and a constraint file. The strength file specifies top-N frame elements under each input category to be used for test frame generation. The constraint file is used to enforce constraints of interaction among frame elements, which can limit the number of (invalid) frames

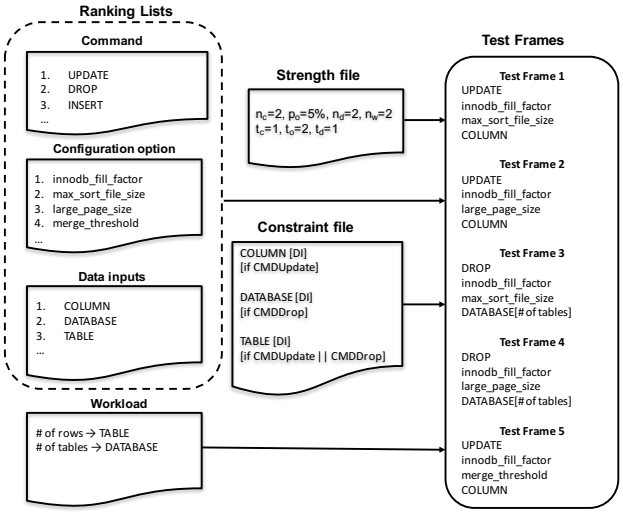

Figure 6: An example of performance test frame generation

> InnoDB performs a bulk load when creating or rebuilding indexes. This method of index creation is known as a "sorted index build". innodb_fill_factor defines the percentage of space on each B-tree page that is filled during a sorted index build, with the remaining space reserved for future index growth.

Figure 7: API description for innodb_fill_factor

to be generated. The constraints are manually derived from user manuals. Both files are provided by users and generic to all bug reports within the same application.

Figure 6 shows a partial constraint file of MySQL. The data definition command `DROP` in the SQL works with DATABASE and TABLE but not with COLUMN. We use `if` to enforce conditions on which frame elements can be combined. To enforce the rule that `UPDATE` works with TABLE but not DATABASE, condition [if CMDUpdate] is added for data inputs COLUMN and TABLE. Condition [if CMDDrop] is added for data inputs DATABASE. Therefore, when `UPDATE` is chosen, it can only be combined with COLUMN and TABLE. Our experiment indicates that adding constraints can reduce 70% of test frames.

In the example of Figure 6, the strength file indicates that top-2 commands (n_c), top-5% configuration options (p_o), and top-2 data inputs (n_d) are selected to generate test frames. Because the number of options is often large, we use a percentage of the total number of configuration options to indicate the selected number of configuration options. The three symbols t_c, t_o, and t_d indicate the interaction strengths for commands, configuration options, and data inputs respectively. In Figure 6, a pair-wise combination (t_o=2) is applied to the configuration options and no combinations are used for the command (t_c=1) and data inputs (t_d=1). Figure 6 also shows the default strength file used by PerfLearner. These strength values are chosen based on our empirical evaluation as they are the minimum requirements for generating test frames achieving up to 90% accuracy. We also evaluated the sensitivity of these values in Section 7.

Algorithm 1 PerfLearner Test Frame Generation

Require: $StrenF, ConsF, BugReports, LPtn$
Ensure: TF_{prio}
1: **for** $br \in BugReports$ **do**
2: $RL_{cmd} \leftarrow$ RankCmd $(br, DB_{cmd}, LPtn.cmd)$
3: $RL_{co} \leftarrow$ RankConfig (br, DB_{co})
4: $RL_{di} \leftarrow$ RankData $(br, DB_{di}, DB_{di}, LPtn.di)$
5: $L_{wl} \leftarrow$ GetWorkload $(br, DB_{di}, LPtn.wl)$
6: $RL \leftarrow$ SelectElements $(RL_{cmd}, RL_{co}, RL_{di}, StrenF)$
7: $TF_{br} \leftarrow$ GenerateFrames $(RL, L_{wl}, ConsF)$
8: $TF \leftarrow TF_{br} \cup TF$
9: **end for**
10: $TF_{prio} \leftarrow$ RankFreq (TF)

Algorithm 2 Combining PerfLearner with Testing Tools

Require: TF_{prio}
Ensure: $TestResults$
1: **for** $tf \in TF_{prio}$ **do**
2: **for** $e \in tf$ **do**
3: $category \leftarrow GetInputCategory(e)$
4: **if** $HasWorkload(e)$ **then**
5: $e \leftarrow IntegrateWorkload(category, e)$
6: **end if**
7: tc.xml $\leftarrow UpdateTestCase(e)$
8: **end for**
9: TestResults $\leftarrow RunPerfTestTool(tc.xml)$
10: **end for**

Algorithm 1 describes the process of generating performance test frames. The algorithm takes as input a list of bug reports from an application, a strength file, and a constraint file. For each bug report, the algorithm obtains a ranked list for each input category (Lines 2-4) and a list of workload (Line 5). It then selects frame elements from the ranked lists with respect to the strengths. Next, a list of candidate test frames is generated given the selected frame elements and the constraints (Line 7). Finally, the algorithm ranks test frames collected from all bug reports (Line 10) in terms of the frequency of their appearance. Test frames ranked higher indicate they may be more likely to cause performance bugs. The last column of Figure 6 shows an example of the five test frames generated.

4.3 Performance Test Case Generation

Algorithm 2 outlines the process of generating performance test cases from test frames. First, PerfLearner iteratively selects a test frame from the prioritized list output by Algorithm 1. For each frame element, the algorithm checks for its input category. If the frame element is workload-sensitive, depending on the input category, the algorithm applies workload in two ways (Line 5). For the command category, the benchmark option that controls workload is included in the test case generation. For other input categories, the input size is included in the test case generation. The algorithm updates the test case as it gets more information from frame elements (Line 7). Specifically, the test frame is converted into an XML file (tc.xml) of which structure is known to the test case generation tools. Finally, the test input (tc.xml) is supplied to the performance testing tool. It is up to the performance testing tool to determine how to assign input values and execute the subject under test to detect performance bugs.

5 IMPLEMENTATION

We implemented a web crawler using the Python Beautiful Soup library [5] to collect raw bug reports and API documentations. We then leveraged Python Natural Language Toolkit (NLTK) [35] to parse the description of the bug reports and match linguistic patterns against the new bug reports with regular expressions on part-of-speech tags. For the information retrieval component, we utilized the Python machine learning library scikit-learn [45] to get the TF-IDF matrix and cosine similarity scores. Lastly, we implemented Python programs to handle the performance test frame generation.

6 EVALUATION OF PERFLEARNER

We evaluated PerfLearner on three open source projects with characteristics described in Section 3.1. We aim to answer the following research questions:

RQ1: How accurate is PerfLearner at detecting performance bug-triggering frame elements and workload?

RQ2: How effective and efficient is PerfLearner at generating performance test frames?

RQ3: Can PerfLearner enhance existing performance testing tools for detecting performance bugs?

6.1 Techniques and Metrics

RQ1: Accuracy of Bug Reports Analysis. To answer RQ1, we evaluate the accuracy of PerfLearner in extracting frame elements and workload. The techniques for extracting commands, configuration options, data inputs, and workload are denoted as CD, CO, DI, WL, respectively. Each technique is compared to a baseline method to evaluate the effects of using advanced techniques such as linguistic patterns and information retrieval (TF-IDF, Cosine Similarity etc.). Specifically, we compare CD, CO, DI to three baseline techniques – CD_s, CO_s, and DI_s. These baseline techniques use a keyword match and count the occurrence of each frame element appearing in a bug report. To evaluate the usefulness of configuration manuals in extracting configuration options, we also compare CO to CO_a. CO_a uses only tokens in the configuration option name without configuration manuals to make the similarity comparison. Since the workload describes whether a frame element is workload-sensitive, the keyword counting is not applicable in this case. Nevertheless, to evaluate the usefulness of linguistic patterns in identifying the workload, the baseline technique WL_r randomly selects a frame element and treats the element as workload-sensitive.

We use two metrics to evaluate the effectiveness of ranking. The first metric is the top-N success rate, which is computed by ranks of ground truths within top N items over all bug reports. For example, if 20 out of 100 performance bug reports rank the ground truth of configuration options in the top 5% of all 600 configuration options, the top-N (N=5%) success rate is 20%. When there are multiple elements specified as the ground truth, we only consider the first one that PerfLearner can find. Since workload is directly identified without ranking, we examine the percentage of bug reports in which ground truth workload is found.

For the second metric, we use MAP (Mean Average Precision). MAP is a single-figure measure of ranked retrieval results independent of the size of the top list [44]. It is designed for general ranked retrieval problems, where a query can have multiple relevant documents. To compute MAP, it first calculates the average precision

(AP) for each individual query Q_i, and then calculates the mean of APs on the set of queries Q:

$$MAP = \frac{1}{|Q|} \cdot \sum_{Q_i \in Q} AP(Q_i)$$

To illustrate the MAP calculation, suppose there are two configuration options *o*1 and *o*2 associated with a bug report. If Technique-I ranks the two options at the 1st and 2nd positions among all 500 options and Technique-II ranks the two options at the 1st and 3rd positions, then the MAP of Technique-I is (1/1 + 2/2)/2 = 1 and the MAP of Technique-II is (1/1 + 2/3)/2 = 0.8.

RQ2: Effectiveness and Efficiency of Generating Performance Test Frames. To answer RQ2, ideally, the comparison should be done with existing approaches that generate performance test frames. However, we cannot find an existing approach with this specific goal. In the absence of such approaches, we instead compare PerfLearner to a combinatorial testing (CT) strategy [19] that employs the category-partition method [37], t-wise testing [39], and the random testing approach. Specifically, CT generates test frames by combining elements under each input category with respect to the constraints. The first difference between PerfLearner and CT is that CT does not analyze bug reports or rank frame elements in terms of their relevance to the report; instead, CT ranks the frame elements in a random order. The second difference is that in CT, the workload is randomly assigned to a frame element. To make a fair comparison, the interaction strength of configuration options and that of data inputs are the same as those used in PerfLearner.

To evaluate the cost-effectiveness of PerfLearner and CT in generating performance test frames, we wish to know whether frame elements frequently appeared in historical bug reports can be used to generate test frame for testing future versions of the programs. For each bug report used for evaluation, we manually inspect and derive the test frame that triggers the performance bug described in the report (Section 3.1). We refer to this test frame as the ground truth test frame. Since test frames cannot be executed directly, we consider an approach detects the bug if the ground truth test frame is included in the generated test frames. To do this, we first list the 100 bug reports from each program in ascending order by the bug creation date. We then select the first 90 bug reports (training set) and apply techniques (PerfLearner and CT) described in Section 4.2 to generate test frames. We compare the test frames generated by each technique against the remaining 10 bug reports (test set) from each subject. Specifically, we examine at which iteration the ground truth test frame of the test set bug report is generated by the technique. To evaluate the efficiency of the two techniques, we evaluate the time they take to generate the ground truth test frames.

RQ3: Detecting Performance Bugs. Besides evaluating PerfLearner on generating performance test frames, we would like to know whether the generated frames are useful for detecting actual performance bugs. PerfLearner is orthogonal to existing performance testing tools. It aims to improve the efficiency of testing by focusing on selecting commands and input parameters that are more likely to expose performance bugs. To answer RQ3, we combine PerfLearner with GA-Prof, a performance test input generation tool to detect performance bugs [46]. We choose GA-Prof because it is the only tool that can evolve both configuration option and data input values. GA-Prof employs a genetic algorithm to explore the space of input combinations among *all* input parameters. We re-implemented the genetic algorithm part of GA-Prof to handle C/C++ applications. We compare two settings of GA-Prof: 1) a default setting (denoted by GA) in which the combinations are evolved for all commands and input parameters, and 2) an enhanced technique, denoted by GP_{PL} where it utilizes test frames generated from PerfLearner to iteratively select and evolve input values to generate performance test cases.

Table 3: RQ1: Test Frame Extraction Accuracy

App.	*Metric*	*Command*		*Data Input*		*Config. Option*			*Metric*	*Workload*	
		CD	CD_s	DI	DI_s	CO	CO_s	PL_a		WL	WL_r
Ap.	Top-N	91%	78%	83%	67%	71%	71%	67%	Acc.	78%	60%
	MAP	0.80	0.70	0.70	0.60	0.37	0.22	0.33			
My.	Top-N	83%	75%	91%	81%	83%	75%	53%	Acc.	67%	43%
	MAP	0.60	0.50	0.80	0.70	0.24	0.23	0.21			
Fi.	Top-N	82%	80%	90%	90%	85%	83%	60%	Acc.	80%	42%
	MAP	0.80	0.60	0.70	0.60	0.28	0.22	0.20			

To evaluate whether the two techniques are able to detect performance bugs within a reasonable time limit, we select real performance bugs that we can reproduce. We iteratively select a bug report from the 1083 performance bug reports (excluding the 300 sampled bug reports in the dataset) and try to reproduce the bug. Because reproducing performance bugs is challenging and expensive, we stop this process after we have 10 bugs successfully reproduced – this process took approximately 400 work hours.

Next, we apply the two techniques to the program versions corresponding to the 10 performance bugs. We evaluate whether the performance bug described in the bug report can be detected and record the time it takes. Specifically, we conduct test experiments on High-Performance Computer (HPC) clusters. The basic HPC node is equipped with a 6 core 2.66 GHz Intel Xeon X5650 Westmere, 36 GB memory, and 256 GB hard drive. This environment enables us to run multiple experiments simultaneously without interruption. Each experiment is repeated 10 times and we report the mean to reduce the bias due to randomness. We default the time limit to 24 hours before terminating the experiment and set the maximum number of GA iterations in each run to be 10.

6.2 Results and Analysis

RQ1: Accuracy of Bug Reports Analysis. Table 3 shows the effectiveness of different techniques at ranking frame elements and extracting workload. The success rates are based on the default values specified in the strength file. The results indicate that commands appear in the top-2 positions for 82-91% of bug reports; the correct data input appears in the top-2 positions for 83-90% of the bug reports; the correct configuration option appears in the top-5% returned results for 71-85% of the reports. Additionally, the workload is identified with 56-80% accuracy. Compared to the baseline approaches, the success rate is higher in each category over all programs.

Where the MAP scores are concerned, PerfLearner is more effective than the baseline techniques over all three types of frame elements across all subject programs. The improvements range from 14% to 40%. These results suggest that *heuristics used by PerfLearner is effective in boosting accuracy.*

Table 4: RQ2: Performance Test Frame Generation

Application	*# of Const.*	*PerfLearner*		*CT*	
		Space	*Count Avg.*	*Space*	*Count Avg.*
Apache	45	445K	2662	64M	10K
MySQL	12	1.4M	1831	2.9B	10K
Firefox	25	443K	7659	3.9B	10K

# of Const.=the number of constraints. Space=the number of total configurations w.r.t. constraints and the default weight. Count Avg.=the average number of test frames generated by the test method before reaching the ground truth.

Table 5: RQ3: Performance Testing with GA

Application	*Bug ID*	*Effectiveness*		*Efficiency*			
		GA	GA_{PL}	GA	Count	GP_{PL}	$Count_{PL}$
Apache	54852	NO	YES	24H	8297	5.2H	1714
	52914	NO	YES	24H	9429	10.1H	3052
	37680	NO	NO	24H	8790	24H	9764
	43081	NO	YES	24H	8822	20.2H	6085
	46749	NO	NO	24H	9037	8.7H	3125
MySQL	21727	NO	YES	24H	8614	14.8H	4097
	44723	NO	YES	24H	9259	11.7H	3015
	74325	NO	YES	24H	8458	11.3H	4055
	15653	NO	YES	24H	7446	7.3H	2910
	26938	NO	NO	24H	9793	24H	9425

RQ2: Effectiveness and Efficiency of Performance Test Frame Generation. Table 4 shows the results of PerfLearner and CT in generating performance test frames. Since CT does not rank test frames, we allow CT to generate test frames among randomly sampled input space for each input category. We limit the number of test frames to 10,000. The threshold number is based on practical considerations as 10,000 tests may take considerable executing time. With the default CT method, all three subjects failed to generate the ground truth test frame before the frame limit threshold. These results suggest that *PerfLearner is more cost-effective at generating performance test frames than the traditional combinatorial testing approach.* Figure 8 shows the distribution of test frames generated in each subject for both PerfLearner (PL) and CT. Firefox has the worst performance of all, this is largely due to Firefox bugs require multiple steps to trigger. Firefox also has the largest number of commands and lowest command extraction accuracy. As a result, the ranking of test frames does not work as effectively as the other two subjects.

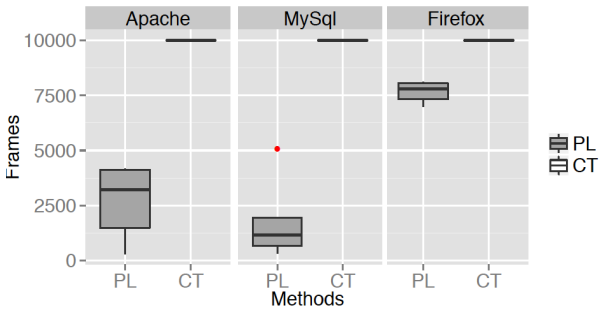

Figure 8: Test frame generation

RQ3: Enhancing Performance Bug Detection. Table 5 shows the results of GA and GP_{PL} (GA enhanced with PerfLearner). GA failed to detect all 10 performance bugs. Like other test case generation techniques, the genetic algorithm for generating input values is applied only after a test frame is selected. However, without knowing which frame element is more likely to cause a performance bug, a random method is used to allow frame elements in each input category to have an equal chance to be selected. As a result, many low-quality test frames are generated. The ground truth test frame often fails to be generated within the time limit (24 hours).

Our results show that the GP_{PL} approach can detect 7 out of 10 performance bugs within an average of 10.9 hours. These results suggest that *PerfLearner can potentially enhance existing performance testing tools.* For the three bugs GP_{PL} failed to detect: 1) Apache bug#37680 requires two entries of the "Listen" option. When selecting configuration options, we do not allow duplications of configuration option since multiple appearances of the same option normally overwrites one another. 2) Apache bug#46749 executes a test frame (server graceful stop) that causes a long response time. This test frame is considered to trigger a performance bug, however, the ground truth test frame of this bug is related to cache utilization. This is the only false positive case appeared in our experiment. 3) For MySQL bug#26938, the "profile" command is required to trigger this bug. However, none of the bug reports used to generate test frames includes the command "profile". We conjecture false negative cases can be reduced as more bug reports are used for mining test frames.

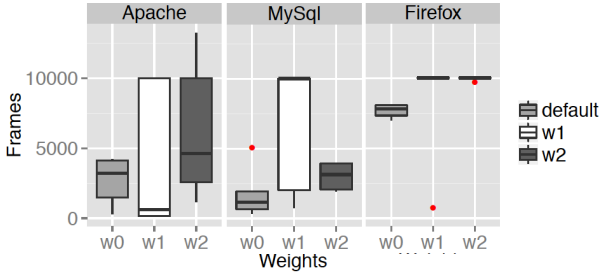

Figure 9: Weight sensitivity analysis

7 DISCUSSION

Sensitivity of Strength. By default, PerfLearner uses strengths $\{ n_c = 2, n_o = 5\%, n_d = 2, n_w = 2, t = 2 \}$. The selected values are based on the empirical study that achieves best test frame element extracting results. To understand the influence of selecting different sets of strengths, we evaluate PerfLearner on two other sets of strengths: w1=$\{ n_c = 1, n_o = 2\%, n_d = 1, n_w = 1$, and $t = 1 \}$ and w2= $\{ n_c = 3, n_o = 10\%, n_d = 3, n_w = 3$, and $t = 3 \}$. Figure 9 reports the results of test frame generation using the three sets of strengths on the test set (10 bug reports) for each of the three subjects. The vertical axis indicates the number of frames generated before reaching the ground truth. The results indicate that, in general, default strengths outperform the other two sets. In Apache, w1 outperforms the default strengths in terms of the average frames generated, but w1 exhibits a larger standard deviation. The weight sensitivity analysis implies that the strengths should not be set too low or too high. Low strength values may cause PerfLearner to miss certain relevant frames, whereas high strength values may result in generating too many performance test frames and thus reduce the efficiency of PerfLearner.

Threats to Validity. The primary threat to the external validity of this study involves the representativeness of our subjects and bug reports. We do reduce this threat to some extent by using several varieties of well studied open source projects and bug tracking

systems for our study. Combining keyword search and manual inspection is an effective technique to identify bugs of a specific type from a large pool of generic bugs and has been used successfully in prior studies [28, 33, 54]. We cannot claim that our results can be generalized to all systems of all domains though. The primary threat to the internal validity involves the manual inspection to identify the ground truth test frame from a bug report. To minimize the risk of incorrect results given by manual inspection, the analysis process was done independently by two trained inspectors.

Limitations. The textual quality of a bug report has substantial impact on the effectiveness of the proposed approach. For example, a bug report may not use the standard names of the frame elements. This can be addressed by integrating advanced NLP techniques, such as Word2Vec [51]. The incompleteness of bug reports is also a major obstacle for PerfLearner to work well, like for many bug report analysis techniques. One strategy is to filter out bug reports containing missing information using an automated approach [9] and apply PerfLearner only to complete bug reports to improve accuracy. Other classification techniques can be integrated with PerfLearner as well, such as detecting reproducible [15] and duplicate [26] bug reports.

PerfLearner takes only labeled performance bug reports. One extension point is to build a prediction model that can automatically predict whether a new bug report is related to performance or not. There has been some research on using text mining to classify bug reports [17, 47, 49], which can be easily tuned to handle performance bug reports. In addition, when a performance bug requires a specific system state (e.g., networking events) to be triggered, the current approach cannot extract such information. For example, a state may be associated with the topology of the target system (e.g., the firewall setup may negatively affect the performance of a system). Nevertheless, we believe PerfLearner can be extended to handle system-level triggering events by defining additional frame databases and linguistic patterns.

8 RELATED WORK

There has been a great deal of research on analyzing, detecting, and fixing performance bugs [7, 28, 29, 34, 36]. Burnim et al. [7] designed a technique to generate worse-case inputs (larger input sizes) to find performance bugs. As discussed in Section 1, these techniques often rely on initial test cases and do not address the challenges of finding the right combination of input parameters to create effective initial test cases. As our empirical study shows, workload only helps to trigger some but not all performance bugs. Although PerfLearner also takes workload into consideration, it focuses more on the combination of elements to be used in the test frame. Our method is orthogonal to the test case generation tools, as our experiment shows, PerfLearner can be integrated into existing performance testing techniques to improve the effectiveness and efficiency of bug detection.

A great body of work has been conducted on applying combinatorial testing (CT) to address the problem of large input space in complex and configurable systems [13, 32, 52, 57]. CT systematically samples the input space and tests only the selected input parameters combinations. Zhang et al. [57] proposed a method to optimize combinatorial testing to generate test cases to find a balanced point of coverage without pressuring on achieving the maximum coverage. Dumlu et al. [13] proposed a feedback-driven approach to detect and avoid masking effect resulted from CT. These techniques focus on sampling combinations from the entire input space. Therefore, it is often inevitable to result in a large sampling space. In the contrast, PerfLearner detects and uses only the error-prone commands and input parameters from the historical bug reports. Empirical results show that our approach can significantly reduce the sampling space when generating test frames for performance bugs.

There has been considerable work on using natural language and information retrieval techniques to improve code documentation and understanding [9, 14, 24, 25] and to create code traceability links [1, 12, 38]. While our work applies some of these same basic techniques, such as tokenization, lemmatization, vector space model with term frequency-inverse document frequency weighting [4], the prior work has not applied these techniques to performance bug reports and has not considered or extracted input parameters to generate test frames.

There has been a large body of work that demonstrates the need for configuration-aware testing techniques and proposes methods to sample and prioritize the configuration space [27, 31, 42, 43, 53] to reduce the cost of testing. For example, Jamshidi et al. [27] conduct an empirical study to evaluate the feasibility of applying the transfer learning technique to reduce the dimensionality of the configuration space when constructing performance models. Nair et al. [31] use inexpensive and inaccurate models to find optimal configurations with less cost compared to the state-of-the-art sampling techniques. Unlike the above technique, our approach focuses on creating test frames to aim performance testing for finding performance bugs instead of performance modeling.

9 CONCLUSIONS

Performance bugs are difficult to expose because they often manifest under special input conditions and system configurations. In this paper, we studied 300 real-world performance bugs from three popular open source projects. Our findings indicate that combinations of input parameters, especially configurations, can play an important role in exposing performance bugs. Guided by these findings, we designed PerfLearner, an automated approach to extract test frame elements, and to generate test frames for performance testing. We evaluated PerfLearner on 300 bug reports and the results show that PerfLearner extracts test frame elements with high accuracy. PerfLearner is also effective in generating performance-bug-triggering test frames. Our evaluation on combining PerfLearner with GA-Prof to detect real-world performance bugs indicates that PerfLearner can enhance existing performance testing tools for generating test cases and detecting performance bugs. For reproducibility and further research, PerfLearner and all the data from the experiments are publicly available at https://github.com/xha225/PerfLearner.

ACKNOWLEDGMENTS

This research is supported in part by the NSF grant CCF-1652149.

REFERENCES

[1] N. Ali, W. Wu, G. Antoniol, M. Di Penta, Y. G. Guéhéneuc, and J. H. Hayes. Moms: Multi-objective miniaturization of software. In *International Conference on Software Maintenance*, pages 153–162, 2011.

[2] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 1–10. IEEE, 2011.

[3] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, pages 307–320, 2012.

[4] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[5] Beautiful soup, 2017. https://www.crummy.com/software/BeautifulSoup/.

[6] Bugzilla keyword descriptions, 2016. https://bugzilla.mozilla.org/describekeywords.cgi.

[7] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. In *Proceedings of the International Conference on Software Engineering*, pages 463–473, 2009.

[8] Andrea Calvagna and Angelo Gargantini. A formal logic approach to constrained combinatorial testing. *Journal of Automated Reasoning*, 45(4):331–358, 2010.

[9] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 396–407, 2017.

[10] Daniel J Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. Perfscope: Practical online server performance bug inference in production cloud computing infrastructures. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[11] Andrea Di Sorbo, Sebastiano Panichella, Corrado A Visaggio, Massimiliano Di Penta, Gerardo Canfora, and Harald C Gall. Development emails content analyzer: Intention mining in developer discussions (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 12–23, 2015.

[12] Bogdan Dit, Latifa Guerrouj, Denys Poshyvanyk, and Giuliano Antoniol. Can better identifier splitting techniques help feature location? In *International Conference on Program Comprehension*, pages 11–20, 2011.

[13] Emine Dumlu, Cemal Yilmaz, Myra B. Cohen, and Adam Porter. Feedback driven adaptive combinatorial testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 243–253, 2011.

[14] Eric Enslen, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *International Working Conference on Mining Software Repositories*, pages 71–80, 2009.

[15] Mona Erfani Joorabchi, Mehdi Mirzaaghaei, and Ali Mesbah. Works for me! characterizing non-reproducible bug reports. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 62–71, 2014.

[16] Christos Faloutsos and Douglas W Oard. A survey of information retrieval and filtering methods. Technical report, 1998.

[17] Michael Gegick, Pete Rotella, and Tao Xie. Identifying security bug reports via text mining: An industrial case study. In *Mining software repositories (MSR), 2010 7th IEEE working conference on*, pages 11–20, 2010.

[18] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 156–166. IEEE, 2012.

[19] Mats Grindal, Jeff Offutt, and Sten F Andler. Combination testing strategies: a survey. *Software Testing, Verification and Reliability*, 15(3):167–199, 2005.

[20] Robert J Grissom and John J Kim. *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers, 2005.

[21] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the International Conference on Software Engineering*, pages 145–155, 2012.

[22] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE*, pages 145–155, 2012.

[23] Xue Han and Tingting 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*, page 23. ACM, 2016.

[24] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. Amap: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *International Working Conference on Mining Software Repositories*, pages 79–88, 2008.

[25] Matthew J. Howard, Samir Gupta, Lori Pollock, and K. Vijay-Shanker. Automatically mining software-based, semantically-similar words from comment-code mappings. In *International Working Conference on Mining Software Repositories*, pages 377–386, 2013.

[26] Nicholas Jalbert and Westley Weimer. Automated duplicate detection for bug tracking systems. In *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC*, pages 52–61, 2008.

[27] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 497–508. IEEE Press, 2017.

[28] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 77–88, 2012.

[29] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: Performance bug detection in the wild. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, pages 155–170, 2011.

[30] Andrew J Ko, Brad A Myers, and Duen Horng Chau. A linguistic analysis of how people describe software problems. In *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*, pages 127–134, 2006.

[31] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. Using bad learners to find good configurations. *arXiv preprint arXiv:1702.05701*, 2017.

[32] Changhai Nie, Huayao Wu, Xintao Niu, Fei-Ching Kuo, Hareton Leung, and Charles J Colbourn. Combinatorial testing, random testing, and adaptive random testing for detecting interaction triggered failures. *Information and Software Technology*, 62:198–213, 2015.

[33] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *Proceedings of the International Conference on Mining Software Repositories*, pages 237–246, 2013.

[34] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the International Conference on Software Engineering*, pages 562–571, 2013.

[35] Natural language toolkit, 2017. http://www.nltk.org/.

[36] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 369–378, 2015.

[37] Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, 31(6):676–686, 1988.

[38] Annibale Panichella, Collin McMillan, Evan Moritz, Davide Palmieri, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. When and how using structural information to improve ir-based traceability recovery. In *European Conference on Software Maintenance and Reengineering*, pages 199–208, 2013.

[39] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 459–468, 2010.

[40] Jay M Ponte and W Bruce Croft. A language modeling approach to information retrieval. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 275–281, 1998.

[41] Michael Pradel, Markus Huggler, and Thomas R. Gross. Performance regression testing of concurrent classes. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 13–25, 2014.

[42] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *ISSTA*, pages 75–86, 2008.

[43] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. Cost-efficient sampling for performance prediction of configurable systems (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 342–352. IEEE, 2015.

[44] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press, 2008.

[45] scikit-learn, 2017. http://scikit-learn.org/stable/.

[46] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Automating performance bottleneck detection using search-based application profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 270–281, 2015.

[47] Wei Wen, Tingting Yu, and Jane Huffman Hayes. Colua: Automatically predicting configuration bug reports and extracting configuration options. In *EEE 27th International Symposium on Software Reliability Engineering*, pages 150–161, 2016.

[48] Alexander Wert, Jens Happe, and Lucia Happe. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In *Proceedings of the International Conference on Software Engineering*, pages 552–561, 2013.

[49] X. Xia, D. Lo, W. Qiu, X. Wang, and B. Zhou. Automated configuration bug report prediction using text mining. In *Computer Software and Applications Conference*, pages 107–116, 2014.

[50] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 90–100, 2013.

[51] Chao Xing, Dong Wang, Chao Liu, and Yiye Lin. Normalized word embedding and orthogonal transform for bilingual word translation. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1006–1011, 2015.

[52] C. Yilmaz. Test case-aware combinatorial interaction testing. *IEEE Transactions on Software Engineering*, 39:684–706, 2013.

[53] Cemal Yilmaz, Myra B. Cohen, and Adam Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *TSE*, 29(4), July 2004.

[54] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *SOSP*, pages 159–172, 2011.

[55] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. A qualitative study on performance bugs. In *MSR*, pages 199–208, 2012.

[56] Shahed Zaman, Bram Adams, and Ahmed E Hassan. A qualitative study on performance bugs. In *IEEE Working Conference on Mining Software Repositories (MSR)*, pages 199–208, 2012.

[57] Zhiqiang Zhang, Jun Yan, Yong Zhao, and Jian Zhang. Generating combinatorial test suite using combinatorial optimization. *Journal of Systems and Software*, 98:191–207, 2014.