CP-Detector: Using Configuration-related Performance Properties to Expose Performance Bugs

Haochen He
National University of Defense Technology, China
hehaochen13@nudt.edu.cn

Zhouyang Jia*
National University of Defense Technology, China
jiazhouyang@nudt.edu.cn

Shanshan Li
National University of Defense Technology, China
shanshanli@nudt.edu.cn

Erci Xu
National University of Defense Technology, China
xuerci@nudt.edu.cn

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

Yue Yu
National University of Defense Technology, China
yuyue@nudt.edu.cn

Ji Wang
National University of Defense Technology, China
wj@nudt.edu.cn

Xiangke Liao
National University of Defense Technology, China
xkliao@nudt.edu.cn

ABSTRACT

Performance bugs are often hard to detect due to their non fail-stop symptoms. Existing debugging techniques can only detect performance bugs with known patterns (e.g., inefficient loops). The key reason behind this incapability is the lack of a general test oracle. Here, we argue that the performance (e.g., throughput, latency, execution time) expectation of configuration can serve as a strong oracle candidate for performance bug detection. First, prior work shows that most performance bugs are related to configurations. Second, the configuration change reflects common expectation on performance changes. If the actual performance is contrary to the expectation, the related code snippet is likely to be problematic.

In this paper, we first conducted a comprehensive study on 173 real-world configuration-related performance bugs (CPBugs) from 12 representative software systems. We then derived seven configuration-related performance properties, which can serve as the test oracle in performance testing. Guided by the study, we designed and evaluated an automated performance testing framework, CP-Detector, for detecting real-world configuration-related performance bugs. CP-Detector was evaluated on 12 open-source projects. The results showed that it detected 43 out of 61 existing bugs and reported 13 new bugs.

CCS CONCEPTS

• **Software and its engineering → Software performance**.

KEYWORDS

Performance bug detection, Software configuration, Performance property

ACM Reference Format:
Haochen He, Zhouyang Jia, Shanshan Li, Erci Xu, Tingting Yu, Yue Yu, Ji Wang, and Xiangke Liao. 2020. CP-Detector: Using Configuration-related Performance Properties to Expose Performance Bugs. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416531

1 INTRODUCTION

Modern software systems are increasingly configurable and thus becoming more adaptive to various scenarios. However, as configuration can (in)directly affect performance (e.g., altering resource allocation), performance bugs' occurrences are also surging. A recent study suggests more than half (59%) of performance bugs are due to incorrect handling of configurations [33]. In this paper, we term these performance bugs as Configuration-handling Performance Bugs (CPBug). Note that a CPBug is different from a misconfiguration where the former focuses on incorrect configuration handling in source code and the latter revolves around user-induced configuration errors.

Figure 1 illustrates a real-world CPBug [17] related to the configuration option `sort_buffer_size` in MySQL, as well as the failure symptom, root cause, and fix method. This option is used to alter the buffer size for sorted results (triggered by `GROUP BY z DESC`). Ideally, a larger buffer should improve the sorting performance, since MySQL can cache more results. However, for the SQL query in the upper-right corner of Figure 1, users actually suffer from up to 4.2× slowdown instead of benefiting from larger buffers (i.e., increase from `2M` to `8M`). The root cause is redundant memory allocation. Specifically, the buffer is allocated before each sub-query (Line 1), and freed immediately at the end of the sub-query (Line 5). The fix method is to allocate the buffer at the first sub-query (Line 2-3) and reuse it at all subsequent sub-queries (Line 6-7). This bug serves as

*Co-first author

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416531

MySQL Bug #21727

Reduced patch in sql/filesort.cc

```
// This code snippet is executed in each sub-SELECT and the first 'if' statement only get satisfied in the first sub-SELECT
1 - sort_keys = my_malloc
      (sort_buffer_size, …);
2 + if(!table_sort.sort_keys)
3 +   sort_keys = my_malloc
        (sort_buffer_size, …);
4   ...
5 - x_free(sort_keys);
6 + if(!subselect)
7 +   x_free(sort_keys);
```

Description:

```
"SELECT a, b, (SELECT x FROM t2 WHERE y=b ORDER BY z DESC LIMIT 1) c FROM t1"
```

will get **4.2x** slower as increasing `sort_buffer_size` from 2M to 8M.

How to fix:
reuse the sort buffer for each sub-SELECT.

Figure 1: Redundant memory allocation in MySQL. *This bug happens because MySQL allocates memory in each sub-*SELECT.

a representative example of performance bugs caused by incorrect configuration handling. This is different from a user misconfiguration, where users may suffer from performance degradations when setting up a buffer size larger than the memory limitation.

There has been much research on detecting performance bugs. Some research has proposed profiling-based techniques, which aims to detect performance bottlenecks that can cause significant slowdowns [28, 55, 67]. However, such slowdowns can be self-induced due to necessarily intensive computation. The lack of test oracles makes it difficult to decide if a slowdown indicates a performance bug [46]. Alternatively, some research has proposed to use inefficient code patterns [46, 59, 61, 65, 66] as test oracles. For example, Toddler [46] identifies loops with inefficiency memory-access patterns which imply potential performance bugs. These works are hard to detect CPBugs, which may or may not contain the patterns. There has also been some research focuses on the relationship between configuration and performance, including configuration-performance modeling [35, 56], and configuration-based performance tuning [43, 47]. The former aims to predict performance for given configurations, while the latter studies the tendency of performance changes when tuning configurations, and finds the optimal configurations with regard to the performance. The performance tendency can be obtained by sampling configuration values and fitting corresponding performances. This is different from a performance bug, which may be triggered by a certain value.

In this paper, we propose CP-Detector[1], an automated testing framework to detect CPBugs. The key insight of CP-Detector is that when tuning a configuration option, a mismatch between the expected and actual performance changes usually indicates a CPBug. For instance, users would expect performance improvement when allocating a larger buffer. If, however, the actual performance drops, the mismatch between the expected and actual performance changes may indicate a CPBug. This kind of performance properties can be used as test oracles to expose CPBugs. For example, the property in Figure 1 can be described as "increasing resource-related configuration options should improve the performance".

To understand CPBugs and guide the design of CP-Detector, we first conducted an empirical study on 173 real-world CPBugs from 12 software systems. We found that 150 (86.7%) CPBugs can be exposed by detecting mismatches between the expected and actual performance changes when tuning configuration options. We further studied the configuration options involved in the 150 CPBugs, and summarized seven performance properties from these options. Each property can be formalized as a three-tuple: <Type, Direction, Expectation>, indicating when tuning configuration options of a given Type (e.g., resource) according to the Direction (e.g., increasing), the software should have the Expectation (e.g., improving) performance change.

CP-Detector contains two major steps to automate the process of exposing CPBugs. Given a configuration option: 1) CP-Detector suggests the performance properties that the option should hold by learning configuration documentation. The challenge is to understand the natural languages and build relationships between the languages and the properties. To address this, CP-Detector applies natural language processing (NLP) and association rule mining (ARM) techniques to automatically derive the properties from the documentation. 2) CP-Detector samples value pairs of the option and test if one value pair can expose a CPBug. This is challenging when the option is numeric, since the value range may be extremely large; thus, it is hard to test all pairs. In this regard, we conduct an empirical study on numeric options to investigate the value ranges that can trigger CPBugs, and propose a heuristic sampling strategy to reduce the sampling space.

To evaluate the effectiveness of CP-Detector in detecting CP-Bugs, we reproduced 38 known CPBugs from the 173 CPBugs in our study, and to avoid over-fitting, we also reproduced 23 known CPBugs not included in our study (61 CPBugs in total so-far all we can reproduce), and evaluated CP-Detector on all 61 bugs. The results show that CP-Detector can successfully expose 43 bugs using the suggested performance properties. The rest cases escape mainly because the information provided by configuration manuals is limited, so CP-Detector can not make the right suggestion on the performance properties. On the other hand, Toddler [46], one of the most effective bug detection tools among existing works, detected 6 out of the 61 CPBugs. In the meantime, CP-Detector detected 13 unknown CPBugs on the same set of software projects. We have reported the 13 bugs to developers, and nine of them have already been confirmed or fixed at the time of writing.

In summary, this paper makes the following contributions:

- We conducted an empirical study on 173 real-world CPBugs from 12 software systems. The findings are used to guide the design of test inputs and test oracles.
- We designed and implemented CP-Detector, an automated framework to detect CPBugs. CP-Detector can automatically suggest performance properties for configuration options and generate configuration values to expose CPBugs.
- We evaluated CP-Detector on 12 software systems. The results show that CP-Detector detected 43 out of 61 known CPBugs and 13 unknown bugs. Ten of the unknown bugs have been confirmed or fixed by developers.

2 UNDERSTANDING CPBUGS

In this section, we take an in-depth look into the CPBugs through an empirical study. We manually collected real-world CPBugs from bug tracking systems, mailing lists, and fix commits of 12 software

[1]The annotated bug data set and tool can be found in our publicly available repository: https://github.com/TimHe95/CP-Detector

Table 1: Software systems and CPBugs used in the studies.

Software	# KW†	# CPBugs	Software	# KW	# CPBugs
MySQL	398	35	Httpd	291	16
MariaDB	451	25	H2	17	1
MongoDB	425	26	Squid	4	1
PostgreSQL	14	5	Tomcat	25	3
RocksDB	16	3	GCC	618	39
Derby	18	3	Clang	192	16

† ***KW**: The CPBug candidates identified using performance-related keywords.*

systems. In these CPBugs, we found a majority of configuration options have expected performance changes when being tuned. We also studied the performance properties of the configuration options involved in these CPBugs, and how the option values triggered the CPBugs. Our findings are used to guide the design of CP-Detector.

Studied Subjects. Table 1 shows the 12 software systems used in our study. These projects cover a variety of domains, including database, web server, and compiler. These projects are performance-critical, highly-configurable (e.g., Httpd has more than 1,100 configuration options), and widely deployed in the field. Therefore, CPBugs from these projects are likely to be rich in numbers and severe in consequences [36]. Also, these projects are open-source and well maintained by the community. This allows us to not only check the buggy code snippets but also gather related details based on the developers' discussions.

CPBug Collection. We collected CPBugs from three sources: bug tracking systems (e.g., JIRA, Bugzilla), mailing lists, and fix commits. We started by searching the above sources using heuristic keywords (e.g., "slow", "long time", "performance"). This process identified 2,469 candidates. We manually analyzed these candidates, each of which is deemed as a CPBug if tuning a configuration option would cause a performance bug. This process yielded 173 (columns 3 and 6 in Table 1) CPBugs.

2.1 Prevalence of Unexpected Performance Changes in CPBugs

Each CPBug has one or multiple triggering configuration option(s). For each option, we use expected performance changes of two values of the configuration option as performance properties, and expose CPBugs by detecting violations of the properties. To evaluate to what extent the property-based approach can expose CPBugs, we study the prevalence of unexpected performance changes when tuning configuration options of the CPBugs.

To achieve this, we manually studied all the 173 CPBugs collected above, and found 150 (86.7%) of them have unexpected performance changes when tuning configuration options. This result indicates the performance expectation can serve as an effective oracle for exposing CPBugs. The remaining 23 (13.3%) CPBugs cannot be exposed mainly because the configuration options have inconsistent expectations. For example, `innodb_fill_factor` defines the percentage of space that is filled during a sorted index build, with the remaining space reserved for future index growth. In production, it should be carefully tuned according to the workloads and hardware. While in MySQL-74325 [19], when setting it to 100, indexed `UPDATE`s get 3.8× slower because page split has to be performed for every `UPDATE` due to no empty space for the changed index. Developer fixes this by preserving 1/16 of the space for any of its value. Increasing or decreasing `innodb_fill_factor` does not necessarily have common expected performance change.

> **Finding 1**: A majority (87.6%) of the CPBugs have unexpected performance changes when tuning configuration options. This result indicates the performance expectation can serve as an effective oracle for exposing CPBugs.

2.2 Performance Properties in CPBugs

We studied the 150 CPBugs to understand the performance properties of the bug-introducing configuration options. The findings can be used to guide the design of CP-Detector for automatically extracting the properties to detect CPBugs for any target software.

We manually studied the semantics of the configuration options involved in the CPBugs and summarized five semantic types, i.e., *optimization on-off*, *non-functional tradeoff*, *resource allocation*, *functionality on-off*, and *non-influence option*. Each configuration type has two tuning directions, e.g., turning on and turning off, or increasing and decreasing. Then, we analyzed the CPBugs and found both *non-functional tradeoff* and *functionality on-off* options triggered CPBugs in two directions, while other types only triggered CPBugs in one direction. After that, we analyzed the expected and actual performance changes from the bug descriptions for each direction of each type.

The result is shown in Table 2, each performance property is associated with its configuration type (Column 2), tuning direction (Column 3-4), and expected performance change (Column 5). These three parts correspond to the factors of the three-tuple <`Type`, `Direction`, `Expectation`> defined in § 1. Besides the properties, Column 6 shows the actual performance change of CPBugs, and Column 7 shows the numbers of CPBugs that break each property. For example, the first configuration type is *optimization on-off*, when tuning an optimization option from OFF to ON, it means turning on an optimization strategy, and the performance is expected to be enhanced. If, however, the actual performance drops, there is a potential CPBug. In our dataset, 18 CPBugs violate this property. Below, we provide details of the properties in each type of configuration options.

Optimization on-off. In this category, a configuration option is used to control an optimization strategy. Specifically, when the optimization is turned on, the application's performance is expected to be improved. Consider a CPBug example, MySQL-67432 [18]. For SQL queries like "`SELECT * FROM t WHERE c1<100 AND (c2<100 OR c3<100)`", MySQL can speed up at least 10% by enabling the optimization strategy `index_merge=ON`, which can merge the indexes of different columns. However, when the queries end with "`ORDER BY c1 LIMIT 10`", the performance degrades by 10× with the optimization turned on. This is because the whole indexes of the columns are merged (i.e., two index range scans, a merge, and a "join"), whereas only the top 10 rows are required. Developers fix this bug by changing the triggering conditions of the optimization strategy controlled by `index_merge`.

Table 2: Performance Properties in CPBugs.

PP-ID	Performance Properties (PP)				CPBugs	
	Configuration Option Type	Tuning Direction		Expected Performance Change	Actual Performance Change	# CPBugs (Pct.)
		Source Value	Target Value			
PP-1	Optimization on-off	OFF	ON	Rise (↑)	Drop (↓)	18 (12.0%)
PP-2	Non-functional tradeoff	Anti-performance†	Pro-performance‡	Rise (↑)	Drop (↓)	35 (23.3%)
PP-3	Non-functional tradeoff	Pro-performance	Anti-performance	Drop (↓)	More-than-expected drop (↓↓)	31 (20.7%)
PP-4	Resource allocation	Small	Large	Rise (↑)	Drop (↓)	24 (16.0%)
PP-5	Functionality on-off	ON	OFF	Rise (↑)	Drop (↓)	6 (4%)
PP-6	Functionality on-off	OFF	ON	Drop (↓)	More-than-expected drop (↓↓)	24 (16.0%)
PP-7	Non-influence option	Random	Random	Keep (–)	Drop (↓)	12 (8.0%)

‡ A value that implies better performance at the cost of lower security, consistency, integrity, and etc. † Opposite to Pro-performance.

Non-functional tradeoff. In this category, a configuration option is used to achieve the balance between performance and other non-functional requirements of the program. We found two properties regarding this configuration type. In PP-2, tuning the configuration option value to relax a requirement is expected to achieve a performance gain. For example, a database can achieve higher performance by relaxing the ACID properties (a set of properties to ensure correctness and consistency). In the CPBug MySQL-77094 [20], `innodb_flush_log_at_trx_commit` controls the ACID property. When setting to 2 (relaxed ACID), however, the performance of the OLTP update benchmark is 10% lower than the performance when setting to 1 (full ACID). The root cause is that two logging functions (i.e., `commit` and `log_write`) both use the `log_sys->mutex` lock to write to the same buffer. This, in return, causes extra lock contentions that hurt the performance. The fix is simply to use two independent buffers and remove this lock. In this case, switching an *anti-perf* value (i.e., 1) to a *pro-perf* value (i.e., 2) leads to performance drops, causing a CPBug.

As for PP-3, tuning the configuration option value to enable a requirement is expected to have a performance loss. If, however, the actual loss is more-than-expected, it still indicates a CPBug. For example, GCC uses `O0, O1, O2, O3` options to control the balance between compilation time and binary execution efficiency. A higher `O` level indicates more compilation time. But in GCC-17520 [11], switching from `O0` to `O2` increases the compilation time from less than 1 second to 1 minute. The cost is more-than-expected. (we describe the thresholds to measure "more-than-expected" in § 3.2.3) This is caused by a sub-optimal algorithm induced by a process called "branch prediction" in the `O2` level. This bug is fixed by adding an early drop condition in the algorithm. It makes the compilation time reduce to less than 1 second with `O2`.

Resource allocation. In this category, a configuration option is used to control resource usages (e.g., RAM, CPU cores). Allocating more resources generally results in better performance. Take the bug in Figure 1 as an example, increasing the memory allocation to the sort buffer is expected to speed up the `"SELECT ORDER BY"` operations. However, the performance of MySQL degrades by 4×, causing user's complaints[2].

[2] *"This is pretty much the opposite of any other case I have seen. In fact, to make the query perform faster, you need to set it to the smallest value"* – The user of MySQL who reported this bug.

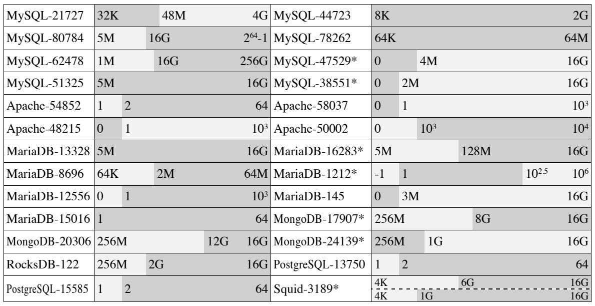
Functionality on-off. In this category, a configuration option is used to control a non-performance functionality but indirectly influences the system's performance. We have two properties in this category. PP-5 suggests that when an option disables a functionality, the system's performance usually increases. For example, in MariaDB, turning on `log_slave_updates` logs the updates of a slave received from a master during replication. However, as described in MariaDB-5802 [13], disabling this functionality increases the time for the slave to catch up during replication by 50%. This happens because the handling code of `log_slave_updates=on` gets optimized as the software evolves, while that of `log_slave_updates=off` escapes. Developers fixed this bug by applying the same optimization to both situations.

While PP-6 suggests that when a configuration option enables a functionality, the induced performance overhead is allowed but should be within a limit. For example, in web servers, `VirtualHost` is used to enable multiple virtual hosts in the server. It is reasonable that setting more virtual hosts causes longer start up time. However, as described in Httpd-50002 [9], the single server startup time grows super-linearly as the number of virtual hosts increases, and it would take 50× more time with 10,000 virtual hosts than with the default setting. The root cause of this bug is Httpd uses a sub-optimal way to parse the `VirtualHost` directives in the configuration file. Developers fixed the bug by optimizing the data structure, which decreases the startup time from several minutes to merely 6 seconds.

Non-influence option. In this category, a configuration option is not supposed to influence the system's performance, i.e., the performance is expected to remain the same after tuning the option. For example, users can choose `proxy_http` (default) or ajp (set `mod_jk`) as the connector between Httpd and Tomcat. The two connectors are expected to have similar file transfer speeds. But in Httpd-33605 [7], in the AIX operating system, the file transfer speed degrades from ~8MB/s to only 2KB/s when switching from default `proxy_http` connector to ajp connector under the same network condition. This is because the implementation of AIX socket buffer is conflicted with the encapsulation of the socket implemented by the `ajp13` connector. Hence, simply removing the encapsulation of the socket solved the bug.

2.3 Triggering Conditions of CPBugs

To test the software against a property of a configuration option, we need to sample at least two values (referred to as V_{src} and V_{tar})



* This bug requires specific workload.

Figure 2: Value ranges that can trigger CPBugs (dark gray).

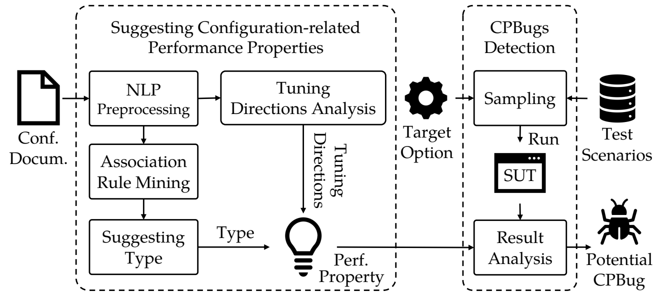


Figure 3: Overview of CP-Detector.

from that configuration option, as shown in the Column 3-4 of Table 2. For numerical options (e.g., *resource allocation* options), it is difficult to enumerate all value pairs, since the value ranges can be extremely large. In this regard, we conducted a study to investigate the value ranges of numeric options that can trigger CPBugs.

In our study, 24.7% (37/150) of the CPBugs are exposed by numerical configuration options. We successfully obtained the value ranges of 26 CPBugs (including 27 configuration options) out of the 37 CPBugs by exhaustively and manually reproducing the bugs with different option values. Note that some upper bounds can be 2^{64}, which is too large and may cause the performance "falls the cliff" [27] due to resource limitations. This behavior is difficult to be distinguished from actual CPBugs. Therefore, we limited the upper bounds to our experimental resource limitations (e.g., CPU cores, RAM).

As illustrated in Figure 2, the dark gray areas show the value ranges where CPBugs can be exposed. Among 26 (96.3%) out of the 27 numeric options, the ranges contain the minimum or maximum value of the option. For example, the CPBug MySQL-21727 can be triggered when the configuration option `sort_buffer_size` is between 32K and 48M. This triggering range contains the minimum value (i.e., 32K) of the overall acceptable range of the option (i.e., [32K, 4G]). The reason behind this is that numerical options usually affect the program control flow much less than data flow. As a result, changing numerical options tends to exaggerate or alleviate an existing bug (if any) in the current program path, instead of triggering a new bug in a different program path.

> **Finding 2**: 96.3% of numeric configuration options can trigger CPBugs when being set to the minimum or maximum values. The sampling numbers can be significantly reduced by fixing V_{src} to the min values or fixing V_{tar} to the max values.

3 CP-DETECTOR DESIGN

Figure 3 shows the overview of CP-Detector, which takes configuration documentation and the Software Under Test (SUT) as inputs and outputs CPBugs. CP-Detector contains two major steps.

First, CP-Detector infers performance properties for each configuration option, i.e., <`Type`, `Direction`, `Expectation`>. Specifically, CP-Detector trains the configuration documentation into a set of rules by natural language processing (NLP) and association rule mining (ARM). These rules are used to distinguish different configuration types. Therefore, given a new configuration option, CP-Detector can label it with a particular configuration type. Next, CP-Detector infers the tuning direction for each labeled configuration option from the property specification (Table 2) and the configuration documentation. As a result, given a value pair of the new configuration option, CP-Detector can determine their tuning direction. The expected performance change is obtained according to the configuration type and the tuning direction.

Second, CP-Detector detects CPBugs by checking whether the outputs of a pair of test executions break the performance properties of the participating configuration option. To achieve this, CP-Detector samples a value pair of the option, serving as two test inputs of the pair of test executions. Besides the participating configuration option, CPBugs may also require other triggering conditions, i.e., workloads, other configuration options, or the running stage of the software (e.g., start, service, shutdown). CP-Detector combines these conditions as test scenarios, and checks the outputs of the execution pair under each scenario. Finally, CP-Detector tests each execution pair under each scenario multiple times, and determines if there is a CPBug based on hypothesis testing [4].

3.1 Suggesting Performance Properties

Since the configuration documentation of an application often describes the names, functions, and usages of configuration options, it is widely-exist, easy to get [63], and a good source to derive configuration-related performance properties. We next describe how to identify the type and the tuning direction for each configuration option.

3.1.1 Identifying Configuration Types. Given a configuration option, CP-Detector can automatically label it with a configuration type. To achieve this, CP-Detector leverages natural language processing and association rule mining to derive a set of classification rules from existing configuration documents. Specifically, we manually analyzed a total of 500 configuration options from 12 applications and assigned each configuration option to one of the five types (Table 2). Next, a set of classification rules are automatically derived from the configuration documents associated with the 500 options, following three steps: 1) pre-processing the documents to normalize words with both syntactic and semantic similarities; 2) mining association rules between word sequences and configuration types, and 3) selecting optimal rules used for labeling new configuration options.

Table 3: Domain specific synonym tags (partial†)

Tag Name	Words in base forms
Resource	memory, buffer, thread, worker, cleaner
Volumn	size, amount, number
PerfPositive	performance, speed, throughput
PerfNegative	latency, CPU time, responses time
OpposePerf	integrity, compression, security, reliability
Self	this option, this directive, <Config Name>

† *The complete tag and word lists can be found in our public repository.*

Normalizing Words. In this step, CP-Detector identifies both syntactic and semantic similarities of different words, and similar words will be regarded as the same one in the following mining process. Each configuration of the dataset is in the form of $\{S_1, S_2, \ldots\} \rightarrow$TYPE, where S_1 and S_2 stand for the first and the second sentences in the configuration description, and TYPE refers to the configuration type. CP-Detector first split the description into sentences sharing the same type: $S_1 \rightarrow$TYPE, $S_2 \rightarrow$TYPE, etc. After that, CP-Detector normalizes the words in each sentence from both syntactic and semantic aspects. In specific, for each word, CP-Detector infers the part-of-speech (POS) tag (e.g., noun, verb) by using spaCy [3]. On the other hand, we manually studied the descriptions of the 500 options, and defined domain-specific synonym tags as shown in Table 3 (traditional methods of identifying synonyms may be hard to deal with words in computer science). We also referenced online resources about domain-specific terminology during the classification [1, 5, 23]. The words will be normalized according to the syntactic and semantic tags. For example, both "size of buffer" and "number of threads" describe resource-related configuration options. CP-Detector normalizes both phrases into the same form: {(NN,Volume), (IN), (NN,Resource)}, where NN and IN are part-of-speech tags, meaning "Noun, singular or mass" and "Preposition or subordinating conjunction"[3]. Volume and Resource are synonym tags in the first column of Table 3.

Mining Association Rules. The goal of this step is to find the word sequences that appear exclusively and frequently in the descriptions of a specific configuration type. We use the design principle of FeatureMine [40], a typical ARM algorithm for classification on sequential data, and implement it by ourselves to achieve our goal. The output of FeatureMine is a set of Class Association Rules (CARs), which are in the form of $\{W_1, W_2, ...\} \rightarrow$TYPE in our case ($W_i$ is the i_{th} word in the sequence). Meanwhile, the algorithm outputs the confidence for each CAR. The confidence is defined as the conditional probability of occurrence of TYPE given $\{W_1, W_2, ...\}$ appears. One important parameter in the algorithm is *min_support*. We assign *min_support* = 3, as used in existing software engineering studies [42, 68], which means $\{W_1, W_2, ...\}$ should appear at least in 3 configurations of type TYPE. Another important parameter is the length limitations of CARs. A short CAR (e.g., 1 or 2) will be less informative; thus we restrict $3 \leq len(\text{CAR}) \leq Len$, where *Len* is a predefined threshold. We will evaluate how to choose *Len* in § 4.4.

[3]The complete part-of-speech tag list and corresponding meanings are available in https://spacy.io/api/annotation#pos-universal

Selecting Optimal Rules. The above mining approach may generate millions of CARs with many of them are repetitive. When a CAR is a sub-sequence of another CAR and the two CARs have the same *support*, CP-Detector rules out the short one, since these two CARs always appear at the same time and the longer one is more informative. This process still leaves tens of thousands of CARs. In this regard, CP-Detector selects a subset of CARs as optimal rules, which are measured by F_{score} – the harmonic mean of 1) the averaged confidences of CARs in the subset, and 2) the proportion of configuration types that can be classified by using the subset of CARs. The CARs are ranked by confidence. Simply choosing the top-n CARs may get a high averaged confidence, but not necessarily have high coverage of configuration options. Instead, CP-Detector randomly samples *Num* CARs for each configuration type. The CARs are weighed by confidences during sampling, therefore, CARs with higher confidences are more likely to be sampled. CP-Detector then calculates the F_{score} of the sampled CARs. The above process will be exhaustively repeated until the current highest F_{score} is the theoretical highest F_{score} with above 99.9% probability, according to the cumulative distribution function (CDF) [2]. Then, CP-Detector selects the subset of CARs with the highest F_{score}. The exhaustive sampling process is a one-time effort; users can directly use the optimal rules. *Num* is a predefined threshold, and we will evaluate how to choose *Num* in § 4.4.

Assigning Configuration Types. With the optimal rules available, CP-Detector defines a voting classifier. Given a configuration description, all rules that match the description will vote for their corresponding TYPE, and the weights are confidences of the rules. CP-Detector suggests the TYPE with the highest weight as the configuration type (when no rules match the description, CP-Detector can not suggest any type). For example, in MySQL, the description of the option innodb_sort_buffer_size is "Specifies the size of sort buffers used to sort data during creation of an InnoDB index". CP-Detector can match the word sub-sequence {size, of, buffers, to, data} using the rule {(NN,Amount), (IN), (NN,Resource), (IN), (NN)} → *Resource*. The complete rules can be found in our public repository.

3.1.2 Identifying Tuning Directions. The tuning direction for a configuration option involves a pair of values (referred to as V_{src} and V_{tar}). For most of the properties (i.e., PP-1, PP-4, PP-5, PP-6, and PP-7), obtaining the value pairs is straightforward as long as the configuration type is known. For example, once a configuration option is labeled with the *Optimization* type, its tuning direction clearly becomes (OFF→ON).

The exceptions are PP-2 and PP-3 of the *Tradeoff* type, because one needs to know tuning V_{src} to V_{tar} is from *Anti-performance* to *Pro-performance* (PP-2) or vice versa (PP-3). To address this, CP-Detector infers the direction of value tuning by analyzing the configuration documentation associated with each configuration option labeled as *Tradeoff*. Specifically, given a configuration option, CP-Detector ranks its values according to their influences on the application's performance. A value is ranked higher if it results in performance improvement. Therefore, for an arbitrary pair of configuration option values, the higher ranked one indicates *Pro-performance* and the lower ranked one is *Anti-performance*.

To do this, similar to § 3.1, CP-Detector extracts performance-related information from documentation. CP-Detector first locates the description of each value by matching its first appearance sentence. Based on the synonym tags in Table 3, CP-Detector then quantifies the degrees to which a configuration option value influences the performance. Specifically, a value will be scored +1 if one of the following sequences in the left side appears in its description:

`Increase` → `PerfPositive`	`Decrease` → `PerfPositive`
`Decrease` → `PerfNegative`	`Increase` → `PerfNegative`
`Decrease` → `OpposePerf`	`Increase` → `OpposePerf`

These sequences explicitly indicate the value can increase the performance. On the contrary, a value will be scored -1 if one of the sequences in the right side appears. For example, in MongoDB, the tradeoff option `compressors` has three values: `snappy` (balanced computation and compression rates), `zlib` (higher compression rates at the cost of more CPU consumption, compared to snappy.), `zstd` (higher compression rates and CPU consumption when compared to zlib). CP-Detector will rank the values as {`snappy`, `zlib`, `zstd`}, since `snappy` has the best performance, while `zstd` has the worse one.

3.2 Exposing CPBugs

CP-Detector generates value pairs <V_{src}, V_{tar}> associated with the tuning direction for the target configuration options (i.e., options labeled by the five configuration types). The performance change (after tuning the target configuration option O_t from V_{src} to V_{tar}) is used to determine if a CPBug is exposed. In addition to O_t, a CPBug may need specific values of other configurations or workloads to be exposed. Therefore, CP-Detector also samples the values of other configuration options and environment parameters (e.g., workloads, stage of program execution) to test O_t under different scenarios. We next describe the process of sampling O_t to generate value pairs, the process sampling test scenarios to test O_t, and results checking.

3.2.1 Sampling Target Configurations Options. Once a configuration option O_t is assigned with one or more specific properties, CP-Detector will generate value pairs, <V_{src}, V_{tar}>, for the tuning direction of each property. According to Table 2, the tuning directions of PP-1, PP-5, and PP-6 involve binary options, so the value pair contains only ON and OFF. The tuning direction of PP-7 is also straightforward so a pair of random values is generated.

The tuning direction of PP-2 and PP-3 exhaustively samples the pairs of enumerated values in terms of their ranking positions, where the higher ranked value is assigned to *Anti-performance* and the lower ranked value is assigned to *Pro-performance*. Suppose O_t has three enumerated values ranked as {V_1, V_2, V_3}, the value pairs for PP-2 are <V_2, V_1>, <V_3, V_1>, <V_3, V_2>. The value pairs for PP-3 are <V_1, V_2>, <V_1, V_3>, <V_2, V_3>.

One challenge is that for numeric options, it is hard to test all combinations of two values because the value ranges may be extremely large. To address this, guided by Finding 2, the sampling numbers can be significantly reduced by fixing V_{src} to the minimum configuration values or fixing V_{tar} to the maximum configuration values. During the sampling process, a small step length between V_{src} and V_{tar} may lead to limited performance change, and cannot expose CPBugs. While a large step length may lead to one of V_{src} or V_{tar} located outside the value range which can trigger CPBugs. In this regard, CP-Detector first assigns V_{src} to the minimum value, and increases V_{tar} exponentially (e.g, <1, 2>, <1, 4>, <1, 8> ...) until the maximum value. Then, CP-Detector assigns V_{tar} to the maximum value, and decreases V_{src} exponentially until the minimum value. This sampling strategy helps CP-Detector find the proper V_{src} and V_{tar} within limited samples.

3.2.2 Sampling Test Scenarios. Given a target configuration option O_t, CP-Detector now has two values (i.e., V_{src} and V_{tar}) of O_t, and the expected performance change when tuning O_t from V_{src} to V_{tar}. Besides O_t, a CPBug may need other triggering conditions, including workloads, other configurations, or the running stages of the software (e.g., start, service, shutdown). In this regard, we define test scenarios *S*=<*Workload*, *Configuration*, *Stage*>. CP-Detector will generate different scenarios, then test V_{src} and V_{tar} of O_t under each scenario.

Workloads: CP-Detector uses both performance benchmark tools and official performance test suite as workloads. Benchmark tools provide a variety of parameters with wide ranges. To generate representative workload commands, CP-Detector applies the state-of-art distance-based sampling method [39] which supports flexible sample size and is more representative [39, 60] than traditional n-wise sampling. After that, the official test suite is integrated with those commands to get the complete set of workload commands. CP-Detector also provides interfaces to accept customized workloads.

Configurations: In § 2, there are 150 CPBugs that show expectation mismatches. We manually analyzed the CPBugs and found 94.0% of them can be triggered by testing the combinations of two options. This result indicates that CP-Detector can expose 94.0% CPBugs by sampling one other option besides O_t. To do this, CP-Detector uses the one-hot sampling strategy, i.e., changing one option at one time while other options remain default values. As for the constraints between configurations, CP-Detector needs to filter out combinations that violate configuration constraints. To achieve this, CP-Detector leverages SPEX [64], which uses the data-flow of program variables corresponding to the configuration options to extract constraints. CP-Detector also allows users to provide customized constraints.

Running stages: CPBugs may only be triggered at specific running stages of software. For example, the CPBug Httpd-50002 shown in § 2.2 happened at the start stage. In this regard, we predefine running stages for each software domain, including {"start", "restart", "service", "shutdown"} for servers and {"binary compilation", "binary execution"} for compilers. Then, CP-Detector tries to expose CPBugs under each stage. For configuration options of the *Resource* type, CP-Detector only uses the *service* stage in servers, since other stages do not use the resources.

3.2.3 Results Checking. CP-Detector finally checks if the actual performance change of V_{src} and V_{tar} indicates a CPBug according to Column 6 in Table 2. For PP-1, PP-2, PP-4, PP-5, and PP-7, it is easy to check if the actual performance drops, i.e., $P(V_{src}) > P(V_{tar})$, where $P(V_{src})$ and $P(V_{tar})$ are performances of V_{src} and V_{tar}, respectively. For PP-3 and PP-6, CP-Detector uses the following rules to determine if the drops are more-than-expected:

$$P(V_{src})/P(V_{tar}) > Tr_1; \qquad P(V_{src}) - P(V_{tar}) > Tr_2.$$

where Tr_1 and Tr_2 are predefined thresholds. It means $P(V_{src})$ is better than $P(V_{src})$ more than Tr_1 times. while the absolute drop from $P(V_{src})$ to $P(V_{tar})$ is large than Tr_2. We will evaluate the thresholds in § 4.4.

Since performance can be influenced by many environment factors, such as network delay and system warm-up, an application running repeatedly on the same machine can produce performance results that differ with each execution. CP-Detector employs a strategy to eliminate the performance bias. CP-Detector tests each case 20 times repeatedly and uses hypothesis testing [4] to eliminate the performance bias. Specifically, CP-Detector assumes the performances of V_{src} and V_{tar} as two random variables, then uses the t-test ($\alpha = 0.05$) to check if the ">" relations hold in the above rules. We set the null hypothesis that the relations do not hold. When the null hypothesis is rejected, a CPBug is alarmed.

4 EVALUATION

To evaluate CP-Detector, we consider four research questions:

RQ1: How accurate is CP-Detector at suggesting performance properties?

RQ2: How effective and efficient is CP-Detector at exposing both known and unknown CPBugs?

RQ3: How does CP-Detector compare with the state-of-the-art performance bug detection tool?

RQ4: How do CP-Detector parameters influence its effectiveness?

4.1 RQ1: Accuracy of Suggesting Performance Properties

To answer RQ1, we evaluated the accuracy of CP-Detector in suggesting performance properties of configuration options. Given a configuration option, this process contains two components: predicting the type of configuration option and identifying the tuning direction. We evaluated the accuracy for each component. We randomly sampled 500 configuration options from the 12 software systems we studied. Three authors manually labeled the types of the options by analyzing the configuration documents. Each label was cross-checked and discussed until there was no disagreement. This process took 70 working hours. The options were split into 10 sets to conduct a stratified 10-fold cross validation. We did not use the configuration options included in our empirical studies, since the options involved in the CPBugs have an unbalanced distribution. By default, we set the parameters during mining CARs with *Len*=7 and *Num*=100, and introduce how to set these parameters in § 4.4.

Predicting Configuration Types. We evaluated the precision and recall of predicting each configuration type. We also calculated the weighted averages [57], which is defined as the averaged precision/recall of each type weighted by the option number of the type. We also compared our approach with a baseline method, i.e., keyword searching. We used the same CAR mining algorithm and restricted *len*(CAR) = 1 to generate keywords for each type. We set *Num* = 25, which has been tested to be able to achieve the best result. A larger *Num* (e.g., 100) may improve the recall, but significantly decreases the precision at the same time.

Table 4: Precision and recall on inferring types of configuration options (average result with stratified 10-fold cross validation).

Type	# Option	Precision		Recall	
		CPD†	Base.‡	CPD	Base.
Resource	143	93.9%	69.4%	92.4%	93.2%
Tradeoff	84	70.7%	61.3%	66.9%	21.4%
Optimization	73	69.4%	27.3%	65.8%	18.8%
Functionality	100	82.2%	56.2%	55.6%	39.1%
Non-influence	100	90.1%	35.0%	67.1%	70.0%
Weighted Average		83.3%	52.4%	71.8%	54.8%

† CP-Detector. ‡ The keyword-based baseline method.

Table 4 shows the precision and recall of CP-Detector in predicting configuration types. CP-Detector is most effective in predicting the *Resource* type. This is because their option descriptions often contain similar semantics, e.g., memory, buffer, CPU, etc. While for *Functionality* options, CP-Detector has a good precision but the lowest recall. This is because the functionalities are highly diverse and only limited common features (e.g., profiling, monitoring) are identified by CP-Detector. Compared to the baseline method, when considering the precision and recall together (i.e., the harmonic mean of the precision and recall), CP-Detector outperforms the baseline method in every type. *This result suggests that CP-Detector is effective (83.3% precision and 71.8% recall) in predicting configuration types.*

Identifying Tuning Directions. The second task is to determine the tuning direction for each value pair of a given option. This task is challenging for *Tradeoff* configuration options, while the tuning directions of other options are straightforward. Among 84 *Tradeoff* options, we need to check 162 pairs of tuning directions (an option with 3 values, say A,B,C, implies 3 pairs: AB, AC, BC). CP-Detector successfully predicted 139/162 (85.8%) of them. Meanwhile, CP-Detector failed to identify 23 cases. This is because the configuration documents do not always contain the relationship among the values of a tradeoff option. For example, a common tradeoff option for database is `storage_engine`. Different engines produce different levels of performance, concurrency, consistency, integrity, etc. But these properties are not described in the documents. The accuracy of other types are: 100% for both *Resource* and *Non-Influence*, 94.4% for *Functionality*, 97.8% for *Optimization* (e.g., some *Functionality* options are not Boolean; thus, CP-Detector can not handle). *This result indicates CP-Detector is effective (96.9% accuracy in average) in identifying tuning directions.*

4.2 RQ2: Effectiveness of Efficiency of Detecting CPBugs

To evaluate CP-Detector in exposing CPBugs, we first applied CP-Detector to a set of existing CPBugs. We then used CP-Detector to find previously unknown CPBugs.

4.2.1 Detecting Existing CPBugs. We evaluate the effectiveness and efficiency of CP-Detector in exposing CPBugs studied in § 2. We tried to reproduce all the 173 studied CPBugs with our best effort. We successfully reproduced 38 bugs. To avoid over-fitting, we

Table 5: The effectiveness of detecting existing CPBugs.

PP Violated	# CPBugs	# Exposed	# FP
Optimization (PP-1)	7	5† / 6‡	2
Tradeoff-1 (PP-2)	12	10 / 12	0
Tradeoff-2 (PP-3)	9	7 / 8	0
Resource (PP-4)	12	11 / 12	1
Functionality-1 (PP-5)	2	1 / 2	0
Functionality-2 (PP-6)	9	5 / 8	2
Non-Influence (PP-7)	6	4 / 6	0
N/A	4	0 / 0	2
TOTAL	61	43 / 54	7

† # CPBugs exposed by CP-Detector. ‡ # CPBugs exposed by CP-Detector given ideal properties.

followed the bug collection steps in § 2 and successfully reproduced 23 bugs that are not included in 173 studied bugs. Reproducing these 61 CPBugs took 500 working hours. We evaluate CP-Detector on these 61 CPBugs. By default, CP-Detector sets $Tr_1 = 3$ and $Tr_2 = 5$. We evaluate how to determine these thresholds in § 4.4.

Effectiveness. To evaluate the effectiveness, we assessed both *completeness*: how many bugs can be exposed by CP-Detector from the 61 known bugs, and *soundness*: how many false positives CP-Detector produces. To measure the false positive, we applied CP-Detector to the software versions where bugs have been fixed by developers, and observed if CP-Detector still reports bugs. The results are shown in Table 5. CP-Detector successfully exposed 43 out of the 61 bugs. Among the exposed bugs, 19 bugs were exposed by using the default workloads of CP-Detector, while 24 bugs required specific workloads collected from bug reports. CP-Detector failed to expose 18 bugs due to the following reasons: 1) The properties suggested by CP-Detector is not correct (11 cases); 2) The bug is not triggered when testing (3 cases), e.g., MongoDB-30643 [16] can be exposed when 7 different options are set to specific values; 3) The option does not have any property (4 cases, row "N/A"), e.g., MySQL-74325 [19] in § 2.1.

CP-Detector reported seven false positives. Three cases are caused by bad application design. For example, in the case of PP-4, allocating larger caches results in worse performances in both buggy and fixed versions. This is because the query cache feature is actually ill-designed: only in rare cases, increasing the cache size improves the performance. This feature is removed since MySQL is upgraded to v8.0. Four cases are caused by incorrect properties. For example, in the "N/A" case, CP-Detector falsely regards `-m32/64` as a non-influence option, which can affect performances depending on CPU architectures. *This result indicates CP-Detector can effectively (43/61) expose existing CPBugs with limited false positives at the same time.*

Efficiency. To measure the efficiency, we used the number of value pairs of each configuration option required to expose CPBugs. We mainly evaluate the numbers for numeric configuration options because the sampling approach for other options is straightforward, i.e., enumerating all combinations. We evaluated the numbers on all 17 CPBugs (from the 61 ones) with numeric options and compared CP-Detector with a baseline method, i.e., uniform sampling (the sampled numbers satisfy uniform distribution). The results showed that CP-Detector exposed 11 out of the 17 bugs. To make a fair comparison, we tuned the sampling density of the baseline method until the same number of bugs are exposed. As a result, CP-Detector generated 106 value pairs (9.6 pairs each bug in average, *stdev*=3.1), whereas the baseline method generated 1,320 pairs (120 pairs each bug in average, *stdev*=0). CP-Detector required fewer pairs since fixing one of the two values on the minimum or maximum value. *This result indicates our sampling strategy can significantly improve the efficiency (i.e., reduce the sampling numbers) while remaining the same effectiveness.*

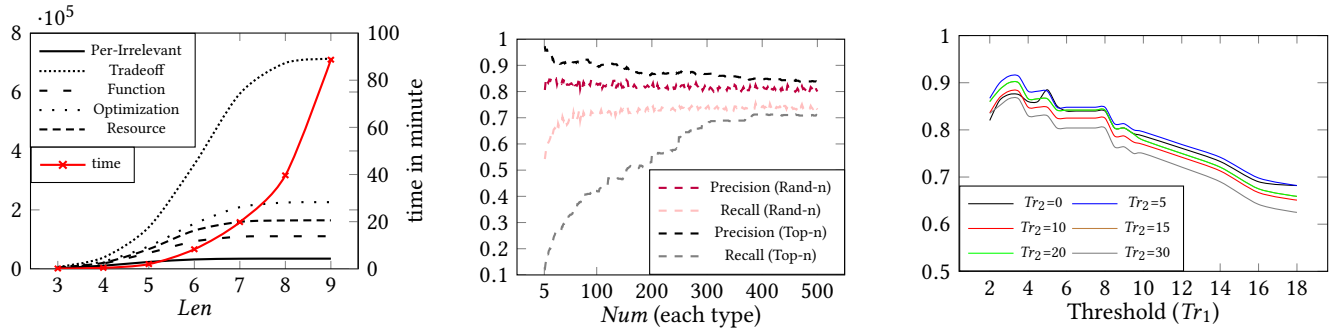
Table 6: New CPBugs detected by CP-Detector

Bug ID	Slowdown‡	Version(s)	Status
Clang #43576(1)	1.19× (E.T.)	v7 - latest	Confirmed
Clang #43576(2)	1.28× (E.T.)	v7 - latest	Confirmed
Clang #43084	2.9× (C.T.)	v3	Fixing
Clang #44359	1.2× (E.T.)	v7 - latest	Pending
Clang #44518	2.0× (C.T.)	v3 - latest	Fixing
GCC #91852	2.8× (C.T.)	v6 - latest	Pending
GCC #91895	4.4× (C.T.)	v4	Confirmed
GCC #91817	44× (C.T.)	v4	Confirmed
GCC #91875	1.85× (E.T.)	v7 - v8	Confirmed
GCC #93037	1.12× (E.T.)	v8 - latest	Pending
GCC #93521	2.52× (E.T.)	v8 - latest	Confirmed
GCC #93535	4.50× (E.T.)	v8 - latest	Confirmed
GCC #94957	hang (C.T.)	v7 - latest	Fixing

‡ **C.T.**: Compiling Time, **E.T.**: Execution Time.

4.2.2 Detecting Unknown CPBugs. We applied CP-Detector on the options sampled in § 4.1 to evaluate if CP-Detector can detect unknown CPBugs. CP-Detector reported 17 CPBugs from Clang and GCC, including 13 true positives and 4 false positives according to our manual analysis. We reported 13 true positives to developers as shown in Table 6. To evaluate the impact of the reported bugs, we calculated their slowdowns by comparing the performances between the fixed version (if any) and the buggy version. If the fixed version is not available, we examined the performance on the other compiler for comparison. The CPBugs found by CP-Detector have significant impacts (1.19× ∼ 1.85× execution times, 2.8× ∼ 44× compilation times) on user's experience and many have been existing for years. Worse still, GCC #94957 [12] hangs for hours to compile 8 lines of C++ code.

Meanwhile, CP-Detector reported four false positives in GCC and Clang. When using a higher level of compile-time optimization, the binary execution time usually decreases. In the two false positives, however, the execution time also depends on the hardware. The higher level optimization generates machine code that is inefficient in the experimental hardware. Adding `-march=native` can solve the problem because it makes the generated code suitable for the compiling machine. These two cases can be eliminated by adding configuration constraints. The other two false positives are caused by incorrect properties suggested by CP-Detector. *This result indicates CP-Detector can expose unknown CPBugs with high impacts (up to 44× slowdowns and existing for years).*



(a) Number of CARs and overhead with different *Len*

(b) Precisions and recall of two CAR selection strategies with different *Num*.

(c) F-score of different Tr_1, Tr_2

Figure 4: The Influence of CP-Detector Parameters.

4.3 RQ3: Comparison with the State-of-the-art

We compare CP-Detector with Toddler [46], which is one of the most effective bug detection tools among existing works. Toddler uses redundant memory access patterns to detect performance bugs that are caused by inefficient loops. We evaluate the effectiveness of Toddler in exposing the same CPBugs. The evaluation shows that Toddler can detect 6 of the 61 existing CPBugs. This is because the CPBugs are caused by a variety of reasons, while Toddler only focuses on the inefficient variable accesses in loops, which account for a small proportion in our dataset. *This result indicates CP-Detector can detect more types of performance bugs than Toddler. CP-Detector can serve as a complementary tool with Toddler in detecting general performance bugs.*

4.4 RQ4: The Influence of Model Parameters

The effectiveness of CP-Detector can be affected by the selection of four parameters: the length *Len* and number *Num* of CARs, and two thresholds (i.e., Tr_1 and Tr_2) to check PP-3 and PP-6. We evaluate how these parameters impact CP-Detector.

The max length (*Len*) of CARs affects the CAR candidates generated by association rule mining process. With longer *Len*, we can get more CAR candidates, but the time spent for mining grows exponentially. To choose a reasonable value, we use the 500 option descriptions in § 4.1 to evaluate the CAR number and overhead with different *Len*. As shown in Figure 4a, when *Len* is larger than 7, the numbers of CAR candidates for all types of options start to converge. This is intuitive because longer sequences are less likely to appear more than *min_support* times to become a CAR candidate.

The number (*Num*) of CARs selected for each type of options can affect the accuracy of the option type classifier. To evaluate this, we use the same options in § 4.1 and split it by 10 to conduct stratified 10-fold cross validation on precision and recall of the classifier with different *Num*. Also, we compare our sampling approach with a baseline method, i.e., using Top-*Num* CARs from each type. Figure 4b shows the averaged precision and recall of the option type classifier. As *Num* grows, the precisions and recalls of both strategies converge. When *Num* is small (i.e., less than 300), our sampling strategy outperforms the Top-*Num* strategy, since the recall of the Top-*Num* strategy is limited. When *Num* is larger than 100, the recall of our approach converges, while the precision remains the same. Thus, we use 100 as the default value.

To choose the best combination of Tr_1 and Tr_2, we evaluated all CPBugs breaking PP-3 and PP-6. For each bug, we collected the performance pairs (i.e., $P(V_{src})$ and $P(V_{tar})$) in both buggy and fixed versions. Given a combination of Tr_1 and Tr_2, a true positive means the combination suggests the performance pair in the buggy version is a bug; a false positive means the combination suggests the pair in the fixed version is a bug; a false negative means the combination suggests the pair in the buggy version is not a bug. We successfully collected data from 41 bugs, and split the data by 10 to do the 10-fold cross validation. Then we calculate the precision, recall and F-score with different Tr_1 and Tr_2. As shown in Figure 4c, larger Tr_1 or Tr_2 implies stricter conditions, thereby reducing false positives but increasing false negatives (and vice versa). The optimal Tr_1 and Tr_2 combination is 3 and 5 (blue line, with best F-score=90.6%). The average precision and recall are 94.1% and 87.3%, respectively. Thus, we use 3 and 5 as default values.

5 DISCUSSION

Impact of Workloads. Software workloads can affect the effectiveness of CP-Detector in exposing new CPBugs. It is hard to automatically predict real-world workloads that can trigger CPBugs. Instead, CP-Detector provides interfaces to accept customized workloads. For example, when the software end users reported a performance issue, CP-Detector can leverage the workload contained in the report and help developers confirm if the issue is caused by a CPBug.

Quality of Performance Properties. We summarized seven properties from 150 CPBugs. These properties may be limited in two aspects: 1) We may miss a property that does not happen in our studied CPBugs; 2) The properties may be affected by other factors and not always hold. For example, in § 4.2.2, CP-Detector reported two false positives which break our properties but are not bugs. In this regard, we will investigate more bugs in the further work to improve the completeness of properties. CP-Detector provides user interfaces to accept customized property constraints, e.g., for the false positives of § 4.2.2, the property of the optimization-level option holds when setting `-march=native`.

Value Bounds of Numeric Options. When sampling numeric options, CP-Detector needs to determine the lower and upper bounds. Simply using the maximum value of an integer variable

(e.g., 2^{64} for unsigned long) as the upper bound may result in a misconfiguration [27]. For example, when a buffer value is larger than the memory size, the system will use the *swap* memory, and the performance drops. This performance loss is not caused by a CPBug. Directly using the memory size may still be problematic, since one software project may have multiple buffers. To avoid this problem, CP-Detector first extracts the lower and upper bounds from user manuals (if any). Otherwise, we empirically set the lower and upper bounds to 0 and 1/4 of the system resource, respectively. And we monitor the resource usage by `top` to avoid resource overloading.

Reproducing Bugs. We successfully reproduced 61 out of 173 CPBugs. Note that reproducing performance bugs following the bug reports are not trivial [32]. The main reasons why we failed to reproduce many of them are missing of important steps and too complicated workload. For instance, httpd #58037 [10] and MongoDB #27753 [15] only show the symptom but miss bug-inducing workload. MongoDB #27700 [14] requires distributed cluster and complicated workload (vaguely described). Few CPBugs need specific environment to trigger (e.g., httpd #42065 [8] require Windows 2003 Server), which, by construction, CP-Detector can not expose. In this paper, only 11.5% (lowest) of MongoDB's CPBug are reproducible, and 56.3% (highest) for Clang.

Future Work. CP-Detector is far from perfect. First, triggering CPBugs sometimes require specific workload, environment or timing [30, 32]. One of our future work will lie in designing automatic workload generation techniques to expose CPBugs for those software systems that have limited or no test suite or benchmark tools. Second, CP-Detector can report unexpected performance drop by tuning options, but can not locate them, still leaving diagnose efforts of developers. So we will explore how to locate the bug-inducing code of CPBugs to help developer fix them.

6 RELATED WORK

Performance Bug Detection. Some works focus on detecting different types of performance problems: inefficient loops [29, 45, 46], redundant roads [61], redundant collection traversals [48], reusable data [44, 62], false sharing in multi-thread programs [41], inefficient synchronization [49, 66], user-interface performance problems [50], architectural impacts among methods [26], performance anti-patterns [65] and tradeoffs [21] in ORM applications. These works are effective in detecting certain types of performance problems, which are different to the configuration-handling performance bugs detected by CP-Detector. Recent short position papers [37, 53, 54] have proposed a proof concept that using metamorphic testing to expose performance bug. While they did not proposed an automatic approach or evaluate on large scale software systems. The difference between our work and metamorphic testing is that they typically use multiple test executions to infer metamorphic relations, and verify those relations on follow-up tests. While we conclude performance properties from bug study and generate them from expert knowledge (e.g., user manuals).

Hotspots Detection. Some works focus on pinpointing hotspots in programs via profiling: Perf [6], YourKit [31]. Similarly, several following works address on generating the most time-consuming workloads via profilers to help expose performance bottlenecks [28, 55, 67]. The limitation of profiling-based methods is that the hotspots are not necessarily caused by performance bugs. While CP-Detector can use performance properties summarized from real-world bugs to confirm if a hotspot if caused by a CPBug.

Performance Modeling and Tuning. Many works [25, 35, 38, 43, 47, 51, 52] aim to predict performance for given configurations, or study the tendency of performance changes to improve performance when tuning configurations. These works focus on building the relationship between performance and configuration, and finding the fastest configuration of a software system. This is different to find performance bugs caused by incorrect configuration handling.

Understanding of Performance Bugs. Previous studies of performance bugs have covered a wide range of characteristics including root causes, fixing complexity, how they are introduced and found [36, 58]. Recently, some empirical studies [33, 34] emphasize the importance of configuration-aware testing techniques and provide insights on reducing the searching space of configurations. Some works help comprehend performance, including performance distributions generation [24], and performance specifications extracting via in-field data [22]. These works help understand performance issues, while CP-Detector can expose CPBugs automatically.

7 CONCLUSIONS

Performance bugs are hard to detect due to their non fail-stop symptom. In this paper, we argue that the performance expectation of configuration tuning can be leveraged to expose CPBugs. We studied 173 real-world CPBugs from 12 software systems and found most (86.7%) of CPBugs can be exposed by using the expectations. Our findings also guide the inferring of performance expectations and sampling of test inputs to trigger CPBugs. We design and implement CP-Detector to detect real-world CPBugs. The result shows that CP-Detector is effective in exposing both known and unknown CPBugs. CP-Detector can be integrated into an IDE as a regression test tool w.r.t. performance, or used as an assistant tool to confirm performance-related bugs in bug tracking systems.

ACKNOWLEDGMENTS

This research was supported by National Key R&D Program of China (Project No.2017YFB1001802); National Natural Science Foundation of China (Project No.61872373, 61702534 and 61872375).

REFERENCES

[1] Categories and subcategories in computer science. https://en.wikipedia.org/wiki/Category:Computing.
[2] Fit CDF for normal distribution. https://en.wikipedia.org/wiki/Cumulative_distribution_function.
[3] spaCy. https://spacy.io.
[4] Statistical hypothesis testing. https://en.wikipedia.org/wiki/Statistical_hypothesis_testing.
[5] IEEE standard glossary of software engineering terminology. *IEEE Std 610.12* (1990).
[6] Linux perf tool. https://perf.wiki.kernel.org/index.php/Main_Page, 2015.
[7] httpd 33605, Acc.: 2020. https://bz.apache.org/bugzilla/show_bug.cgi?id=33605.
[8] httpd 42065, Acc.: 2020. https://bz.apache.org/bugzilla/show_bug.cgi?id=42065.
[9] httpd 50002, Acc.: 2020. https://bz.apache.org/bugzilla/show_bug.cgi?id=50002.
[10] httpd 58037, Acc.: 2020. https://bz.apache.org/bugzilla/show_bug.cgi?id=58037.
[11] GCC 17520, Accessed: 2020. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=17520.
[12] GCC 94957, Accessed: 2020. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=94957.
[13] MariaDB 5802, Accessed: 2020. https://jira.mariadb.org/browse/MDEV-5802.

[14] MongoDB 27700, Accessed: 2020. https://jira.mongodb.org/browse/server-27700.
[15] MongoDB 27753, Accessed: 2020. https://jira.mongodb.org/browse/server-27753.
[16] MongoDB 30643, Accessed: 2020. https://jira.mongodb.org/browse/server-30643.
[17] MySQL 21727, Accessed: 2020. https://bugs.mysql.com/bug.php?id=21727.
[18] MySQL 67432, Accessed: 2020. https://bugs.mysql.com/bug.php?id=67432.
[19] MySQL 74325, Accessed: 2020. https://bugs.mysql.com/bug.php?id=74325.
[20] MySQL 77094, Accessed: 2020. https://bugs.mysql.com/bug.php?id=77094.
[21] Atlee, J. M., Bultan, T., and Whittle, J. View-centric performance optimization for database-backed web applications. In *International Conference on Software Engineering (ICSE)* (2019).
[22] Brüninк, M., and Rosenblum., D. S. Mining performance specifications. In *European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)* (2016).
[23] Butterfield, A., Ngondi, G. E., and Kerr, A. *A Dictionary of Computer Science*. Oxford University Press, 2016.
[24] Chen, B., Liu, Y., and Le., W. Generating performance distributions via probabilistic symbolic execution. In *International Conference on Software Engineering (ICSE)* (2016).
[25] Chen, T.-H., Shang, W., Hassan, A. E., Nasser, M., and Flora, P. Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications. In *European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)* (2016).
[26] Chen, Z., Chen, B., Xiao, L., Wang, X., and Xu, B. Speedoo: prioritizing performance optimization opportunities. In *International Conference on Software Engineering (ICSE)* (2018).
[27] Coady, Y., Cox, R., DeTreville, J., Druschel, P., Hellerstein, J., Hume, A., Keeton, K., Nguyen, T., Small, C., Stein, L., and Warfield, A. Falling off the cliff: When systems go nonlinear. In *Workshop on Hot Topics in Operating Systems (HotOS)* (2005).
[28] Coppa, E., Demetrescu, C., and Finocchi., I. Input-sensitive profiling. In *Conference on Programming Language Design and Implementation (PLDI)* (2012).
[29] Dhok, M., and Ramanathan, M. K. Directed test generation to detect loop inefficiencies. In *European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)* (2016).
[30] Ding, Z., Chen, J., and Shang, W. Towards the use of the readily available tests from the release pipeline as performance tests. are we there yet? In *International Conference on Software Engineering (ICSE)* (2020).
[31] GmbH, Y. The industry leader in .NET & Java profiling. https://www.yourkit.com.
[32] Han, X., Carroll, D., and Yu, T. Reproducing performance bug reports in server applications: The researchers' experiences. *Journal of Systems and Software 156* (2019), 268–282.
[33] Han, X., and Yu, T. An empirical study on performance bugs for highly configurable software systems. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)* (2016).
[34] Han, X., Yu, T., and Lo, D. Perflearner: Learning from bug reports to understand and generate performance test frames. In *International Conference on Automated Software Engineering (ASE)* (2018).
[35] Jamshidi, P., Siegmund, N., Velez, M., Kästner, C., Patel, A., and Agarwal, Y. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *International Conference on Automated Software Engineering (ASE)* (2017).
[36] Jin, G., Song, L., Shi, X., Scherpelz, J., and Lu, S. Understanding and detecting real-world performance bugs. In *Conference on Programming Language Design and Implementation (PLDI)* (2012).
[37] Johnston, O., Jarman, D., Berry, J., Zhou, Z. Q., and Chen, T. Y. Metamorphic relations for detection of performance anomalies. In *International Workshop on Metamorphic Testing (MET)* (2019).
[38] Juliana, A. P., Mathieu, A., Hugo, M., and Jean-Marc, J. Sampling effect on performance prediction of configurable systems: A case study. In *International Conference on Performance Engineering (ICPE)* (2020).
[39] Kaltenecker, C., Grebhahn, A., Siegmund, N., Guo, J., and Apel, S. Distance-based sampling of software configuration spaces. In *International Conference on Software Engineering (ICSE)* (2019).
[40] Leshl, N., Zaki, M. J., and Ogihara3, M. Mining features for sequence classification. In *ACM Knowledge Discovery and Data Mining (SIGKDD)* (1999).
[41] Liu, T., and Berger., E. D. Sheriff: Precise detection and automatic mitigation of false sharing. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (2011).
[42] Michail, A. Data mining library reuse patterns in user-selected applications. In *International Conference on Automated Software Engineering (ASE)* (1999).
[43] Nair, V., Menzies, T., Siegmund, N., and Apel., S. Using bad learners to find good configurations. In *European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)* (2017).
[44] Nguyen, K., and Xu, G. Cachetor: Detecting cacheable data to remove bloat. In *European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)* (2013).
[45] Nistor, A., Chang, P.-C., Radoi, C., and Lu, S. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *International Conference on Software Engineering (ICSE)* (2015).
[46] Nistor, A., Song, L., Marinov, D., and Lu, S. Toddler: Detecting performance problems via similar memory-access patterns. In *International Conference on Software Engineering (ICSE)* (2013).
[47] Oh, J., Batory, D., Myers, M., and Siegmund, N. Finding near-optimal configurations in product lines by random sampling. In *European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)* (2017).
[48] Olivo, O., Dillig, I., and Lin, C. Static detection of asymptotic performance bugs in collection traversals. In *Conference on Programming Language Design and Implementation (PLDI)* (2015).
[49] Pradel, M., Huggler, M., and Gross, T. R. Performance regression testing of concurrent classes. In *International Symposium on Software Testing & Analysis (ISSTA)* (2014).
[50] Pradel, M., Schuh, P., Necula, G., and Sen, K. Eventbreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (2014).
[51] Ravjot, S., Cor-Paul, B., Weiyi, S., and E., H. A. Optimizing the performance-related configurations of object-relational mapping frameworks using a multi-objective genetic algorithm. In *International Conference on Performance Engineering (ICPE)* (2016).
[52] Sarkar, A., Guo, J., Siegmund, N., Apel, S., and Czarnecki, K. Cost-efficient sampling for performance prediction of configurable systems. In *International Conference on Automated Software Engineering (ASE)* (2015).
[53] Segura, S., Troya, J., Durán, A., and Cortés, A. R. Performance metamorphic testing: A proof of concept. *Information & Software Technology 98* (2018), 1–4.
[54] Segura, S., Troya, J., Durán, A., and Ruiz-Cortés, A. Performance metamorphic testing: Motivation and challenges. In *International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER)*, (2017).
[55] Shen, D., Luo, Q., Poshyvanyk, D., and Grechanik., M. Automating performance bottleneck detection using search-based application profiling. In *International Symposium on Software Testing & Analysis (ISSTA)* (2015).
[56] Siegmund, N., Grebhahn, A., Apel, S., and Kästner, C. Performance-influence models for highly configurable systems. In *European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)* (2015).
[57] Sokolovaa, M., and Lapalmeb, G. A systematic analysis of performance measures for classification tasks. *Information Processing & Management* (2009).
[58] Song, L., and Lu, S. Statistical debugging for real-world performance problems. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (2014).
[59] Song, L., and Lu, S. Performance diagnosis for inefficient loops. In *International Conference on Software Engineering (ICSE)* (2017).
[60] Souto, S., d'Amorim, M., and Gheyi, R. Balancing soundness and efficiency for practical testing of configurable systems. In *International Conference on Software Engineering (ICSE)* (2017).
[61] Su, P., Wen, S., Yang, H., Chabbi, M., and Liu, X. Redundant loads: A software inefficiency indicator. In *International Conference on Software Engineering (ICSE)* (2019).
[62] Toffola, L. D., Pradel, M., and Gross, T. R. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (2015).
[63] Xiang, C., Huang, H., Yoo, A., Zhou, Y., and Pasupathy, S. Pracextractor: Extracting configuration good practices from manuals to detect server misconfigurations. In *USENIX Annual Technical Conference (ATC)* (2020).
[64] Xu, T., Zhang, J., Huang, P., Zheng, J., Sheng, T., Yuan, D., Zhou, Y., and Pasupathy, S. Do not blame users for misconfigurations. In *Symposium on Operating Systems Principles (SOSP)* (2013).
[65] Yang, J., Subramaniam, P., Lu, S., Yan, C., and Cheung, A. How *not* to structure your database-backed web applications: a study of performance bugs in the wild. In *International Conference on Software Engineering (ICSE)* (2018).
[66] Yu, T., and Pradel, M. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *International Symposium on Software Testing & Analysis (ISSTA)* (2016).
[67] Zaparanuks, D., and Hauswirth., M. Algorithmic profiling. In *Conference on Programming Language Design and Implementation (PLDI)* (2012).
[68] Zimmermann, T., Weisgerber, P., Diehl, S., and Zeller, A. Mining version histories to guide software changes. In *International Conference on Software Engineering (ICSE)* (2005).