2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

Faster Configuration Performance Bug Testing with Neural Dual-level Prioritization

Youpeng Ma[1], Tao Chen[2*], Ke Li[3]
[1] School of Computer Science and Engineering, University of Electronic Science and Technology of China, China
[2] IDEAS Lab, School of Computer Science, University of Birmingham, United Kingdom
[3] Department of Computer Science, University of Exeter, United Kingdom
myp@std.uestc.edu.cn, t.chen@bham.ac.uk, k.li@exeter.ac.uk

***Abstract*—As software systems become more complex and configurable, more performance problems tend to arise from the configuration designs. This has caused some configuration options to unexpectedly degrade performance which deviates from their original expectations designed by the developers. Such discrepancies, namely configuration performance bugs (CPBugs), are devastating and can be deeply hidden in the source code. Yet, efficiently testing CPBugs is difficult, not only due to the test oracle is hard to set, but also because the configuration measurement is expensive and there are simply too many possible configurations to test. As such, existing testing tools suffer from lengthy runtime or have been ineffective in detecting CPBugs when the budget is limited, compounded by inaccurate test oracle.**

In this paper, we seek to achieve significantly faster CPBug testing by neurally prioritizing the testing at both the configuration option and value range levels with automated oracle estimation. Our proposed tool, dubbed `NDP`, is a general framework that works with different heuristic generators. The idea is to leverage two neural language models: one to estimate the CPBug types that serve as the oracle while, more vitally, the other to infer the probabilities of an option being CPBug-related, based on which the options and the value ranges to be searched can be prioritized. Experiments on several widely-used systems of different versions reveal that `NDP` can, in general, better predict CPBug type in 87% cases and find more CPBugs with up to 88.88× testing efficiency speedup over the state-of-the-art tools.

***Index Terms*—Performance bug testing, software debugging, testing prioritization, configuration testing, SBSE.**

I. Introduction

Modern software systems typically have a high degree of configurability wherein the configuration options directly (e.g., certain optimization) or indirectly (e.g., resource allocation) affect software performance, such as throughput and latency [1]–[7]. As software configurability continues to improve, they are also more likely to be buggy. We refer to these performance bugs caused by configuration errors as Configuration Performance Bugs (CPBugs). It is worth noting that CPBugs differ from the typical misconfigurations that concern user-induced configuration errors [8]; instead, they are the errors of the configuration design in the source code that are unintentionally introduced by the developers of the systems [2]. A typical example of CPBugs has been illustrated in Table I. Here, we can see that the option `read_buffer_size` in MySQL is

TABLE I: A real-world example of CPBugs for MySQL.

ID: MySQL-44723; **CPBug-related Option:** `read_buffer_size`
Expected Performance: Each thread that does a sequential scan for a `MyISAM` table allocates a buffer of this size (in bytes) for each table it scans. If you do many sequential scans, you might want to increase this value, which defaults to 131072.......
Actual Performance: The performance decreases if the option `read_buffer_size` is set to be larger than 256K......

mainly used to change the size of the read buffer allocated for each sequential table scan request. In the developers' expectation, increasing this option value should allow MySQL to cache more results, hence a larger buffer should improve performance. However, the performance actually drops when increasing the value beyond 256K. This is because, in the code logic, `MyISAM` initializes an `IO_CACHE` for writing and uses `read_buffer_size` for bulk inserts. `my_malloc` is called with the `MY_ZEROFILL` flag, which causes `memset` to be called on the size of `read_buffer_size`, hence mistakenly restricting the permitted memory quote for processing SQL commands and resulting in a large performance decrease.

CPBugs can lead to devastating outcomes [2], [9], [10]. For example, there have been several large-scale flight delays, which were mainly caused by problematic configuration designs of the systems [11]. Systems from Google and Meta [12], [13] have also suffered performance degradation or outage due to configuration issues, leading to a huge loss of revenue.

However, testing and finding CPBugs are challenging, due primarily to the fact that (1) there are simply too many configurations to examine (MySQL has hundreds of options with more than millions of configurations [14]); (2) measuring configuration performance is highly expensive, e.g., it can take up to 166 minutes to merely measure one configuration on MariaDB [15]; and (3) the test oracle is often unclear, i.e., we do not know when a CPBug occurs. While some tools exist for testing CPBugs [2], [16], [17], they are limited in the efficiency of testing and the accuracy of oracle estimation. That is, they have not effectively handled the discriminative importance of the options and their value range with respect to the CPBugs, together with restricted rule-driven oracle inference. Therefore, those tools suffer from issues such as long running times or are ineffective in detecting CPBugs

*Tao Chen is the corresponding author. Youpeng Ma is also supervised in the IDEAS Lab.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00201

TABLE II: Categorization of CPBugs types from He et al. [2]. Anti-Performance implies better performance at the cost of lower security, consistency, integrity, and etc; Pro-Performance means otherwise. Their value change can be in any direction.

CPBug Type	Option Purpose	Source Option Value	Target Option Value	Expected Performance	Actual Performance
Type-1	Optimization on-off	OFF	ON	Rise	Drop
Type-2	Non-functional trade-off	Ant- Performance	Pro-Performance	Rise	Drop
Type-2	Non-functional trade-off	Pro-Performance	Anti-Performance	Drop	Drop beyond expectation
Type-3	Resource allocation	Small	Large	Rise	Drop
Type-4	Functionality trade-off	ON	OFF	Rise	Drop
Type-4	Functionality trade-off	OFF	ON	Drop	Drop beyond expectation
Type-5	Non-influential option	Any	Any	Keep	Drop

when the testing budget is limited while can be largely mislead by incorrect oracle.

In this paper, we propose to test CPBugs with what we call Neural Dual-level Prioritization, dubbed `NDP`, a framework that can be paired with different heuristic generators. `NDP` leverages neural language models for estimating test oracle and, more importantly, for prioritizing CPBug testing, which is motivated by our observations that only a small proportion of the options are responsible for CPBugs and there are specific ranges of the numeric options' values that are more CPBugs-prone. As such, by leveraging on the probabilities of being CPBug-related to the options, we prioritize the test at two levels: at the options level and at the level of search depth for numeric options, aiming to significantly accelerate the detection of CPBugs. Specifically, our contributions are:

- Instead of leveraging on rule mining [2], we build a neural language model, i.e., RoBERTa [18], that predicts the CPBug type which serves as the oracle in testing.
- We fine-tune the other RoBERTa model for estimating the probabilities of an option being CPBug-related, which prioritizes the options to be tested.
- For testing numeric options, we design three search depths that bound an underlying heuristic generator with different search spaces. During the actual testing, those depths (and hence their corresponding search spaces) are prioritized according to the commonality of the CPBug-relatedness and code semantic of an option.
- We evaluate `NDP` over several widely-used systems with various versions, 2–10 workloads, containing 66 known CPBugs, and against state-of-the-art testing tools.

The results show that, compared with state-of-the-art tools/approaches, `NDP` more accurately predicts the oracle of CPBug type in 87% types/metrics; tests CPBugs with up to 1.73× speedup by prioritizing the tested options; while finding more CPBugs with from 3.13× to 88.88× speedup through prioritizing the search depths of numeric options. All data and source code are publicly accessible via our repository: https://github.com/ideas-labo/ndp.

This paper is organized as: Section II presents the preliminaries. Section III delineates the known and newly discovered characteristics of CPBugs, which derive our designs. Section IV illustrates `NDP`. Section V presents the experiment setup followed by the results in Section VI. Sections VII, VIII and IX present the discussion, threats to validity, and related work, respectively. Section X concludes the paper.

II. PRELIMINARIES

A. CPBugs in Configurable Systems

In general, CPBugs naturally incur from the mismatch between the expected performance (as specified in the documentation) and the actual performance observed by changing a configuration option. Therefore, measuring the performance deviation between the source and target option value serves as a strong oracle for identifying the CPBugs. In particular, the performance deviations that cause the CPBugs can be mainly observed from a common scenario: *the direction of expected and actual performance changes is different, e.g., the expectation is performance raises but the actual effect is a performance drop*[1] *under at least one workload,* implying defects in the code segments of the configuration option.

B. CPBugs Types

We follow the categorization of the CPBug types proposed by He et al. [2], as articulated in Table II and below:

- **Optimization Switch:** When enabled, an optimization strategy is activated, and the performance is expected to improve. Yet, a performance drop implies a likely CPBug.
- **Non-functional Trade-off:** Configuration options are used to balance the performance and other non-functional needs, such as the ACID properties. Whether the option needs to be increased or decreased is case-dependent. This involves two subtypes (see *Type-2* in Table II). In this work, we do not distinguish these two as the threshold for the "drop beyond expectation" is highly subjective. Instead, for an option under this type, there is a CPBug as long as a performance drop is observed[2].
- **Resource Allocation:** Options influence resource allocation; more resources are expected to boost performance.
- **Functional Switch:** Options control non-performance functionalities but indirectly affect the system's performance. When an option disables a function, system performance usually improves. This also involves two situations (see *Type-4* in Table II). We do not distinguish those two cases due to the same aforementioned reason.

[1]We follow the de facto standard that a drop is significant only when the change is greater than 5% on any concerned performance attribute [19], [20].

[2]Distinguishing the subtypes does not affect the testing designs of the approach. This is because, if an option is classified into either subtype, then what we seek to find during testing is mainly whether there is an actual performance drop for the two subtypes regardless of the expected performance. As such, a CPBug can be found as long as we see a performance drop between configurations with the changes in the option's value.

- **Non-influential Option:** These options should not affect the system's performance, i.e., performance is expected to remain unchanged after adjusting the values.

Each pattern in the above CPBug types determines the oracle of identifying whether there is a CPBug. For example, with *Type-3*, we can interpret it as *"if a resource allocation related option is changed from a small value to a larger value, then we expect the performance to be improved or otherwise there is a CPBug"*. Therefore, it is essential to estimate which CPBug type is the most relevant to a tested option.

C. Testing CPBugs

In essence, CPBug testing aims to generate diverse test cases, represented as a pair of configurations (a source $\mathbf{c}_s$ and a target $\mathbf{c}_t$), which differ only on the configuration option to be tested, i.e., o at index 1 (highlighted in red):

$$\begin{aligned} \mathbf{c}_s &= \{0, 10, 15, 43, 1\} \\ \mathbf{c}_t &= \{0, 20, 15, 43, 1\} \end{aligned} \tag{1}$$

An automated CPBug testing tool would perturb the values of the option o, such that the performance deviation from the source configuration to the target one under at least one workload matches with a CPBug type from Table II, which serves as the oracle. For example, if changing from $\mathbf{c}_s$ to $\mathbf{c}_t$ (from a smaller value of a resource option to a larger value) under a workload causes a performance drop while in the documentation it should have been a rise, then we find a CPBug of *Type-3*. Yet, due to the large number of options and their values, testing CPBugs is extremely expensive.

III. Characteristics of CPBugs

CPBugs naturally come with certain characteristics that can help us design more effective testing. From the systems/versions/options tested in this work, which are taken from a prior study [2] and it is worth noting that the number of total options per system we tested has already exceeded what is considered for them. We have discovered in Table III[3] that:

> ***Characteristic 1:*** *Overall, only 13.39% of the configuration options can trigger CPBugs.*

This means that, although CPBugs can be devastating, testing on all unique options to find them is not cost-effective. The numeric configuration options[4] also have known characteristics. For example, He et al. [2] have shown that:

> ***Characteristic 2:*** *The majority of the numeric configuration options studied can trigger CPBugs when they are changed to near one extreme of their values.*

From Table IV, we have identified a similar pattern from the systems tested: when fixing the source of an option as close to either its maximal or minimal values, all the 18 numerical options can trigger CPBugs when the option in the target is

[3]Due to the scale of systems and limited resources, for each system, we conducted a preliminary study to identify the most recent designed/discussed options for actual testing instead of examining all options.

[4]Note that the numeric options are often discretized by a set of values, e.g., `cache_size` can be 8M, 16M, 32M, etc.

TABLE III: Percentage of unique options that cause CPBugs.

System	All Options (Unique)	CPBugs-related (Unique)	%
MySQL	139	24	17.27%
MariaDB	127	9	7.09%
Apache	121	6	4.96%
Gcc	38	16	42.11%
Clang	38	7	18.42%
Total	463	62	13.39%

TABLE IV: Ranges in the numeric options in the target that cause CPBugs when fixing the source of an option as close to either its maximal or minimal values.

CPBug	Range	CPBug	Range
MySQL-21727	0%—1.60%	MySQL-38511	0%—0.01%
MySQL-44723	0%—100%	MySQL-47529	0%—0.01%
MySQL-51325	0%—100%	MySQL-60074	0%—100%
MySQL-62478	98.44%—100%	MySQL-74325	90%—100%
MySQL-78262	0%—100%	MySQL-80784	5.82E-9%—100%
MariaDB-145	0%—0.01%	MariaDB-8696	3.13%—100%
MariaDB-12556	0.10%—100%	MariaDB-13328	0%—100%
MariaDB-16283	0.80%—100%	Apache-48215	0%—0.10%
Apache-50002	10%—100%	Apache-54852	1.56%—100%

TABLE V: Number of CPBugs triggered under extreme and middle value of numeric options based on their categorization.

Category of Numeric Option	Trigger at Extreme	Trigger at Middle
Buffer	12	8
Memory	2	1
Network	1	1
Thread	1	1
Loop	2	1

Different values of an option on the same version might trigger the same CPBug.

changed to 10% of the values at the opposed extreme. Further to the above, we have additionally found that:

> ***Characteristic 3:*** *66.67% (12 out of 18) of the numeric configuration options can trigger CPBugs when they are changed to their middle values.*

Table IV shows that a considerable proportion of the numeric options can trigger CPBugs when their values are changed to be within the middle 80% and the opposed extreme. The above is because, while most values of numeric options impact the data flow, they usually do not change the control flow. That is, changes in the numeric configuration options need to hit certain critical values that trigger new execution paths, hence more likely to reveal CPBugs.

To further understand the CPBugs-related numeric options, we manually analyze their source code. Inspired from a recent work [19], we classify those options based on the main type of performance sensitive operations that they can control, i.e., operations related to buffer, memory, network, thread or loops (exclude the other types). From Table V, we found that:

> ***Characteristic 4:*** *Numeric options can trigger CPBugs at extreme and middle values most commonly when they control operations related to buffer.*

This is because the buffer operations often cause immediate implication to many parts in the system, hence the corresponding numeric options are highly performance sensitive.

While ***Characteristic 1-2*** have been known, ***Characteristic 3-4*** are newly discovered information for CPBugs in this work.

IV. Dually Prioritized CPBug Testing with Neural Language Model

`NDP` seeks to expedite the CPBug testing via dual-prioritization at two levels: the option level that determines which option to test earlier and the search space level of the numeric options, which sets the order of search space to explore. This is supported by two fine-tuned neural language models, one for estimating the probability of an option being CPBug-related and the other for predicting the most relevant CPBug type that determines the test oracle. Similar to the other tools [2], `NDP` tests the system's configuration option one by one, in which all the combinations of workloads and related versions are also examined in turn. Specifically, we design the two phases in `NDP`, as shown in Figure 1. For *Initialization*, `NDP` focuses on a one-off process that fine-tunes two neural language models using existing data from different systems:

- **CPBug Types Prediction:** Here, the goal is for a neural language model to parse the documentation and predict which CPBug type is most relevant to an option. This then serves as the essential oracle for CPBug testing.
- **Option-CPBugs Relevance Estimation:** We fine-tune another neural language model that takes both the description of options from the documentation and the related code snippet as inputs and estimates the probabilities of those options being CPBug-related. This allows us to handle the identified characteristics of CPBugs.

In the *Testing* phase, `NDP` contains four components:

- **Options Prioritization (high-level):** The probabilities of whether the options are CPBug-related are used to prioritize their testing order (due to ***Characteristic 1***).
- **Exhaustive Generator:** This is mainly for non-numeric options in which all the possible pairs will be covered under all workloads and related versions considered.
- **Search Prioritization (low-level):** For numeric options, the actual testing would also need to be conducted via a certain search depth. In `NDP`, we design three search depths, which are prioritized differently depending on the commonality of the probabilities for being CPBug-related. Each of the search depths would trigger an independent run of the heuristic generator, which can be any search algorithm, to generate the test cases. This fits with ***Characteristic 2*** and ***Characteristic 3***. According to taint analysis of code semantic, buffer-related numeric options are specifically handled given ***Characteristic 4***.
- **Heuristic Generator:** A stochastic search algorithm that samples the values of numeric options in CPBug testing under all workloads and related versions.

For each option under a version, if its value alteration and the performance change (on any concerned performance attribute) in both configurations of a pair match with the pattern in the predicted CPBug type (which serves as the oracle) under at least one workload, then we found a CPBug; otherwise, we stop testing the option for a version when all pairs/workloads have been explored or a budget has been exhausted. In all cases, `NDP` then switches to test the option under the next related version, if any. That is, we only need to find the CPBugs caused by the option on at least one workload under a version but all the related versions would be examined regardless of how many CPBugs are found on the said option. When all related versions have been tested for an option, we move to the next option as prioritized by `NDP`.

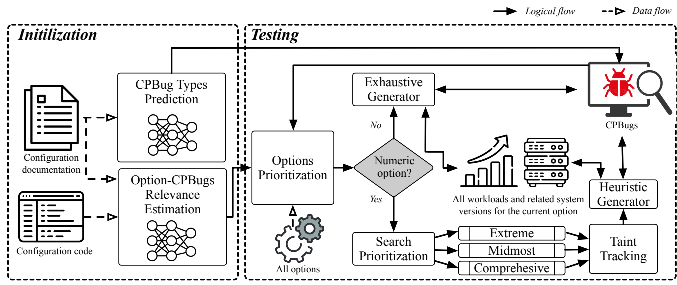


Fig. 1: Workflow overview of `NDP` for CPBug testing.

TABLE VI: An exampled option's description from MySQL.

Option: `innodb_flush_log_at_trx_commit`
Documentation (partial): The `innodb_flush_log_at_trx_commit` controls the balance between strict ACID compliance for commit operations and higher performance that is possible when commit-related I/O operations are rearranged and done in batches. You can achieve better performance by changing the default value......

A. Neural CPBug Types Inference

An essential task in CPBug testing is to determine the oracle, i.e., identifying what types of CPbugs an option is most likely to be associated with, hence triggering the corresponding way to verify whether such an option can trigger CPBugs. To that end, we leverage a single-modal RoBERTa (denoted $\mathcal{M}_s$), a particular type of neural language model, to predict the CPBug type of a given option to be tested. In `NDP`, we fine-tune the RoBERTa using the data collected from previous work [2], such that the inputs are the natural description of an option from the documentation with a known label from the five CPBug type or a label of "no CPBug", since CPBugs are mainly related to the deviation from the expected performance of an option specified in the documentation (e.g., Table VI). Notably, the fine-tuning process is naturally cross-project since the naturalness of the documents ensures its generalization.

In particular, RoBERTa is chosen for three reasons:

- Compared with the rule mining approach [2], it exhibits a stronger generalization ability that can learn hidden information in the documentation in the latent space. In Section VI-A, we will experimentally verify this.
- In contrast to LLM, RoBERTa fits our problem better: we need a classification of CPBugs type rather than text generation. Further, RoBERTa is more cost-effective [21].
- It has been reported that RoBERTa is the generally most promising BERT variant for software engineering [22].

TABLE VII: An example of an option's texts from the documentation and the mapped code snippet from MYSQL.

Option: `innodb_buffer_pool_size`
Documentation (partial): The `innodb_buffer_pool_size` system variable specifies the size of the buffer pool. If your buffer pool is small and you have sufficient memory, making the pool larger can improve performance by reducing the disk I/O as queries access `InnoDB` tables......
Code Snippet Mapped (partial):

```
if (srv_dedicated_server && sysvar_source_svc !=
    nullptr) {
   static const char *variable_name = "
    innodb_buffer_pool_size";
   enum enum_variable_source source;
   if (!sysvar_source_svc->get(variable_name,
     static_cast<unsigned int>(strlen(variable_name
     )),&source)) {
      if (source == COMPILED) {
         double server_mem = get_sys_mem();
```

B. Neural Multi-Modal Option-CPBugs Relevance Estimation

From ***Characteristic 1***, we note that only a small number of options can potentially be the cause of CPBugs. As a result, a natural idea is to estimate which options are more related to CPBugs. `NDP` leverage both documentation and the corresponding code of an option (see Table VII), together with the corresponding label of whether the option is CPBug related, to fine-tune another multi-modal RoBERTa model, $\mathcal{M}_m$, in a binary classification problem. Our goal here, however, is not to use the model to make a binary prediction but to extract its probability related to the likelihood of an option being CPBug-related, based on which we can rank those options. As such, forming this binary classification to train yet another new RoBERTa model has the benefit of simplicity without producing much noise to fulfill our goal.

To that end, we locate the code for a corresponding option in the documentation using a pattern matching-based heuristic:

- **Direct way:** Some systems have a centralized file (or a few files) to maintain all configuration options, such as MYSQL. In those cases, we look at the variable with a similar name to those in the documentation and identify the relevant code snippets from the centralized file(s).
- **Indirect way:** Other systems might not have a file(s) that share the same name as those in the documentation. We consider two cases: (1) there are mechanisms that allow access to those configuration variables via `setter()` and `getter()`. In those cases, we can write a script that searches through the relevant files and outputs the similarity of the `setter()` and `getter()` to each option in the documentation. We can then manually identify the corresponding code snippets. (2) there are no `setter()`/`getter()`, in which case we scan all variables and related functions in the related files.

Once the code snippets and the texts of an option are mapped, our heuristic uses the rules below to clean the code:

- Remove useless comments, e.g., those with timestep, certain license information, etc.
- Remove duplicated comments or code snippets.
- Remove usage examples of the options in the comments. e.g., formats or order of changes.
- Remove code snippets of incomplete functions extracted.

In `NDP`, the texts from documentation and the code snippets of an option are concatenated together (e.g., Table VII) and we use standard steps such as text cleaning, tokenization, and serialization to parse the data. Those inputs, together with a label of whether the option is CPBug-related, form the data to fine-tune the RoBERTa model. To make RoBERTa work for our simplify binary classification problem, we add a task-specific classification layer with a cross-entropy loss function in the fine-tuning process. Upon predicting a given option, the probability of being a true label (CPBug-related) extracted from the `softmax` layer is what we are interested in. Again, we use the CPBugs data that has been reported previously [2].

C. All Options Prioritization (High-level)

In `NDP`, at the high level, we firstly leverage the probabilities of all the options produced by $\mathcal{M}_m$ to determine their order in CPBug testing. This is important as if an option is more likely to cause the CPBugs, then prioritizing it beforehand would help us to identify the bugs quicker. Here, although we do not distinguish the type of options in this prioritization, e.g., numeric and non-numeric ones, their actual testing strategies can be different: for non-numeric options, we generate and test all the combinatorial values of the configurations in the pair using an exhaustive generator, since often those possible values are of limited range [23]. Notably, all workloads and related versions are considered: we at first pick a version and test all workloads therein in turn; if a CPBug is found for a non-numeric option under a workload, then the remaining untested workloads would be skipped and we switch to the next related version. Finally, `NDP` moves to the next option when all related versions have been tested for the current option.

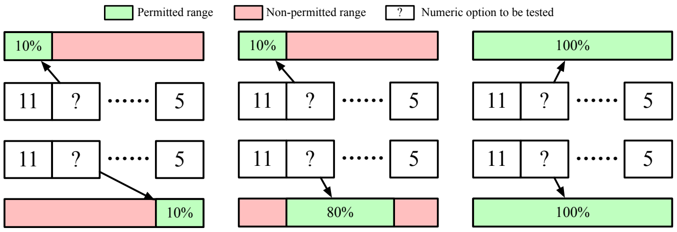
In contrast, for numeric values, we need a stronger way to prioritize their sampling, which we will describe as follows.

D. Search Prioritization for Numeric Options (Low-level)

When the option to be tested is a numeric option, we propose three different search depths that bound the search spaces of the underlying heuristic generator:

- **Extreme search:** As in Figure 2a, this is the search with the most restricted search space: the search for test cases happens within 10% of the upper/lower bounds range of values[5] for both configurations in the pair. Yet, we ensure that the two configurations in the pair are searched over the opposed bounds (***Characteristic 2***).
- **Midmost search:** Here, in Figure 2b, one configuration in the pair is searched within 10% of the upper or lower bound range values while the other can be changed within the middle 80% of the values (***Characteristic 3***).

[5]For those options without explicitly defined upper/lower bounds, we set them using our understanding of the domain, e.g., the capacity of our hardware or the extreme values that are commonly set in practice.



(a) Extreme (b) Midmost (c) Comprehensive

Fig. 2: Illustrations of the search space bounded by different search depths for CPBug testing with NDP.

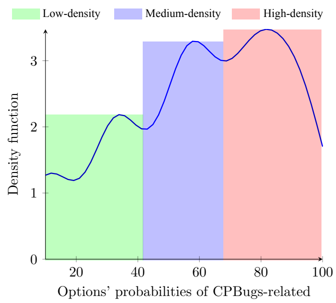


Fig. 3: Exampled kernel density function on the numeric options' probabilities of being CPBug-related for a system.

- **Comprehensive search:** This is the search that basically means all possible values of the configurations in the pair can be explored (Figure 2c).

Those search depths differ in terms of the number of tests (for the pairs) required. According to the probabilities produced from $\mathcal{M}_m$ for the numeric options, we can then prioritize how the above three search depths are used on them. In NDP, the idea is to divide the probabilities of all numeric options into three divisions, based on which different prioritization of the search depth is used. To systematically perform such a division, we leverage the Gaussian Kernel Density Estimation (GKDE). In essence, GKDE serves as a one-dimensional binning algorithm that divides the probability density of options being CPBug-related into three bins with no thresholds. This is achieved by dividing the options into the top three peaks[6], which are separated by a local trough, based on their closeness of probabilities to those peaks.

Figure 3 shows an example: we see that the numeric options are divided into three divisions according to their commonality on the probabilities of being CPBug-related. These divisions derive three prioritizations of the search depths:

- **High-density:** For the numeric options belonging to the most frequent bin, we prioritize the more restricted search depth: starting from extreme search, midmost search, and finally comprehensive search. This will speedup testing if CPBugs can be detected within an extreme/middle value.
- **Medium-density:** Here, we prioritize the midmost search first, followed by the extreme search, and then the comprehensive search. The reason is that since the numeric options are of medium density, we start from the midmost search that also assumes a medium size of search space.
- **Low-density:** For the least frequent bin of numeric options, we adopt the comprehensive search only.

According to ***Characteristic 4***, numeric options that control buffers are most likely to cause CPBugs at their extreme or middle values. Hence, in NDP, we adopt taint tracking[7] using the option as the source while the buffer-related operations as the sink (e.g., `release_sysvar_source_service()` for MySQL), hence analyzing the code semantic to identify whether an option controls buffer. These sinks, which are system-dependent, are domain knowledge specified by software engineers. For all buffer-related numeric options, we force their search to follow the depth order of high-density.

For a given numeric option, NDP follows the steps below:

1) Pick a related version of the system under test.
2) If the numeric option controls buffer, then make it uses the order of search depths for high-density; otherwise, following the order of the assigned density level.
3) Test the option with the order of search depths via the heuristic generator for all workloads (see Section IV-E).
4) If a CPBug is found, then jump to 5); otherwise, return to 2) and move to the next search depth.
5) Repeat from 1) for the next related version, if any; otherwise, move to the next option.

E. Heuristic Generator

NDP can be paired with any heuristic algorithms for generating test cases for the numeric options. In this work, we use the population-based Genetic Algorithm (GA) [25], but it can be easily replaced by other algorithms. Since NDP tests one numeric option under a version each time, the solution representation is a pair of values for the tested option.

To determine which pairs to preserve, we use the fitness function below to compare the pairs:

$$fitness = \max_{i \in \{1 \dots m\}} |f(\mathbf{c_1}, w_i) - f(\mathbf{c_2}, w_i)| \tag{2}$$

whereby $\mathbf{c_1}$ and $\mathbf{c_2}$ are the configurations in a pair with different values on the numeric option to be tested; w_i denoted the ith workloads out of a total of m ones. Since based on the CPBug types, either configurations in the pair can be the source and the options would trigger CPBugs if the performance drops, this fitness reflects the maximum performance deviation between the two paired configurations with different values of the tested numeric option across different workloads—the larger the deviation, the higher possibility of triggering more CPBugs, which should be preserved in testing.

Under each of the above search depths, the search space of GA is bounded correspondingly. Whether a configuration in the pair starts from the upper or lower sides (for extreme

[6]Given the complexity of configurable systems, we have not seen a case with less than three peaks.

[7]The tracking (built on `LibASTMatchers` [24] for C/C++) is highly efficient: it takes a few seconds to around one minute for a system studied.

search); or whether it is searched on the middle range (for midmost search) is decided randomly. When GA consumes all of its budget or all pairs within the bound have been explored, NDP terminates the GA and checks whether a CPBug has been found according to the estimated CPBug type.

F. Handling Dependency

When changing the tested options, their dependencies need to be complied [26]. For example, in MySQL, there is a dependency that option innodb_buffer_pool_size (the buffer pool size) must be set as an integer product of that of the option innodb_buffer_pool_chunk_size (the granularity of buffer pool resizing) while being greater.

NDP leverages GPTuner [27]—a large language model-based tool that predicts configuration dependency based on the documentation. If, when a value of the tested option violates any dependency, we then (randomly) change the other affected option correspondingly. For example, if we change the value of innodb_buffer_pool_chunk_size to 128MB, then we should also set the innodb_buffer_pool_size to a value that is an integer product of 128MB, e.g., 128MB, 256MB, or 384MB. Note that, in that case, if either of the two options triggers CPBug, then both are CPBug-related with the same CPBug type. We chose GPTuner for two reasons:

- it is highly flexibility and can be conveniently used without any fine-tuning.
- thanks to the GPT3.5, it is generalizable to different systems. This is the key advantage compared with other rule-based tools such as cDep [28].

V. Experiments Setup

A. Research Questions

In this work, we answer the following research questions:

- **RQ1:** How well can NDP estimate the CPBug types?
- **RQ2:** How effective dose NDP in options prioritization against the state-of-the-art tools?
- **RQ3:** How well dose the prioritized search in NDP perform over the state-of-the-art tools?
- **RQ4:** Can NDP discover unknown CPBugs?

B. Systems, Versions, Workloads, and Known CPBugs

In this work, we use the datasets of 12 systems provided by He et al. [2] for assessing the CPBug type prediction. For the actual testing, we conduct experiments on five widely used configurable systems therein with known CPBugs, as shown in Table VIII. The reason is that we have not been able to reproduce the CPBugs for all 12 systems used by He et al. [2] because, e.g., the related versions are discarded; or the CPBugs have not been documented clearly. Yet, the five systems used for testing are still of diverse domains and scales, including database systems (i.e., MySQL and MariaDB), web servers (i.e., Apache), and compilers (i.e., Gcc and Clang).

To reproduce the CPBugs in testing, we test the options of each system under various versions. Note that not all the options would go through the same versions, since some do not exist in certain versions, hence NDP maintains a mapping between the options and related versions. We selected the related versions that can run successfully, including those that can produce the CPBugs in the ground truth and used previously [2]; as well as other stable versions that have not been discarded. In practice, we believe that software engineers would come with some domain knowledge about which versions are more likely to have CPBugs, or use all deployable versions that are of interest. Therefore, the selection of versions is case-dependent. NDP does not make assumptions on the nature and number of versions to be tested.

TABLE VIII: Configurable software with reproduced CPBugs.

Software	Version	$\mathcal{W}$	# CPBugs	Type-1	Type-2	Type-3	Type-4	Type-5	$\neg\mathcal{N}$	$\mathcal{N}$
MySQL	5.0 - 8.0	10	30	8	7	8	6	1	17	7
MariaDB	5.3 - 10.3	10	10	3	0	4	2	1	5	4
Apache	2.2 - 2.4	6	5	0	2	1	1	1	3	3
Gcc	3.4 - 7.3	2	15	1	10	0	2	2	16	0
Clang	3.2 - 5.0	2	6	0	6	0	0	0	7	0

An option might trigger multiple CPBugs, e.g., different values across different versions; Two options might also lead to a CPBug due to dependency. $\neg\mathcal{N}$ and $\mathcal{N}$ count the number of non-numeric and numeric CPBug-related options, respectively. $\mathcal{W}$ counts the number of workloads.

Each system is tested under different workloads generated by standard benchmarks. For MySQL and MariaDB, we use Sysbench—a powerful multi-threaded benchmark that is frequently employed [20], [29]—to generate 10 workloads that are of various data scales, number of concurrent threads, and test duration. For Apache, we use ApacheBench to create 6 workloads of different types. For Gcc and Clang, we use 2 standard programs with different types and scales. All above are important for revealing the CPBugs and have been used in prior work [20], [29], [30]. The workloads, combined with the versions, led to a high number of cases in the CPBug testing.

Derived from prior work [2], Table VIII shows that all systems/versions studied contain various known CPBugs, which are sufficiently complex to challenge CPBug testing tools.

C. Compared Approaches

Our experiments make comparisons with respect to the following state-of-the-art CPBug testing/prediction approaches:

- **CP-Detector (CPD) [2]**: A state-of-the-art tool that uses rule mining and keyword search to estimate the CPBug types. During the actual testing, it follows the greedy method with a random order of the tested options: fixing each tested option of a configuration in the pair at its minimal value and exponentially increasing the value of the same option for the other configuration.
- **Keyword Searching (KS)**: A baseline that predicts CPBugs type by keyword matching in the documentation.
- **Uniform Sampling (US) [16]**: A tool that samples uniformly on the values of the option in a pair with randomly sorted options to be tested. We set the same budget for each option as the GA in NDP, i.e., 100 tests, and the same way as NDP for CPBug type prediction.

Like NDP, when testing an option under a version, CPD and US also consider all workloads; stop whenever a CPBug is found or cover all pairs/exhaust the budget, then move to the next related version/option. We have omitted some other tools, e.g., Toddler [17], as they have been shown to be significantly inferior to CPD [2]. For **RQ1**, we use CPD and KS; while both CPD and US are used for the remaining RQs.

Note that, indeed, various sampling methods exist for testing other configuration issues [31]; however, CPBug testing differs from those as it considers different versions and workloads, making it too expensive to be tested by current sampling approaches (which more or less favor diversity).

D. Testing Budgets and Other Settings

In NDP, when testing numeric options, we set a budget of 100 tests for the heuristic generator (GA in this work). This means that, under a search depth, if the heuristic generator has consumed 100 tests (i.e., testing a pair under a workload for a system version would consume one test), then NDP would stop testing for the corresponding option. This is the same budget for US but not for CPD since it leverages a greedy search method. In contrast, the testing of non-numeric options is always exhaustive. All other settings of the compared approaches are left as default specified in their work.

For setting the GA, we use a mutation and crossover rate of 0.1 and 0.9, respectively, together with a soft population size capped at 10 (i.e., the number of pairs to explore might be less than 10 on the more restricted search depth). The crossover operator is a uniform crossover, i.e., one of the tested option's values in a pair might be swapped with that of the other; the mutation is a random mutation that randomly changes the tested option's value to a different permissible value. All above are standard settings from prior studies [15], [32], [33]. As for GKDE, we set all parameters as their default values.

For training the two neural language models in NDP, we use all the CPBugs data (including CPBug types) that have been previously reported [2] from different systems, except the system (and its versions) under test; unless otherwise stated. This is the same setup for the rule mining process in CPD [2].

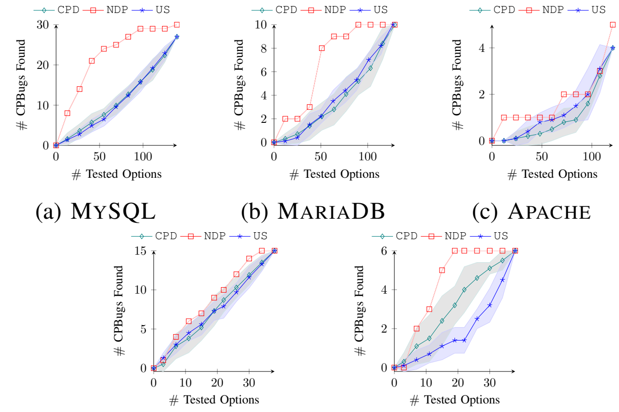
VI. Experimental Evaluation

A. RQ1: CPBugs Type Estimation

1) Method: To examine oracle prediction via **RQ1**, we compare NDP with CPD and KS under the same 500 samples from 12 systems used by He et al. [2] with no sampling method change, following the same training (fine-tuning)/testing splits for 10-fold cross-validation. He et al. [2] state that those are randomly sampled from the systems, but they have ensured data quality and representative nature. The mean recall, precision, and F1 scores for each of the five CPBug types are reported, i.e., a total of 15 types/metrics.

2) Results: From Table IX, we clearly see that the neural language model in NDP achieves considerably better results than the others, particularly on the F1 score, leading to superior results on 13 out of 15 types/metrics. In particular, the KS is clearly insufficient due to the limitation of a human-defined keywords sets; CPD is also restricted by the rule mining capability: due to the naturalness, the vast ways of describing the potential CPBugs in the documentation cannot be fully captured by the rules identified. Indeed, CPD marginally performs better than NDP on the precision of Type-3 and Type-4. However, this is mainly due to the fact that CPD tends not to include the samples in those two types, as naturally, their descriptions can be more complex. This has led to better precision (better false positive) but serenely comprised recall (worse false negative), which, together have worsened the F1 score in general. NDP, in contrast, can handle complex cases with good performance over both false positives/negatives, thanks to the reasoning ability in the latent space provided by the neural language model RoBERTa. Overall, we say:

NDP better predicts CPBugs type than the state-of-the-art approaches over 87% (13/15) types/metrics.



(a) MySQL (b) MariaDB (c) Apache

(d) Gcc (e) Clang

Fig. 4: Effectiveness of testing CPBugs over all options.

B. RQ2: Tested Option Prioritization

1) Method: To verify high-level option prioritization in **RQ2**, we use five systems (and their versions) for which we have successfully reproduced the CPBugs. We compare NDP against CPD and US, which test the options in random order. The mean/deviation of the cumulative number of CPBugs found with respect to the number of options tested for each system over 10 runs are reported. We also calculate the speedup of NDP via $\frac{o}{o'}$, where o is the number of options tested to find the most CPBugs by the other tool and o' is the number of tested options tested for NDP to achieve the same.

2) Results: As can be seen in Figure 4, NDP can reveal more CPBugs for MySQL and Apache due to the prioritization in testing numeric options (if we compare the last point), which we will evaluate in **RQ3**. More importantly, it runs CPBug testing with much better efficiency: we see that for all systems, NDP exhibits much stepper slops, meaning that many more CPBugs are discovered in an earlier stage of the testing—a significant contribution made by the prioritization at the tested options level. In particular, NDP achieves speedup range from 1.11× to 1.73× against both CPD and US.

Notably, testing a single option can be rather expensive, i.e., 2 hours on average; some can be days (since we need to go through many versions/workloads), hence there will be significant savings if we can find the same (or more) CPBugs

TABLE IX: Effectiveness of estimating CPBugs types. red cells denote the best-performing approach.

CPBug Type	Option Purpose	# Sample Options	Precision			Recall			F1 Score		
			NDP	CPD	KS	NDP	CPD	KS	NDP	CPD	KS
Type-1	Optimization	73	73.2%	69.4%	27.3%	71.2%	65.8%	18.8%	0.72	0.68	0.22
Type-2	Tradeoff	84	85.9%	70.7%	61.3%	72.6%	66.9%	21.4%	0.79	0.69	0.32
Type-3	Resource	143	92.2%	93.9%	69.4%	99.3%	92.4%	93.2%	0.96	0.93	0.80
Type-4	Functionality	100	78.4%	82.2%	56.2%	80.0%	55.6%	39.1%	0.79	0.66	0.46
Type-5	Non-influence	100	91.1%	90.1%	35.0%	93.0%	67.1%	70.0%	0.92	0.77	0.47

TABLE X: The least (average) tested options/clock time required to find per CPBug; red cells denote the best.

System	# Tested Options			Time (min)		
	NDP	CPD	KS	NDP	CPD	KS
MySQL	1.6	5.1	5.1	119.1	380.2	380.2
MariaDB	6.0	12.7	12.7	153.0	324.7	324.7
Apache	12.0	30.3	30.3	536.2	1351.6	1351.6
Gcc	2.1	2.5	2.3	18.0	21.0	19.4
Clang	3.0	5.5	6.3	25.2	46.2	53.2

by testing even slightly fewer options. It can be seen from the Table X, which shows the least tested options/clock time required to find per CPBug, that NDP only needs to test 1.6–12 options (18–536.2 minutes) against the 2.5–30.3 tested options (19.4–1351.6 minutes) for the state-of-the-art tools.

The reduced improvement of NDP for Gcc is due to its larger CPBugs ratio: 16 out of 38 options can trigger CPBugs. Indeed, since NDP speedups testing by prioritizing the options, clearly a high ratio of CPBugs can blur the benefits.

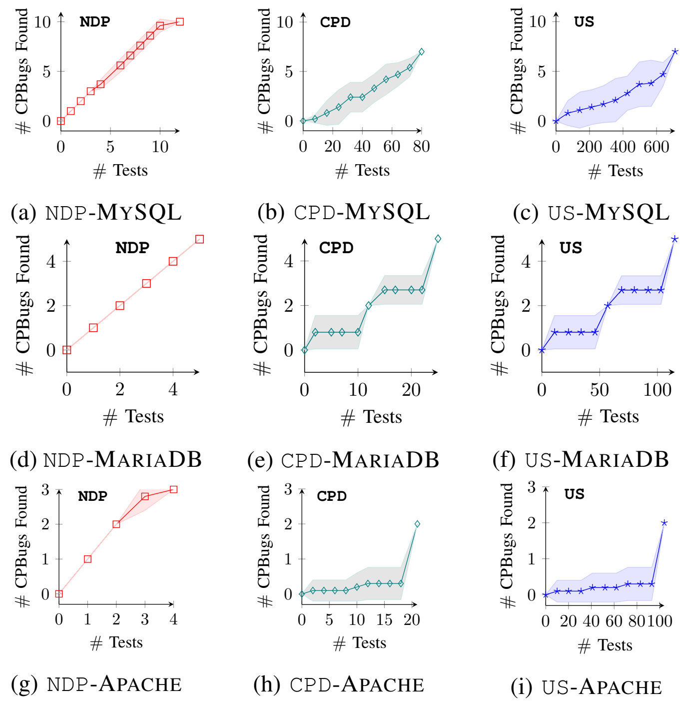
All those results suggest that:

NDP produces significantly better efficiency than state-of-the-art tools by prioritizing the order of tested options on all systems, achieving up to 1.73× speedup.

C. RQ3: Search Prioritization for Numeric Options

1) Method: In **RQ3**, we examine the search prioritization when testing numeric options on the systems/versions from **RQ2** (only MySQL, MariaDB, and Apache contain numeric options) against others. Since the number of tests—testing a pair of configurations is one test—is crucial for testing numeric options, for all systems, we report on the mean/deviation of the cumulative number of CPBugs found along with the number of tests for numeric options across 10 runs. All tools follow the same order of testing the numeric options prioritized by NDP: among all the numeric options of each system, NDP remarkably prioritizes the CPBug-related ones before the others. We measure the cumulative CPBugs found for every 10% tests (rounded) when the number of tests required is greater than 10; otherwise, we report every test. We calculate the speedup of NDP via the same way as for **RQ2**.

2) Results: Figure 5 shows the traces of testing numeric options. Clearly, we see that NDP exhibits remarkably better results compared with the others: it discovers the same (MariaDB) or more numeric options-related CPBugs (MySQL and Apache) than CPD and US, e.g., 10 for NDP while the other two can only find 7 CPBugs on MySQL.



(a) NDP-MySQL (b) CPD-MySQL (c) US-MySQL
(d) NDP-MariaDB (e) CPD-MariaDB (f) US-MariaDB
(g) NDP-Apache (h) CPD-Apache (i) US-Apache

Fig. 5: Testing CPBugs over numeric options.

In particular, NDP achieves such with significantly less number of tests—for all three systems, NDP does so with as few as 4–13 tests while CPD and US need 21–80 tests and 104–711 tests to reach their maximum number of CPBugs, respectively. Notably, to find the same maximum number of CPBugs as achieved by others, NDP has 3.13× to 10.50× and 14.38× to 88.88× speedup over CPD and US, respectively.

Since a single test run is highly expensive, the saving is thereby significant. Table XI shows the least number of tests/clock time required to find per CPBug: NDP only needs 1 test runs (as small as 1.1–4.4 minutes) against the 2.5–88.8 test runs (4.25–432.4 minutes) for the state-of-the-art tools.

All above demonstrate the effectiveness of the search level prioritization for numeric options in NDP. Thus, we conclude:

For all systems, NDP finds considerably more numeric options-related CPBugs than the state-of-the-art tools with 3.13× to 88.88× speedup by prioritizing the search.

D. RQ4: Detecting New CPBugs

1) Method: To verify whether NDP can reveal unknown CPBugs, we apply NDP to further extended sets of versions

TABLE XI: The least (average) test counts/clock time required to find per CPBug on numeric options; red cells are the best.

System	# Test Counts			Time (min)		
	NDP	CPD	KS	NDP	CPD	KS
MySQL	1.0	7.6	88.8	4.4	37.0	432.4
MariaDB	1.0	2.5	23.0	1.1	4.25	39.1
Apache	1.0	10.5	52.0	4.2	44.1	218.4

TABLE XII: New CPBugs discovered by NDP.

CPBug	System	Version	Performance Degradation	CPBug Type
#Pending (link)	Gcc	v12	1.12× Execution Time	Type-4
#Pending (link)	Gcc	v9	2.75× Execution Time	Type-2
#Pending (link)	Gcc	v12	1.06× Execution Time	Type-2
#Pending (link)	Gcc	v12	1.10× Compiling Time	Type-2
#Pending (link)	Gcc	v12	1.17× Execution Time	Type-2
#Pending (link)	Gcc	v9	1.14× Compiling Time	Type-2
#Pending (link)	Gcc	v12	1.17× Compiling Time	Type-2
#Pending (link)	Gcc	v9,v12	1.06–1.10× Execution Time	Type-2
#Pending (link)	Gcc	v9,v12	1.31× File Size	Type-2
#117992(1)	Clang	v14	1.09× Compiling Time	Type-2
#117992(2)	Clang	v9,v14	1.06× Compiling Time	Type-2
#117993	Clang	v9,v14	1.06–1.13× Execution Time	Type-2

We published the new Gcc bugs on our repository since Gcc has stopped bug reporting.

compared with those used for **RQ1–RQ3**, and left the testing runs. For any CPBugs discovered, we also compute the measured performance drop by setting the target option value against the performance obtained via the source option value.

2) Results: From Table XII, we see that NDP has successfully discovered 12 CPBugs that are previously unknown on Gcc and Clang, which we have reported. These CPBugs can lead to significant performance impact, e.g., ranging between 1.06×–2.75× and 1.06×–1.17× degradations on the execution time and compiling time, respectively. As such, we say that:

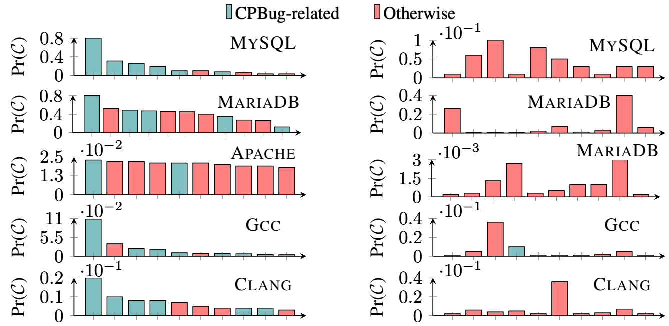
NDP can discover previously unknown CPBugs given sufficient resources and versions/workloads.

VII. Discussion: Why NDP Works?

RQ1–RQ3 serve as the ablation analysis of NDP. Here, we further explain why NDP work with a qualitative analysis.

A. Predicting CPBugs Types

A key benefit of the neural language model in NDP is the significant reduction of false negatives compared with CPD and KS. For example, option innodb_fill_factor for MySQL has the description of "*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. For example, setting innodb_fill_factor to 80 reserves 20 percent of the space on each B-tree page for future index growth...*". This option should belong to *Type-2* since both a too small or a too large value could downgrade the performance as the former creates many recursions while the latter processes too many pages. Yet, for consistency, a smaller value is preferred since fewer pages need to be maintained, hence there is a trade-off. However, the description has no clear pattern to indicate such, hence both CPD and KS have wrongly classified it as *Type-3* due to the presence of the word "space". NDP, in contrast, has correctly estimated the option since it has learned that texts with "B-tree" and "page" are likely to be related to trade-offs.



(a) Prioritization in NDP (b) Random in CPD and US

Fig. 6: Order of options to be tested for all systems (left to right). Pr($\mathcal{C}$) denotes the probability of being CPBug-related.

B. Prioritizing Options

To understand why prioritizing at the options level helps NDP to significantly improve the testing efficiency, Figure 6 plots the first few options to be tested by all approaches. We see that, with NDP, the prioritized options generally have higher probabilities of being CPBug-related than those of the other two, within which up to 60% options can trigger CPBugs (for MySQL) while only one option from CPD and US can do so (for Gcc). This considerably impacts the CPBug testing.

C. Prioritizing Search

We also analyze why prioritizing the different bounds of search space when testing the numeric options can help. We found that the extreme search and mid-most search are of great benefit therein. For example, innodb_buffer_pool_size is a CPBug-related numeric option on MySQL. Yet, such a CPBug can only be discovered when we change it from a near-minimal value, i.e., 10MB, to one that is closer to its maximum value, i.e., 256GB. With NDP, such an option is categorized as the high-density region (***Characteristic 2*** and ***4***), thus NDP would prioritize its bounds as extreme search first. This fits perfectly with its range of values that causes a CPBug, since with the extreme search, a configuration in a pair would be explored within 10% close to its minimum value while the other would take a value close to 10% of its maximum extreme. In contrast, CPD would fix one configuration to the option value of 1MB while increasing the other as 2MB, 4MB, and 8MB, etc, each pair of which needs to be tested. Unlike the others, US does not use a heuristic as it aims to sample randomly and uniformly. Therefore, for options like innodb_buffer_pool_size, NDP needs significantly less number of tests to reveal the CPBugs compared with the others, which might even fail to find the CPBugs due to exhaustion of budget.

VIII. Threats to Validity

Threats to internal validity: We set the parameters either adopting pragmatic values or following widely-used defaults, e.g., the inner budget of 100 tests for GA under a bound is a pragmatic setting, achieving a good balance between quality and cost. For confirming performance drop in the oracle, we set a minimum of 5% change as prior work [19]. However, we agree that some settings might not be the best.

Threats to external validity: For evaluating the estimation of CPBugs types, we use prior datasets of 12 systems [2]. For testing the CPBugs, we use five systems with reproduced CPBugs. In both cases, the systems are of diverse languages, domains, and scales. We have also considered a wide range of workloads (2–10) and versions (4–31), which are the most commonly used ones from existing work [2], [34]–[36]. Indeed, more subjects might strengthen the conclusion.

Threats to construct validity: We use several metrics, including precision, recall, and F1 score, together with the trajectory of finding CPBugs and the best efficiency of each approach. Yet, unintended programming errors or misconsiderations are always possible.

IX. Related Work

A. Implication of Configuration to Performance Issues

A vast amount of early work has been conducted to understand the implications of configurations for performance issues. For example, Jin et al. [37] and Han et al. [38] reveal that 59% of the performance problems can be traced back to configuration errors. Xiang et al. [39] further suggest that configuration option documentation is a significant resource for analyzing configuration-related performance expectations, which serve as a foundation for identifying CPBug oracle. Those studies provide insights into how configuration caused performance issues while NDP automatically testing CPBugs.

B. Performance Bug Testing

There exist tools that detect general performance bugs using a fixed set of patterns, such as loops and memory access [17], [40]–[44]. To tackle unforeseen bottleneck patterns, Shen et al. [45] propose a GA-based testing framework with contrast data mining. However, they are not related to configurations.

Among configuration-related testing approaches, `ctest` [8] is a tool that leverages existing regression testing code to prioritize the execution of test cases for misconfiguration-related performance issues. `DiagConfig` [19] leverages static code analysis and machine learning to detect performance bugs caused by misconfiguration. Yet, they aim for misconfiguration, which is user-induced performance issues while NDP reveal CPBugs—the configuration performance issues that are unintentionally introduced by the developers of configurable systems. This work also advances CPD [2]—a state-of-the-art CPBug testing tool—in several aspects:

- We additionally summarize ***Characteristic 3*** (on the commonality of median range value for CPBug-related numeric options) and ***Characteristic 4*** (on the more detailed categorization of CPBug-related numeric options), which have not been revealed by the work of CPD.
- CPD predicts the oracle using rule mining and keyword search while NDP does so via a RoBERTa, fine-tuned by configuration documentation. This, as shown in Section VI-A, has led to much superior accuracy.
- CPD does not prioritize the options to be tested and a numeric option is tested by fixing it in one configuration as maximal/minimal value, while exponentially changing the value of the same option in the other configuration. In contrast, through exploiting the other RoBERTa fine-tuned by both documentation and code, NDP designs dual-level prioritization that (1) prioritizes the options that are more likely to cause CPBugs to be tested first while (2) stochastically exploring the values of a numeric option in the pair using differently prioritized search bounds, according to the likelihood of the option being CPBugs-related and the observations from the ***Characteristics 2–4***. This has resulted in considerably improved efficiency.

C. Configuration Performance Tuning

Unlike testing for configuration-related performance bugs, configuration performance tuning aims to find the best configuration that reaches the optimal performance for deployment time benchmarking [15], [32], [46]–[49] or runtime self-adaptation [46], [50]–[55]. Among others, FLASH [47] and BOCA [48] are tuners based on Bayesian optimization to find optimal configuration. Chen and Li propose MMO [15], [32], [49]—an alternative way to tune configuration via tuner-agnostic multi-objectivization.

Yet, configuration tuning differs from the testing that NDP focus on in several aspects: the representation in configuration tuning is often a single configuration while for CPBug testing, we need to test a pair of configurations for revealing whether the actual performance matches the expectation. Further, CPBug testing examines each option in turn while configuration tuning changes several options simultaneously.

X. Conclusion

This paper presents NDP, a general framework aiming to expedite CPBug testing via neural dual-level prioritization. NDP builds two neural language models for estimating the CPBugs types oracle and inferring the probabilities of the options being CPBug-related, respectively. These models serve as the foundation for prioritization at two levels— prioritizing the order of tested options and the order of search bounds for numeric options. Experiments on several real-world systems and against state-of-the-art tools/approaches reveal that:

- NDP estimates more reliable oracle of CPBug types;
- while significantly expedites the testing at both the options level and the search level for numeric options, with up to 1.73× and 88.88× speedup, respectively.

For future work, the static handling of workloads in NDP can also be prioritized, placing the more vulnerable ones to be tested first. Extending NDP to detect CPBugs by testing multiple options simultaneously is also fruitful.

Acknowledgement

This work was supported by a NSFC Grant (62372084) and a UKRI Grant (10054084).

REFERENCES

[1] J. Gong and T. Chen, "Deep configuration performance learning: A systematic survey and taxonomy," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 1, Dec. 2024. [Online]. Available: https://doi.org/10.1145/3702986

[2] H. He, Z. Jia, S. Li, E. Xu, T. Yu, Y. Yu, J. Wang, and X. Liao, "Cp-detector: Using configuration-related performance properties to expose performance bugs," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 623–634. [Online]. Available: https://doi.org/10.1145/3324884.3416531

[3] P. Chen, J. Gong, and T. Chen, "Accuracy can lie: On the impact of surrogate model in configuration tuning," *IEEE Trans. Software Eng.*, 2025.

[4] J. Gong, T. Chen, and R. Bahsoon, "Dividable configuration performance learning," *IEEE Trans. Software Eng.*, vol. 51, no. 1, pp. 106–134, 2025.

[5] J. Gong and T. Chen, "Predicting software performance with divide-and-learn," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*. ACM, 2023, pp. 858–870. [Online]. Available: https://doi.org/10.1145/3611643.3616334

[6] T. Chen and M. Li, "Do performance aspirations matter for guiding software configuration tuning? an empirical investigation under dual performance objectives," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 3, pp. 68:1–68:41, 2023. [Online]. Available: https://doi.org/10.1145/3571853

[7] J. Gong and T. Chen, "Predicting configuration performance in multiple environments with sequential meta-learning," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 359–382, 2024. [Online]. Available: https://doi.org/10.1145/3643743

[8] R. Cheng, L. Zhang, D. Marinov, and T. Xu, "Test-case prioritization for configuration testing," in *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, C. Cadar and X. Zhang, Eds. ACM, 2021, pp. 452–465. [Online]. Available: https://doi.org/10.1145/3460319.3464810

[9] M. Velez, P. Jamshidi, N. Siegmund, S. Apel, and C. Kästner, "On debugging the performance of configurable software systems: Developer needs and tailored tool support," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1571–1583. [Online]. Available: https://doi.org/10.1145/3510003.3510043

[10] T. Wang, Z. Jia, S. Li, S. Zheng, Y. Yu, E. Xu, S. Peng, and X. Liao, "Understanding and detecting on-the-fly configuration bugs," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 628–639. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00062

[11] M. Nygard, "Release it!: design and deploy production-ready software," 2018.

[12] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2018. [Online]. Available: https://doi.org/10.2200/S00874ED3V01Y201809CAC046

[13] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic configuration management at facebook," in *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, E. L. Miller and S. Hand, Eds. ACM, 2015, pp. 328–343. [Online]. Available: https://doi.org/10.1145/2815400.2815401

[14] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, E. D. Nitto, M. Harman, and P. Heymans, Eds. ACM, 2015, pp. 307–319. [Online]. Available: https://doi.org/10.1145/2786805.2786852

[15] T. Chen and M. Li, "Adapting multi-objectivized software configuration tuning," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 539–561, 2024. [Online]. Available: https://doi.org/10.1145/3643751

[16] Q. Plazar, M. Acher, G. Perrouin, X. Devroey, and M. Cordy, "Uniform sampling of SAT solutions for configurable systems: Are we there yet?" in *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. IEEE, 2019, pp. 240–251. [Online]. Available: https://doi.org/10.1109/ICST.2019.00032

[17] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: detecting performance problems via similar memory-access patterns," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013, pp. 562–571. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606602

[18] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[19] Z. Chen, P. Chen, P. Wang, G. Yu, Z. He, and G. Mai, "Diagconfig: Configuration diagnosis of performance violations in configurable software systems," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 566–578. [Online]. Available: https://doi.org/10.1145/3611643.3616300

[20] S. Mühlbauer, F. Sattler, C. Kaltenecker, J. Dorn, S. Apel, and N. Siegmund, "Analysing the impact of workloads on modeling the performance of configurable software systems," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2085–2097. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00176

[21] K. I. Roumeliotis, N. D. Tselikas, and D. K. Nasiopoulos, "Llms in e-commerce: A comparative analysis of GPT and llama models in product review evaluation," *Nat. Lang. Process. J.*, vol. 6, p. 100056, 2024. [Online]. Available: https://doi.org/10.1016/j.nlp.2024.100056

[22] T. Zhang, B. Xu, F. Thung, S. A. Haryono, D. Lo, and L. Jiang, "Sentiment analysis for software engineering: How far can pre-trained transformer models go?" in *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 2020, pp. 70–80. [Online]. Available: https://doi.org/10.1109/ICSME46990.2020.00017

[23] P. Jamshidi and G. Casale, "An uncertainty-aware approach to optimal configuration of stream processing systems," in *24th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2016, London, United Kingdom, September 19-21, 2016*. IEEE Computer Society, 2016, pp. 39–48. [Online]. Available: https://doi.org/10.1109/MASCOTS.2016.17

[24] https://clang.llvm.org/docs/LibASTMatchers.html.

[25] S. Forrest, "Genetic algorithms," *ACM computing surveys (CSUR)*, vol. 28, no. 1, pp. 77–80, 1996.

[26] H. Liang, Y. Huang, and T. Chen, "The same only different: On information modality for configuration performance analysis," in *47th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2025.

[27] J. Lao, Y. Wang, Y. Li, J. Wang, Y. Zhang, Z. Cheng, W. Chen, M. Tang, and J. Wang, "Gptuner: A manual-reading database tuning system via gpt-guided bayesian optimization," *Proc. VLDB Endow.*, vol. 17, no. 8, pp. 1939–1952, 2024. [Online]. Available: https://www.vldb.org/pvldb/vol17/p1939-tang.pdf

[28] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu, "Understanding and discovering software configuration dependencies in cloud and datacenter systems," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 362–374. [Online]. Available: https://doi.org/10.1145/3368089.3409727

[29] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, E. D. Nitto, M. Harman, and P. Heymans, Eds. ACM, 2015, pp. 284–294. [Online]. Available: https://doi.org/10.1145/2786805.2786845

[30] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. S. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds.

IEEE Computer Society, 2012, pp. 167–177. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227196

[31] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 643–654. [Online]. Available: https://doi.org/10.1145/2884781.2884793

[32] P. Chen, T. Chen, and M. Li, "MMO: meta multi-objectivization for software configuration tuning," *IEEE Trans. Software Eng.*, vol. 50, no. 6, pp. 1478–1504, 2024. [Online]. Available: https://doi.org/10.1109/TSE.2024.3388910

[33] T. Chen and M. Li, "The weights can be harmful: Pareto search versus weighted search in multi-objective search-based software engineering," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, pp. 5:1–5:40, 2023. [Online]. Available: https://doi.org/10.1145/3514233

[34] S. Tongkaw and A. Tongkaw, "A comparison of database performance of mariadb and mysql with oltp workload," in *2016 IEEE conference on open systems (ICOS)*. IEEE, 2016, pp. 117–119.

[35] C. Curino, E. P. C. Jones, S. Madden, and H. Balakrishnan, "Workload-aware database monitoring and consolidation," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, Eds. ACM, 2011, pp. 313–324. [Online]. Available: https://doi.org/10.1145/1989323.1989357

[36] L. Lesoil, M. Acher, A. Blouin, and J. Jézéquel, "Input sensitivity on the performance of configurable systems an empirical study," *J. Syst. Softw.*, vol. 201, p. 111671, 2023. [Online]. Available: https://doi.org/10.1016/j.jss.2023.111671

[37] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, J. Vitek, H. Lin, and F. Tip, Eds. ACM, 2012, pp. 77–88. [Online]. Available: https://doi.org/10.1145/2254064.2254075

[38] X. Han and T. Yu, "An empirical study on performance bugs for highly configurable software systems," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2016, Ciudad Real, Spain, September 8-9, 2016*. ACM, 2016, pp. 23:1–23:10. [Online]. Available: https://doi.org/10.1145/2961111.2962602

[39] C. Xiang, H. Huang, A. Yoo, Y. Zhou, and S. Pasupathy, "Pracextractor: Extracting configuration good practices from manuals to detect server misconfigurations," in *Proceedings of the 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, A. Gavrilovska and E. Zadok, Eds. USENIX Association, 2020, pp. 265–280. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/xiang

[40] L. Song and S. Lu, "Performance diagnosis for inefficient loops," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE / ACM, 2017, pp. 370–380. [Online]. Available: https://doi.org/10.1109/ICSE.2017.41

[41] P. Su, S. Wen, H. Yang, M. Chabbi, and X. Liu, "Redundant loads: a software inefficiency indicator," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 982–993. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00103

[42] J. Yang, P. Subramaniam, S. Lu, C. Yan, and A. Cheung, "How *not* to structure your database-backed web applications: a study of performance bugs in the wild," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 800–810. [Online]. Available: https://doi.org/10.1145/3180155.3180194

[43] T. Yu and M. Pradel, "Syncprof: detecting, localizing, and optimizing synchronization bottlenecks," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, A. Zeller and A. Roychoudhury, Eds. ACM, 2016, pp. 389–400. [Online]. Available: https://doi.org/10.1145/2931037.2931070

[44] E. Coppa, C. Demetrescu, and I. Finocchi, "Input-sensitive profiling," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, J. Vitek, H. Lin, and F. Tip, Eds. ACM, 2012, pp. 89–98. [Online]. Available: https://doi.org/10.1145/2254064.2254076

[45] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik, "Automating performance bottleneck detection using search-based application profiling," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, M. Young and T. Xie, Eds. ACM, 2015, pp. 270–281. [Online]. Available: https://doi.org/10.1145/2771783.2771816

[46] Y. Ye, T. Chen, and M. Li, "Distilled lifelong self-adaptation for configurable systems," in *47th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2025.

[47] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel, "Finding faster configurations using FLASH," *IEEE Trans. Software Eng.*, vol. 46, no. 7, pp. 794–811, 2020. [Online]. Available: https://doi.org/10.1109/TSE.2018.2870895

[48] J. Chen, N. Xu, P. Chen, and H. Zhang, "Efficient compiler autotuning via bayesian optimization," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 1198–1209. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00110

[49] T. Chen and M. Li, "Multi-objectivizing software configuration tuning," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 453–465. [Online]. Available: https://doi.org/10.1145/3468264.3468555

[50] T. Chen, "Planning landscape analysis for self-adaptive systems," in *International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2022, Pittsburgh, PA, USA, May 22-24, 2022*, B. R. Schmerl, M. Maggio, and J. Cámara, Eds. ACM/IEEE, 2022, pp. 84–90. [Online]. Available: https://doi.org/10.1145/3524844.3528060

[51] Tao Chen, "Lifelong dynamic optimization for self-adaptive systems: Fact or fiction?" in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 2022, pp. 78–89. [Online]. Available: https://doi.org/10.1109/SANER53432.2022.00022

[52] S. Kumar, T. Chen, R. Bahsoon, and R. Buyya, "DATESSO: self-adapting service composition with debt-aware two levels constraint reasoning," in *SEAMS '20: IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Seoul, Republic of Korea, 29 June - 3 July, 2020*, S. Honiden, E. D. Nitto, and R. Calinescu, Eds. ACM, 2020, pp. 96–107. [Online]. Available: https://doi.org/10.1145/3387939.3391604

[53] T. Chen, "All versus one: an empirical comparison on retrained and incremental machine learning for modeling performance of adaptable software," in *Proceedings of the 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, M. Litoiu, S. Clarke, and K. Tei, Eds. ACM, 2019, pp. 157–168. [Online]. Available: https://doi.org/10.1109/SEAMS.2019.00029

[54] T. Chen, K. Li, R. Bahsoon, and X. Yao, "FEMOSAA: feature-guided and knee-driven multi-objective optimization for self-adaptive software," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 2, pp. 5:1–5:50, 2018. [Online]. Available: https://doi.org/10.1145/3204459

[55] T. Chen and R. Bahsoon, "Self-adaptive trade-off decision making for autoscaling cloud-based services," *IEEE Trans. Serv. Comput.*, vol. 10, no. 4, pp. 618–632, 2017. [Online]. Available: https://doi.org/10.1109/TSC.2015.2499770