ConfProf: White-Box Performance Profiling of Configuration Options

Xue Han
University of Southern Indiana
Indiana, USA
xhan@usi.edu

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

Michael Pradel
University of Stuttgart
Stuttgart, Germany
michael@binaervarianz.de

ABSTRACT

Modern software systems are highly customizable through configuration options. The sheer size of the configuration space makes it challenging to understand the performance influence of individual configuration options and their interactions under a specific usage scenario. Software with poor performance may lead to low system throughput and long response time. This paper presents ConfProf, a white-box performance profiling technique with a focus on configuration options. ConfProf helps developers understand how configuration options and their interactions influence the performance of a software system. The approach combines dynamic program analysis, machine learning, and feedback-directed configuration sampling to profile the program execution and analyze the performance influence of configuration options. Compared to existing approaches, ConfProf uses a white-box approach combined with machine learning to rank performance-influencing configuration options from executions traces. We evaluate the approach with 13 scenarios of four real-world, highly-configurable software systems. The results show that ConfProf ranks performance-influencing configuration options with high accuracy and outperform a state of the art technique.

CCS CONCEPTS

• **Software and its engineering** → *Software notations and tools.*

KEYWORDS

Performance Profiling, Software Performance

ACM Reference Format:
Xue Han, Tingting Yu, and Michael Pradel. 2021. ConfProf: White-Box Performance Profiling of Configuration Options. In *Proceedings of the 2021 ACM/SPEC International Conference on Performance Engineering (ICPE '21), April 19–23, 2021, Virtual Event, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3427921.3450255

1 INTRODUCTION

Modern software systems are highly-configurable. Users can customize a large number of configuration options to change program functionalities. The complexity of the configuration space and the sophisticated interactions among configuration options can easily cause performance issues. A typical configurable software system may have thousands of configuration options, and this generates numerous possible configurations. Unfortunately, developers often do not know how configuration options and their interactions influence the performance of a system [12].

Prior work has examined the prevalence of configuration issues that have led to performance problems. Han et al. [12] found that more than half of the performance problems (59%) are due to configuration issues. Figure 1 shows a real-world, configuration-related performance bug in Apache. When a user sets a large value for the configuration option `StartServers` (e.g., 60), restarting Apache takes a longer time to complete than usual. The root cause of this bug is an expensive method call dummy_connection() (line 6) inside a for loop (line 2). This dummy_connection() method invokes system calls such as poll() and select() to wake Apache child server processes. An if statement (line3) is added to the for loop to fix this bug. The if statement checks the status of child server processes (line 4) before invoking the dummy_connection() method (line 6) to wake child server processes.

```
1  ap_mpm_pod_killpg(ap_pod_t *pod, int num){
2   for (i=0;i<num && rv==APR_SUCCESS;i++) {
3  + if(ap_scoreboard_image->servers[i][0].pid == 0 ||
4  +  ap_scoreboard_image-> servers[i][0].status!=SERVER_READY)
5  +  continue;
6     rv=dummy_connection(pod);}}
```

Figure 1: Apache Bug #54852

This paper presents ConfProf, a white-box performance profiling approach that analyzes and ranks configuration options for highly-configurable software systems. ConfProf chooses a white-box performance profiling approach over the black-box approach so that developers can pinpoint inefficient, configuration-dependent code locations. ConfProf consists of two major phases. In phase I, the approach identifies individual code locations for which performance depends on configuration options. To this end, ConfProf gathers execution profiles with different configuration option values and infers a complexity model that uses the option values to predict the execution cost of a performance-sensitive code location [19] such as loops and system calls. The intuition is that such code locations are more likely to cause performance problems [3, 14]. In phase II, ConfProf summarizes the performance impact of each configuration option across all performance-sensitive code locations to report a ranked list of performance-influencing options. The rank can help developers to identify which configuration options have the highest performance impact on the subject program.

ICPE '21, April 19–23, 2021, Virtual Event, France

ACM ISBN 978-1-4503-8194-9/21/04…$15.00
https://doi.org/10.1145/3427921.3450255

We envision to apply ConfProf in at least three cases. First, a developer who may not be aware of performance-influencing configuration options in a program can use ConfProf to rank the configuration options in terms of their performance impact. ConfProf differs from prior work [4, 14, 31, 36] on performance profiling by considering the configuration space beyond the default configuration of a software system. Similar to existing profiling techniques, ConfProf is based on dynamic analysis and therefore limited to observing the executions triggered by a given set of inputs. The problem of finding suitable inputs for performance analysis [6, 8, 35] is orthogonal to the issue addressed here.

Second, developers can use ConfProf to pinpoint code locations for which the performance depends on a configuration option value. ConfProf guides developers toward understanding configuration-related performance issues by identifying performance-influencing code locations. An alternative way to identify configuration options that are relevant to specific code locations is to use static analysis [16]. However, such techniques work best for single-language, self-contained systems with source code [1]. In contrast, ConfProf is a dynamic approach in which the identification of performance-influencing configuration options is from program execution profiles. Thus, ConfProf can scale with large and heterogeneous software systems without source code.

Third, ConfProf can help developers and researchers who build performance prediction models for a software system using just configuration options [29]. They can combine existing performance modeling techniques with ConfProf by sampling only performance-influencing configuration options identified by ConfProf.

To evaluate the effectiveness of ConfProf, we apply the approach to four popular real-world C/C++ programs. Our results show that ConfProf effectively identifies performance-influencing configuration options. In summary, this paper makes the following contributions:

- An automated white-box and dynamic performance analysis approach that ranks the performance influence of configuration options for highly-configurable software systems.
- A technique that associates specific code locations with configuration options that have a high-performance influence on the subject program.
- A practical approach with open-source implementation toolsets that works in real-world C/C++ programs.

2 BACKGROUND

This section provides definitions and background on configurable software systems and performance bugs.

2.1 Definitions

A *configurable software system S* consists of a set of configuration options *OPT* including both numeric options and binary options. The function $v(O_i)$ represents the current value of a configuration option O_i. A *usage scenario* of *S* corresponds to a major function in a software system. ConfProf aims to profile the performance influence of configuration options on a given usage scenario. For example, typical usage scenarios of Parallel BZIP2 (PBZIP2) are compressing data and decompressing data. Each usage scenario associates with a set of configuration options OPT', where $OPT' \subseteq OPT$. The configuration options used in one scenario may not apply to another scenario. For instance, PBZIP2 uses the configuration option `-z` to compress data, whereas the configuration option `-d` is used for decompressing data. Since performance bugs often require specific workloads to manifest, we identify the workload for each usage scenario accordingly. For example, serving HTTP requests is one usage scenario in the Apache server. The number of concurrent HTTP requests is one type of workload in this usage scenario.

2.2 Configuration-Related Performance Bugs

Prior work has studied the challenges for handling performance bugs in highly-configurable software systems [12]. The study reports that more than half of 193 studied performance bugs (59%) are triggered by configuration options. There are many cases in which a misconfiguration causes poor software performance. The root cause can fall into two categories. The first category is software bugs due to coding errors [18, 19]. The software bug in Figure 1 is an example of this category. The second category of configuration-related performance bugs is system environment-specific related to hardware, system topology, and the choice of system core libraries. For example, in Apache Bug #45834, a misconfiguration of the firewall cuts authentication communications, which freezes the system. A prior study [12] shows that the system environment-specific configurations performance bugs account for only a small portion (8% to 17%) of all problems. Therefore, in this paper, we focus on performance coding errors, i.e., the first category.

3 APPROACH

This section presents ConfProf, a performance profiling approach that helps developers understand how configuration options influence the performance of a system. Figure 2 gives an overview of the approach. The input to ConfProf is a configurable program and a usage scenario that exercises the program.

ConfProf consists of two phases. In phase I, ConfProf analyzes how the performance of individual code locations, i.e., loops and system calls (e.g., write(), poll(), and select()) depend on configuration options. For illustration purposes, we discuss the approach with loops only. To this end, ConfProf collects execution profiles for different configurations and infers *code location-level complexity models* (*location-level* models for short). A location-level model describes the execution time spent at the code location under different values of a single or interacting configuration options. For example, the model $m_{L,O_i} = k * v(O_i)$ describes that the execution time in a code location (e.g., a loop) is linearly dependent on the value of a configuration option O_i, where the coefficient k is a constant. In phase II, ConfProf summarizes the location-level models for each configuration option obtained from phase I and computes its performance impact. The output of phase II includes both a ranked list of individual options and a ranked list of interacting options. A high ranking option has a stronger influence on software system performance. To illustrate the idea, consider the example below.

O_1	Perf(L_1)	Perf(L_2)	Perf(L_1, L_2)
1	10	100000	100010
2	20	100000	100020
3	30	100000	100030
4	40	100000	100040

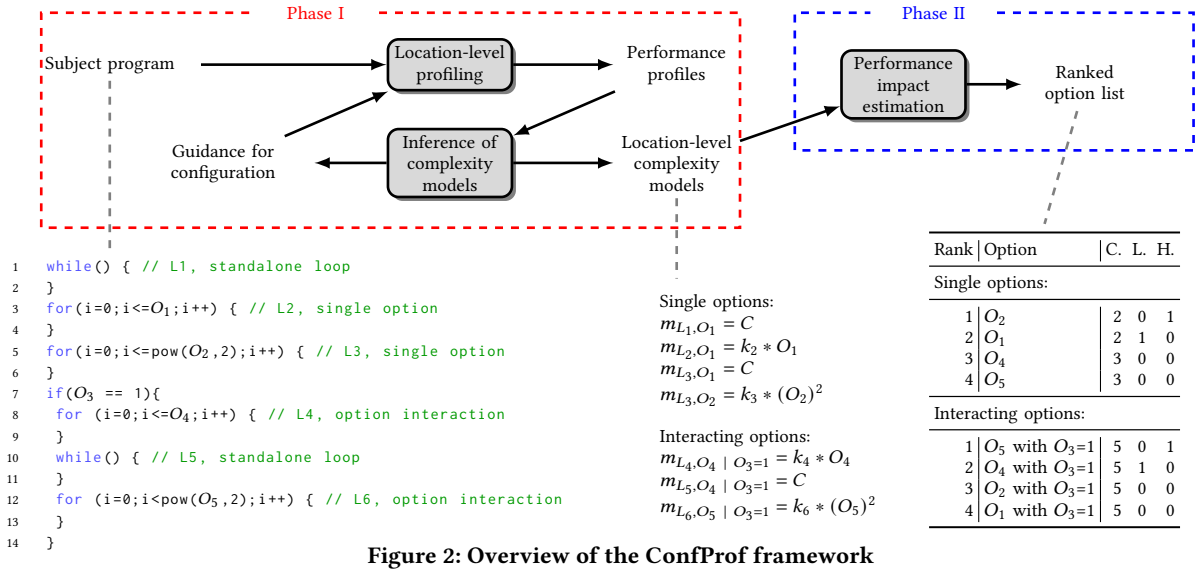

Figure 2: Overview of the ConfProf framework

Column one indicates the values of option O_1. Column two and column three indicate the location-level performance measures for code locations L_1 and L_2, and the last column indicates the overall performance measurements using a black-box approach. Suppose that the current sample data for O_1 is 1, 2, 3, and 4, ConfProf infers that the complexity model of L_1 is a linear (L.) model $m_{L_1,O_1} = 10 * O_1$ and the complexity model of L_2 is a constant (C.) model $m_{L_2,O_1} = 100000$. ConfProf concludes that O_1 has a performance impact on the system because it has a positive linear relationship with L_1. As the value of O_1 increases, it will influence the overall system performance. In contrast to the current sample data, O_1 has a nearly constant relationship with the overall performance measurements and a black box approach would consider it to have no performance impact on the system.

We use the example in the lower part of Figure 2 to illustrate ConfProf in the rest of this paper. This example contains five configuration options, i.e., $O_1 - O_5$. Option O_3 is a binary configuration option, and others are numeric configuration options. The default value of each option is 0.

3.1 Inferring Complexity Models

In phase I, ConfProf analyzes how the performance of individual code locations depends on the configuration options. We focus on loops and system calls as code locations in this work because they often influence the performance of a system in significant ways, and they are at the core of various performance problems [14, 28].

3.1.1 Performance Measurements. Given a usage scenario and a set of configuration options, ConfProf measures the performance of each loop and system call in a program using the wall-clock execution time. Recent work on performance modeling [29] and performance bug detection [38] techniques also used wall-clock execution time as performance measurements. While other performance measurements, such as the number of executed conditional instructions (e.g., direct/indirect calls and direct/indirect branches) or the number of instruction counts, can also be used [24, 27], they may not be representative of the actual performance.

3.1.2 Location-Level Complexity Models. Based on performance measurements obtained for different configurations, ConfProf infers a location-level model for performance-sensitive code locations, i.e., loops and system calls.

Definition 3.1 (Complexity Model). A complexity model m is a function that predicts the execution time of a code location l using one or more configuration options. We consider three kinds of complexity models:

- A *constant model* $m_L = C$ expresses the performance cost of the measured code location L is C, that is, it is independent of any options.
- A *single-option model* $m_L = f(O_i)$ expresses that the performance cost of the measured code location L is a function of an option O_i.
- A *model of interacting options* $m_{L \mid O_b=1} = f(O_i)$ expresses that the performance cost of the measured code location L is a function of the option O_i given that the binary option O_b is enabled.

To infer the function f of such a model, ConfProf analyzes a sequence of performance measurements. Specifically, the inference takes a sequence of performance measurements $p_1, .., p_k$, where each measurement is gathered with a different configuration value $v_1, .., v_k$ for a configuration option O_i. We explain below how ConfProf obtains this sequence of performance measurements. Given the sequence, the approach uses machine learning models to fit the performance measurements. To that end, configuration options are learning features, and performance measurements are learning labels. ConfProf fits the data points (v_k, p_k) to linear regression models and non-linear complexity models. Specifically, ConfProf uses linear regression (LR) to learn the linear complexity models and both the support vector machines (SVM) and multilayer perceptron (MP) to learn the non-linear complexity model [23]. The

mean absolute error (MAE) is widely used to assess model performance [7, 37]. ConfProf computes MAE and associates the code location with the model that has the lowest MAE.

3.1.3 Feedback-Driven Profiling and Model Inference. To infer complexity models, ConfProf uses performance measurements obtained from different configuration option values. One possible approach is to first measure performance for a sufficiently large set of configurations and then infer models. However, the downside of this approach is that performance measurements are taken without knowing what and how many data points are sufficient for inferring an accurate model. Given the high cost of measuring performance that involves executing the program with the given usage scenario, this approach may suffer from scalability. Instead of the first-measure-then-infer approach, ConfProf uses a feedback-driven profiling and model inference approach. Specifically, ConfProf incrementally obtains new performance measurements to expand the existing profile data and iteratively improves the inferred model. The main benefit is that the feedback-driven approach requires fewer performance measurements than the alternative approach, which significantly reduces the model inference overall cost.

Algorithm 1 summarizes the main steps of the iterative model inference approach for ConfProf. The algorithm takes a set L of code locations (i.e., loops and system calls) and a set OPT of configuration options as input and outputs an inferred model for each code location. The complexity model inference algorithm starts by obtaining an initial sequence P of performance measures for configuration option values $v_1, v_2, \ldots v_k$ of each option O_i (line 2). ConfProf executes different values of O_i while keeping the other options as default to assess the performance influence of O_i individually. Therefore, after the execution, we obtain k profiles and have a sequence of performance measurements $P = p_1, \ldots, p_k$ for each code location.

Algorithm 1 Iterative inference of location-level complexity models.

Require: Set L of code locations, set OPT of options
Ensure: Map $\mathcal{U}$ of code locations to models
1: $err = err_{pre} = \infty$
2: **for** each $O_i \in OPT$ **do**
3: **for** each $L \in LOC$ **do**
4: **while** time < LIMIT **do**
5: $P \leftarrow$ `GetMeasures`(L, O_i)
6: $\mathcal{M} \leftarrow$ `LearnModel`(P)
7: $err =$ `ComputeError` $(\mathcal{M}, P)$
8: **if** $err < thresh_e \parallel (err\text{-}err_{pre}) < thresh_{inc}$ **then**
9: $\mathcal{U} \leftarrow \mathcal{U} \cup \{L \mapsto \mathcal{M}\}$
10: **break**
11: **end if**
12: $P' \leftarrow$ `GenerateNewProfile`$(O_{i_{new}})$
13: $err_{pre} = err$
14: $P \leftarrow P \cup P'$
15: **end while**
16: **end for**
17: **end for**

The main part of the algorithm is the iteration cycle of inferring and refining complexity models (lines 3 - 16). ConfProf first obtains the performance measurements for the code location L across all execution profiles (line 5). It then infers complexity models (line 6) based on the configuration option values and performance measurements using the machine learning models. In the next step, ConfProf computes err, the errors for the inferred model (line 7). Based on the prediction error, the algorithm terminates and returns the model when passing either one of two conditions: 1) The average error is less than a threshold of the mean absolute prediction error; 2) The improvement of the average error is less than a threshold of model improvement $thresh_{inc}$ (line 8). If neither of the above conditions holds, ConfProf continues to refine the model by generating new profiles using a new set of values for O_i (line 12) until it reaches a predefined time limit (e.g., 24 hours). In the presence of nested loops, the algorithm selects and profiles the outer and inner loops independently, which improves the transparency of model inference. This strategy provides details in the individual loop and avoids misrepresentations in cases when the whole program runs inside a single loop.

Selecting Values. While the number of sampled option values should be large enough to infer an accurate model, an excessively large number would require a much longer time to infer the model. Therefore, the initial set of values for a configuration option O_i is generated incrementally by following a fixed percentage $N\%$, and additional values are generated by adjusting the value of N. Specifically, for each numeric configuration option with a value range of $[min, max]$, if $max > 10$, ConfProf begins with min, and iteratively selects the next value by an increment of $max * N\%$ until max is reached. Each value is rounded up to an integer. If the prediction error does not reach the threshold, ConfProf automatically increases the sampling density by reducing $N\%$ to $\frac{N}{2}\%$ to generate more option values. The intuition is that as more data points are used, it often leads to a lower error in the inferred models. If the option has a smaller value range (e.g., $max \leq 10$), ConfProf selects all integer values within the range. When the algorithm completes, ConfProf updates the average error err_{pre} (line 13). Both new and old performance profiles are used for learning in the next iteration (line 14).

Interactions of Options. Prior work shows that binary options can enable/disable certain functionalities [25]. Switching a binary option O_b to a different value may cause a numeric option O_n to cover new performance-sensitive code locations. To identify such numeric options, ConfProf employs a pair-wise strategy, in which each value of the numeric option O_n is combined with all boolean values of each binary option O_b while keeping other options with their default values. If the number of performance-sensitive code location is different than when O_b is at its default value, ConfProf determines that O_b and O_n have a potential interaction. In this case, ConfProf learns complexity models for code locations that are controlled by O_b. If the order of a complexity model is linear or higher, O_b and O_n have a confirmed interaction.

ConfProf considers only pairwise interactions between numeric and binary options. A higher-strength interaction (among more than two configuration options) may cause performance problems. However, exhaustively exploring the combinations of all configuration options is not practical due to limited system resources. As a previous study [12] shows, the majority (72% to 73%) of configuration-related performance bugs is related to only one option, 13% to 15% of the bugs are related to two options where 12% to 15% of the examined parameter configuration bugs involve more than two options. Therefore, we focus on single options and

pairwise interactions. Yet, ConfProf has advantages over most existing performance modeling techniques [9, 29, 30] that support only binary options and cannot identify performance-related numeric options, which are commonly used in the real-world configurable systems [25].

A Running Example. In the example program of Figure 2, ConfProf first infers complexity models for configuration option O_1. Suppose the value range of O_1 is $[0, 1000]$ and the fixed percentage $N\%$ is 10% [17] (i.e., O_1 increases by 100 in each iteration). As such, the initial sequence of option values for O_1 is $0, 100, 200, \ldots, 1000$. The program is then exercised on 11 configurations: {0,0,0,0,0}, {100,0,0,0,0}, …, {1000,0,0,0,0}. As a result, ConfProf obtains 11 performance profiles: P_1, P_2, …, P_{11} for each performance-sensitive code location such as Loop1 (L_1), Loop2 (L_2), and Loop3 (L_3). If the prediction error does not reach the threshold, $N\%$ is set to $\frac{N}{2}\%$ (i.e., 5%). Additional configurations {50,0,0,0,0}, …, {950,0,0,0,0} are generated and exercised to produce more performance profiles. When either the prediction error reaches the threshold or a timeout occurs, ConfProf stops generating performance profiles and infers the complexity models (in terms of O_1) for three code locations: m_{L_1,O_1}, m_{L_2,O_1}, and m_{L_3,O_1}. As Figure 2 shows, the three code locations correspond to two constant models and one linear model ($m_{L_2,O_1} = k_2 * O_1$). Following the same process, ConfProf continues to infer complexity models for other numeric options (i.e., O_2, O_4, O_5).

Next, ConfProf infers the complexity models for interacting options. ConfProf first changes the default value of O_3 from 0 to 1 and pairs O_3 with other options (i.e., (O_1,O_3), (O_2,O_3), (O_4,O_3), (O_5,O_3)). For example, when pairing O_3 with O_1, the program is exercised on 11 new configurations: {0,0,1,0,0}, {100,0,1,0,0}, …, {1000,0,1,0,0}. ConfProf begins to infer complexity models on all configuration options: $m_{L_4,O_1 \mid O_3=1}$, …, $m_{L_5,O_4 \mid O_3=1}$, …, $m_{L_6,O_5 \mid O_3=1}$. ConfProf discovers three new performance-sensitive code locations (i.e., Loop4, Loop5, Loop6) on four interacting option pairs. Among these models, $m_{L_4,O_4 \mid O_3=1}$ is linear, $m_{L_5,O_4 \mid O_3=1}$ is constant, and $m_{L_6,O_5 \mid O_3=1}$ is higher-order (H.).

3.2 Estimating the Performance Impact

Upon completion of the location-level model inference, each configuration option O_i corresponds to a set of complexity models across all distinct code locations, $\{m_{L_1,O_1}, m_{L_2,O_1}, \ldots, m_{L_n,O_1}\}$, where n is the index of code locations.

Next, ConfProf ranks the single configuration options and option interactions based on their performance influence. ConfProf employs an idea similar to performance summarization [3] to rank options. The intuition is that a single option or an option interaction associated with more higher-order complexity models is more likely to have a significant performance impact and thus have a higher performance ranking. In particular, we assign different weights to location-level models according to their complexity orders (i.e., constant order, linear order, and higher-order). The final performance impact of an option O is a weighted sum of cost across all code locations: $\mathcal{P}_O = \sum_{i=1}^{n} (w_i C(L_i))$. L_i refers to a code location. $C(L_i)$ is the performance cost of L_i, a value of the recorded performance measurements. w_i is the weight of the complexity model at location L_i.

A Running Example. When choosing weights for models, we conduct a sensitivity study in which we tried out different combinations of weights (in Section 7). The weights used in the illustration are the most straightforward to understand and yield a better result. In Figure 2, we assign weights 1 (2^0), 2 (2^1), and 4 (2^2) to the constant, (positive) linear, and (positive) higher-order models. We use configuration options O_1 and O_2 as an example. O_1 corresponds to one linear model (Loop2) and two constant models (Loop1 and Loop3), and O_2 corresponds to one quadratic model (Loop3) and two constant models (Loop1 and Loop2). Suppose the performance impact of O_1 (with the binary configuration option O_3 disabled) across Loop1 to Loop3 is $C(L_1) = 5$, $C(L_2) = 10$, $C(L_3) = 5$ and that of O_2 is $C(L_1) = 2$, $C(L_2) = 4$, $C(L_3) = 10$. The performance impact for the two options is computed as: $\mathcal{P}_{O_1} = 1*5 + 2*10 + 1*5 = 30$; $\mathcal{P}_{O_2} = 1*2 + 1*4 + 4*10 = 46$. Therefore, O_2 has a higher performance impact than O_1. The table in Figure 2 summarizes the number of complexity models under different orders and the ranking of both single configuration options and their interactions.

4 IMPLEMENTATION

We conduct all experiments in a High-Performance Computer (HPC) cluster. Each node is exclusively reserved by the experiment and runs only ConfProf and subject programs to minimize any perturbation. ConfProf first classifies the configuration options into numeric options and binary options. This step is semi-automated. We develop a parsing tool to extract all configuration options and their descriptions from the documentation, and then manually identify the value range of these options. This is a one-time effort. ConfProf uses the Intel PIN dynamic binary instrumentation framework to identify loops and systems calls to collect performance measurements. ConfProf uses Weka [10] to learn location-level models.

5 EXPERIMENTAL SETUP

ConfProf aims to address three research questions: **RQ1:** How effective is ConfProf at ranking performance-influencing configuration options and identifying the corresponding code segments? **RQ2:** How well does ConfProf work compare to a baseline approach? **RQ3:** How effective is ConfProf at refining the location-level complexity models with the feedback-driven iterative approach?

5.1 Subject Programs

We use four highly-configurable, and popular open-source programs to evaluate ConfProf: Apache Server, PBZIP2, PostgreSQL (PSQL), and Lighttpd. These programs cover different application domains. For example, we use web servers, database engines, and data compression utilities. Each program has various numeric and binary options. We extract these options from the respective project documentation. For each project, we summarize several common usage scenarios that exercise different functionalities. Therefore, ConfProf ranks configuration options according to the usage scenario.

5.2 RQ1: Ranking Options

To evaluate the effectiveness of ranking, we use the mean average precision $MAP = \frac{1}{|Q|} \cdot \sum_{Q_i \in Q} AP(Q_i)$. MAP is a single-figure measurement of ranked retrieval results independent of the size of the top list [26]. It is designed for general ranked retrieval problems, where a query can have multiple relevant documents. To compute MAP, it first calculates the average precision (AP) for individual query Q_i and then calculates the mean of APs on the set of queries Q. To illustrate the MAP calculation, suppose that two configuration options O_1 and O_2 are both considered to be the ground truth options. If Technique-I ranks the two options at the 1st and 2nd positions among all 500 options and Technique-II ranks the two options at the 1st and 3rd positions, then the MAP of Technique-I is (1/1 + 2/2)/2 = 1, and the MAP of Technique-II is (1/1 + 2/3)/2 = 0.8.

To determine the ground truth options, we manually inspect the documentation and bug reports of the subject programs to identify configuration options that have a performance influence and use these options as the ground truth options. For example, the Apache documentation states that when setting the option `DeflateCompressionLevel`, "the higher the value, the better the compression, but the more CPU time is required to achieve this." Therefore, we consider `DeflateCompressionLevel` as a performance-influencing option. Hence, `DeflateCompressionLevel` is a ground truth option for the Apache deflate encoding usage scenario. Furthermore, we manually examine bug reports and code patches to verify whether any of the code locations associated with the performance-influencing options have bugs. We consider a code location that has bugs if (i) it is reported and confirmed by developers, and (ii) developers fix the bug in subsequent releases.

5.3 RQ2: Comparison with A Baseline

Ideally, to answer RQ2, we should compare ConfProf with existing approaches that detect and/or rank performance-influencing configuration options. However, we cannot find an existing approach with this specific goal. As discussed in Section 1, performance modeling techniques use configuration options to build models to predict a system's performance. The options used to construct the model indicate their impact on performance. SPLConqueror [29] is a black-box approach that uses machine learning and heuristic sampling to learn a performance model for the subject program from a set of configuration option values. Therefore, we can use the orders associated with each term generated from the SPLConqueror performance model to rank performance-influencing configuration options.

To illustrate SPLConqueror, we consider a web server program with options for defining the maximum number of client requests r and a binary option p for enabling HTTP persistent connection. A configuration-aware performance model for this system is: $\mathcal{M} = 3 * v(r)^2 - 2 * v(p) + 10$. In this example, the two options used to build the model influence software performance in different degrees. The first term $3 * v(r)^2$ indicates that the maximum number of client requests influences the overall running time in a quadratic order. The second term $2 * v(p)$ denotes that the HTTP persistent connection option if enabled, speeds up the execution time by two units, a linear order. Since the option r associates with the higher-order term $(3 * v(r)^2)$, r ranks higher than p in terms of performance influence.

5.4 RQ3: Effectiveness of Learning

To answer RQ3, we use the Mean Absolute Error $MAE = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n}$ for location-level models. MAE computes the average of the absolute difference between model prediction (y_i) and the actual outcome (x_i) across all data instances. It is a widely used metric to evaluate the accuracy of machine learning models [5, 33].

5.5 Study Operation

We set the threshold of the mean absolute error to 0.1 and the threshold of model accuracy improvement to 25%. These thresholds are empirically chosen because they yield an accurate model in a reasonable time (within 24 hours). To obtain the execution profiles (Section 3.1.3), the percentage used for generating the initial set of values of each numeric option is set to 10%. We assign weights 2^0, 2^1, 2^2 to the constant, (positive) linear, and (positive) higher-order models, respectively.

To control the fluctuation due to the randomness in each technique, we run each experiment three times to report the mean result.

6 RESULTS

6.1 RQ1: Ranking Options

ConfProf ranks configuration options in terms of their performance impact in a given scenario. We assess the effectiveness of ConfProf to rank configuration options based on the aforementioned ground truth. Column "Top-5" of Table 1 shows the top-5 configuration options ranked by each approach. The options marked in bold font indicate that they are known performance-influencing options. The reference links to the ground truth source. Options marked with "★" correspond to known performance bugs as discussed below.

AP-S1. In Apache Bug #34508 and Apache Bug #54852, users reported that a higher value of configuration option `StartServers` would cause a slow down during a graceful server restart. ConfProf ranked configuration option `StartServers` at the top among all 224 configuration options. The option `StartServers` was associated with 50 constant models, eight linear models, and five higher-order models. The loop implementation that had caused the bug (Figure 1) was among one of the linear location-level models. Also, the function "dummy_connection()" called inside the loop utilized system calls poll() and select() internally. The system call select() took on average 500 milliseconds to complete while most system calls use less than one millisecond. Therefore, the configuration option `StartServers` has the highest performance impact during the server restart.

AW-S2. In Apache Bug #42031, when the number of HTTP requests reached the `MaxClient` limit, Apache child processes would start to freeze. This bug was related to the configuration options `KeepAliveTimeout` and `MaxClients`, which were ranked by ConfProf at positions four and five, respectively. The bug had been

Table 1: Performance-Influencing Configuration Options Ranking

Scenario	ConfProf		SPLConqueror	
	Top-5	MAP	Top-5	MAP
AE-S1	G.S.Timeout, **KeepAliveTimeout [2]**, LimitI.R.,LimitReqBody, LimitRequestFields	**0.5**	MaxClients, RLimitNPROC, MinSpareServers, ReceiveBufferSize	0
AE-S2	MaxKeepAliveRequests, Timeout, SendBufferSize, **KeepAliveTimeout [2]***, **MaxClients***	0.3	**MaxClients**, MinSpareServers, FileETag	**1**
PZ-S1	**FileSize [20]**, **MaxMem(m)**, BlockSize(b), **NumOfProcessor(p)**, LoadAvg(l)	**0.9**	**FileSize**, BlockSize(b), **NumOfProcessor(p)**	0.8
PZ-S2	BWT, BlockSize(b), ChildStackSize(S), **MaxMem(m) [20]**, **NumOfProcessor(p)**	0.3	**MaxMem(m)**, FileSize, BlockSize(b), **NumOfProcessor(p)**	**0.8**
PZ-S3	**NumOfProcessor(p) [20]**, BlockSize(b), BWT, **MaxMem(m)**, LoadAvg(l)	**0.8**	ChildStackSize(S), **NumOfProcessor(p)**, BlockSize(b), **MaxMem(m)**	0.5
PS-S1	auto_work_mem, temp_file_limit, **commit_delay [21]**, **max_connections**, **maintenance_work_mem**	**0.5**	ssl, maxpreparedtrans, sslpreferscs, autovacuum, logdisconnections	0
PS-S2	commit_delay, max_files_per_process, auto_work_mem, **max_connections [22]**, temp_file_limit	**0.25**	max_parallel_workers_per_gather, ssl	0
LH-S1	connection.kbps, server.kbps, server.mc, server.listen-backlog, **server.max-fds [15]**	**0.2**	debuglogresponseheader, connkbps, servermaxreq, serverlistenbacklog	0
LH-S2	server.mrs, **server.max-keep-alive-requests [15]**, server.listen-backlog, server.mc, server.max-fds	**0.5**	server.max-read-idle, server.listen-backlog	0

The bold fonts are known to be performance-influencing. Due to space limitation, we only list nine scenarios. The full list can be found on our website.

fixed by adding locks to a while loop that associated the option `KeepAliveTimeout` with a linear complexity model.

Overall, the results show that the top-5 options ranked by ConfProf in each usage scenario include at least one known performance-influencing option. Thus, ConfProf is effective in ranking performance-influencing configuration options.

6.2 RQ2: Comparison with a Baseline

In Table 1, column "Top-5" under "SPLConqueror" shows the top-5 performance-influencing options in the performance model built by SPLConqueror. Both column "MAP" under ConfProf and SPLConqueror show the MAP scores. Compared to SPLConqueror, the MAP score in ConfProf is higher in 11 out of 13 scenarios ranging from 0.2 to 0.9 and averaging 0.5. The results show that ConfProf can achieve better configuration option ranking than SPLConqueror.

Table 2: Location-Level Complexity Models

Scenario	MAE		Models			Cost
	Start	End	C.	L.	H.	
AE-S1	13.9	9.1	26	22	34	1.8h
AE-S2	2.6	1.7	17	25	38	0.6h
PZ-S1	13.8	4.6	6	4	7	5.3h
PZ-S2	21.8	11.1	5	3	7	2.7h
PZ-S3	13.4	2.1	4	2	6	0.5h
PS-S1	32.1	2.5	49	17	58	2.1h
PS-S2	7.4	1.7	36	12	50	1.3h
LH-S1	14.5	3.3	11	6	20	1.5h
LH-S2	10.8	2.3	21	12	15	1.6h

6.3 RQ3: Effectiveness of Learning

To answer RQ3, we measured prediction errors of the learned location-level models and to what extent the error was reduced due to the iterative model refinement. Table 2 shows the result. The "Start" and "End" columns show the prediction errors before and after the model refinement. When applying the model refinement, errors of the inferred models were substantially lower (57%, on average) across location-level models in all usage scenarios. In summary, the results show that the iterative inference algorithm (Algorithm 1) improved the prediction accuracy of learning the location-level models.

7 THREATS TO VALIDITY

The primary threat to external validity for this study involves the representativeness of the selected subject programs. Other subject programs may exhibit different behaviors. We reduce this threat to some extent by studying subjects from different application domains, and those have varying numbers of configuration options.

A threat to the internal validity of this study is the possible faults in the implementation and tools that we use to perform the evaluation. We controlled this threat by testing our tools extensively and verifying their results against a smaller program for which we can manually determine the correct result.

One threat involves the human factor for determining whether a configuration option is performance-influencing. It may introduce false-positive options. To reduce this threat, we first searched the application documentation and issue trackers for performance-influencing options and then compared these options to the top-5 options reported by ConfProf. The fact that most performance bugs require less than five options [11, 12] to manifest, our evaluation with the top-5 options is reasonable and sufficient. When multiple options are involved in the ground truth, the absolute option ranks cannot be determined, and it does not matter either. We use MAP to evaluate the effectiveness of identifying performance-influencing options. The exact rank among ground truth options does not change the MAP calculation. A second threat is the relationship between performance impact and option values in a model. ConfProf considers constant, linear, and higher-order relations. The performance impact and the option values can have an inverse relationship. We did not observe such relationships in our study. We plan to study the cost-effectiveness of using inverse models in future work. A third threat is that ConfProf focuses on performance-sensitive code locations because performance bugs due to loops and system calls are more pervasive [14]. Other code locations may also cause performance problems, such as multi-threaded code that causes lock contention [32, 39] and (indirect) recursion. We plan to incorporate these code locations into ConfProf in future work.

8 RELATED WORK

Configuration-Aware Research. There has been a good amount of research work on configuration-aware techniques [25, 40]. Configuration in the context is not the same as software configuration management (SCM). Configuration-aware research targets on parameters that control various software runtime behavior in the runtime. Zhang et al. [40] have proposed a technique ConfDiagnoser,

to diagnose crashing and non-crashing errors related to configuration misconfigurations. Rabkin et al. [25] propose a static analysis technique to extract configuration options from Java code to bridge the discrepancy between configuration options and software documentation. Our work lies in configuration-aware research. But unlike our research, previous work focuses on the functional aspect of a software system instead of performance problems.

Performance Modeling. There has been a good body of work on constructing performance models for various purposes [9, 13, 29, 34]. Guo et al. [9] predict a configuration's performance based on random sampling and a statistical learning method CART. Huang et al. [13] build sparse polynomial regression performance models using sampled input files and command-line arguments. Siegmund et al. [29] propose performance models that predict end-to-end system performance with individual configuration options and their interactions. Tarvo et al. [34] build performance models for multithreaded programs. Unlike our work that targets ranking performance-influencing configuration options, prior work focuses on program performance prediction. Specifically, such work builds end-to-end performance models while ConfProf builds location-level performance models for configuration options. Building performance models is the vehicle to evaluate the performance impact of configuration options when profiling a scenario. It is not the goal of ConfProf to build an end-to-end performance model for the subject program. Also, when building performance models using sampled configurations, these techniques deem the subject program as a black-box while ignoring the implementation of the program. Although these techniques may provide some insights about factors involved in analyzing configuration-related performance problems, they are incapable of pinpointing code location-level implementations that can influence the software system performance.

9 CONCLUSIONS

We present ConfProf, a white-box performance profiling approach for software configuration options. ConfProf outputs a ranked list to identify and understand the performance influence of configuration options. Our evaluation shows that ConfProf can successfully identify performance-influencing configuration options and outperform the state of the art approach in 11 out of 13 usage scenarios.

ACKNOWLEDGEMENT

This work was supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and Perf4JS projects.

REFERENCES

[1] Paul Anderson. The use and limitations of static-analysis tools to improve software quality. *CrossTalk: The Journal of Defense Software Engineering*, 21(6):18–21, 2008.
[2] http://httpd.apache.org/docs/current/misc/perf-tuning.html.
[3] Mona Attariyan, MIchael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, pages 307–320, 2012.
[4] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, pages 1–11, 2010.
[5] David Bailey and Allan Snavely. Performance modeling: Understanding the past and predicting the future. *Euro-Par 2005 Parallel Processing*, pages 620–620, 2005.
[6] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. In *ICSE*, pages 463–473, 2009.
[7] Tianfeng Chai and Roland R Draxler. Root mean square error (rmse) or mean absolute error (mae)?–arguments against avoiding rmse in the literature. *Geoscientific model development*, 7(3):1247–1250, 2014.
[8] Monika Dhok and Murali Krishna Ramanathan. Directed test generation to detect loop inefficiencies. In *FSE*, 2016.
[9] Jianmei Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-aware performance prediction: A statistical learning approach. In *ASE*, pages 301–311, 2013.
[10] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009.
[11] Xue Han, Daniel Carroll, and Tingting Yu. Reproducing performance bug reports in server applications: The researchers' experiences. *Journal of Systems and Software*, 156:268–282, 2019.
[12] Xue Han and Tingting Yu. An empirical study on performance bugs for highly configurable software systems. In *ESEM*, pages 215–224, 2016.
[13] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. Predicting execution time of computer programs using sparse polynomial regression. In *Advances in Neural Information Processing Systems*, pages 883–891, 2010.
[14] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *PLDI*, pages 77–88, 2012.
[15] https://redmine.lighttpd.net/projects/1/wiki/docs-performance.
[16] Max Lillack, Christian Kästner, and Eric Bodden. Tracking load-time configuration options. *IEEE Transactions on Software Engineering*, 44(12):1269–1291, 2018.
[17] https://machinelearningmastery.com/much-training-data-required-machine-learning/.
[18] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *MSR*, pages 237–246, 2013.
[19] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *ICSE*, pages 562–571, 2013.
[20] http://compression.ca/pbzip2/.
[21] https://wiki.postgresql.org/wiki/Tuning-Your-PostgreSQL-Server.
[22] http://www.revsys.com/writings/postgresql-performance.html.
[23] John C Platt. 12 fast training of support vector machines using sequential minimal optimization. *Advances in kernel methods*, pages 185–208, 1999.
[24] Michael Pradel, Parker Schuh, George Necula, and Koushik Sen. EventBreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *OOPSLA*, 2014.
[25] Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *ICSE*, pages 131–140, 2011.
[26] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press, 2008.
[27] Marija Selakovic, Thomas Glaser, and Michael Pradel. An actionable performance profiler for optimizing the order of evaluations. In *ISSTA*, pages 170–180, 2017.
[28] Marija Selakovic and Michael Pradel. Performance issues and optimizations in JavaScript: An empirical study. In *ICSE*, pages 61–72, 2016.
[29] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. Performance-influence models for highly configurable systems. In *ESEC*, pages 284–294, 2015.
[30] Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *ICSE*, pages 167–177, 2012.
[31] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. In *OOPSLA*, pages 561–578, 2014.
[32] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. In *PPoPP*, pages 269–280, 2010.
[33] Chong Tang. System performance optimization via design and configuration space exploration. In *FSE*, pages 1046–1049, 2017.
[34] Alexander Tarvo and Steven P. Reiss. Automated analysis of multithreaded programs for performance modeling. In *ASE*, pages 7–18, 2014.
[35] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Synthesizing programs that expose performance bottlenecks. In *CGO*, 2018.
[36] Alexander Wert, Jens Happe, and Lucia Happe. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In *ICSE*, pages 552–561, 2013.
[37] Cort J Willmott. Some comments on the evaluation of model performance. *Bulletin of the American Meteorological Society*, 63(11):1309–1313, 1982.
[38] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *ISSTA*, pages 90–100, 2013.
[39] Tingting Yu and Michael Pradel. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *ISSTA*, pages 389–400, 2016.
[40] Sai Zhang and Michael D. Ernst. Automated diagnosis of software configuration errors. In *ICSE*, pages 312–321, 2013.