DiagConfig: Configuration Diagnosis of Performance Violations in Configurable Software Systems

Anonymous Author(s)

ABSTRACT

Performance degradation due to misconfiguration in software systems that violates SLOs (service-level objectives) is commonplace. Diagnosing and explaining the root causes of such performance violations in configurable software systems is often challenging due to their increasing complexity. Although there are many tools and techniques for diagnosing performance violations, they provide limited evidence to attribute causes of observed performance violations to specific configurations. This is because the configuration is not originally considered in those tools. This paper proposes DiagConfig, specifically designed to conduct configuration diagnosis of performance violations. It leverages static code analysis to track configuration option propagation, identifies performance-sensitive options, detects performance violations, and constructs cause-effect chains that help stakeholders better understand the relationship between configuration and performance violations. Through experimental evaluations on eight real-world open-source software, we demonstrate that DiagConfig effectively identifies performance-sensitive options and constructs cause-effect chains. Specifically, DiagConfig identifies 93.3% (14/15) of performance-sensitive options on the test set and significantly outperforms the state-of-the-art method. We also show that DiagConfig can accelerate auto-tuning by compressing configuration space.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; **Software performance**.

KEYWORDS

Configuration diagnosis, Static analysis, Performance violation, Taint tracking

1 INTRODUCTION

Modern software systems are highly configurable to meet users' requirements in various scenarios. Take the popular open source database MySQL as an example. Its latest version has reached around 1,000 configuration options. It is always challenging to make the proper configurations for the software deployment. Studies have shown that misconfiguration is one of the primary culprits responsible for production system failures and performance problems [2, 28, 70, 66]. As system performance is becoming more and more critical in enterprise business [17, 31, 27], misconfiguration can lead to millions of dollars cost [53]. It is of essence to quickly find out the improperly configured options when SLOs (service-level objectives) violations occur.

However, diagnosing and pinpointing configuration-related performance problems is time-consuming [74, 44, 8, 7, 21, 29, 58] due to its huge search space. Search-based techniques are exploited to change the value of configuration options by trial-and-error [40, 43] to resolve performance violations and to find the optimal configurations. But without explicit cause-effect relationships between options and performance, search-based techniques could be easily trapped in the massive configuration space generated by too many options [41, 55, 10].

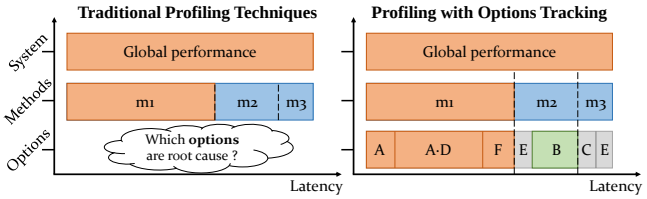


Figure 1: The weakness of traditional profiling techniques for configuration-related performance violations diagnosis.

The cause-effect relationships between options and performance help understand *which, where, how,* and *why* configured options influence system performance behind the screen. But such studies are still in the early stage and rely greatly on experts and domain knowledge. One human-centric approach [58] to diagnose performance problems is to collect software hotspots with CPU profiling, identify related options and locate option hotspots with performance-influence models. Then they manually investigate how related configuration options affect the performance. This approach is non-trivial and cannot guarantee to find the correct root cause for the following reasons: (1) the relationship between hotspots detected by profilers and configuration options is complex and uncertain; (2) not all options need to be tuned when a performance violation occurs; (3) some options are continuous values, making it difficult for performance modeling; and (4) options evolve software updates.

When performance violations occur, there are substantial monitoring and profiling techniques whose goal is to locate performance hotspots [48, 56, 42, 52, 30]. However, hotspots only imply performance bottlenecks but not always performance violations. For example, some code logic is destined to occupy more execution time than others, and thus are highlighted as hotspots by the profilers. On the other side, which options to tune remains an open problem in performance analysis with traditional monitors and profilers, given that those profiling and monitoring tools consider only one instance of all configuration options at a time to evaluate their performance effects. As illustrated in Figure 1, in a configurable system with six options, method *m1* contains complex computational logic and is highlighted as a hotspot by profilers when a performance violation occurs. However, traditional profilers cannot dive into the option level and it totally depends on stakeholders (users, developers, and site reliability engineers) to navigate the source code and then to find the options *A, D, F* are relevant to *m1*. But in fact, option *B* is the critical option causing the performance violation.

Performance-influence models [49] help understand the influence of options on system performance [19, 20, 25, 26, 29, 34, 57, 59, 62, 58]. The accuracy of performance models heavily depends on the selected model [34] and the subset of the configuration space generated by various sampling strategies [39, 32]. To make it trickier for sampling-based performance modeling, software systems are sensitive to configuration changes, meaning that systems with similar configurations would have dramatic performance differences [41, 10]. Previous attempts at building white-box performance models of configurable systems [57, 59, 62, 58] are deficient for general-purpose dynamic workloads and environments. Besides, as software evolves and the number of system configurable options increases, it is becoming more expensive to keep the performance models updated.

Given the discussion above, our goal is to devise a general low-cost, high-precision technique for configurable software systems that can not only interprets cause-effect relationships between options and performance violations but also infers which configured options are responsible for those violations. To achieve our goal, we propose DiagConfig, a white-box diagnosis tool to (1) identify performance-sensitive options, (2) tame cause-effect relationships between the options and performance violations, (3) figure out the options that are responsible for performance violations. DiagConfig leverages both runtime profiling information and static code analysis via taint tracking to build cause-effect chains which can help stakeholders explain performance issues. We implemented DiagConfig using FlowDroid [3] and Soot [54] and evaluated it with eight real-world open-source projects. Our results show that DiagConfig identifies 86.95% of performance-sensitive options and finds out 10 more performance-sensitive options missed by the state-of-the-art method. DiagConfig is fast and can diagnose performance violations in seconds. We integrate DiagConfig into an auto-tuner and demonstrate its feasibility of underpinning prior works on configuration performance tuning.

Our key contributions are as below:

- A summary of information needed for configuration diagnosis of performance violations, including identification of performance-sensitive options, localization of performance violations, and root causes inference.
- A white-box approach and prototype, DiagConfig, for building cause-effect chains between configuration options and profiled hotspots to diagnose performance violations.
- A dataset of performance-sensitive options on eight real-world configurable software systems in diverse domains, which can be used to evaluate DiagConfig and its comparable alternatives [1].

The rest of this paper is organized as follows. We begin with background concepts in the white-box analysis of configurable software system and research question definition in Section 2, propose our methodology in section 3, and describe the implementation in section 4. We evaluate our approach in section 5, and present limitations with threats to validity in section 6. We discuss related work in 7 and conclude in section 8.

[1] All implementation and data can be found in archived repository [13].

2 BACKGROUND AND RESEARCH QUESTIONS

In this section, we first introduce basic concepts on profiling and taint tracking, which are exploited in the white-box analysis of configurable software systems. After that, we introduce important information needed to diagnose performance violations, according to which we define and describe our research questions.

2.1 Background

Profiling. Profiling aims to reveal the runtime behavior of program execution with regard to resource consumption [15]. Profiling investigates how much of a resource each program element consumes and reports performance-critical program elements as **hotspot**s. A **program element** refers to a statement or method (function) in a program. It is a basic sampling unit in profiling. There are many approaches to detect performance problems caused by resource consumption, such as CPU-time profiling [18] and unnecessarily high memory consumption profiling [65, 64, 68]. These approaches track how hotspots are triggered; in particular, stakeholders follow the traces (e.g. call-chains) to diagnose *unexpected* performance behaviors. However, limited evidence in the profiling clarifies the relationships between configuration options and hotspots. Stakeholders have to navigate the source code to find hotspots-related options, which is not efficient and could go beyond the scope of human reasoning due to the complexity in the dependencies of program elements.

Taint tracking. Taint tracking is typically used in information security detection to track the **information-flow** from user inputs (**sources**) to specific security-sensitive locations (**sinks**). From the perspective of performance analysis in configurable systems, the configuration is equal to user inputs. Well-designed systems have standard APIs for loading configuration option values to program variables [14, 37, 46, 47, 67, 66]. Variables are then propagated along the program's data-flow paths via assignments, string operations, and arithmetic operations, until they are consumed in program elements that change runtime behaviors. Taint tracking with configuration as source helps understand different attributes of configurable software systems, including performance attributes.

2.2 Research Questions

The performance of a configurable software system can be defined as $P_i = p(c_i, w_i, v_i)$, where $c = [o_1, o_2, ..., o_n]$ is a valid configuration, o is an option, w is a specific workload, v is a specific production environment, and i is used to distinguish between cases. In this paper, our ultimate goal is to find the set of options c^* responsible for the performance violation $|P_i - P_{SLO}| \geq \delta$ to help configuration performance tuning and cause-effects explanation, where P_{SLO} denotes the predefined SLO and δ is a significant factor.

Specifically, a configuration diagnosis needs sufficient information to answer the following research questions.

RQ_1: Which options are performance-sensitive?

Not all options are performance-sensitive. Stakeholders usually identify and interpret performance-sensitive options based on documentation rather than source code [58]. However, any option that affects performance during runtime must have a data- or control-flow dependency with performance-related operations [36].

Performance-related operations refer to code snippets in a program that contribute positively to execution time (time-expensive) and memory consumption (memory-expensive). And the **data- and control-flow dependencies** between an option and a performance-related operation can be grouped into three categories.

(1) A direct data dependency where program variables derived from the option's value are used in the operation and affect every dynamic execution of the operation.
(2) An if/switch-related control dependency where program variables derived from the option's value determine whether the operation is executed by influencing control-flow decisions of the if/switch statement.
(3) A loop-related control dependency where program variables derived from the option's value determine the number or frequency of the operation executions.

In this paper, we have identified four types of performance-related operations: one type of **memory-expensive operations** and three types of **time-expensive operations**. The **memory-expensive operations** are heap or static array allocation operations. And the three types of **time-expensive operations** are: (1) I/O operations; (2) lock-synchronization and threads start/pause operations; (3) operations that affect system concurrency (e.g., creation of threads or thread pools). These operations are denoted by the notation $\boldsymbol{PerfOps_i}$ where $i \in [0, 3]$.

To answer this question, we first treat configuration as source for taint tracking and identify performance-related operations. Since not all performance-related operations have a significant performance impact, we utilize the dependencies between options and performance-related operations, and identify performance-sensitive options via the random forest. The details of this approach are presented in Section 3.1 and the evaluation of its effectiveness is discussed in Section 5.2.

RQ_2: Which hotspot functions are performance-violating?

End-to-end performance metrics can tell when performance violations occur but could not help explain the reason. In contrast, profilers report hotspot functions within the program concerning execution time, memory consumption, invocations, etc. To answer this question, we do a profiling comparison between poor executions and the normal baseline execution. The former has significantly deteriorated performance while the latter is a high-performance execution that meets SLO, usually given by domain experts. By comparing hotspot functions in poor execution to those in the normal baseline execution, profiling can help locate performance-violating hotspot functions under poor execution.

RQ_3: How do performance-sensitive options lead to performance violations?

RQ_1 discovers performance-sensitive options and RQ_2 identifies hotspot functions causing performance violations. RQ_3 tries to build connections between those two in order to explain the cause-effect relationship between configuration options and performance. To answer this question, we use performance-related operations as the intermediary. Once program variables derived from the configuration options are used in statements that control the hotspot functions (e.g., branch/loop conditions, invocations), both performance-sensitive options and performance-violating hotspot functions become traceable. We build cause-effect chains by correlating information-flow paths (between options and operations) in Section 3.3 and call-chains (between operations and hotspot functions), and evaluate the effectiveness of this approach in Section 5.3.

3 METHODOLOGY

In this section, we first briefly introduce our prototype tool DiagConfig and then describe the workflow steps in each subsection.

DiagConfig is a white-box configuration diagnosis system and is general enough to adapt to software systems under different configurations, workloads, and environments. Figure 2 shows the overview of DiagConfig. It consists of two parts: 1) *offline analysis*, and 2) *online diagnosis*. **Offline analysis** identifies performance-sensitive options and tracks them in the source code. This procedure requires two inputs: 1) the target system's source code, and 2) a list of specific prerequisites. The prerequisites contain statements that load configuration option values (as sources) and performance-related operations (as sinks) in the source code. It constructs information-flow paths from option values to performance-related operations via taint tracking, and then extracts options' static performance properties, and classifies performance-sensitive options and information-flow paths. **Online diagnosis** monitors system runtime behavior, locates performance-violating hotspot functions, and collects call-chains for the hotspot functions from the profiling. It builds cause-effect chains with information-flow paths and call-chains, which reveals the data- or control-flow dependency between individual options and performance-violating hotspot functions. When a new performance violation occurs, DiagConfig can apply these cause-effect chains to guide diagnosis and to recommend crucial configuration options for performance tuning. It can work as a daemon process that continuously analyzes performance-violating configuration options for auto-tuning.

3.1 Identification of Performance-sensitive Options

The identification of performance-sensitive options is the goal of the offline analysis. The data- and control-flow dependencies between variables derived from configuration options and performance-related operations are contained in source code, thus DiagConfig identifies the dependencies via static taint tracking.

Taint tracking. The basic idea of taint tracking is to track the propagation of variables in source code with a "coloring" technique. It first tags each program variable that stores an option value and then analyzes the dependencies between the colored variables and performance-related operations. The initial taints are program variables obtained from configuration loading functions, which are called **sources**. Taints are then propagated and transformed along the program's data-flow paths, until they are consumed in sink statements. DiagConfig automatically discovers sinks. Besides pre-defined performance-related operations, it leverages program-dependence graph [16] to mark *if/switch* statements with branches containing performance-related operations and *loop* startup/jump-out statements with the body containing performance-related operations as **sinks**. Once the taints reach sinks, DiagConfig records the **information-flow paths** containing the corresponding options, taints propagation paths, and location (e.g., source code file name, source code line number) of code snippets where dependencies between options and performance-related operations occur.

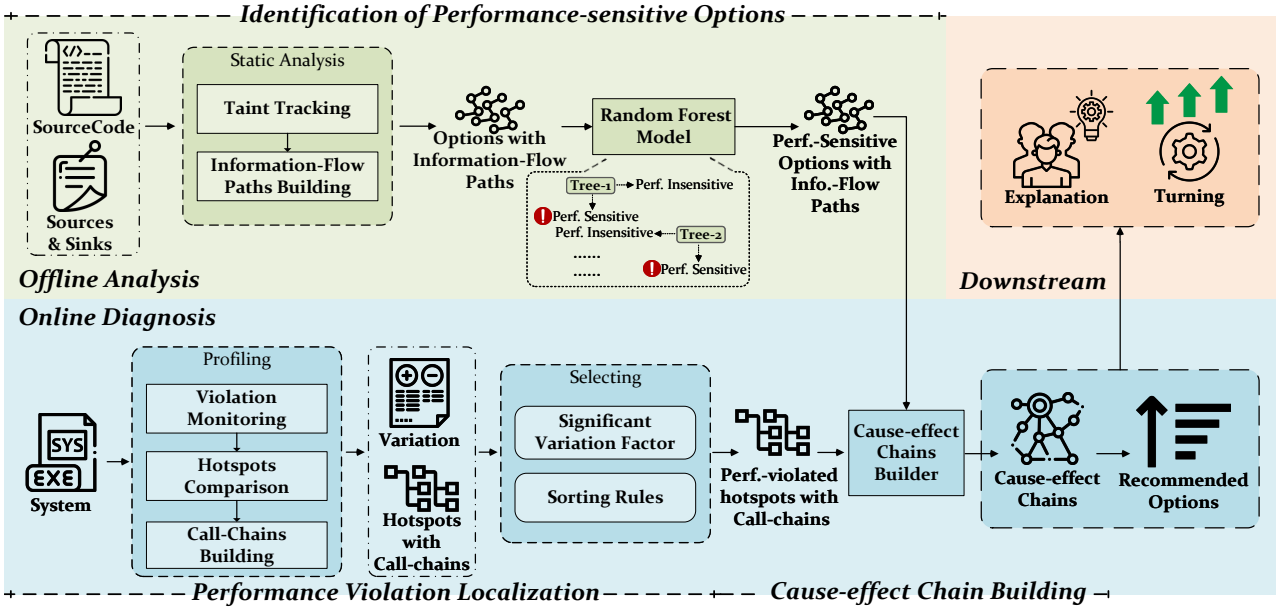
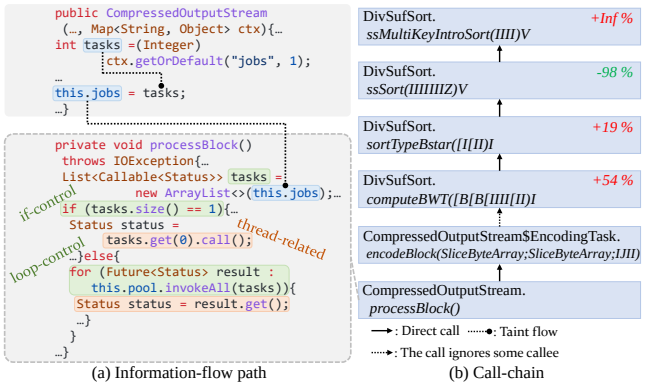


Figure 2: Overview of DiagConfig



(a) Information-flow path (b) Call-chain

Figure 3: Example of building a cause-effect chain using an information-flow path (a) and a call-chain of the performance-violating hotspot *ssMultiKeyIntroSort* (b) to diagnose *where, how, and why* the option *jobs* causing a performance violation in Kanzi [38]. DiagConfig backtracks the call-chain and analyzes each callsite (b). In the method *processBlock*, it identifies the program dependencies between the option *jobs* and thread-related operations (*call, get*) that invoke *encodeBlock* (the callee of the *processBlock*). Then, it connects the information-flow path and the call-chain to a cause-effect chain.

If the dependency belongs to the if/switch-related or loop-related dependency, the code snippet contains the boundary of branches or loop bodies.

Example. As shown in Figure 3.(a), DiagConfig captures the *if-related* control and *loop-related* control dependencies between the option *jobs* and *thread-related* operations with static taint tracking. Meanwhile, DiagConfig records the information-flow path and the location where the dependencies occur, including the method signature, range of branch and loop body, and source code line number.

The critical challenge here is that there are some, but not all, performance-related operations that do not significantly affect performance during system execution. Thus, we characterize the information-flow paths of the configuration options as **performance properties**. Next, we will build a random forest classification model with the labeled options' performance properties to identify performance-sensitive options in the following section.

Characterization. Given information-flow paths for one configuration option, we count the number of performance-related operations and characterize the performance property of the option by constructing the counter vector V. We note that the performance property is a sparse vector and each count is relatively small. It's plausible that for some nodes, the random subsets of the bootstrapped samples will produce an invariant feature space. These nodes have limited productive splitting, resulting in the children of them may not helping. Therefore, we cluster the counts based on performance-related operations and dependencies categories, denoted as $PerfOps_i^k$, where $i \in [0, 3]$ refer to memory-expensive operations and three categories of time-expensive operations we mentioned in section 2.2, $k \in \{data, if/switch, loop\}$ corresponds the categories of dependencies, and $\sum_k PerfOps_i^k = PerfOps_i$. Next, we add the following aggregated items in front of the performance property V. First, we introduce the aggregated items for four categories of performance-related operations,

$$\forall i, 0 \leq i < 4, v_i = PerfOps_i,$$

where each aggregated item is an accumulated counts of performance-related operations based on the operation categories. For example, the aggregated item $v_1 = PerfOps_1$ is obtained by accumulating the counts of performance-related operations belong to I/O operations.

Then, we introduce three aggregated items that combine memory-expensive operations with other three time-expensive operations

categories,

$$\forall i, 0 < i < 4, v_{3+i} = PerfOps_0 + PerfOps_i,$$

where each aggregated item is an accumulated counts of memory-expensive operations with other three time-expensive operations categories. For example, $v_5 = PerfOps_0 + PerfOps_2$ is obtained by accumulating the counts of memory-expensive operations plus lock-synchronization and threads start/pause operations. Similarly, we introduce three aggregated terms,

$$\forall i, j, 0 < i < j < 4, v_{4+i+j} = PerfOps_i + PerfOps_j.$$

After that, we introduce one aggregated item for all performance-related operations,

$$v_{10} = \sum_{0 \le i < 4} PerfOps_i,$$

which is an accumulated counts of all performance-related operations dependent on the given option. The above aggregation items do not distinguish between dependencies categories.

Finally, we introduce twelve aggregated items for individual performance-related operations based on the data- and control-flow dependencies categories,

$$\forall i, 0 \le i < 4, v_{10+i\times3+1} = PerfOps_i^{data},$$
$$v_{10+i\times3+2} = PerfOps_i^{if/switch},$$
$$v_{10+i\times3+3} = PerfOps_i^{loop}.$$

For example, the aggregated item $v_{16} = PerfOps_1^{loop}$ is obtained by accumulating the counts of I/O operations that loop-related control dependent on the given option. Such 23-dimensional aggregated items are reasonable to introduce because they contain information about performance-related operations and dependencies categories, rather than a single performance-related operation.

Random Forest. Random forest [6] is an appropriate algorithm for our binary classification task. It combines multiple randomized decision tree predictors and distinguishes classes by aggregating their predictions. Therefore, it is more robust than a single tree predictor. Moreover, it is also more interpretable than deep learning models because of explicit decision inference paths in a tree. Lastly, training a random forest model only needs a small sample data which is readily satisfied in our scenario. To identify performance-sensitive options and filter corresponding information-flow paths, we train a random forest [6] model which approximates the following function.

$$g(\text{performance property}) \rightarrow \text{performance type}$$

The training data of random forest are performance properties of the target configuration options with class labels, namely performance-sensitive (i.e., 1) or not (i.e., 0). During the training, the algorithm randomly samples fractions from the training data, builds a randomized decision tree predictor on each data partition, and then aggregates these trees together as a forest. Once the random forest model is obtained, for a new configuration option with its information-flow paths, DiagConfig characterizes the performance property and feeds it to the trained random forest model. The random forest aggregates the scores of each decision tree predictor for the performance property to determine whether the option is performance-sensitive. *All performance-sensitive options and their corresponding information-flow paths are persisted in a file for cause-effect chains building in the online diagnosis.*

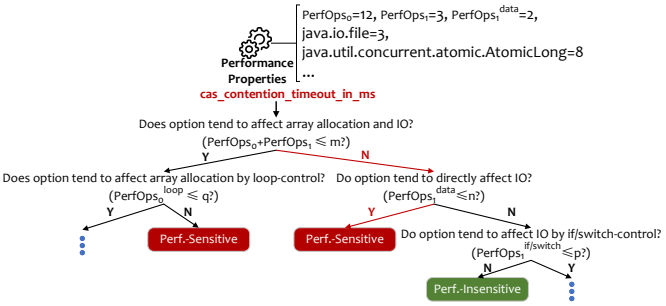


Figure 4: An example of identifying performance-sensitive options based on a possible decision tree in the random forest.

Example. Figure 4 shows one decision tree of a trained random forest model for prediction from the performance property. The performance property of the option *cas_contention_timeout_in_ms* in the Cassandra database shows that this option tends to influence the system performance by memory-expensive operations, I/O operations, and lock-synchronization operations. Given the performance property of the option, the predictor descends along the tree and finally predicts the option is performance-sensitive.

3.2 Performance Violation Localization

Performance violation localization is the first step of the online diagnosis. The goal of this step is to detect the performance-violating hotspot functions based on performance measurements for the selected metrics. DiagConfig first leverages an off-the-shelf profiler, Jprofiler [52], to continuously monitor the end-to-end performance of a system. When an SLO violation is detected in the end-to-end performance measurement, it collects hotspot functions from the profiler. By comparing the execution time of hotspot functions in the performance violated situation and the normal baseline execution, DiagConfig calculates the performance variation of each hotspot function. Most off-the-shelf profilers are capable to measure the execution time of each hotspot function excluding the callee's performance influence. Thus, DiagConfig determines that a hotspot function with a significant performance variation is performance-violating. Here we choose the significant variation factor to be 5% (see Section 4).

Besides the measurement of performance variations, we also extract call-chains for each hotspot function from the profiler. The call-chains of a hotspot function can provide information about how the hotspot function is triggered and impacts the system performance. Since a system has many hotspot functions and a hotspot function can be associated with multiple call-chains, we use some rules to filter and sort the hotspot functions and their call-chains. We pre-set a significant variation factor (5%), and those hotspot functions whose performance variations under the factor will be discarded. Then, we sort the hotspot functions and call-chains according to performance variation and performance contribution to the system performance.

Example. A call-chain is shown in Figure 3.(b), each box is a hotspot function and the corresponding performance variation is

marked in the upper right corner. Hotspot functions with significant performance deterioration are culprits of the performance violation.

3.3 Cause-effect Chain Building

Despite information-flow paths showing how performance-sensitive options affect performance-related operations, not all performance-related operations lead to performance violations at runtime. Besides, quantifying the importance of options for system performance violations is still an open question, which cannot be answered by static code analysis alone. Moreover, some function calls are dynamic binding (e.g., thread invocation, reflection, callbacks) that static analysis tends to miss. Therefore, our approach leverages the call-chains of the performance-violating hotspot functions from the profiler to complement the runtime information of the system lacking by the static code analysis. DiagConfig backtracks the call-chains of the hotspot functions and analyzes the callsite at the statement/block-level to determine whether there are program-dependencies between the performance-violating hotspot functions and performance-related operations. When a hotspot function in the call-chains involves performance-related operations or is called by performance-related operations, DiagConfig correlates the corresponding information-flow paths and the call-chains to construct trackable cause-effect chains. Associating performance variations of hotspot functions with options corresponding to information-flow paths allows for quantifying and ranking the importance of options.

The algorithm 1 describes how we build cause-effect chains based on information-flow paths and call-chains. The information-flow path records the configuration option (source) and the location of performance-related operation (sink). For each call-chain of the selected hotspot function (line 4), we backtrack it from callee to caller and check whether the caller is contained by the information-flow paths (line 5). If all callers in the call-chain of a hotspot function are not contained by the information-flow paths, this means that the hotspot function is not influenced by the options. Since the branch or loop body is involved in if/switch- or loop-related dependency, we get the code snippet where dependency between the option and the performance-related operation occurs (line 10). Then, we build a cause-effect chain by connecting the call-chain and information-flow path if the caller directly invokes the performance-related operation or the callee is invoked by the performance-related operation based on the call graph (line 11).

Example. Figure 3 shows that DiagConfig builds a cause-effect chain between the option *jobs* and the hotspot function *ssMultiKeyIntroSort*. It backtracks the call-chain and finds that the option *jobs* influences the frequency of the hotspot function *ssMultiKeyIntroSort* by thread-related operation and if/loop-related control dependency in the method *processBlock*. Then it builds a cause-effect chain by connecting the call-chain and information-flow path starting from the option *jobs*.

4 IMPLEMENTATION

Our prototype tool **DiagConfig** is built on the top of the Soot compiler infrastructure [54], FlowDroid [3], Scikit-learn [45], and Jprofiler [52] and targets particularly configurable Java ecosystems.

Algorithm 1: Cause-effect Chains Building

```
Input: Perf.-Sensitive Info.-flow paths ℙ, Selected hotspots with call-chains ℍ
Output: Cause-effect chains ℂ
1  ℂ = emptySet()
2  for h ∈ ℍ do
       /* Get the call-chains for each hotspot                          */
3      Call − chains ← parse(h)
4      for call − chain ∈ Call − chains do
           /* Backtrack the call-chain from callee to caller.
              Caller doesn't appear in ℙ, indicating that no
              option-related performance operations involved            */
5          for caller ∈ call − chain ∩ ℙ do
6              Paths ← ℙ.get(caller)
7              cg ← Scene.v().getCallGraph()
8              callee ← Prev(caller)
9              for path ∈ Paths do
10                 block ← getPerfOpRelatedBlock(path)
                   /* Check whether caller invokes perf.
                      operations or callee is invoked by perf.
                      operations                                        */
11                 if isInfluenced(cg, caller, callee, block) then
12                     ℂ.add(link(call − chain, path))
13                 end
14             end
15         end
16     end
17 end
18 return ℂ
```

To ensure its scalability, we consider only standard library APIs and bytecode instructions as they are the basic performance-related operations that are application-independent in the Java ecosystem. DiagConfig identifies performance-related operations and control-flow statements related to them as *sinks* through method signatures, type analysis, and program dependence graph [16] based on Jimple, the intermediate-representation provided by the Soot. For example, array allocation-related operations are represented by *JNewArrayExpr* and *JNewMultiArrayExpr*, I/O-related time-expensive operations are usually prefixed with *java.io* or *java.nio* for their standard library API signatures.

In the offline analysis, we leverage the static taint analysis framework FlowDroid for configuration options taint tracking. We build the random forest with *Scikit-learn* for identifying performance-sensitive options. In the online diagnosis, we use a profiler well-known in the industry for its low overhead, Jprofiler, to monitor the execution time of each method in the system.

We consider a performance violation occurs when the system takes more than 5% of the benchmark execution time to process a task. Then we compute performance variation for each hotspot method by hotspot comparison built within the profiler and collect call-chains of the hotspot methods with performance variation more than 5% for root cause inference. The 5% is our empirical unacceptable value based on measurement variance which is up to 4% known from the prior work [59]. We implement an inter-procedural callsite analysis using the call graph provided by the Soot to build cause-effect chains.

5 EVALUATION

In this section, we evaluate the effectiveness of our summarized information above (Section 2.2) to help stakeholders diagnose the performance violations of configurable software systems. Moreover,

Table 1: Overview of target systems

System	Domain	#Opt.	#KLOC	V/ID	Overhead
BATIK	SVG rasterizer	31	360	1.14	6h
Cassandra	Database	172	697	4.0.5	10h
Catena	Password hashing	12	6.6	9c89da4	≤1h
DConverter	Image Density Converter	24	49[1]	bdf1535	≤1h
H2	Database	16	340	2.1.210	3h
Kanzi	Data compressor	40	28.8	2.0.0	≤1h
Prevayler	Database	12	14.4	2.6	3h
Sunflow	Rendering engine	6	27.4	0.07.2	≤1h

[1] : Includes source code for several libraries invoked by image processing; Opt: The number of options; V/ID: Version/Commit ID; Overhead: Time required for static taint analysis.

we further answer the following research questions to evaluate the effectiveness of our approach.

RQ$_4$: The performance of DiagConfig. Can DiagConfig work well for performance violation diagnosis of configurable systems? Can DiagConfig speed up the existing auto-tuning process?

5.1 Experiment Setup

Hardware. We use two environments, one with 128GB of RAM, 24 cores and 48 threads of Intel Xeon Silver 4116 processor running Ubuntu 18.04, which is only used for static taint analysis, and the other with 48GB of RAM, 4 cores and 8 threads of Intel Core i7-7700 processor running Ubuntu 20.04 desktop version.

Target Systems. We select eight configurable, real-world, open-source Java systems from different domains, including databases, image processing, and data compression. Table 1 provides an overview of all target configurable software systems. All of them satisfy the following criteria: (1) medium- to large-scale systems with binary, enumerated, and continuous configuration options; (2) systems used for evaluation by previous research on performance modeling. We reuse the workloads and benchmarks evaluated in the existing literature [59] in each target system.

5.2 RQ$_1$: Accuracy of DiagConfig in identifying performance-sensitive options

To collect the ground truth of performance-sensitive options for each system, we first conducted an experimental study to investigate which options influence the performance of the system in the respective benchmark. To achieve that, we deployed each system and changed each option one by one, and checked whether the average time variation over 5% in different executions (5 repetitions to reduce measurement bias [62]). Finally, we follow the same methodology as in 3.1 to manually understand the relationship between each option and the performance-related operations at the source code level and reviewed the results of experimental study.

We divide the target systems into two categories, one as a training set for building classification models, including Batik, Catena, H2, Prevayler, and Sunflow, and the other as a test set for testing the accuracy of the models, including Dconverter, Kanzi. Once the classification model performs well enough on the test set, we evaluate the effectiveness of the model by identifying performance-sensitive options for the large database system Cassandra and comparing it to the state-of-art.

Comparison with state-of-the-art. SafeTune [22], a state-of-the-art approach for selecting performance-related parameters

Figure 5: Results of performance-sensitive options identification.

by building models and analyzing on system configuration-related documentation. We compare DiagConfig with SafeTune by evaluating them on Cassandra application since it is one of the publicly released datasets in SafeTune. Specifically, the ground truth collected by running Cassandra in the popular YCSB [12] benchmark.

Result and Analysis. Our classification model constructed from the training set figured out 93.3% (14/15) of performance-sensitive options on the test set, so our model is valid. Moreover, we leverage it to identify Cassandra's performance-sensitive options. We fed 172 options to taint tracking, 67 of which influence pre-defined performance-related operations. And we identified 37 options are performance-sensitive based on the model. In comparison, SafeTune identified 116 performance-related parameters out of 117 parameters, which is available in their public dataset.

The comparison results are shown in Figure 5, both approaches produce false negatives. DiagConfig misses 17 performance-sensitive options (i.e., region *001* + region *011*) and SafeTune misses 14 (i.e., region *001* + region *101*). There are 10 performance-sensitive options (i.e., region *101*) identified by our approach but missed by SafeTune. And DiagConfig misses 13 performance-sensitive options (i.e., region *011*) but were accurately classified by SafeTune. Furthermore, SafeTune produces 77 (i.e., region *010* + region *110*) false positives and DiagConfig produces 5 false positives (i.e., region *100* + region *110*). We study the 17 performance-sensitive options that we misclassified and summarize some of the sources that resulted in the misclassification as follows: (1) the considered performance-related operations are not complete; e.g., the option *ideal_consistency_level* dependent on consistency maintenance-related operations that are not included in the Java standard library; (2) performance property is a simple count, which does not fully reflect runtime performance behavior; e.g., the performance property of the option *periodic_commitlog_sync_lag_block_in_ms* indicates that it is thread-related, but the small count causes it to be misclassified.

Discussion on documentation and source code. Current interpretations of the relationship between configuration and performance are mostly based on documentation rather than source code. The information that documentation can provide is mostly systematic and macroscopic, while the information provided by the source

code is mostly rational-logical and microscopic. The trade-off between the information provided by documentation and source code can facilitate the development of interpretability related to options and performance.

Summary *for RQ_1: For performance-sensitive option identification, our static code analysis-based approach can introduce fewer false positives than the state-of-the-art documentation-based approach. The data- and control-flow dependencies between options and performance-related operations in source code can better reflect the runtime behavior of the system than documentation.*

5.3 RQ_2 and RQ_3: Effectiveness of cause-effect chains identified by DiagConfig

In this paper, we focus on the diagnosis of performance violations caused by configuration changes. Therefore, performance violations caused by other reasons such as dynamic workload and environment migration are out of the scope of this paper.

Diagnosis of Configuration Changes. First, we select a baseline or default configuration based on the relevant document and treat the application performance under this configuration as the SLO for each system. Then, we determine the range of each option. With the same sampling strategy as Weber et al. [62], which has been evaluated for accurate performance modeling, we select feature-wise and pair-wise sampling for binary options as well as Plackett-Burman sampling [60] for enumeration and continuous options to obtain a set of valid configurations. We load these configurations into the system and get corresponding performance measurements in a specific benchmark. Note that in the evaluation, the goal is not to find the optimal configurations, but to diagnose performance violations.

We run DiagConfig to obtain the set of configuration options responsible for performance violations. To evaluate the effectiveness of diagnosing performance violations, two widely used metrics are adopted, namely Precision and Recall. Precision is the ratio between the set of correctly identified options and the set of predicted options, which can be formulated as $Precision = \frac{TP}{TP+FP}$. Recall is calculated as the percentage of correctly identified options causing a performance violation and can be defined as $Recall = \frac{TP}{TP+FN} = \frac{TP}{Diff}$. $Diff$ represents the set of options whose values have changed compared to the baseline. The options in $Diff$ that we identified are TP, and those we did not identify are FN. For example, if the configuration is different from the baseline configuration with two options, A and B. Identifying both is considered exactly correct, and identifying only one or neither corresponds to a recall of 50% and 0% respectively.

Result and Analysis. Table 2 summarizes the average precision and recall of each system in diagnosing performance violations. The result shows the effectiveness of our approach. For the reasons of false negatives (which lead to loss of recall), we manually double-checked the offline analysis corresponding to the missed options and concluded that (1) the information-flow is lost due to the small depth of alias analysis in taint tracking. The larger the depth the higher the offline analysis overhead required, which we set to 5 in our experiments. (2) there are few options that influence compute-intensive operations (e.g., operations of primitive numeric types) without involving the performance-related operations we have agreed upon, leading to incomplete information-flow paths. Also, we found more options responsible for performance violations (which lead to loss of precision) that deserve further tuning.

Table 2: Average of Precision and Recall.

	Batik	Catena	H2	Kanzi	Prevayler
Configurations	128	2433	573	494	118
Precision	0.911	0.809	0.905	0.806	0.788
Recall	0.807	0.956	0.815	0.932	0.952

Configurations: The total number of valid configurations (invalid ones are filtered out) obtained by sampling, which is traded off against the performance measurement overhead.

Interpretability of cause-effect chains. The information-flow inherently carries how options affect performance-related operations. And call-chains contain dynamic dependencies of various methods (functions) within the program. Given the cause-effect chains like Figure 3, stakeholders can follow the path and finally understand the root cause of performance violations. However, not every stakeholder desires such detailed cause-effect information, but rather seeks interpretability. For example, they focus mainly on the contribution of options to the performance-violating hotspot methods and distinguish the importance of the options. Associating cause-effect chains with performance metrics can help understand different performance violations. To show the interpretability of the cause-effect chains without loss of generality, in addition to execution time, we also consider throughput. We collect one execution time violation in Catena and one throughput violation in the H2 database to demonstrate the interpretability of cause-effect chains. Then, we rank the hotspot methods based on their performance variations and calculate the relative importance of the influencing options by accumulating performance variation.

Result and Analysis. We present the results of Catena and H2 separately in Figure 6. There are influencing options and their interactions responsible for performance-violating hotspot methods, and the options vary in importance. In Catena (top half of Figure 6), the options *gHigh* and *gLow* are primarily responsible for the execution time violation, and they are statically or dynamically dependent on almost all hotspot methods with significant performance variations. Moreover, half of the options are irrelevant to the violation. Similarly, the bottom half of Figure 6 shows that three-quarters of the options are irrelevant to the throughput violation in H2. Such summary information indicates that the root cause of performance violations lies in some internal crucial components with significant performance variation. Besides, this kind of information not only helps pre-select some crucial options for tuning and interpretation but also locates hotspot methods that may suffer from performance bugs, facilitating the code refactoring of the system.

Summary *for RQ_2 and RQ_3: Treating the system as a white-box and taking the advantage of performance-related operations, DiagConfig effectively diagnoses performance violations by building cause-effect chains. In addition, the cause-effect chains achieve coarse- and fine-grained interpretability, which helps stakeholders understand the root causes of performance violations.*

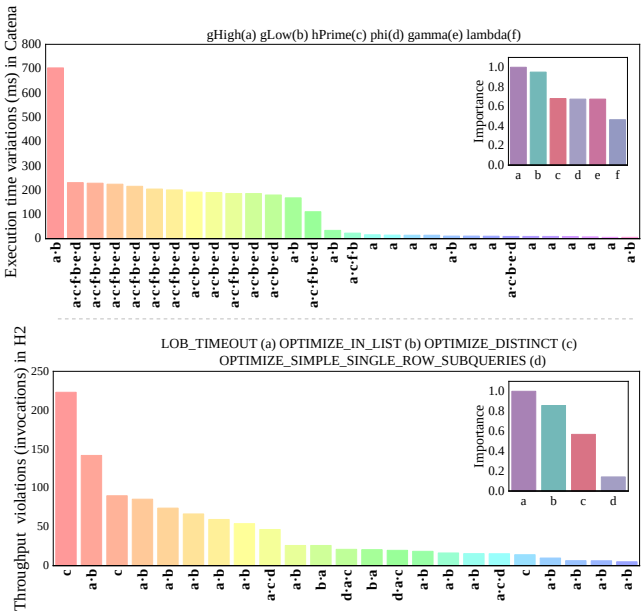


Figure 6: Overview of the impact of options and interactions on Catena execution time violation (top) and on H2 Database throughput violation (bottom). The background plot shows the ordering of hotspot methods (we omitted the method signature) by performance variations. Each bar is a hotspot. The influencing options and interactions are listed below the hotspot methods. The small internal plot shows the importance of the options, obtained by accumulating and normalizing the associated hotspots' performance variation.

5.4 RQ$_4$: Performance of DiagConfig

Overhead. As Figure 2 shows, DiagConfig consists of offline analysis and online diagnosis. Thus, the overhead of DiagConfig comes from these two parts. The overhead required for offline analysis includes static taint analysis, classification model building, and performance-sensitive information-flow paths filtering. Table 1 lists the overhead for static taint analysis of each target system in our evaluation. It is relatively heavy but acceptable because we only need to run static taint analysis once. Moreover, the software vendors generally can provide the results of this part. The other overhead in the offline analysis is almost negligible compared to the static taint analysis. We mentioned in section 7 that there are extensive research and tools focused on online monitoring that can be lightweight while balancing overhead and completeness. Similarly, many profilers in the industry have proven to be efficient in debugging the runtime behavior of software systems. Therefore, we do not need to replace them but build on industry-class profilers to achieve performance violation detection in online diagnosis. Given this, we mainly discuss the overhead of the building of cause-effect chains. When diagnosing performance violations, we record the execution time of each diagnosis and summarize it into the metrics shown in Table 3.

Result and Analysis. Experimental results show that our method has an acceptable cost. The cold start is that DiagConfig loads the necessary program static information to build the call graph when constructing the cause-effect chains for the first time. It is a one-shot cost. The acceptable cost shows that our approach is suitable for the vast majority of production environments and can be integrated into the existing auto-tuners.

Table 3: Overhead of Cause-effect Chains Building.

	Batik	Catena	H2	Kanzi	Prevayler
Total Time	666.105s	37.419s	324.705s	46.569s	23.020s
Cold Start (Maximum)	68.248s	24.207s	47.920s	19.903s	20.638s
Second Largest	6.104s	73ms	3.716s	531ms	155ms
Minimum	2.915s	3ms	220ms	3ms	10ms
Mean	4.7233s	5.65ms	549.8ms	56.2ms	20.9ms
Standard Deviation	769.1ms	101.8ms	267.3ms	88.5ms	21.8ms

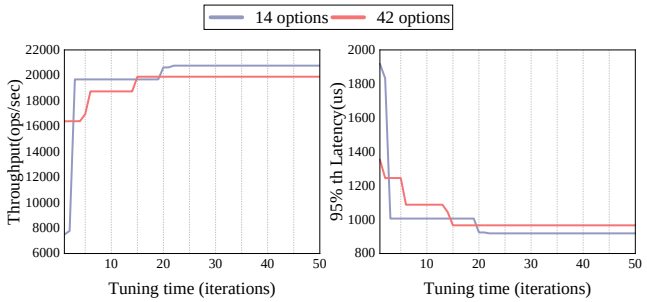


Figure 7: Tunning Example. Comparison of OtterTune tuning process with and without DiagConfig assistance.

Case Study. DiagConfig can recommend crucial options to guide tuning tools (e.g., *SmartConf* [61], OtterTune [55], DAC [72], BestConfig [75]) to reduce the huge configuration space. We conduct a case study with OtterTune, a representative auto-tuner, to show the effectiveness of DiagConfig. OtterTune supports MySQL and PostgreSQL, but neither of them is a Java system. Thus, we extend OtterTune to work for the Java database Cassandra. We apply DiagConfig to figure out the crucial options when a performance violation occurs and then run OtterTune to improve performance. We record OtterTune's iterations to validate how much DiagConfig accelerates the tuning process. In the case study, we simulate a situation where Cassandra suddenly suffered significant throughput and latency degradation under YCSB benchmark. The stakeholders point out the options related to this performance violation and then leverage OtterTune to improve the system's performance. By contrast, we run DiagConfig to select crucial options before tuning.

Result and Analysis. DiagConfig recommended 14 options, while the stakeholders offered 42 relevant options according to their experience. We fed these two sets of options to OtterTune separately. The tuning process is shown in Figure 7. OtterTune gets stuck in the configuration space that consists of the 42 options. Tuning with DiagConfig requires only 2 iterations to achieve 98% of the throughput obtained by tuning without DiagConfig through 14 iterations. This is an almost 7× acceleration. Moreover, after the 19th iteration, the tuning with DiagConfig further achieves a better throughput. Similarly, the acceleration of OtterTune by DiagConfig is also reflected in the latency.

***Summary** for RQ$_4$: The auto-tuners can be stuck in a huge configuration space leading to a slow tuning speed; DiagConfig with acceptable overhead is complementary to them, which accelerates the tuning process by compressing configuration space.*

6 LIMITATIONS AND THREATS TO VALIDITY

Limitations of the Static Taint Analysis. DiagConfig computes information-flow between options and performance-related operations in the offline analysis may produce inaccurate results. The main source of inaccuracy is that static taint analysis requires a trade-off between accuracy and overhead when confronted with path explosion and alias analysis, leading to overtainting or loss of taint. If the analysis misses all information-flow, then DiagConfig will fail to construct cause-effect chains. In contrast, if the analysis falsely reports too many information-flow paths, resulting in many redundant cause-effect chains. Additionally, while FlowDroid [3] holds a high accuracy, the analysis is challenged by the explosion of paths between source and sink as well as the size of the call graph. As a result, these challenges limit the scale of the target system that DiagConfig can analyze. Our evaluation demonstrates the overhead of DiagConfig based on FlowDroid for static taint tracking in the target system. Although the cost of analyzing large-scale configurable software systems is relatively heavy, it is acceptable.

Threats to Validity. The selection of the profiler for performance violation detection is a threat to construct validity. Profiling generally indicates a overhead, resulting in performance degradation of the software system. We mitigated this threat by selecting a lightweight profiler, Jprofiler, for performance-violating hotspot functions detection and localization. Besides, the setting of the profiler is also a threat. Our evaluation of the target system Batik SVG Rasterizer (128 valid configurations generate 51GiB call-chains) shows that persisting profiling information for each hotspot function leads to expensive storage costs. Mitigating this threat requires the user to understand the target system well enough and set the profiler blacklist to ignore specific program elements. The selection and setting of the profiler are threats to internal validity.

The choice of target systems threatens external validity, on which we evaluate the effectiveness of our approach. To alleviate this threat, we introduced various systems with multiple options from different areas in our evaluation. They are collected from previous work [62, 59, 57] and usually used to evaluate sampling strategies and performance-modeling methods. We further run our approach with OtterTune on the Cassandra database to show the feasibility of large-scale configurable software systems.

7 RELATED WORK

Generally speaking, software configuration tuning has three steps, namely detection of performance violations, identification of root causes, and searching for optimal configurations. DiagConfig mainly targets on the second step.

Performance violations occur frequently due to changes in workload and environment as well as misconfigurations. There is substantial literature on detecting [24, 73], testing [33, 67], diagnosing [4, 5], and fixing [63, 35] misconfigurations. While these solutions help reduce misconfigurations introduced by users' mistakes, interpretability is still an open problem. Our goal is to restore the cause-effect relationships between performance-sensitive options and performance violations based on the program logic. In particular, stakeholders want a clear explanation of why there was a performance violation when they had set up a configuration that seems better according to the documentation.

There is no silver bullet to finding a configuration that performs well in all situations. Off-the-shelf profilers [42, 52, 30], targeted profiling techniques [71, 11], and visualizations [11, 1] help detect performance problems and locate performance bottlenecks. However, there is not enough evidence to explain **why** options cause performance violations, particularly to determine **which** options are responsible for performance violations. DiagConfig strives to recommend crucial options by cause-effect chains.

Similarly, most previous works target at helping stakeholders understand why, where, and how options and their interactions affect the performance behavior of configurable software systems by building white-box performance-influence models [57, 59, 62, 49]. ConfigCrusher [57] first relies on static taint to determine which options affect which code regions. Then it leverages option-affected code region expansion and merging with instrumentation to reduce the cost of measurement and construct interpretable performance-influence models. However, the instrumentation is overhead and does not support numeric options and multi-threaded programs. COMPREX [59] builds white-box performance-influence models based on expensive dynamic taint analysis and incomplete configuration-specific local code performance measurement. Weber et al. [62] propose an approach based on SPLConqueror [49, 51, 50] to build white-box performance models over binary and numeric options at the method level for understanding options and their interactions. It achieves relatively high-precision because it combines coarse and fine profiling to reduce the influence of performance variance on the models. All approaches based on performance models involve repeated performance measurements of the system in specific workloads and environments. In addition to the difficulty of model transfer, the performance of the models themselves varies depending on the sampling strategy and learning tricks.

Optimizers for software configuration tuning that treat the system as a black box and contain limited interpretable information about the relationship between options and performance behavior. They can be classified into two categories: control-theory-based [61, 23] and machine-learning-based [10, 9, 69, 55, 72, 75] approach.

8 CONCLUSION

We propose a white-box static code analysis-based approach to diagnose performance violations of configurable software systems. This approach combines static configuration-related performance information from source code and runtime performance behaviors from profiling. Moreover, we implement a novel prototype, DiagConfig, to diagnose performance violations. It performs option tracking, performance violation localization, and construction of cause-effect chains. Our evaluation on eight open-source systems demonstrates the effectiveness and efficiency of DiagConfig. More importantly, DiagConfig can restore the complete evidence chain of performance violations, highlight the contribution of configuration options to performance violations, help stakeholders explain the causes of performance violations, and accelerate the configuration tuning process regardless of workloads and environments.

9 DATA AVAILABILITY

All implementation and data can be found in archived repository [13].

REFERENCES

[1] Juan Pablo Sandoval Alcocer, Fabian Beck, and Alexandre Bergel. 2019. Performance evolution matrix: visualizing performance variations along software versions. In *2019 Working conference on software visualization (VISSOFT)*. IEEE, 1–11.

[2] George Amvrosiadis and Medha Bhadkamkar. 2016. Getting back up: understanding how enterprise data backups fail. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 479–492.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49, 6, 259–269.

[4] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: automating {root-cause} diagnosis of performance anomalies in production software. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 307–320.

[5] Mona Attariyan and Jason Flinn. 2010. Automating configuration troubleshooting with dynamic information flow analysis. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*.

[6] Leo Breiman. 2001. Random forests. *Machine learning*, 45, 1, 5–32.

[7] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. 2010. Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, 301–310.

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

[9] Chi-Ou Chen, Ye-Qi Zhuo, Chao-Chun Yeh, Che-Min Lin, and Shih-Wei Liao. 2015. Machine learning-based configuration parameter tuning on hadoop system. In *2015 IEEE International Congress on Big Data*. IEEE, 386–392.

[10] Tao Chen and Miqing Li. 2021. Multi-objectivizing software configuration tuning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 453–465.

[11] Jürgen Cito, Philipp Leitner, Christian Bosshard, Markus Knecht, Genc Mazlami, and Harald C Gall. 2018. Performancehat: augmenting source code with runtime performance traces in the ide. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 41–44.

[12] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, 143–154.

[13] DiagConfig. 2022. Diagconfig - configuration diagnosis of performance violations in configurable software systems. https://anonymous.4open.science/r/diagconfig-AEDE. Accessed June 6, 2022. (2022).

[14] Zhen Dong, Artur Andrzejak, David Lo, and Diego Costa. 2016. Orplocator: identifying read points of configuration options via static analysis. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 185–195.

[15] Jiaqing Du, Nipun Sehrawat, and Willy Zwaenepoel. 2011. Performance profiling of virtual machines. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments*, 3–14.

[16] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9, 3, 319–349.

[17] Google. 2019. Google cloud storage incident #19002. https://status.cloud.google.com/incident/storage/19002. Accessed March 13, 2020. (2019).

[18] Susan L Graham, Peter B Kessler, and Marshall K McKusick. 1982. Gprof: a call graph execution profiler. *ACM Sigplan Notices*, 17, 6, 120–126.

[19] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-aware performance prediction: a statistical learning approach. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 301–311.

[20] Huong Ha and Hongyu Zhang. 2019. Performance-influence model for highly configurable software with fourier learning and lasso regression. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 470–480.

[21] Xue Han and Tingting Yu. 2016. 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*, 1–10.

[22] Haochen He, Zhouyang Jia, Shanshan Li, Yue Yu, Chenglong Zhou, Qing Liao, Ji Wang, and Xiangke Liao. 2022. Multi-intention-aware configuration selection for performance tuning. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 1431–1442.

[23] Henry Hoffmann et al. 2012. Self-aware computing in the angstrom processor. In *Proceedings of the 49th Annual Design Automation Conference*, 259–264.

[24] Peng Huang, William J Bolosky, Abhishek Singh, and Yuanyuan Zhou. 2015. Confvalley: a systematic configuration validation framework for cloud services. In *Proceedings of the Tenth European Conference on Computer Systems*, 1–16.

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

[26] Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. 2018. Learning to sample: exploiting similarities across environments to learn performance models for configurable systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 71–82.

[27] Santosh Janardhan. 2021. Update about the october 4th outage. https://engineering.fb.com/2021/10/04/networking-traffic/outage/. Accessed October 4, 2021. (2021).

[28] Dongpu Jin, Xiao Qu, Myra B Cohen, and Brian Robinson. 2014. Configurations everywhere: implications for testing and debugging in practice. In *Companion Proceedings of the 36th International Conference on Software Engineering*, 215–224.

[29] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47, 6, 77–88.

[30] Tomas Hurka Jiri Sedlacek. 2019. Visualvm: all-in-one java troubleshooting tool. https://visualvm.github.io/index.html. Accessed April 6, 2022. (2019).

[31] Wall Street Journal. 2019. Facebook, google and apple hit by unusual outages. https://www.wsj.com/articles/facebook-and-instagram-suffer-lengthy-outages-11552539752. Accessed March 14, 2019. (2019).

[32] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. 2020. The interplay of sampling and machine learning for software performance prediction. *IEEE Software*, 37, 4, 58–66.

[33] Lorenzo Keller, Prasang Upadhyaya, and George Candea. 2008. Conferr: a tool for assessing resilience to human configuration errors. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 157–166.

[34] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. 2019. Tradeoffs in modeling performance of highly configurable software systems. *Software & Systems Modeling*, 18, 3, 2265–2283.

[35] Nate Kushman and Dina Katabi. 2010. Enabling {configuration-independent} automation by {non-expert} users. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*.

[36] Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. 2020. Statically inferring performance properties of software configurations. In *Proceedings of the Fifteenth European Conference on Computer Systems*, 1–16.

[37] Max Lillack, Christian Kästner, and Eric Bodden. 2014. Tracking load-time configuration options. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 445–456.

[38] Mahoney Matt, Collet Yann, Ondrus Jan, Mori Yuta, Muravyov Ilya, Burns Neal, Giesen Fabian, Duda Jarek, and Grebnov Ilya. 2019. Kanzi: a lossless data compressor implemented in java. https://github.com/flanglet/kanzi. Accessed July 13, 2014. (2019).

[39] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 643–654.

[40] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2018. Faster discovery of faster system configurations with spectral learning. *Automated Software Engineering*, 25, 2, 247–277.

[41] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. 2018. Finding faster configurations using flash. *IEEE Transactions on Software Engineering*, 46, 7, 794–811.

[42] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *ACM Sigplan notices*, 42, 6, 89–100.

[43] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding near-optimal configurations in product lines by random sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 61–71.

[44] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, 199–209.

[45] F. Pedregosa et al. 2011. Scikit-learn: machine learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.

[46] Ariel Rabkin and Randy Katz. 2011. Precomputing possible configuration error diagnoses. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 193–202.

[47] Ariel Rabkin and Randy Katz. 2011. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering*, 131–140.

[48] Marija Selakovic, Thomas Glaser, and Michael Pradel. 2017. An actionable performance profiler for optimizing the order of evaluations. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 170–180.

[49] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 284–294.

[50] Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. 2012. Predicting performance via automated feature-interaction detection. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 167–177.

[51] Norbert Siegmund, Marko Rosenmuller, Christian Kastner, Paolo G Giarrusso, Sven Apel, and Sergiy S Kolesnikov. 2011. Scalable prediction of non-functional properties in software product lines. In *2011 15th International Software Product Line Conference*. IEEE, 160–169.

[52] ej-technologies. 2019. Jprofiler: the award-winning all-in-one java profiler. https://www.ej-technologies.com/products/jprofiler/overview.html. Retrieved December10,2019. (2019).

[53] Xinhui Tian, Rui Han, Lei Wang, Gang Lu, and Jianfeng Zhan. 2015. Latency critical big data computing in finance. *The Journal of Finance and Data Science*, 1, 1, 33–41. DOI: https://doi.org/10.1016/j.jfds.2015.07.002.

[54] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: a java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, 214–224.

[55] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*, 1009–1024.

[56] André Van Hoorn, Jan Waller, and Wilhelm Hasselbring. 2012. Kieker: a framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC international conference on performance engineering*, 247–248.

[57] Miguel Velez, Pooyan Jamshidi, Florian Sattler, Norbert Siegmund, Sven Apel, and Christian Kästner. 2020. Configcrusher: towards white-box performance analysis for configurable systems. *Automated Software Engineering*, 27, 3, 265–300.

[58] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2022. On debugging the performance of configurable software systems: developer needs and tailored tool support. *arXiv preprint arXiv:2203.10356*.

[59] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2021. White-box analysis over machine learning: modeling performance of configurable systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1072–1084.

[60] JC Wang and CF Jeff Wu. 1995. A hidden projection property of plackett-burman and related designs. *Statistica Sinica*, 235–250.

[61] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. 2018. Understanding and auto-adjusting performance-sensitive configurations. *ACM SIGPLAN Notices*, 53, 2, 154–168.

[62] Max Weber, Sven Apel, and Norbert Siegmund. 2021. White-box performance-influence models: a profiling and learning approach. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1059–1071.

[63] Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen. 2017. Replication-driven live reconfiguration for fast distributed transaction processing. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 335–347.

[64] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the flow: profiling copies to find runtime bloat. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 419–430.

[65] Guoqing Xu and Atanas Rountev. 2010. Detecting inefficiently-used containers to avoid bloat. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 160–173.

[66] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early detection of configuration errors to reduce failure damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 619–634.

[67] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 244–259.

[68] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2012. Uncovering performance problems in java applications with reference propagation profiling. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 134–144.

[69] Nezih Yigitbasi, Theodore L Willke, Guangdeng Liao, and Dick Epema. 2013. Towards machine learning-based auto-tuning of mapreduce. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 11–20.

[70] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. 2011. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 159–172.

[71] Tingting Yu and Michael Pradel. 2018. Pinpointing and repairing performance bottlenecks in concurrent programs. *Empirical Software Engineering*, 23, 5, 3034–3071.

[72] Zhibin Yu, Zhendong Bei, and Xuehai Qian. 2018. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 564–577.

[73] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. 2011. Context-based online configuration-error detection. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, 28–28.

[74] Andreas Zeller. 1999. Yesterday, my program worked. today, it does not. why? *ACM SIGSOFT Software engineering notes*, 24, 6, 253–267.

[75] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, 338–350.