DiagConfig: Configuration Diagnosis of Performance Violations in Configurable Software Systems

Zhiming Chen
Sun Yat-sen University
Guangzhou, China

Pengfei Chen
Sun Yat-sen University
Guangzhou, China

Peipei Wang
ByteDance US Infrastructure System Lab
Seattle, USA

Guangba Yu
Sun Yat-sen University
Guangzhou, China

Zilong He
Sun Yat-sen University
Guangzhou, China

Genting Mai
Sun Yat-sen University
Guangzhou, China

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. Experimental evaluations with eight real-world software demonstrate that DiagConfig produces fewer false positives than a state-of-the-art documentation analysis-based tool (i.e., 5 vs 41) in the identification of performance-sensitive options, and outperforms a statistics-based debugging tool in the diagnosis of performance violations caused by configuration changes, offering more comprehensive results (recall: 0.892 vs 0.289). Moreover, 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, Program analysis, Performance violation, Taint tracking

ACM Reference Format:
Zhiming Chen, Pengfei Chen, Peipei Wang, Guangba Yu, Zilong He, and Genting Mai. 2023. DiagConfig: Configuration Diagnosis of Performance Violations in Configurable Software Systems. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616300

1 INTRODUCTION

Modern software systems are highly configurable to meet users' requirements in various scenarios. Take the popular open-source database management system MySQL as an example. Its latest version has reached around 1,000 configuration options. It is always challenging to make the proper configurations for software deployment. Studies have shown that misconfiguration is one of the primary culprits responsible for production system failures and performance problems [2, 36, 79, 84]. As system performance is becoming more and more critical in enterprise business [24, 35, 39], misconfiguration can lead to millions of dollars cost [66]. 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 [9, 11, 28, 37, 55, 72, 88] due to its huge search space. Search-based techniques are exploited to change the value of configuration options by trial-and-error [32, 50, 54] 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 [13, 51, 68]. This motivates us to explore other techniques to tame the cause-effect relationships for eliminating the effort of trial-and-error and figuring out the crucial options.

The cause-effect relationships between configuration 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 [72] 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

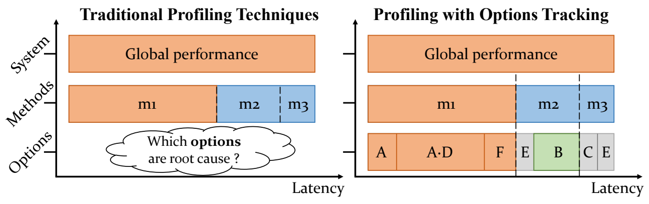

Traditional Profiling Techniques
Profiling with Options Tracking
System
Methods
Options
Global performance
m1
m2
m3
Which options are root cause ?
A
A·D
F
E
B
C
E
Latency

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

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 with software updates. Thus, we seek to diagnose and restore the evidence for performance violations via automatic cause inference.

When performance violations occur, there are substantial monitoring and profiling techniques whose goal is to locate performance hotspots [20, 38, 52, 60, 69]. 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 [61] help understand the influence of options on system performance [26, 27, 33, 34, 37, 42, 70–72, 75]. The accuracy of performance models heavily depends on the selected model [42] and the subset of the configuration space generated by various sampling strategies [40, 47]. 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 [13, 51]. Previous attempts at building white-box performance models of configurable systems [70–72, 75] are deficient for general-purpose dynamic workloads and environments. Besides, as the software evolves and the number of options increases, it is becoming more expensive to keep the performance models up to date.

Given the discussion above, our goal is to devise a general low-cost, high-precision technique for configurable software systems that can not only interpret cause-effect relationships between options and performance violations but also infer 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 static code information from taint tracking and runtime profiling information to build cause-effect chains which can help stakeholders explain performance issues. We evaluated DiagConfig with eight real-world open-source projects and the results show that DiagConfig produced fewer false positives than SafeTune [29] (i.e., 5 vs 41), a documentation-analysis based tool, in the identification of performance-sensitive options. Moreover, DiagConfig is fast and supports a more comprehensive diagnosis of performance violations compared to Unicorn [32] (recall: 0.892 vs 0.289), a statistics-based debugging tool. We also integrated DiagConfig into an auto-tuner and demonstrated its feasibility of underpinning prior works on configuration performance tuning.

Our key contributions are as follows.

- 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.

2 BACKGROUND AND RESEARCH QUESTIONS

In this section, we first introduce basic concepts of 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 [19]. Profiling investigates how much of a resource each program element consumes and reports performance-critical program elements as **hotspots**. 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 [25] and unnecessarily high memory consumption profiling [77, 78, 81]. These approaches track how hotspots are invoked; in particular, stakeholders follow the traces (e.g., call-chains) to diagnose *unexpected* performance behavior. However, limited evidence in the profiling clarifies the relationships between options and hotspots. Stakeholders have to navigate the source code to find hotspot-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 [18, 45, 57, 58, 79, 80]. These 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 behavior. Taint tracking with configuration as a 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 [72]. However, any option that affects performance during runtime must have a data- or control-flow dependency with performance-related operations [44]. **Performance-related operations** refer to code snippets in a program contributing positively to execution time (time-expensive) and memory consumption (memory-expensive). In this paper, we follow the prior work [44], and consider 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; and (3) operations that affect system concurrency (e.g., creation of threads or thread pools). These operations are computationally expensive, and potentially need paging or swapping with low-speed devices or context-switching. 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.

To answer this question, we first treat configuration as the source and mine the information-flow paths that record the dependencies between performance-related operations and options via taint tracking. Since not all performance-related operations have a significant performance impact, we characterize the dependency information and then utilize the random forest [8] to identify performance-sensitive options. The details of this process are presented in § 3.1 and the evaluation of its effectiveness is discussed in § 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 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 options and performance violations. To answer this question, we use performance-related operations as the intermediary. Once the options are used in statements that influence 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 § 3.3 and call-chains (between operations and hotspot functions), and evaluate the effectiveness of this approach in § 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** (§ 3.1) identifies performance-sensitive options by revealing the dependencies between configuration options and performance-related operations. 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 builds information-flow paths from option values to performance-related operations via taint tracking, extracts options' static performance properties, and then classifies performance-sensitive options and information-flow paths. **Online diagnosis** (§ 3.2 and § 3.3) continuously monitors system runtime behavior, locates performance-violating hotspot functions, and collects call-chains for the performance-violating hotspot functions from the profiling. It builds cause-effect chains with information-flow paths and call-chains, which further 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.

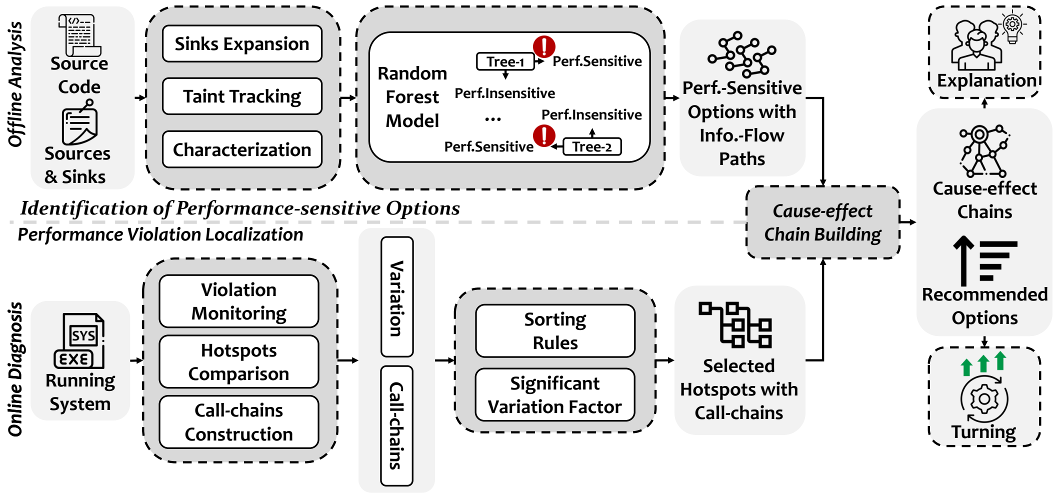


Figure 2: Overview of DiagConfig

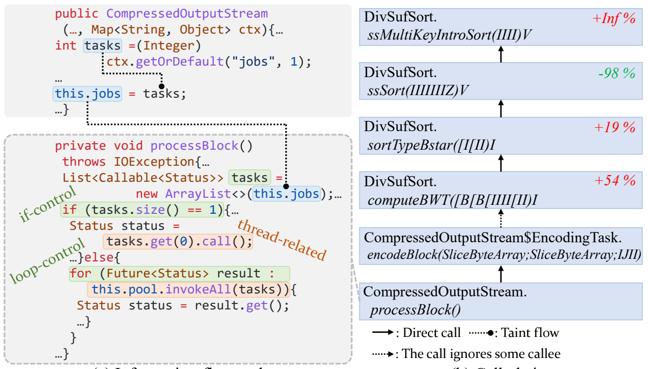
3.1 Identification of Performance-sensitive Options

The identification of performance-sensitive options is the goal of the offline analysis. For each option, DiagConfig first computes the information-flow paths that record the data- and control-flow dependencies between variables derived from the option and performance-related operations via taint tracking. Subsequently, DiagConfig characterizes these paths as performance property (i.e., feature vector), which is utilized as input for a random forest model to determine whether the option is performance-sensitive or not. The training process of the random forest is presented in § 5.2.

Taint tracking. Taint tracking tracks 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 standard configuration loading statements, which are called **sources**. Taints are then propagated and transformed along the program's data-flow paths, until they are consumed in sink statements. Besides pre-defined performance-related operations (i.e., **sinks**), DiagConfig marks *if/switch* statements with branches containing performance-related operations and *loop* startup/jump-out statements with the body containing performance-related operations as **expanded sinks** (see§ 4). Once the taints reach sinks, DiagConfig builds the **information-flow paths** for the corresponding options that record performance-related operations, taints propagation paths, and location (e.g., source code file name, source code line number) of code snippets where the dependencies between options and performance-related operations occur.

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 (i.e., *call(), invokeAll()*) via taint tracking. Then, DiagConfig builds information-flow paths for the option *jobs* that records the *thread-related* operations and the location where the dependencies occur, including method signature, if statement, loop startup statement, and source code line number.

Note that not all performance-related operations significantly affect system runtime performance. To identify performance-sensitive



```
public CompressedOutputStream
 (…, Map<String, Object> ctx){…
int tasks =(Integer)
       ctx.getOrDefault("jobs", 1);
…
this.jobs = tasks;
…}
```

```
private void processBlock()
 throws IOException{…
 List<Callable<Status>> tasks =
           new ArrayList<>(this.jobs);…
 if (tasks.size() == 1){…
  Status status =
      tasks.get(0).call();
  …}else{
  for (Future<Status> result :
     this.pool.invokeAll(tasks)){
   Status status = result.get();
   …}
  }
…}
```

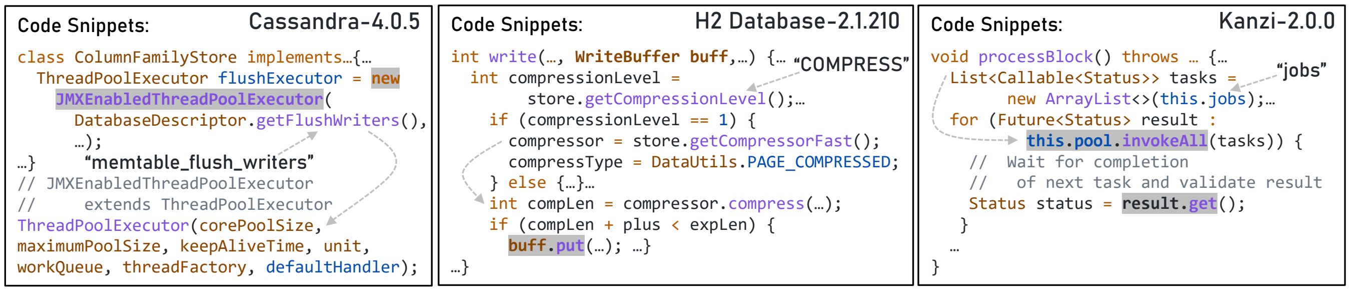
if-control
thread-related
loop-control
DivSufSort. ssMultiKeyIntroSort(IIII)V +Inf %
DivSufSort. ssSort(IIIIIIIZ)V -98 %
DivSufSort. sortTypeBstar([I[II)I +19 %
DivSufSort. computeBWT([B[B[IIII[II)I +54 %
CompressedOutputStream$EncodingTask. encodeBlock(SliceByteArray;SliceByteArray;IJII)
CompressedOutputStream. processBlock()
: Direct call
: Taint flow
: The call ignores some callee

(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* to diagnose *where, how, and why* the option *jobs* causing a performance violation in Kanzi [46]. DiagConfig backtracks the call-chain (b) and analyzes each callsite. In the method *processBlock*, it identifies the program dependencies between the option *jobs* and thread-related operations that invoke *encodeBlock* (the callee of the *processBlock*). Then, it connects the information-flow path and the call-chain to a cause-effect chain.

options, we still need to characterize the information-flow paths of the configuration options as **performance properties** and build a random forest classification model with the labeled configuration options' performance properties.

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. The original performance property is a high-dimensional (e.g., at least 176 classes and their 2221 methods under the *java.nio* package belonging to I/O operations for Java applications) and sparse vector where each count is relatively small. It influences the splitting decision for the random forest and makes the random forest struggle to generalize well. Inspired by the best practices of feature abstraction for sparse high-dimensional feature spaces in text classification [7, 10, 64], we study the information-flow paths of configuration options and conclude



```
class ColumnFamilyStore implements…{…
  ThreadPoolExecutor flushExecutor = new
    JMXEnabledThreadPoolExecutor(
      DatabaseDescriptor.getFlushWriters(),
      …);
…}
// JMXEnabledThreadPoolExecutor
//     extends ThreadPoolExecutor
ThreadPoolExecutor(corePoolSize,
maximumPoolSize, keepAliveTime, unit,
workQueue, threadFactory, defaultHandler);
```

```
int write(…, WriteBuffer buff,…) {…
  int compressionLevel =
        store.getCompressionLevel();…
    if (compressionLevel == 1) {
      compressor = store.getCompressorFast();
      compressType = DataUtils.PAGE_COMPRESSED;
    } else {…}…
    int compLen = compressor.compress(…);
    if (compLen + plus < expLen) {
      buff.put(…); …}
…}
```

```
void processBlock() throws … {…
  List<Callable<Status>> tasks =
        new ArrayList<>(this.jobs);…
  for (Future<Status> result :
          this.pool.invokeAll(tasks)) {
    //  Wait for completion
    //   of next task and validate result
    Status status = result.get();
   }
  …
}
```

(a) direct data dependency (b) if/switch-related control dependency (c) loop-related control dependency

Figure 4: Real-world examples of three target systems (see table 1 for more details) to illustrate the dependencies between performance-related operations and options. The arrows show the data-flow of option propagation. Configuration options are quoted in the figure, and the performance-related operations are shaded.

four heuristic strategies for dimensionality reduction and transformation. We cluster the counts based on performance-related operations and dependency categories that we mentioned in § 2.2.

(1) We first introduce $C_4^1 = 4$ aggregated features by clustering counts based on performance-related operation categories. As an illustration, we cluster all performance-related operations that pertain to the *java.io* or *java.nio* packages for Java applications in our implementation.
(2) We next introduce $C_4^2 = 6$ aggregated features by clustering counts according to the pair-wise combination of performance-related operations categories. This strategy is motivated by our observation that one information-flow path of options influences two categories of performance-related operations at the same time. Figure 4.(c) shows a real-world example that the option *jobs* in Kanzi determines the concurrency and synchronization of tasks (the operation *invokeAll()* creates threads for tasks execution and holds for them to complete).
(3) We then introduce one aggregated feature by accumulating the counts of four performance-related operations categories.
(4) We finally introduce $C_4^1 \cdot C_3^1 = 12$ aggregated features by categorizing the dependencies between performance-related operations and options. Figure 4 shows three real-world examples of the dependencies. Figure 4.(a) shows that the option *memtable_flush_writers* directly determines the core pool size of a thread pool; Figure 4.(b) gives an example that the option *COMPRESS* decides whether *WriteBuffer.put()* is executed by an *if* statement; and Figure 4.(c) shows that the option *jobs* determines the task synchronization via loop control. These aggregated features would prompt the random forest to learn fine-grained information about dependencies between options and performance-related operations.

These heuristics are set to mitigate the risk of overfitting caused by the sparse high-dimensional feature space for random forest classification model building.

Random Forest. Random forest is an appropriate algorithm for our binary classification task. It combines multiple tree predictors and distinguishes classes by aggregating their predictions. It is also more robust than a single tree predictor and more interpretable than deep learning models because of explicit decision inference paths in a tree. Besides, training a random forest model only needs a small sample data which is readily satisfied in our scenario. Given these, to identify performance-sensitive options, we train a random forest model which approximates the following function.

$$g(\text{performance property}) \longrightarrow \text{performance-sensitive or not}$$

The training data are performance properties of the configuration options with class labels (§ 5.2), namely performance-sensitive (i.e., 1) or not (i.e., 0). Then, DiagConfig characterizes the information-flow paths of a new option as the performance property and feeds it to the trained model. The random forest determines whether the option is performance-sensitive or not. *All performance-sensitive options and their corresponding information-flow paths are persisted in a file for cause-effect chains building in the online diagnosis.*

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 [20], to continuously monitor the end-to-end performance of a system in sampled-based [3, 59] mode (see § 5.4). 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 § 5.1).

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 trace 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 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 performance-violating hotspot functions from the profiler to complement the runtime information of a system that is missing from static code analysis. DiagConfig backtracks the call-chains of 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 the operations, DiagConfig correlates the corresponding information-flow paths and the call-chains to construct trackable cause-effect chains (i.e., conditional pairs consisting of information-flow paths and call-chains). Associating performance variations of hotspot functions with options corresponding to information-flow paths allows for quantifying and ranking options' importance.

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 the information-flow path.

Algorithm 1: Cause-effect Chains Building

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

4 IMPLEMENTATION

Our prototype tool **DiagConfig** is built on the top of the Soot compiler infrastructure [67], FlowDroid [4], Scikit-learn [56], and Jprofiler [20] and specially targets configurable Java applications.

To ensure its scalability, we focus on the standard library APIs and bytecode instructions that contribute positively to execution time and memory consumption. These are the basic application-independent performance-related operations in the Java ecosystem. DiagConfig treats these performance-related operations as sinks through method signatures and type analysis based on Jimple, the intermediate-representation provided by the Soot. For example, the array allocation-related operations are represented by *JNewArrayExpr* and *JNewMultiArrayExpr*, and the 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 utilize the popular static taint analysis framework, FlowDroid [4], to support configuration options taint tracking. By treating configuration loading statements as **sources** and performance-related operations as **sinks**, it reveals the direct data dependency between the configuration options and the performance-related operations. But the if/switch-related and loop-related control dependencies are missed out. Therefore, we slightly modified the sink manager component of FlowDroid with program-dependence graphs [23] for our purposes to mark *if/switch* statements with branches containing performance-related operations and *loop* startup/jump-out statements with body containing performance-related operations as **expanded sinks**. Then, if the conditions of if/switch statements or loop startup/jump-out statements have a direct data dependency with an option, there is if/switch-related or loop-related dependency between the option and the operations inside the branches or loop body. We also customized FlowDroid's source manager component to support those systems without standard configuration loading operations. Additionally, we set the depth of alias analysis at five for FlowDorid to balance the taint tracking precision and computational overhead. This is the default value provided by FlowDroid, and the larger the depth the higher the taint tracking overhead required. Next, we build the random forest with *Scikit-learn* for identification of performance-sensitive options. In the online diagnosis, we use a

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	28	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;
* :The system targets at low execution time (latency);
+ :The system targets at high throughput;
Opt: The number of options; V/ID: Version/Commit ID; Overhead: Time required for static configuration options taint tracking;

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 suffers a performance degradation over 5% compared to the baseline under its benchmark. 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 over 5% for root cause inference. The 5% is our empirical unacceptable value, based on the measurement variation (see § 5.1) which is up to 4% known from the prior work [75]. 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 (§ 2.2) to help stakeholders diagnose the performance violations of configurable software systems. Moreover, 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 used two environments, one with 128GB of RAM, 24 cores, and 48 threads of Intel Xeon Silver 4116 processor running Ubuntu 18.04, which was 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 selected eight configurable, real-world, open-source Java systems from various domains shown in Table 1. All of them satisfied the following criteria: (1) systems with binary, enumerated, and continuous configuration options; (2) systems used for evaluation by previous research on performance modeling. We reused the workloads and benchmarks evaluated in the existing literature [75] for each target system.

Measurement and Configuration Variation. To confirm the measurement stability in our environment, we chose one configuration with normal execution (i.e., normal configuration) for each of the four target systems according to their artifacts (e.g., documentation, release notes, benchmark results), and repeated 50 times in the benchmarks. Figure 5.(a) shows that execution time variations of Batik, H2 database, Kanzi, and throughput variation of Cassandra

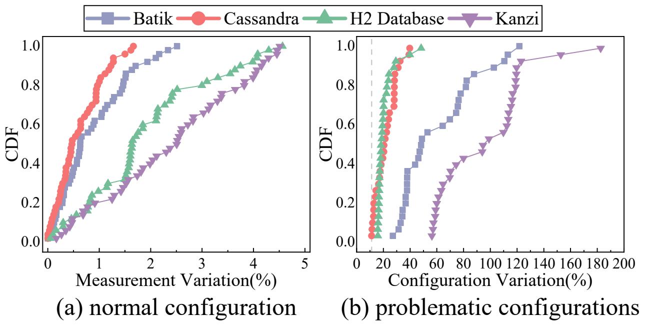


Figure 5: Performance variation of normal configuration and problematic configurations in 50 repetitions.

in 50 repetitions with normal configuration are all below 5%. The $measurement\ variation = |\frac{p_i - mean}{mean}|$ where $i \in [1, 50]$, p_i is the performance of the i-th execution and *mean* is the average performance of 50 repetitions. Each point in Figure 5.(a) represents the performance variation of a single repetition compared to the average performance. Moreover, to capture the variation in a system's performance due to loading problematic configuration, we first randomly generated 100 configurations derived from the normal configuration for each system, repeated measurement 50 times, and filtered out the configurations with average performance variation below 5% compared to the normal configuration. This leaves us with 49, 31, 32, and 38 configurations for Batik, Cassandra, H2, and Kanzi, respectively. Then we randomly selected 30 problematic configurations (i.e., average performance variation over 5%) from these filtered configurations to understand the sensitivity of performance violations. Figure 5.(b) shows that the performance degradation (i.e., configuration variation) caused by problematic configurations with 50 repetitions varies from 11.2% in Batik to 196% in Kanzi. The $configuration\ variation = |\frac{c_j - normal}{normal}|$ where $j \in [1, 30]$, c_j is the average performance for the j-th problematic configuration and *normal* is the average performance for the normal configuration. Each point in Figure 5.(b) represents an average performance variation of 50 repetitions for one problematic configuration. The dashed vertical line denotes the minimal configuration variation (i.e., 11.2%). From Figure 5, we can conclude that in our environment, the performance variation under repeated executions is below 5% and the configuration variation is above 11.2%. Therefore, we regarded the measurement result of 5 repetitions as approximate to the result of 50 repetitions and use 5% as the threshold of performance violation conservatively. Note that measurement and configuration variation is the rationale for our derived performance violation threshold.

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

In this section, we describe how we labeled data for random forest classification modeling, and the details about the identification of performance-sensitive options, followed by the result analysis. Note that we also conduct data labeling for Cassandra and use the labeled data as the ground truth for the evaluation of identification results.

Data Labeling. To label performance-sensitive options accurately, we conducted experiments to study the performance influence of each option. For each option in each system, we changed

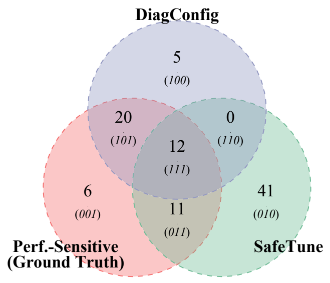


Figure 6: Results of performance-sensitive options identification.

its value, re-deployed the system, followed common practices [32, 33, 75] to measure five repetitions, and calculated the average performance variation before and after the value change. Then, the options with any observed variation larger than 5% were labeled as performance-sensitive. Specifically, for options of binary and enumerated types, we explored all possible values, and for continuous options, we obtained a set of their values by Plackett-Burman sampling [73]. For some continuous options that were not identified as performance-sensitive in the sampling space, we further tried our best to analyze their influence in small incremental steps (e.g., we used 2mb as a step from 1mb to 4096mb to analyze the influence of the option *file_cache_size_in_mb* of Cassandra).

Random forest modeling. To build a well-generalized random forest classification model, we simulated a scenario where the model is trained on existing systems and tested on unseen systems. Given this, we used Batik, Catena, H2, Prevayler, and Sunflow (i.e., 89 options) as the training set and Dconverter, Kanzi (i.e., 64 options) as the test set. 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 and used for further evaluation.

Comparison. The remaining target system, Cassandra, was used to further evaluate the effectiveness of our classification model. We chose Cassandra since it was evaluated in another state-of-the-art tool SafeTune [29], which identifies performance-related parameters by building learning-based models and analyzing on system configuration-related documentation. SafeTune made its dataset publicly available, so we used it to compare with our results.

Result and Analysis. We collected the set of performance-sensitive options by running Cassandra in NoSQLBench [22], tlp-stress [21], cassandra-stress [53] and the YCSB [17] benchmark. We fed 172 options to taint tracking, 67 of which influence pre-defined performance-related operations. And we identified 37 performance-sensitive options based on the random forest model. In comparison, SafeTune identified 64 performance-related parameters.

The results are shown in Figure 6. The circled area labeled 'Perf.-Sensitive' provides the ground truth for the comparison analysis. Among the 37 reported performance-sensitive options by DiagConfig, 32 are true positives (i.e., region *101* + region *111*) and 5 are false positives (i.e., region *100* + region *110*), having a precision of 86.4% and a recall of 65.3%. SafeTune reported 64 performance-sensitive options. Among them, 23 are true positives (i.e., region *011* + region *111*) and 41 are false positives (i.e., region *010* + region *110*), having a precision of 35.9% and a recall of 46.9%.

Both approaches produce false negatives. DiagConfig missed 17 performance-sensitive options (i.e., region *001* + region *011*) and SafeTune missed 26 (i.e., region *001* + region *101*). We studied the 17 performance-sensitive options that we misclassified and summarized 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.

DiagConfig and Safetune are complementary to each other since each has identified new options missing in the other. DiagConfig identified 20 performance-sensitive options (i.e., region *101*) missed by SafeTune, and SafeTune identified 11 performance-sensitive options (i.e., region *011*) that were missed in DiagConfig.

***Summary** for RQ$_1$: For performance-sensitive option identification, our static code analysis-based approach introduces fewer false positives (i.e., 5 vs 41) than the documentation-based approach. This is because 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 section, we focus on the diagnosis of performance violations caused by configuration changes and evaluate our code analysis-based approach by comparing it with a statistic-based approach.

Diagnosis of Configuration Changes. First, we selected a baseline or default configuration based on the relevant document and treated the application performance under this configuration as the SLO for each system. Then, we mutated the configuration to produce scenarios of performance violations for further diagnosis. Similar to the sampling strategy for accurate performance modeling which is described in Weber et al [75], we selected feature-wise and pair-wise sampling for options of binary type and Plackett-Burman sampling [73] for enumeration and continuous types. We first manually filtered out the invalid configurations derived from sampling, loaded only valid ones into the system, and got corresponding performance measurements in a specific benchmark. Those configurations with an average performance in five repetitions over 5% compared to the SLO would be fed to the process of diagnosis.

Note that in the evaluation, the goal is not to find the optimal configurations, but to diagnose performance violations. The effectiveness of diagnosing performance violations was reported via precision and recall metrics. Precision is the ratio between the set of correctly identified options and the set of predicted options while recall is the percentage of correctly identified options causing a performance violation and the set of options whose values had changed compared to the baseline.

Comparison. We ran DiagConfig to obtain the set of configuration options responsible for performance violations, and compared our results to Unicorn, a statistics-based approach for configurable systems performance debugging and optimization via causal performance models. In debugging mode, Unicorn repeatedly generates new configurations to replace the loaded one for the system in the

Table 2: Average Precision and Recall of Comparison.

	System Configurations	Batik 128	Catena 2433	H2 573	Kanzi 494	Prevayler 118	**Average**
Precision	Unicorn	0.787	0.705	0.871	0.964	0.756	0.817
	DiagConfig	**0.911**	**0.809**	**0.905**	**0.806**	**0.788**	**0.844**
Recall	Unicorn	0.116	0.774	0.049	0.274	0.234	0.289
	DiagConfig	**0.807**	**0.956**	**0.815**	**0.932**	**0.952**	**0.892**

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

deployed environment, measures the performance, then pinpoints the critical options related to performance issues.

Result and Analysis. Table 2 summarizes the average precision and recall of Unicorn and DiagConfig for each system in diagnosing performance violations. Catena has a much bigger configuration size because in the fixed time duration sampling runs faster due to its small source code size and number of options. It also uncovers the reason for Unicorn's exception recall in Catena since a small number of options means that the correlation between options and performance metrics is easier to learn during training.

The relatively low recall of Unicorn reveals its limitations. When generating one new configuration to debug, Unicorn only considers one crucial option at a time, changes its value, deploys and measures the system, and targets normal metrics. The list of candidate options is short and relies on the accuracy of causal correlations between options and performance metrics captured by causal performance models, resulting in an incomplete diagnosis. In contrast, DiagConfig captures the cause-effect relationships between options and performance behaviors and supports a more comprehensive diagnosis via code analysis. However, DiagConfig also produced false negatives and false positives.

For the reasons of false negatives, we double-checked the offline analysis corresponding to the missed options and concluded that (a) some information-flow paths of options are lost due to the level of variables indirectly referencing the options exceeding the depth of alias analysis we set (i.e., 5) in taint tracking; (b) a few options influence compute-intensive operations (e.g., operations of primitive numeric types and hash computation) 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.

***Summary** for RQ$_2$ and RQ$_3$: Treating the system as a white box and taking advantage of performance-related operations, DiagConfig effectively diagnoses performance violations by building cause-effect chains. In addition, the cause-effect chains achieve fine-grained interpretability (like Figure 3), which helps stakeholders understand the root causes of performance violations.*

5.4 RQ$_4$: Performance of DiagConfig

Overhead. As Figure 2 shows, DiagConfig consists of offline analysis and online diagnosis. The offline analysis overhead includes data labeling, static configuration options taint tracking, classification model building, and performance-sensitive options classification with their information-flow paths filtering. Among them, data labeling and classification model building are only required in the preparation stage so that when a new system is fed into DiagConfig, it can directly apply the classification model after taint tracking and characterization. Thus, data labeling and classification model building are conducted only once and used forever. The performance-sensitive options classification with their information-flow paths filtering are comparatively negligible compared to the static taint tracking, which is required for each new system. Table 1 lists the overhead for static taint tracking of each target system in our evaluation. It is relatively heavy but acceptable because we only need to run static taint tracking once. Moreover, the software vendors generally can provide the results of this part.

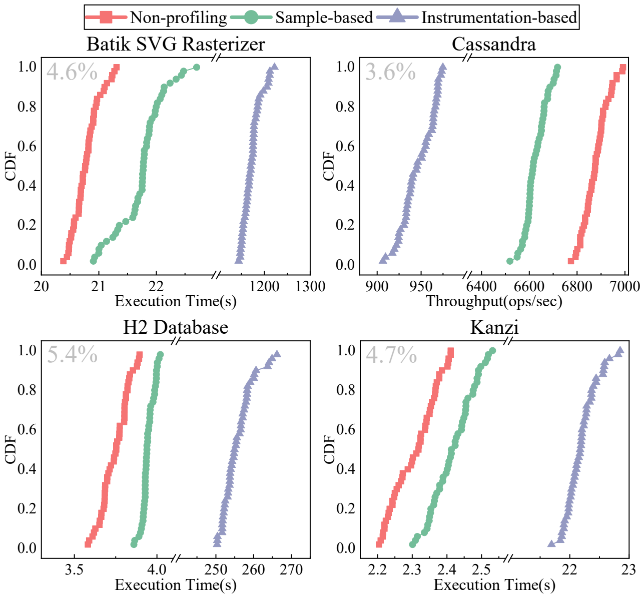


Figure 7: Performance distribution of four software systems in non-profiling, sample-based profiling, and instrumentation-based profiling. The small top left plot reports the average performance degradation caused by sample-based profiling for each system, while instrumentation-based profiling incurs unacceptable overhead.

The overhead of online diagnosis includes profiling overhead and cause-effect chain building overhead. To check the overhead incurred by our chosen profiler, we selected 50 configurations for each of the four target software systems. We measured these software systems with 50 repetitions using non-profiling, sample-based profiling [3, 59], and instrumentation-based profiling [48], respectively. We visualized the performance distribution of these software systems loaded one configuration in Figure 7 to show the variation of the results. Each point in Figure 7 represents the performance of a single repetition. For all four software systems, the performance degradation incurred by sample-based profiling (i.e., $|\frac{mean^s - mean^n}{mean^n}|$ where $mean^n$ is the average performance under non-profiling and $mean^s$ is the average performance under sample-based profiling) is below 6%. Additionally, the performance distribution is similar for all configurations, indicating that the configuration does not impact the measurement stability, which is consistent with the insight of Weber et al [75]. Despite the extensive research on lightweight online monitoring tools and sophisticated industrial profilers [83], our experimental profiling results suggest that the overhead of sample-based profiling is acceptable and DiagConfig may not need to replace industry-class profilers but build on them to achieve performance violation detection in online diagnosis.

Table 3: Overhead of Cause-effect Chains Building.

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

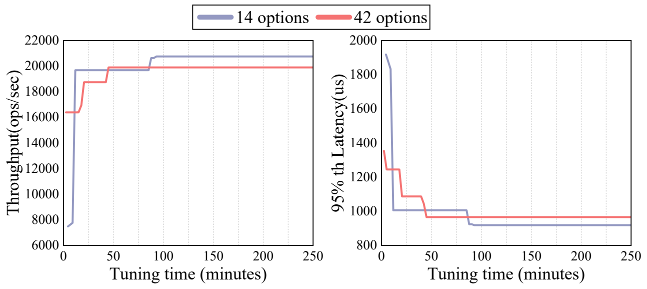


Figure 8: Tuning Example. Comparison of OtterTune tuning process with (14 options) and without (42 options) DiagConfig aid.

Regarding the overhead of the building of cause-effect chains, we recorded the execution time of each diagnosis when diagnosing performance violations (§ 5.3). A summary of the metrics is shown in Table 3. 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 online production environments and can be integrated into the existing auto-tuners for diagnosis before tuning starts.

Case Study. DiagConfig can recommend crucial options to guide tuning tools (e.g., *SmartConf* [74], OtterTune [68], DAC [86], BestConfig [90]) to reduce the huge configuration space. We conducted a case study with OtterTune, a representative auto-tuner, to show the effectiveness of DiagConfig's recommendation. OtterTune supports MySQL and PostgreSQL, but neither of them is a Java system. Thus, we extended OtterTune to work for the Java database Cassandra. We applied DiagConfig to figure out the crucial options when a performance violation occurs and then ran OtterTune to improve performance. We recorded OtterTune's tuning time spent to validate how much DiagConfig accelerates the tuning process. In the case study, we simulated a situation where Cassandra suddenly suffered significant throughput and latency degradation under the YCSB benchmark. The stakeholders pointed out the options related to this performance violation and then leveraged OtterTune to improve the system's performance. By contrast, we ran 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 8. OtterTune got stuck in the configuration space that consists of the 42 options. Tuning with DiagConfig required only 11 minutes to achieve 98% of the throughput obtained by tuning without DiagConfig through 45 minutes. This was an almost 4× acceleration. Moreover, after the 87th minute, the tuning with DiagConfig further achieved better throughput. Similarly, the acceleration of OtterTune by DiagConfig was also manifested 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 over-tainting 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 [4] 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 an 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 51 GiB call-chains) showed 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 were collected from previous work [70, 71, 75] and were usually used to evaluate sampling strategies and performance-modeling methods. We further ran our approach with OtterTune on the Cassandra database to show the feasibility of large-scale configurable software systems.

7 DISCUSSION

Interpretability of documentation and source code. Accurately identifying performance-sensitive options requires understanding the relationship between configuration options and performance behaviors. This information can be obtained from system documentation and source code analysis. SafeTune is a documentation analysis-based tool, while DiagConfig emphasizes source code analysis. 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. Both SafeTune and DiagConfig can identify performance-sensitive options that the other has missed. Therefore, the information provided by documentation and source code is complementary to each other.

Dynamic workload and environment. Dynamic workloads and environments are non-trivial for evaluating system performance. For RQ_1, both SafeTune and our evaluation were limited by the workload and environment. In our evaluation, we first confirmed the measurement and configuration variation (§ 5.1), then tried our best to identify performance-sensitive options in multiple rich workloads. In contrast, previous work [29] only considered multiple workloads without accounting for measurement and configuration variation, thus may lead to inaccurate results. For RQ_2 and RQ_3, due to the wide variety of workloads and environments, the performance violations caused by dynamic workloads and environments migration are out of the scope of this paper.

Multiple metrics for performance issues troubleshooting. A wide range of metrics has been spawned for monitoring various aspects of systems' runtime behaviors. While DiagConfig focuses on the diagnosis of single-objective (most prior works also target single-objective [61, 68, 70–72, 75, 86, 90]) performance violations , we will accommodate multiple and mixed metrics with the appropriate modification of monitoring components for in-depth performance issues diagnosis in the future. For instance, thread activity and concurrency metrics (e.g., thread on-CPU cycles, synchronization delays) for unusual thread behavior and CPU contention detection, and lock contention metrics (e.g., the level of locks) for problematic code-sharing designs uncover.

8 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 the second step.

Performance violations occur frequently due to changes in workload and environment as well as misconfigurations. There is substantial literature on detecting [31, 87, 89], testing [15, 41, 65, 80], diagnosing [5, 6], and fixing [43, 76] misconfigurations. Specifically, ConfigX [89] employs a tailored static analysis of configuration-related code snippets to extract the specification constraint among options. It does not consider runtime performance behaviors and is therefore well-suited for detecting misconfiguration that may result in unexpected and hard-to-observed functional behavior rather than performance behavior before the configuration is loaded in. 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 [20, 38, 52], targeted profiling techniques [16, 85], and visualizations [1, 16] 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 aim to 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 [61, 70, 71, 75]. ConfigCrusher [70] first relies on static taint analysis 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 [71] builds white-box performance-influence models based on expensive dynamic taint analysis and incomplete configuration specific local code performance measurement. Weber et al. [75] propose an approach based on SPLConqueror [61–63] 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 [33, 49], the performance of the models themselves varies depending on the sampling strategy and learning tricks.

Optimizers for 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 [30, 74] and machine-learning-based [12, 13, 32, 68, 82, 86, 90] approach.

9 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 with 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 configuration options for performance violations, help stakeholders explain the causes of performance violations, and accelerate the configuration tuning process regardless of workloads and environments.

10 DATA-AVAILABILITY STATEMENT

All implementation and data can be found in archived repository [14].

ACKNOWLEDGMENT

We thank the anonymous reviewers for their feedback. The research is supported by the Key-Area Research and Development Program of Guangdong Province (No. 2020B010165002), the Guangdong Basic and Applied Basic Research Foundation (No. 2023B1515020054), the Natural Science Foundation of China (No. 62272495), the Tencent Rhino-Bird Joint Research Program and the Fundamental Research Funds for the Central Universities, Sun Yat-sen University (No. 22qntd1004). The corresponding author is Pengfei Chen.

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. https://doi.org/10.1109/VISSOFT.2019.00009

[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] Jennifer M Anderson, Lance M Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R Henzinger, Shun-Tak A Leung, Richard L Sites, Mark T Vandevoorde, Carl A Waldspurger, and William E Weihl. 1997. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 357–390.

[4] 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 (2014), 259–269.

[5] 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.

[6] 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)*.

[7] L Douglas Baker and Andrew Kachites McCallum. 1998. Distributional clustering of words for text classification. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*. 96–103.

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

[9] 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. https://doi.org/10.1145/1718918.1718973

[10] Cornelia Caragea, Adrian Silvescu, and Prasenjit Mitra. 2012. Combining hashing and abstraction in sparse high dimensional feature spaces. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 26. 3–9.

[11] 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.

[12] 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. https://doi.org/10.1109/BigDataCongress.2015.64

[13] 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. https://doi.org/10.1145/3468264.3468555

[14] Zhiming Chen, Pengfei Chen, Peipei Wang, Guangba Yu, and et al. 2023. *Reproduction Package for Article 'DiagConfig: Configuration Diagnosis of Performance Violations in Configurable Software Systems'*. https://doi.org/10.5281/zenodo.8279414

[15] Runxiang Cheng, Lingming Zhang, Darko Marinov, and Tianyin Xu. 2021. Test-case prioritization for configuration testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 452–465. https://doi.org/10.1145/3460319.3464810

[16] 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.

[17] 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.

[18] 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. https://doi.org/10.1109/ISSRE.2016.37

[19] 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.

[20] ej technologies. 2019. JPROFILER: The AWARD-WINNING ALL-IN-ONE JAVA PROFILER. https://www.ej-technologies.com/products/jprofiler/overview.html.

[21] Jon Haddad et al. 2023. tlp-stress: A workload centric stress tool and framework. https://github.com/thelastpickle/tlp-stress. Accessed May 13, 2023.

[22] Jonathan Shook et al. 2023. NoSQLBench: The Open Source, Pluggable, NoSQL Benchmarking Suite. https://github.com/nosqlbench/nosqlbench.

[23] 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 (1987), 319–349.

[24] Google. 2019. Google Cloud Storage Incident #19002. https://status.cloud.google.com/incident/storage/19002. Accessed March 13, 2020.

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

[26] 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. https://doi.org/10.1109/ASE.2013.6693089

[27] 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. https://doi.org/10.1109/ICSME.2019.00080

[28] 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. https://doi.org/10.1145/2961111.2962602

[29] 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. https://doi.org/10.1145/3510003.3510094

[30] Henry Hoffmann, Jim Holt, George Kurian, Eric Lau, Martina Maggio, Jason E Miller, Sabrina M Neuman, Mahmut Sinangil, Yildiz Sinangil, Anant Agarwal, et al. 2012. Self-aware computing in the Angstrom processor. In *Proceedings of the 49th Annual Design Automation Conference*. 259–264.

[31] 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.

[32] Md Shahriar Iqbal, Rahul Krishna, Mohammad Ali Javidian, Baishakhi Ray, and Pooyan Jamshidi. 2022. Unicorn: reasoning about configurable system performance through the lens of causality. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 199–217. https://doi.org/10.1145/3492321.3519575

[33] 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. https://doi.org/10.1109/ASE.2017.8115661

[34] 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. https://doi.org/10.1145/3236024.3236074

[35] Santosh Janardhan. 2021. Update about the October 4th outage. https://engineering.fb.com/2021/10/04/networking-traffic/outage/.

[36] 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.

[37] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices* 47, 6 (2012), 77–88. https://doi.org/10.1145/2345156.2254075

[38] Tomas Hurka Jiri Sedlacek. 2019. VisualVM: All-in-One Java Troubleshooting Tool. https://visualvm.github.io/index.html. Accessed April 6, 2022.

[39] 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.

[40] 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 (2020), 58–66.

[41] 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. https://doi.org/10.1109/DSN.2008.4630084

[42] 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 (2019), 2265–2283.

[43] 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)*.

[44] 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. https://doi.org/10.1145/3342195.3387520

[45] 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. https://doi.org/10.1145/2642937.2643001

[46] 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.

[47] 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. https://doi.org/10.1145/2884781.2884793

[48] Tipp Moseley, Alex Shye, Vijay Janapa Reddi, Dirk Grunwald, and Ramesh Peri. 2007. Shadow profiling: Hiding instrumentation costs with parallelism. In *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, 198–208. https://doi.org/10.1109/CGO.2007.35

[49] CKJDSA Stefan Mühlbauer, Florian Sattler, and N Siegmund. 2023. Analyzing the impact of workloads on modeling the performance of configurable software systems. In *Proceedings of the International Conference on Software Engineering (ICSE), IEEE*. https://doi.org/10.1109/ICSE48619.2023.00176

[50] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2018. Faster discovery of faster system configurations with spectral learning. *Automated Software Engineering* 25, 2 (2018), 247–277. https://doi.org/10.1007/s10515-017-0225-2

[51] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. 2018. Finding faster configurations using flash. *IEEE Transactions on Software Engineering* 46, 7 (2018), 794–811. https://doi.org/10.1109/TSE.2018.2870895

[52] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100.

[53] Apache Cassandra Official. 2023. Cassandra Stress Benchmark. https://cassandra.apache.org/doc/4.0/cassandra/tools/cassandra_stress.html.

[54] 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. https://doi.org/10.1145/3106237.3106273

[55] 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.

[56] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[57] 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. https://doi.org/10.1109/ASE.2011.6100053

[58] Ariel Rabkin and Randy Katz. 2011. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering*. 131–140. https://doi.org/10.1145/1985793.1985812

[59] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE micro* 30, 4 (2010), 65–79. https://doi.org/10.1109/MM.2010.68

[60] 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. https://doi.org/10.1145/3092703.3092716

[61] 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. https://doi.org/10.1145/2786805.2786845

[62] 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.

[63] 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. https://doi.org/10.1109/SPLC.2011.20

[64] Noam Slonim and Naftali Tishby. 1999. Agglomerative information bottleneck. *Advances in neural information processing systems* 12 (1999).

[65] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. 2020. Testing configuration changes in context to prevent production failures. In *USENIX Symposium on Operating Systems Design and Implementation*.

[66] 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 (2015), 33–41. https://doi.org/10.1016/j.jfds.2015.07.002

[67] 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.

[68] 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. https://doi.org/10.1145/3035918.3064029

[69] 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. https://doi.org/10.1145/2188286.2188326

[70] 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 (2020), 265–300. https://doi.org/10.1007/s10515-020-00273-8

[71] 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. https://doi.org/10.1109/ICSE43902.2021.00100

[72] 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. In *Proceedings of the 44th International Conference on Software Engineering*. 1571–1583. https://doi.org/10.1145/3510003.3510043

[73] JC Wang and CF Jeff Wu. 1995. A hidden projection property of Plackett-Burman and related designs. *Statistica Sinica* (1995), 235–250.

[74] 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 (2018), 154–168.

[75] 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. https://doi.org/10.1109/ICSE43902.2021.00099

[76] 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.

[77] 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. https://doi.org/10.1145/1543135.1542523

[78] 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.

[79] 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.

[80] 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. https://doi.org/10.1145/2517349.2522727

[81] 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.

[82] 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.

[83] Fangxi Yin, Denghui Dong, Sanhong Li, Jianmei Guo, and Kingsum Chow. 2018. Java performance troubleshooting and optimization at alibaba. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. 11–12. https://doi.org/10.1145/3183519.3183536

[84] 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.

[85] Tingting Yu and Michael Pradel. 2018. Pinpointing and repairing performance bottlenecks in concurrent programs. *Empirical Software Engineering* 23, 5 (2018), 3034–3071. https://doi.org/10.1007/s10664-017-9578-1

[86] 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.

[87] 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.

[88] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? *ACM SIGSOFT Software engineering notes* 24, 6 (1999), 253–267.

[89] Jialu Zhang, Ruzica Piskac, Ennan Zhai, and Tianyin Xu. 2021. Static detection of silent misconfigurations with deep interaction analysis. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021). https://doi.org/10.1145/3485517

[90] 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. https://doi.org/10.1145/3127479.3128605

Received 2023-02-02; accepted 2023-07-27