On Debugging the Performance of Configurable Software Systems: Developer Needs and Tailored Tool Support

Miguel Velez
Carnegie Mellon University

Pooyan Jamshidi
University of South Carolina

Norbert Siegmund
Leipzig University

Sven Apel
Saarland Informatics Campus -
Saarland University

Christian Kästner
Carnegie Mellon University

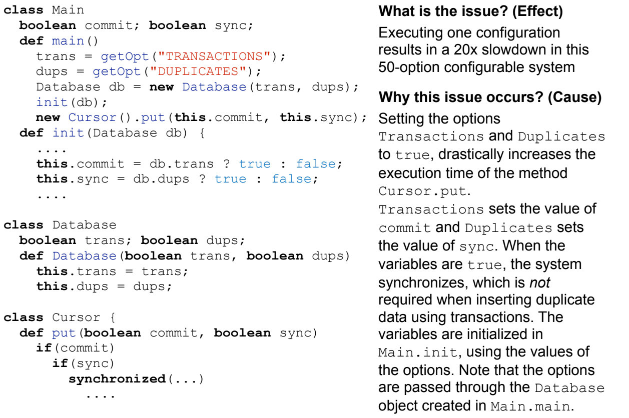
ABSTRACT

Determining whether a configurable software system has a performance bug or it was misconfigured is often challenging. While there are numerous debugging techniques that can support developers in this task, there is limited empirical evidence of how useful the techniques are to address the actual needs that developers have when debugging the performance of configurable software systems; most techniques are often evaluated in terms of technical accuracy instead of their usability. In this paper, we take a human-centered approach to identify, design, implement, and evaluate a solution to support developers in the process of debugging the performance of configurable software systems. We first conduct an exploratory study with 19 developers to identify the information needs that developers have during this process. Subsequently, we design and implement a tailored tool, adapting techniques from prior work, to support those needs. Two user studies, with a total of 20 developers, validate and confirm that the information that we provide helps developers debug the performance of configurable software systems.

ACM Reference Format:
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 *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510043

1 INTRODUCTION

Developers often spend a substantial amount of time diagnosing a configurable software system to localize and fix a performance bug, or to determine that the system was misconfigured [12, 15, 29, 34, 35, 42, 56, 59, 60, 85]. This struggle is quite common when maintaining configurable software systems. Some empirical studies find that 59% of performance issues are related to configuration errors, 88% of these issues require fixing the code [29, 30], of which 61% take an average of 5 weeks to fix [42], and that 50% of patches in open-source cloud systems and 30% of questions in forums are related to configurations [74]. Regardless of how developers find the root cause of the issue or misconfiguration, performance issues impair user experience, which often result in long execution times or increased energy consumption [29, 32, 34, 42, 45, 69, 77].



```
class Main
  boolean commit; boolean sync;
  def main()
    trans = getOpt("TRANSACTIONS");
    dups = getOpt("DUPLICATES");
    Database db = new Database(trans, dups);
    init(db);
    new Cursor().put(this.commit, this.sync);
  def init(Database db) {
    ....
    this.commit = db.trans ? true : false;
    this.sync = db.dups ? true : false;
    ....

class Database
  boolean trans; boolean dups;
  def Database(boolean trans, boolean dups)
    this.trans = trans;
    this.dups = dups;

class Cursor {
  def put(boolean commit, boolean sync)
    if(commit)
      if(sync)
        synchronized(...)
          ....
```

What is the issue? (Effect)

Executing one configuration results in a 20x slowdown in this 50-option configurable system

Why this issue occurs? (Cause)

Setting the options `Transactions` and `Duplicates` to `true`, drastically increases the execution time of the method `Cursor.put`. `Transactions` sets the value of `commit` and `Duplicates` sets the value of `sync`. When the variables are `true`, the system synchronizes, which is *not* required when inserting duplicate data using transactions. The variables are initialized in `Main.init`, using the values of the options. Note that the options are passed through the `Database` object created in `Main.main`.

Figure 1: Artificial example contrasting effects and causes when debugging the performance of configurable software systems.

When performance issues occur, developers typically use profilers to identify the locations of performance bottlenecks [14, 16, 17, 24, 83]. Unfortunately, locations where a system spends the most time executing are not necessarily the sign of a performance issue. Additionally, traditional profilers only indicate the locations of the *effect* of performance issues (i.e., where a system spends the most time executing) for one configuration at a time. Developers are left to inspect the code to analyze the *root cause* of the performance issues and to determine how the issues relate to configurations.

With an example scenario in Figure 1, we illustrate the kind of performance challenge that developers may face in configurable software systems and that we seek to support: A user executes a configuration in this system with 50 configuration options, which results in an *unexpected* 20× slowdown. The *only visible effect* is the excessive execution time. While, in some situations, developers might be able to change some options to work around the problem, users might not know which options cause the problem, and may want to select certain options to satisfy specific needs (e.g., enable encryption, use a specific transformation algorithm, or set a specific cache size). In these situations, developers need to determine whether the system has a potential *bug*, is *misconfigured*, or *works correctly*, but the user has a *different expectation* about what the performance of the system should be under the configuration that they execute. To

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510043

determine the cause of the potential problematic performance behavior, developers would need to debug the system and, most likely, the *implementation* to identify which options or interactions in this configuration of 50 options are the *root cause* of the *unexpected performance behavior* (e.g., the system works as expected, setting a specific Cache_Size results in a misconfiguration, or setting the options Transactions and Duplicates to `true` results in a bug.).

When performance issues as in our example occur, there are numerous techniques that developers could use to determine whether there is a performance bug or the system was misconfigured. In addition to off-the-shelf profilers [3, 4, 54], developers could use more targeted profiling techniques [8, 14, 17, 82, 83], visualize performance behavior [6, 10, 16, 24, 63, 71], search for inefficient coding patterns [11, 47, 55, 57, 69], use information-flow analyses [44, 46, 49, 70, 78, 80, 87], or model the performance of the systems in terms of its options and interactions [27, 39, 65, 72, 73, 75]. Likewise, developers could use established program debugging techniques, such as delta debugging [84], program slicing [7, 40, 76], and statistical debugging [9, 68] for some part of the debugging process. One reason why we cannot reliably suggest developers to use any of the above techniques is that there is limited empirical evidence of how useful the techniques are to help developers debug the performance of configurable software systems; the techniques typically solve a specific technical challenge that is usually evaluated in terms of accuracy, not usability [60]. Hence, we could only, at best, speculate which techniques might support developers' needs to debug unexpected performance behaviors in configurable software systems.

In this paper, we take a human-centered approach [19, 52] to *identify*, *design*, *implement*, and *evaluate* a solution to support developers in the process of debugging the performance of configurable software systems; particularly, in situations such as our example in Figure 1. Our human-centered research design consists of three steps, summarized in Figure 2: We first conduct an *exploratory user study* to *identify the information needs* that developers have when debugging the performance of configurable software systems. Our study reveals that developers *struggle* to find relevant information to (a) identify influencing options; the options or interactions causing an unexpected performance behavior, (b) locate option hotspots; the methods where options affect the performance of the system, and (c) trace the case-effect chain; how influencing options are used in the implementation to directly and indirectly affect the performance of option hotspots. Afterwards, we *design* and *implement information providers* to support developers' needs, adapting and tailoring **G**lobal and **L**ocal performance-**I**nfluence **M**odels, CPU **P**rofiling, and program **S**licing, in a tool called **GLIMPS**. Finally, we conduct two user studies to *validate* and *confirm* that the designed information providers are useful to developers when debugging the performance of complex configurable software systems, in terms of supporting their information needs and speeding up the process.

In summary, we make the following contributions:

- The information needs – influencing options, option hotspots, and cause-effect chain– that developers have when debugging the performance of configurable software systems.
- The design of information providers, adapted from **G**lobal and **L**ocal performance-**I**nfluence **M**odels, CPU **P**rofiling, and program **S**licing, to support the above needs.
- Two empirical evaluations to demonstrate the usefulness of the designed information providers.
- A prototype tool, **GLIMPS**, that implements the designed information providers to help developers debug.

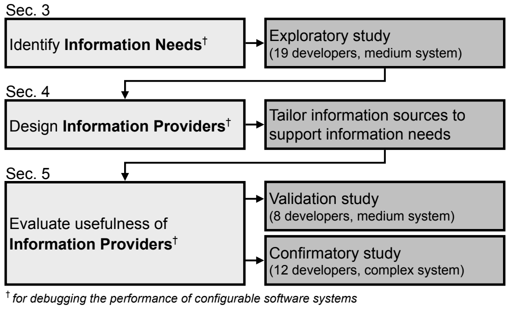

Sec. 3
Identify **Information Needs**†
Exploratory study
(19 developers, medium system)
Sec. 4
Design **Information Providers**†
Tailor information sources to
support information needs
Sec. 5
Evaluate usefulness of
Information Providers†
Validation study
(8 developers, medium system)
Confirmatory study
(12 developers, complex system)
† for debugging the performance of configurable software systems

Figure 2: Overview of our human-centered approach to support the information needs that developers have when debugging the performance of configurable software systems.

2 PERFORMANCE DEBUGGING IN CONFIGURABLE SOFTWARE SYSTEMS

There is substantial literature on debugging the performance of software systems [e.g., 28, 32, 47, 57, 68]. Our goal is to support developers in the process of debugging the performance of configurable software systems; in particular, when developers do not even know which options or interactions in their current configuration cause an unexpected performance behavior.

When performance issues occur in software systems, developers need to identify relevant information to debug the unexpected performance behaviors [12, 15, 30, 56]. For this task, in addition to using off-the-shelf profilers [3, 4, 54], some researchers suggest using more targeted profiling techniques [14, 16, 17, 24, 83] and visualizations [6, 10, 16, 24, 63, 71] to identify and analyze the locations of performance bottlenecks. Alternatively, some researchers suggest using techniques to search for inefficient coding patterns [11, 17, 47, 55, 57, 69]. While these techniques are quite useful, there is limited evidence of their usefulness when debugging the performance of configurable software systems; particularly, to determine how performance issues are related to options and their interactions.

In addition to performance debugging techniques, there are several established *program debugging techniques*, such as delta debugging [84], program slicing [7, 40, 76], and statistical debugging [9, 68], that can help developers isolate relevant parts of a system to focus their debugging efforts. For this reason, the techniques have been implemented in the backend of tools to help developers debug [13, 38, 43, 61]. For example, Whyline [38] combines static and dynamic program slicing to allow developers to ask questions about a system's output. While these tools have been evaluated in terms of their technical accuracy and usability with numerous user studies, there is limited evidence of which and how these *program debugging techniques* can be used or adopted for debugging the performance of configurable software systems.

Performance issues in configurable software systems: Performance issues are often caused by misconfigurations or software bugs, both of which impair user experience. Misconfigurations are errors in which the system and the input are correct, but the system does not behave as desired, because the selected configuration is inconsistent or does not match the intended behavior [81, 86, 88]. By contrast, a software bug is a programming error that degrades a system's behavior or functionality [9, 84, 85]. Regardless of the root cause of the unexpected behavior (misconfigurations or software bugs), systems often misbehave with similar symptoms, such as crashes, incorrect results [9, 48, 84, 85], and, in terms of performance, long execution times or increased energy consumption [29, 32, 34, 42, 45, 69, 77].

Research has repeatedly found that configuration-related performance issues are common and complex to fix in software systems [29, 30, 32, 42, 74]. For example, Han and Yu [29] found that 59% of performance issues concern configurations, 72% of which involved one option, and 88% require fixing the code. Similarly, Krishna et al. [42] found that 61% of performance issues take an average of 5 weeks to fix, and Wang et al. [74] found that 50% of patches in open-source cloud systems and 30% of questions in forums are related to configurations and performance issues. Performance issues were primarily caused by incorrect implementation of configurations, synchronization issues, and misconfigurations.

Debugging performance in configurable software systems: Similar to debugging performance in general, identifying relevant information is key when debugging unexpected performance behaviors in configurable software systems. Ideally, developers would have relevant information to debug how performance issues are related to specific options and their interactions. Unfortunately, there are situations in which developers only know the *effect* of an unexpected performance behavior (e.g., a long execution time as in Figure 1). In these situations, developers need to debug the system to determine whether the system has a potential *bug*, is *misconfigured*, or *works correctly*, but the user has a *different expectation* about the performance behavior of the system. For these reasons, our goal is to support developers in *finding relevant information* to debug unexpected performance behaviors in configurable software systems.

There are some research areas that aim to help developers understand how options affect a system's behavior. For instance, some researchers argue that information-flow analyses can help developers understand how options affect a system's behavior [44, 46, 49, 70, 78, 80, 87]. For example, Lotrack [46] used static taint analysis to identify under which configurations particular code fragments are executed. While these techniques can solve specific challenges, there is limited evidence of the usefulness of these techniques, particularly, for debugging the performance of configurable software systems.

In terms of understanding performance, some researchers suggest that performance-influence models can help developers debug unexpected performance behaviors [27, 39, 65, 72, 73, 75], as the models describe a system's performance in terms of its options and their interactions. For example, the model $8 \cdot A \cdot B + 5 \cdot C$ explains the influence of the options A, B, and C, and their interactions on the performance of a system; for instance, enabling C increases the execution time by 5 seconds, and enabling A and B, *together*, further increase the execution time by 8 seconds. Some approaches also model the performance of individual methods [31, 72, 73, 75], which can be useful to locate *where* options affect a system's performance. However, while performance-influence models have been evaluated in terms of accuracy [37, 65–67, 72, 73] and optimizing performance [25, 53, 58, 89], they have not been evaluated in terms of usability; in particular, to support developers' needs when debugging the performance of configurable software systems.

In this paper, we take a human-centered approach [19, 52] to identify, design, implement, and evaluate a solution to support developers when debugging the performance of configurable software systems. We first conduct an exploratory user study to *identify* the *information needs* that developers have when debugging the performance of configurable software systems. Afterwards, we *design* and *implement information providers* to support developers' needs, adapting techniques from prior work. Finally, we conduct a *validation* user study and a *confirmatory* user study to evaluate that the designed information providers actually support developers' information needs and speed up the process of debugging the performance of complex configurable software systems.

3 EXPLORING INFORMATION NEEDS

While there are numerous performance and program debugging techniques, there is limited empirical evidence of how useful the techniques are to support developers' needs when debugging the performance of configurable software systems. Hence, we first investigate the *information needs* that developers have and the process that they follow to debug the performance of configurable software systems. Specifically, we answer the following research questions:

RQ1: What information do developers look for when debugging the performance of configurable software systems?

RQ2: What is the process that developers follow and the activities that they perform to obtain this information?

RQ3: What barriers do developers face during this process?

3.1 Method

We conducted an exploratory user study to *identify* the *information needs* that developers have when debugging the performance of configurable software systems. Using Zeller's terminology [85], we want to understand how developers *find possible infection origins*: where options affect performance, and *analyze the infection chain*: what are the causes of an unexpected performance behavior, when debugging the performance of configurable software systems.

Study design. We conducted the exploratory user study, combining a think-aloud protocol [33] and a Wizard of Oz approach [18], to observe how participants debug a performance issue for 50 minutes: We encourage participants to verbalize what they are doing or trying to do (i.e., think-aloud component), while the experimenter plays the role of some tool that can provide performance behavior information (e.g., performance profiles and execution time of specific configurations) on demand (i.e., Wizard of Oz component), thus avoiding overhead from finding or learning specific tools.

We decided to provide additional information to participants halfway through the study, after we found, in a pilot study with 4 graduate students from our personal network, that participants spend an extremely long time (~60 minutes in a relatively small system) just identifying relevant options and methods. To additionally explore how participants search for the cause of performance issues once they have identified options and methods, we told participants, after 25 minutes, which options cause the performance issue and the methods where the options influence performance. In this way, we can both observe how participants start addressing the problem and analyze how options affect the performance in the implementation.

After the task, we conducted a brief semi-structured interview to discuss the participants' experience in debugging the system, as well as the information that they found useful and would like to have when debugging the performance of configurable software systems.

Due to the COVID-19 pandemic, we conducted the studies remotely over Zoom. We asked participants to download and import the source code of the subject system to their favorite IDE, to avoid struggles with using an unfamiliar environment. We also asked participants to share their screen. With the participants' permission, we recorded audio and video of the sessions for subsequent analysis.

Task and subject system. Based on past studies [48, 50, 51, 60] that have shown how time-consuming debugging even small configurable software systems is, we prepared one performance debugging task for one configurable software system of moderate size and complexity. We selected *Density Converter* as the subject system, which transforms images to different dimensions and formats. We selected this Java system because it is medium-sized, yet non-trivial (over 49K SLOC and 22 binary and non-binary options), and has many options that influence its performance behavior (execution time on the same workload ranged from few seconds to a couple of minutes, depending on the configuration). The task involved a user-defined configuration that spends an excessive amount of time executing. We introduced a bug caused by the incorrect implementation of one option, representative of bugs reported in past research [5, 29, 34] (the system was spending a long time to transform and output a JPEG image). Participants were required to identify and explain which and how options caused the unexpected performance behavior. We, however, did not explicitly tell participants that the performance problem was related to configurations.

Participants. We recruited 14 graduate students and 5 professional software engineers with extensive experience analyzing the performance of configurable Java systems. We stopped recruiting when we observed similar information needs and patterns in the debugging process. We used our professional network and LinkedIn for recruiting. The graduate students had a median of 6.5 years of programming experience, a median of 5 years in Java, a median of 3 years analyzing performance and a median of 4.5 years working with configurable software systems. The software engineers had a median of 13 years of programming experience, a median of 13 years in Java, a median of 5 years analyzing performance and a median of 5 years working with configurable software systems.

Analysis. We analyzed transcripts of the audio and video recordings of the debugging task and interviews using standard qualitative research methods [62]. The author who conducted the study coded the sessions using open and descriptive coding, summarizing observations, discussions, and trends [2]. All authors met weekly to discuss the codes and observations. When codes were updated, previously analyzed sessions were reanalyzed to update the sessions' coding.

Threats to Validity and Credibility. We observe how developers debug the performance of a system that they had not used before. Developers who are familiar with a system might have different needs or follow different processes. While readers should be careful when generalizing our findings, the needs help us identify the information that, at the very least, developers want to find when debugging the performance of unfamiliar configurable software systems.

Conducting a study with one system in which one option causes a performance issue has the potential to overfit the findings to this scenario, even though the scenario mirrors common problems in practice [29]. While our later study intentionally varies some aspects of the design to observe whether our solutions generalize to other tasks, generalizations about our results should be done with care.

3.2 Results

We observed that participants struggle for a long time looking for relevant information to debug the performance of the configurable subject system. In fact, no participant was able to finish debugging the system within 50 minutes! In what follows, we present the information needs that participants had, the process that they followed, and the barriers that they faced during the debugging process.

RQ1: Information Needs. Table 1 lists the four information needs that we identified and the number of participants that looked for each need. We refer to the information needs as influencing options, option hotspots, cause-effect chain, and user hotspots. The participants referred to these needs using varying terms.

When participants faced a non-trivial configuration space, they all tried to identify the influencing options – the option or interaction causing the unexpected performance behavior. More specifically, the participants tried to identify *which options* in the *problematic configuration* caused the unexpected performance behavior.

Some participants tried locating option hotspots – the methods where options affect the performance of the system. More specifically, the participants tried to *locate* where the *effect* of the problematic configuration could be observed; the methods whose execution time increased under the problematic configuration.

When we told participants (a) which options cause the unexpected performance behavior (i.e., the influencing options) and (b) the methods where these options influence performance (i.e., the option hotspots), all participants tried tracing the cause-effect chain – how influencing options are used in the implementation to directly and indirectly affect the performance of option hotspots. More specifically, as the participants (a) knew which options were *causing* an unexpected performance behavior and (b) had observed the *effect* of those options on the system's performance, the participants tried to find the *root cause* of the unexpected performance behavior.

Some participants also looked for user hotspots – the methods that spend a long time executing under the user-defined problematic configuration. However, as we will discuss in RQ2, these participants looked for this information trying to locate option hotspots (i.e., how options might affect the execution time of the expensive methodsunder the user-defined configuration).

RQ1: Developers look for information to (1) identify influencing options, (2) locate option hotspots, and (3) trace the cause-effect chain of how options influence performance in the implementation.

Table 1: Information needs, activities, and information sources for debugging the performance of configurable software systems.

Information Need	Description	Frequency	Activities	Frequency	Information Source
Influencing Options	Which options influence the performance of the system?	19/19	Read options' documentation	19/19	Global performance-influence models
			Measure the performance of numerous configurations	10/19	
User Hotspots	What are the hotspots under the problematic configuration?	9/19	Profile the system under the problematic configuration	9/19	CPU Profiling
Option Hotspots	Where do options influence the performance of the system?	10/19	Trace options in the implementation	4/10	Local performance-influence models
			Analyze the user hotspots' source code	6/10	
			Inspect the user hotspots' call stacks	6/10	
Cause-Effect Chain	How are influencing options used in the implementation to directly and indirectly influence the performance of option hotspots?	19/19	Analyze the option hotspots source code	19/19	CPU Profiling Program Slicing
			Inspect the option hotspots call stacks	12/19	
			Use a debugger to analyze how the influencing options affect the values of the variables used in the option hotspots	10/19	
			Manually trace how influencing options are used in the implementation to directly and indirectly affect the performance of option hotspots	19/19	

RQ2: Process and Activities. Table 1 lists the activities that participants performed when looking for relevant information and the number of participants that performed each activity. Overall, all participants *compared* the problematic configuration to the default configuration, to understand the causes of the unexpected performance behavior. In particular, the participants compared the values selected for each option and analyzed how the changes were affecting the performance of the system in the implementation.

When looking for the influencing options, the participants mainly *read documentation* and *executed* the system under *multiple configurations*, primarily *comparing* execution times. With these approaches, the participants tried to identify *which options* in the *problematic configuration* were causing the unexpected behavior.

When looking for option hotspots, the participants mainly *profiled* the system under the problematic configuration, and *analyzed* the *call stacks* and *source code* of hotspots, trying to *locate the methods* where *options* might be *affecting* the performance of hotspots.

When looking for the cause-effect chain, some participants *analyzed* the option hotspots' *source code*, whereas others used a *debugger*; trying to understand *how* the influencing options are *used* in the implementation to affect the performance of option hotspots. Several participants also *compared* the hotspots' *call stacks* under the problematic and default configurations, trying to understand how the influencing options affected how the option hotspots were called. Ultimately, all participants tried to *manually trace how* the influencing options were being *used* in the implementation to directly and indirectly affect the performance of the option hotspots.

While identifying the influencing options and locating the option hotspots is needed to trace the cause-effect chain, the order in which the first two pieces of information was acquired did not affect the debugging process. For instance, 9 participants started looking for influencing options, but gave up trying after a while. Then, the participants looked for and were able to identify user hotspots. Six of these participants subsequently started looking for option hotspots (i.e., how options might affect the execution time of the expensive methods under the user-defined configuration).

> *RQ2: Overall, developers compare the problematic configuration to a non-problematic baseline configuration to understand the causes of an unexpected performance behavior. Initially, developers compare execution times to identify influencing options, and analyze call stacks and source code to locate option hotspots. These two pieces of information are necessary to trace the cause-effect chain of how influencing options are used in the implementation to directly and indirectly influence the performance of option hotspots.*

RQ3: Barriers. Our participants struggled for a long time trying to find relevant information to debug how options influence the performance of the system in the implementation. Most participants discussed the *"tedious and manual"* process of executing multiple configurations when looking for influencing options. For instance, only 10 out of the 19 participants identified the influencing options. While we told participants the system's execution time under any configuration that they wanted, several participants mentioned that finding the problematic option would have *"taken me hours."*

Most participants also mentioned the struggle to locate option hotspots. In fact, no participant found any option hotspot! Several participants mentioned that *locating* these methods is challenging since options are not typically directly used in expensive methods.

The participants struggled the most when trying to trace the cause-effect chain. In fact, no participant could establish the cause-effect chain, even when our task consisted of tracing *a single* influencing option and we explicitly told participants the influencing option and the option hotspots they needed to analyze. Most participants mentioned that manually tracing even one option through a relatively small system is *"error-prone."* Additionally, some participants discussed that *identifying differences* in the option hotspots' *call stacks* was difficult for determining whether the influencing options were affecting how the option hotspots were called. As mentioned by several participants: Variables used in option hotspots are often a *"result of several computations"* involving influencing options. Since the influencing options are used in various parts of the system, *"tracing which paths to follow is very challenging."*

RQ3: Developers struggle for a substantial amount of time looking for relevant information to identify influencing options, locate option hotspots, and trace the cause-effect chain.

4 SUPPORTING INFORMATION NEEDS

We aim to support developers in identifying influencing options, locating option hotspots, and tracing the cause-effect chain. To this end, we design *information providers*, adapting *information sources*, to support the above needs. We implement the designed information providers in a tailored and cohesive prototype called **GLIMPS** [1], which can assist developers debug the performance of configurable software systems. Table 1 shows which information needs are supported by the information sources **G**lobal and **L**ocal performance-**I**nfluence **M**odels, CPU **P**rofiling, and program **S**licing that we adapt for designing information providers.

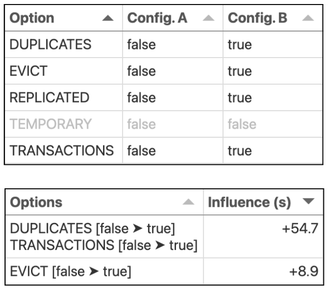
4.1 Identifying Influencing Options

To help developers identify the influencing options that cause an unexpected performance behavior, we select global performance-influence models [27, 39, 65, 72, 73, 75], which describe a system's performance in terms of its options and their interactions. For instance, the model 4.6 + 54.7 · Duplicates · Transactions + 8.9 · Evict + 3.5 · Temporary explains the influence of Duplicates, Transactions, Evict, and Temporary, and their interactions on a system's performance; for instance, enabling Evict increases the execution time by 8.9 seconds, and enabling Duplicates and Transactions, *together*, further increase the execution time by 54.7 seconds. The model also shows that other options (e.g., Replicated) do not influence the performance of the system.

We adapt this information source to design an *information provider* that shows developers influencing options; specifically, *which and how differences* between configurations (e.g., a problematic and a non-problematic configuration) influence a system's performance. In our implementation, this information provider highlights the differences in the values of options selected between two configurations, and shows the influencing options between the configurations. If changes between the configurations are not shown, then the changes do not influence the performance of the system.[1]

Global performance-influence models are typically built by measuring execution time under different configurations [65]. The models can be built using white-box techniques [72, 73, 75], machine-learning approaches [23, 25–27], or a brute-force approach. Details on these techniques are beyond the scope of this paper. In our implementation, we generate the models using white-box techniques, as these approaches first generate local performance-influence models (which our tool also uses) to obtain the global model for the system [72, 73, 75].

Example. Figure 3 shows a screenshot of our tool highlighting the differences of the options selected between two configurations (e.g., a non-problematic and problematic configurations). Our tool shows the influencing options between these two configurations. For instance, changing both options Duplicates and Transactions from `false` to `true` results in an interaction that increased the execution time by 54.7 seconds. Based on this information, most developers would consider Duplicates and Transactions as influencing options that are causing an unexpected performance behavior.



Option ▲	Config. A ▲	Config. B ▲
DUPLICATES	false	true
EVICT	false	true
REPLICATED	false	true
TEMPORARY	false	false
TRANSACTIONS	false	true

Options ▲	Influence (s) ▼
DUPLICATES [false ➤ true] TRANSACTIONS [false ➤ true]	+54.7
EVICT [false ➤ true]	+8.9

Figure 3: Our tool highlights differences in the options selected between two configurations, and shows the influencing options of the changes from one configuration to (►) another configuration.

Note that individual changes to Duplicates and Transactions did not influence the performance of the system; only the *interaction* increased the execution time. Additionally, any other changes between the configurations – Replicated – do not influence the performance of the system. Likewise, the influence of Temporary is not shown, as both configurations selected the same value.

4.2 Locating Option Hotspots

After helping developers identify the influencing options, we help developers locate the option hotspots where these options cause an unexpected performance behavior. To this end, we select local performance-influence models [72, 73, 75]. Analogous to how global performance-influence models describe the influence of options and interactions on a *system*'s performance, local models describe the influence of options on the performance of *individual methods*. Hence, local models indicate *where* options affect the performance in the implementation [72, 73, 75]. For instance, the local model of a method 0.9 + 42.9 · Duplicates · Transactions explains the influence of Duplicates and Transactions on the performance of the specific method, rather than the entire system.

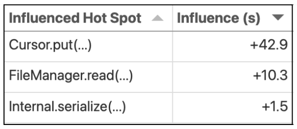
We adapt this information source to design an *information provider* that shows developers option hotspots; specifically, *where* and *by how much* options influence the performance of the system. In our implementation, this information provider shows (a) the *methods* whose performance is influenced by changes made between configurations (e.g., a problematic and a non-problematic configuration) and (b) the *influence* of the changes on each method's performance.[2]

Local performance-influence models are usually built with white-box approaches as by-products of global models [72, 73, 75]. In our implementation, we generate local models using these approaches.

Example. Figure 4 shows as screenshot of our tool indicating the option hotspots where the influencing options Duplicates and Transactions affect the system's performance. Note that the influence on all methods equals the influence in the entire system (see Figure 3). Based on this information, most developers would

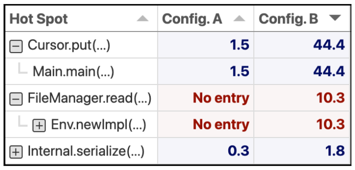
[1] Any performance-influence model is shown relative to one configuration (e.g., the default configuration), which explains the impact of changes to that configuration. In our tool, developers can select that one configuration.

[2] Our tool allows developers to select a single configuration to analyze individual local performance-influence models.



Influenced Hot Spot ▲	Influence (s) ▼
Cursor.put(...)	+42.9
FileManager.read(...)	+10.3
Internal.serialize(...)	+1.5

Figure 4: Our tool shows option hotspots affected by influencing options, as well as the influence on performance in each method.



Hot Spot ▲	Config. A ▲	Config. B ▼
⊟ Cursor.put(...)	1.5	44.4
└ Main.main(...)	1.5	44.4
⊟ FileManager.read(...)	No entry	10.3
└ ⊞ Env.newImpl(...)	No entry	10.3
⊞ Internal.serialize(...)	0.3	1.8

Figure 5: Our tool helps developers trace the cause-effect chain by highlighting the differences in the option hotspots' execution time and call stacks affected by influencing options. While the call stacks of `Cursor.put` are the same under both configurations, `FileManager.read` is only called under the second configuration.

consider `Cursor.put` as an option hotspot; the location where the *effect* of the influencing options is observed.

4.3 Tracing the Cause-Effect Chain

After helping developers identify the influencing options and locate the option hotspots, we help developers trace the cause-effect chain. To this end, we select CPU profiling and program slicing.

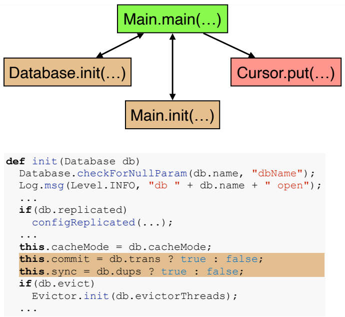
4.3.1 CPU Profiling

We select CPU profiling to collect the hotspot view of the problematic configuration and a non-problematic configuration. The hotspot view is the inverse of a call tree: A list of all methods sorted by their total execution time, cumulated from all different call stacks, and with back traces that show how the methods were called.

We adapt this information source to design an *information provider* that helps developers trace the cause-effect chain; specifically, *compare* the hotspot view of two configurations (e.g., a non-problematic and a problematic configuration) to help developers determine whether the influencing options *affect* how option hotspots are called. In our implementation, this information provider highlights *differences* in the option hotspots' execution time and call stacks.[3]

CPU profiles are collected with most off-the-shelf profilers. White-box approaches that build global and local performance-influence models collect these profiles [72, 73, 75]. In our implementation, we use the CPU profiles collected by these approaches.

Example. Figure 5 shows a screenshot of our tool, which helps developers trace the cause-effect chain by highlighting differences in the option hotspots' execution time and call stacks based on the influencing options Duplicates and Transactions. For instance, the changes increased `Cursor.put`'s execution time, but did not affect how the method was called. By contrast, `FileManager.read` is only executed under the problematic configuration. This information can help developers understand how the influencing options are used in the implementation to affect the option hotspots's performance.



```
def init(Database db)
  Database.checkForNullParam(db.name, "dbName");
  Log.msg(Level.INFO, "db " + db.name + " open");
  ...
  if(db.replicated)
    configReplicated(...);
  ...
  this.cacheMode = db.cacheMode;
  this.commit = db.trans ? true : false;
  this.sync = db.dups ? true : false;
  if(db.evict)
    Evictor.init(db.evictorThreads);
  ...
```

Figure 6: Our tool helps developers trace the cause-effect chain by displaying a method-level dependence graph from the method where the influencing options are first loaded into the system (green box) to and option hotspot (red box). Other relevant methods are shown in brown boxes. When clicking on a box, GLIMPS opens the file with the method and highlights the statements of the slice. The position of the nodes in the graph (left to right and top to bottom) does not represent the order of execution of methods.

4.3.2 Program Slicing

We select program slicing [41, 76, 79, 85]; an approach to compute relevant fragments of a system based on a criterion. Several debugging tools have been implemented on top of program slicers [20, 38, 43, 79] to help developers narrow down relevant parts of a system where developers should focus their debugging efforts.

We adapt this information source to design an *information provider* that helps developers trace the cause-effect chain; specifically, *tracking how* influencing options are *used in the implementation* to directly and indirectly *influence* the performance of option hotspots. In our implementation, this information provider slices (chops) a system from where influencing options are first loaded into the system to the option hotspots, and shows (a) a *method-level dependence graph* and (b) *highlighted statements* of the slice in the source code.

Ideally, we would slice the program dynamically, as we are analyzing a system's dynamic behavior and to avoid approximations in the results. However, after exploring various dynamic and static slicing research tools, we settled on the state-of-the-art static slicer provided by JOANA [22], as it is the most mature option.

Example. Figure 6 shows a screenshot of our tool, which helps developers trace the cause-effect chain by showing a method-level dependence graph from the `main` method, in which the influencing options Duplicates and Transactions are first loaded into the system, to the option hotspot `Cursor.put`. The graph can help developers track dependences across methods in the system. When clicking on a method on the graph, our implementation opens the file with the method and highlights the statements in the slice, such as in Figure 6. The highlighted statements can help developers trace the cause-effect chain by tracking how influencing options are used in the implementation to directly and indirectly cause an unexpected performance behavior in option hotspots.

4.4 Implementation

We implemented the information providers in a Visual Studio Code extension called **GLIMPS** [1]. Our prototype adapts **G**lobal and

[3] In our tool, developers can also analyze the CPU profile of one configuration.

Local performance-**I**nfluence **M**odels, CPU **P**rofiles, and a program **S**licer. The first three items are collected prior to debugging, using an infrastructure where developers configure and run the system. Subsequently, developers use **GLIMPS** to identify influencing options, locate option hotspots, and trace the cause-effect chain.

GLIMPS is agnostic to the information sources used to adapt and implement information providers. In fact, **GLIMPS** is entirely built on existing infrastructure of information sources for global and local performance-influence modeling, CPU profiling, and program slicing. The novelty to **GLIMPS** is in the *design* and *integration of information providers*, from multiple *information sources*, into a *cohesive* infrastructure and user interface, which can help developers debug the performance of configurable software systems.

Our implementation uses Comprex [73] to build the global and local performance-influence models, which uses JProfiler [3] to collect the CPU profiles. Our implementation uses JOANA to slice the system from influencing options to option hotspots, using a fixed-point chopper algorithm, which first computes a backward slice from the option hotspots, and then computes a forward slice, on the backward slice, from the influencing options [21]. For scalability and to reduce approximations, we modified JOANA to consider code coverage under the problematic and a non-problematic configurations.

5 EVALUATING USEFULNESS OF INFORMATION PROVIDERS

We evaluate the usefulness of our designed information providers to help developers debug the performance of configurable software systems. Specifically, we answer the following research question:

RQ4: To what extent do the designed information providers help developers debug the performance of configurable software systems?

We answer this research question with two user studies using different designs. We first evaluate the extent that our designed information providers support the information needs that we identified in our exploratory study. To this end, we conduct a *validation* study, in which we ask the participants of our exploratory study to debug a *comparable* unexpected performance behavior using **GLIMPS** on the same subject system (Sec 5.1). Afterwards, we replicate the study, intentionally *varying* some aspects of the designs (theoretical replication [36, 64]), to evaluate to which extent our information providers generalize for a more complex task with an interaction in a larger system. Specifically, we conduct a *confirmatory* study, in which we ask a *new set of participants* to debug a *more complex task* on a *more complex subject system* (Sec 5.2). The validation and confirmatory studies, *together*, provide evidence that our information providers help developers debug the performance of complex configurable software systems *because* the information providers *support* the information needs that developers have in this process.

5.1 Validating Usefulness of Information Providers

We first conducted a *validation* study to evaluate the extent that the designed information providers support the information needs that we identified in our exploratory study.

5.1.1 Method

Study design. We invited the participants from our exploratory study, after 5 months, to solve another problem in the same system, but now with the help of our information providers. This design can be considered as a within-subject study, where subjects perform tasks both in the control and in the treatment condition: Specifically, we consider our exploratory study as the *control* condition, in which participants debugged a system *without* **GLIMPS**, and consider the new study as the *treatment* condition, in which participants debug a *comparable* performance issue for 50 minutes in the same subject system with **GLIMPS**. Similar to the exploratory study, we use think-aloud protocol [33] to identify whether our information providers actually support the information needs that developers have when debugging the performance of the subject system.

Prior to the task, participants worked on a warm-up task for 20 minutes using **GLIMPS**. We tested the time for the warm-up task, as well as **GLIMPS**'s design and implementation in a pilot study with 4 graduate students from our personal network.

After the task, we conducted a brief semi-structured interview to discuss the participants' experience in debugging the system, as well as the usefulness of the information providers, and to contrast their experience to debugging without the information providers.

Due to the COVID-19 pandemic, we conducted the studies remotely over Zoom. Participants used Visual Studio Code through their preferred Web browser. The IDE was running on a remote server and was configured with **GLIMPS**. We asked participants to share their screen. With the participants' permission, we recorded audio and video of the sessions for subsequent analysis.

Task and subject system. We prepared a comparable, *but different*, debugging task to the task in our exploratory study. Similar to the exploratory study, the task involved a user-defined configuration in *Density Converter* that spends an excessive amount of time executing. We introduced a bug caused by the incorrect implementation of one option (the system was using a larger scale of the input image instead of using a fraction). Participants were required to identify and explain which and how options caused the unexpected performance behavior. Similar to the exploratory study, we did not explicitly tell participants that the performance problem was related to configurations. In contrast to the exploratory study, the user-defined configuration, bug, and problematic option, were *different*.

Participants. We invited the participants from our exploratory study to work on our task. After conducting the study with 8 participants, we observed a *massive effect size* between debugging with and without our information providers (correctly debugging within 19 minutes compared to failing after 50 minutes, see Sec. 5.1.2). Hence, we did not invite the remaining participants.

Analysis. We analyzed and compared transcripts of the audio and video recordings of the exploratory and validation studies to measure the time participants spend working on the task and their success rates. Based on our exploratory study, participants needed to identify influencing options, locate option hotspots, and trace the cause-effect chain to correctly debug the system. We also analyzed the interviews using standard qualitative research methods [62]. The author who conducted the study analyzed the sessions independently, summarizing observations, discussions, and trends during the task and interviews. All authors met weekly to discuss the observations.

Threats to Validity and Credibility. We invited the same participants and use the same subject system as in our exploratory study. Such a design might only validate the information needs when debugging performance in the selected subject system. Additionally, the exploratory and validation studies were conducted 5 months apart, which might result in learning effects that help participants in the latter study. Furthermore, there is the threat that the task in the validation study is simpler. While our later study varies these aspects to observe whether our solutions generalize to other tasks, generalizations about our results should be done with care.

5.1.2 Results

RQ4: Validating Usefulness of Information Providers. Figure 7 shows the time that each participant spent looking for each piece of information while debugging the performance of *Density Converter* with (treatment) and without (control) **GLIMPS**. Overall, all participants who used our information providers identified the influencing options, located option hotspots, and traced the cause-effect chain, and correctly explained the root cause of the performance issue in less than 19 minutes. By contrast, the 8 participants could debug the unexpected behavior without our information providers in 50 minutes, when they struggled to find relevant information.[4]

All 8 participants who used our information providers identified the influencing options and located the option hotspots in a few minutes. Afterwards, all participants traced the cause-effect chain and explained how the influencing options caused the unexpected performance behavior in the option hotspots.

When these 8 participants did not use our information providers, no participant found a single piece of information in the same timeframe as they did when using our information providers. In fact, during a 25-minute window, only 3 participants found the influencing options, and no participant found any option hotspots. Furthermore, as described in our exploratory study, even when we *explicitly told* participants (a) the one influencing option that was causing the unexpected performance behavior and (b) the option hotspots whose execution times drastically increased as a result of the problematic option, no participant could trace the cause-effect chain and find the root cause of the unexpected behavior within 25 minutes.

After completing the task, the participants discussed how the information providers helped them debug the performance of the system, and contrasted their experience to debugging without our tool. All participants mentioned that the information providers helped them obtain relevant information for debugging. The consensus was that the information providers *"helped me focus on the relevant parts of the system"* to debug the unexpected performance behavior. The participants contrasted this experience to the struggles that they faced when debugging without our tool. In particular, some participants remembered *"being lost"* on what methods to follow or knowing *"which parts of the program are relevant."*

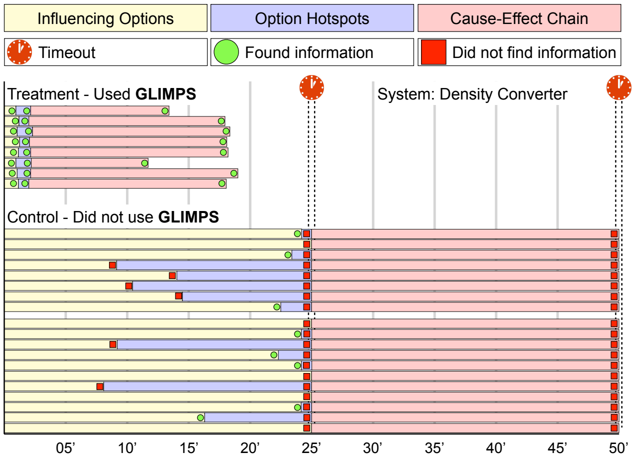


Figure 7: Time looking for each piece of information when debugging performance with and without GLIMPS. The first 8 participants who did not use GLIMPS are the same participants who used our tool. The data for the other participants who did not use GLIMPS is included for reference.

> *RQ4: The validation study provides evidence that the designed information providers support the information needs that we identified in our exploratory study. Specifically, the information providers support developers' needs to (a) identify influencing options, (b) locate option hotspots, and (c) trace the cause-effect chain.*

5.2 Confirming Usefulness of Information Providers

After validating that our information providers support the information needs that we identified, we conducted a *confirmatory* study to evaluate the extent that the information providers can potentially generalize to support the information needs of debugging the performance of complex configurable software systems.

5.2.1 Method

Study design. We replicated the validation study, intentionally *varying* some aspects of the designs: We used a *between-subject design* where we ask a *new set of participants* to debug a *more complex task* on a *more complex subject system*, all working on the same task but using *different tool support*. With these variations, we evaluate that the results and *massive effect size* in our previous studies are not due to, for example, a simpler task or learning effects, but rather, that the information providers help developers debug the performance of complex configurable software systems, *because* the information providers support the needs that developers have in this process.

As in our prior study, we conducted the confirmatory study using a think-aloud protocol [33], to compare how two new sets of participants debug the performance of a complex configurable software system using different tool support in 60 minutes. The treatment group used **GLIMPS**, while the control group used a simple plugin, which profiles and provides the execution time of the system under any configuration. This information is the same that we gave participants in our exploratory study using a Wizard of Oz approach. For this confirmatory study, however, we did not

[4] We did not conduct a statistical significance test, since comparing completion rates is obvious: All participants correctly debugged with our tool, but no participants correctly debugged without our tool; comparing completion times cannot be done since nobody completed the task without our tool.

use a Wizard of Oz approach, as we wanted both groups to access information for debugging using a tool and the same IDE.

Prior to the task, participants worked on a warm-up task for 20 minutes using either **GLIMPS** or the simple plugin to learn how to use the information providers or the components that provided performance behavior information, respectively. We tested the simple plugin's design and implementation in a pilot study with 4 graduate students from our personal network.

After the task, we conducted a brief semi-structured interview to discuss the participants' experience in debugging the system. In particular, we asked participants in the treatment group about the usefulness of the information providers and whether there was additional information that they would like to have in the debugging process. Similarly, we asked participants in the control group for the information that they would like to have when debugging the performance of configurable software systems.

Due to the COVID-19 pandemic, we conducted the studies remotely over Zoom. Participants used Visual Studio Code through their preferred Web browser, which was running on a remote server and was configured with **GLIMPS** and the simple plugin. We asked participants to share their screen. With the participants' permission, we recorded audio and video of the sessions for subsequent analysis.

Task and subject system. We prepared a more complex performance debugging task for a more complex configurable software system than the task and subject system in our exploratory and validation studies. Similar to the prior studies, the task involved a user-defined configuration that spends an excessive amount of time executing. Participants were required to identify and explain which and how options caused the unexpected performance behavior. Similar to the previous studies, we did not explicitly tell participants that the performance problem was related to configurations. In contrast to the previous studies, we introduced a bug caused by the incorrect implementation of *an interaction of two options* (The system spent a long time inserting duplicate data using transactions). We selected *Berkeley DB* as the subject system due to the following reasons: (1) the system is implemented in Java, is open source, and is more complex than *Density Converter* (over 150K SLOC and over 30 binary and non-binary options) and (2) the system has a complex performance behavior (execution time ranges from a couple of seconds to a few minutes, depending on the configuration).

Participants. We recruited 12 graduate students, *independent* of our exploratory and validation studies, with extensive experience analyzing the performance of configurable Java systems.

When determining the number of participants for the control group, we made some ethical considerations, while also ensuring that we obtain reliable results. In our exploratory study, we observed 19 *experienced* researchers and professional software engineers who *could not debug* the performance of a *medium-sized* system with a performance bug caused by a *single option* within 50 minutes (see Figure 7). With *Berkeley DB*, we want to observe how participants debug the performance of *a more complex system*; a significantly larger system, in terms of SLOC and configuration space size, in which the unexpected behavior is caused by an *interaction of two options*. Based on (a) the fact that we have *strong empirical evidence* that debugging the performance of configurable software systems without relevant information is frustrating and is highly likely to not be completed under 60 minutes and (b) the massive effect size in our validation study between debugging with and without our information providers, we decided to minimize the number of participants that we expect to struggle and fail to complete the task, while still having a reasonable number participants in the control group.

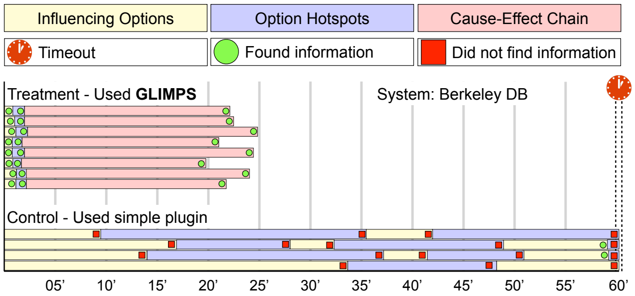


Figure 8: Time looking for each piece of information when debugging performance with different tool support.

Ultimately, we randomly assigned 4 out of the 12 participants to the control group, making sure to balance the groups in terms of the participants' debugging experience: The median programming experience for both groups is 6 years, a median of 3.5 years in Java, a median of 2.2 years of performance analysis experience, and a median of 2.7 years working with configurable software systems.

Analysis. We analyzed transcripts of the audio and video recordings to measure the time participants spend working on the task and their success rates. Based on our exploratory and validation studies, participants needed to identify influencing options, locate option hotspots, and trace the cause-effect chain to successfully debug the system. Additionally, we analyzed the interviews using standard qualitative research methods [62]. The author who conducted the study analyzed the sessions independently, summarizing observations, discussions, and trends during the debugging task and the interviews. All authors met weekly to discuss the observations.

Threats to Validity and Credibility. While we aimed to increase the complexity of the performance debugging task, readers should be careful when generalizing our results to other complex systems.

Our control group consisted of 4 participants. As argued previously, we did not recruit more participants due to the struggles that we observed in our exploratory study on a simpler system and the massive effect size in our validation study between debugging with and without our information providers. Nevertheless, readers should be careful when generalizing our results.

While the control group had access to the IDE's debugger and used a simple plugin, we might obtain different results if the participants had used other debugging tools and techniques.

5.2.2 Results

RQ4: Confirming Usefulness of Information Providers. Figure 8 shows the time each participant spent looking for each piece of information while debugging the performance of *Berkeley DB* with **GLIMPS** (treatment) and the simple plugin (control). Similar to our validation study, all participants who used our information providers identified the influencing options, located the option hotspots, and traced the cause-effect chain in less than 25 minutes. By contrast, the participants who did not use our information providers struggled for 60 minutes and could not debug the system.[5]

While working on the task, we observed the participants in the treatment group looking for the same information as in Table 1 and using our information providers similarly to the participants in the validation study to find information to debug the subject system. Likewise, the participants in the control group struggled while performing the same activities as those listed in Table 1 when trying to identify the influencing options and locate the option hotspots.

After working on the task, all participants discussed their experience in debugging the performance of the system using tool support. Similar to the discussion in our validation study, all participants in the treatment group commented how the information providers helped them identify influencing options, locate option hotspots, and trace the cause-effect chain. Likewise, the participants who used the simple plugin described similar struggles and barriers as those mentioned in our exploratory study. All participants in this group mentioned that identifying the influencing options that cause the expected behavior is *"difficult"* and locating the option hotspots is *"challenging."* However, none of the participants in this group commented on tracing the cause-effect chain, as they never got to that point in the debugging process.

RQ4: The confirmatory study provides evidence that the designed information providers help developers debug the performance of complex configurable software systems because the information providers support the needs that developers have in this process.

6 CONCLUSION

We identified the information needs – influencing options, option hotspots, and cause-effect chain – that developers have when debugging the performance of configurable software systems. Subsequently, we designed and implemented information providers, adapted from **G**lobal and **L**ocal performance-**I**nfluence **M**odels, CPU **P**rofiling, and program **S**licing, that support the above needs. Two user studies, with a total of 20 developers, validate and confirm that our designed information providers help developers debug the performance of complex configurable software systems.

7 ACKNOWLEDGMENTS

We want to thank James Herbsleb, for his guidance on conducting a Wizard of Oz experiment, Thomas LaToza, for his advice on how to analyze and code sessions, and Rohan Padhye, Janet Siegmund, and Bogdan Vasilescu, for their feedback on the design of our confirmatory user study. This work was supported in part by the NSF (Awards 2007202, 2107463, and 2038080), NASA (Awards 80NSSC20K1720 and 521418-SC), the Software Engineering Institute, the German Research Foundation (SI 2171/2, SI 2171/3-1, and AP 206/11-1, and Grant 389792660 as part of TRR 248 – CPEC), and the German Federal Ministry of Education and Research (AgileAI: 01IS19059A and 01IS18026B) by funding the competence center for Big Data and AI "ScaDS.AI Dresden/Leipzig".

REFERENCES

[1] [n.d.]. *Link to Visual Studio Code extension. Omitted for double-blind review.*
[2] [n.d.]. *See supplementary material.* https://bit.ly/3mgFhty
[3] 2019. *JProfiler 10.* Retrieved December 10, 2019 from https://www.ej-technologies.com/products/jprofiler/overview.html
[4] 2020. *VisualVM.* Retrieved November 24, 2020 from https://visualvm.github.io/
[5] Iago Abal, Jean Melo, Ştefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* 26, 3, Article 10 (Jan. 2018), 34 pages.
[6] Andrea Adamoli and Matthias Hauswirth. 2010. Trevis: A Context Tree Visualization and Analysis Framework and Its Use for Classifying Performance Failure Reports. In *Proc. Int'l Symposium Software Visualization (SOFTVIS)* (Salt Lake City, UT, USA). ACM, New York, NY, USA, 73–82.
[7] Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. *ACM SIGPlan Notices* 25, 6 (1990), 246–256.
[8] Mohammad Mejbah ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. 2017. SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs. In *Proc. European Conference on Computer Systems (EuroSys)* (Belgrade, Serbia). ACM, New York, NY, USA, 298–313.
[9] David Andrzejewski, Anne Mulhern, Ben Liblit, and Xiaojin Zhu. 2007. Statistical Debugging Using Latent Topic Models. In *Proc. European Conf. Machine Learning* (Warsaw, Poland). Springer-Verlag, Berlin, Heidelberg, 6–17.
[10] Cor-Paul Bezemer, J.A. Pouwelse, and Brendan Gregg. 2015. Understanding software performance regressions using differential flame graphs. In *Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER).* IEEE, 535–539.
[11] James Bornholt and Emina Torlak. 2018. Finding Code That Explodes Under Symbolic Evaluation. *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* 2, Article 149 (Oct. 2018), 26 pages. https://doi.org/10.1145/3276519
[12] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. 2010. Information Needs in Bug Reports: Improving Cooperation between Developers and Users. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)* (Savannah, GA, USA). ACM, New York, NY, USA, 301–310.
[13] Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. 2013. Interactive Record/Replay for Web Application Debugging. In *Proc. Symposium User Interface Software and Technology (UIST)* (St. Andrews, Scotland, United Kingdom). ACM, New York, NY, USA, 473–484.
[14] Pablo De Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. 2015. CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization. *ACM Trans. Archit. Code Optim. (TACO)* 12, 1, Article 6 (April 2015), 24 pages.
[15] 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 *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)* (Paderborn, Germany). ACM, New York, NY, USA, 396–407.
[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 *Proc. Int'l Conf. Software Engineering: Companion Proceeedings* (Gothenburg, Sweden). ACM, New York, NY, USA, 41–44.
[17] Charlie Curtsinger and Emery D. Berger. 2016. COZ: Finding Code that Counts with Causal Profiling. In *USENIX Annual Technical Conference (ATC).* USENIX Association, Denver, CO, USA.
[18] Nils Dahlbäck, Arne Jönsson, and Lars Ahrenberg. 1993. Wizard of Oz studies—why and how. *Knowledge-Based Systems* 6, 4 (1993), 258–266.
[19] Tayba Farooqui, Tauseef Rana, and Fakeeha Jafari. 2019. Impact of Human-Centered Design Process (HCDP) on Software Development Process. In *Int'l Conf. Communication, Computing and Digital systems (C-CODE)* (Islamabad, Pakistan). 110–114. https://doi.org/10.1109/C-CODE.2019.8680978
[20] Xiaoqin Fu, Haipeng Cai, and Li Li. 2020. Dads: Dynamic Slicing Continuously-Running Distributed Programs with Budget Constraints. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)* (Virtual Event, USA). ACM, New York, NY, USA, 1566–1570. https://doi.org/10.1145/3368089.3417920
[21] Dennis Giffhorn. 2011. Advanced chopping of sequential and concurrent programs. *Software Quality Journal* 19, 2 (2011), 239–294.
[22] Jürgen Graf, Martin Hecker, and Martin Mohr. 2012. *Using JOANA for Information Flow Control in Java Programs - A Practical Guide.* Technical Report 24. Karlsruhe Institute of Technology.
[23] Alexander Grebhahn, Norbert Siegmund, and Sven Apel. 2019. Predicting Performance of Software Configurations: There is no Silver Bullet. arXiv:1911.12643 [cs.SE]

[5] We did not conduct a statistical significance test, since comparing completion rates is obvious: All participants in the treatment group correctly debugged the system, but no participant in the control group did; comparing completion times cannot be done since nobody in the control group completed the task.

[24] Brendan Gregg. 2016. The Flame Graph. *Commun. ACM* 59, 6 (May 2016), 48–57.

[25] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wąsowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *Proc. Int'l Conf. Automated Software Engineering (ASE)* (Silicon Valley, CA, USA). ACM, New York, NY, USA, 301–311.

[26] Huong Ha and Hongyu Zhang. 2019. DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Montreal, Quebec, Canada). IEEE, 1095–1106.

[27] H. Ha and H. Zhang. 2019. Performance-Influence Model for Highly Configurable Software with Fourier Learning and Lasso Regression. In *Proc. Int'l Conf. Software Maintance and Evolution (ICSME)*. 470–480.

[28] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. 2012. Performance Debugging in the Large via Mining Millions of Stack Traces. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Zurich, Switzerland). IEEE, Piscataway, NJ, USA, 145–155.

[29] Xue Han and Tingting Yu. 2016. An Empirical Study on Performance Bugs for Highly Configurable Software Systems. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)* (Ciudad Real, Spain). ACM, New York, NY, USA, Article 23, 10 pages.

[30] Xue Han, Tingting Yu, and David Lo. 2018. PerfLearner: Learning from Bug Reports to Understand and Generate Performance Test Frames. In *Proc. Int'l Conf. Automated Software Engineering (ASE)* (Montpellier, France). ACM, New York, NY, USA, 17–28.

[31] Xue Han, Tingting Yu, and Michael Pradel. 2021. ConfProf: White-Box Performance Profiling of Configuration Options. In *Proc. Int'l Conf. Performance Engineering (ICPE)*. ACM, New York, NY, USA, 1–8.

[32] Haochen He, Zhouyang Jia, Shanshan Li, Erci Xu, Tingting Yu, Yue Yu, Ji Wang, and Xiangke Liao. 2020. CP-Detector: Using Configuration-related Performance Properties to Expose Performance Bugs. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*.

[33] Riitta Jääskeläinen. 2010. *Think-aloud protocol.* John Benjamins Publishing Amsterdam/Philadelphia. 371–374 pages.

[34] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In *Proc. Conf. Programming Language Design and Implementation (PLDI)* (Beijing, China). ACM, New York, NY, USA, 77–88.

[35] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. 2011. Catch Me If You Can: Performance Bug Detection in the Wild. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (Portland, Oregon, USA). ACM, New York, NY, USA, 155–170.

[36] Natalia Juristo and Omar S. Gómez. 2011. *Replication of Software Engineering Experiments.*

[37] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-Based Sampling of Software Configuration Spaces. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Montreal, Quebec, Canada). IEEE.

[38] Andrew J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior. In *Proc. Conf Human Factors in Computing Systems (CHI)* (Vienna, Austria). ACM, New York, NY, USA, 8 pages.

[39] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. 2018. Tradeoffs in modeling performance of highly configurable software systems. *Software and System Modeling (SoSyM)* (08 Feb. 2018).

[40] Bogdan Korel and Janusz Laski. 1988. Dynamic program slicing. *Information processing letters* 29, 3 (1988), 155–163.

[41] Jens Krinke. 2003. Barrier Slicing and Chopping. In *Int'l Workshop Source Code Analysis and Manipulation (SCAM)*. IEEE, Amsterdam, Netherlands.

[42] Rahul Krishna, Md Shahriar Iqbal, Mohammad Ali Javidian, Baishakhi Ray, and Pooyan Jamshidi. 2020. CADET: A Systematic Method For Debugging Misconfigurations using Counterfactual Reasoning. arXiv:2010.06061 [cs.SE]

[43] T. D. LaToza and B. A. Myers. 2011. Visualizing call graphs. In *Symposium Visual Languages and Human-Centric Computing (VL/HCC)*. 117–124.

[44] Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. 2020. Statically Inferring Performance Properties of Software Configurations. In *Proc. European Conf. Computer Systems (EuroSys)* (Heraklion, Greece). ACM, New York, NY, USA, Article 10, 16 pages.

[45] Ding Li, Yingjun Lyu, Jiaping Gui, and William G.J. Halfond. 2016. Automated Energy Optimization of HTTP Requests for Mobile Applications. ACM, New York, NY, USA.

[46] Max Lillack, Christian Kästner, and Eric Bodden. 2018. Tracking Load-time Configuration Options. *IEEE Transactions on Software Engineering* 44, 12 (12 2018), 1269–1291.

[47] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 1013–1024.

[48] Jens Meinicke, Chu-Pan Wong, Christian Kästner, and Gunter Saake. 2018. Understanding differences among executions with variational traces. *arXiv preprint arXiv:1807.03837* (2018).

[49] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity: Measuring Interactions in Highly-configurable Systems. In *Proc. Int'l Conf. Automated Software Engineering (ASE)* (Singapore, Singapore). ACM, New York, NY, USA, 483–494.

[50] Jean Melo, Claus Brabrand, and Andrzej Wąsowski. 2016. How Does the Degree of Variability Affect Bug Finding?. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Austin, TX, USA). ACM, New York, NY, USA, 679–690.

[51] Jean Melo, Fabricio Batista Narcizo, Dan Witzner Hansen, Claus Brabrand, and Andrzej Wasowski. 2017. Variability through the Eyes of the Programmer. In *Proc. Int'l Conference Program Comprehension (ICPC)* (Buenos Aires, Argentina). IEEE, 34–44.

[52] Brad A. Myers, Andrew J. Ko, Thomas D. LaToza, and YoungSeok Yoon. 2016. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. 49, 7 (July 2016), 44–52.

[53] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. Using Bad Learners to Find Good Configurations. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 257–267.

[54] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. Conf. Programming Language Design and Implementation (PLDI)* (San Diego, CA, USA). ACM, New York, NY, USA, 89–100.

[55] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and Fixing Performance Problems That Have Non-intrusive Fixes. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Florence, Italy). IEEE, Piscataway, NJ, USA, 902–912.

[56] Adrian Nistor, Tian Jiang, and Lin Tan. 2013. Discovering, Reporting, and Fixing Performance Bugs. In *Proc. Int'l Conf. Mining Software Repositories* (San Francisco, CA, USA). IEEE, Piscataway, NJ, USA, 237–246.

[57] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In *Proc. Int'l Conf. Software Engineering (ICSE)* (San Francisco, CA, USA). IEEE, Piscataway, NJ, USA, 562–571.

[58] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding Near-optimal Configurations in Product Lines by Random Sampling. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)* (Paderborn, Germany). ACM, New York, NY, USA, 61–71.

[59] J. Park, M. Kim, B. Ray, and D. Bae. 2012. An empirical study of supplementary bug fixes. In *Proc. Int'l Conf. Mining Software Repositories* (Zurich, Switzerland). 40–49.

[60] Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)* (Toronto, Canada). ACM, New York, NY, USA, 199–209.

[61] Guillaume Pothier, Éric Tanter, and José Piquer. 2007. Scalable Omniscient Debugging. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (Montreal, Quebec, Canada). ACM, New York, NY, USA, 535–552.

[62] Johnny Saldaña. 2015. *The coding manual for qualitative researchers.* Sage.

[63] J. P. Sandoval Alcocer, F. Beck, and A. Bergel. 2019. Performance Evolution Matrix: Visualizing Performance Variations Along Software Versions. In *Conf. Software Visualization (VISSOFT)*. IEEE, 1–11. https://doi.org/10.1109/VISSOFT.2019.00009

[64] Stefan Schmidt. [n.d.]. Shall we Really do it Again? The Powerful Concept of Replication is Neglected in the Social Sciences. *Review of General Psychology* ([n. d.]).

[65] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence Models for Highly Configurable Systems. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)* (Bergamo, Italy). ACM, New York, NY, USA, 284–294.

[66] 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 *Proc. Int'l Conf. Software Engineering (ICSE)* (Zurich, Switzerland). IEEE, Piscataway, NJ, USA, 167–177.

[67] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012. SPLConqueror: Toward Optimization of Non-functional Properties in Software Product Lines. *Software Quality Journal* 20, 3-4 (Sept. 2012), 487–517.

[68] Linhai Song and Shan Lu. 2014. Statistical Debugging for Real-World Performance Problems. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (Portland, OR, USA). ACM, New York, NY, USA, 561–578.

[69] Linhai Song and Shan Lu. 2017. Performance Diagnosis for Inefficient Loops. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Buenos Aires, Argentina). IEEE, Piscataway, NJ, USA, 370–380.

[70] John Toman and Dan Grossman. 2016. Staccato: A Bug Finder for Dynamic Configuration Updates. In *Proc. European Conf. Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

[71] Jonas Trümper, Jürgen Döllner, and Alexandru Telea. 2013. Multiscale visual comparison of execution traces. In *Proc. Intl Conf. Program Comprehension (ICPC)* (San Francisco, CA, USA). 53–62.

[72] Miguel Velez, Pooyan Jamshidi, Florian Sattler, Norbert Siegmund, Sven Apel, and Christian Kästner. 2020. ConfigCrusher: Towards White-Box Performance Analysis for Configurable Systems. *Autom Softw Eng* (2020).

[73] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2021. White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Madrid, Spain). IEEE, 11 pages.

[74] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. 2018. Understanding and Auto-Adjusting Performance-Sensitive Configurations. In *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Williamsburg, VA, USA). ACM, New York, NY, USA, 154–168.

[75] Max Weber, Sven Apel, and Norbert Siegmund. 2021. White-Box Performance-Influence Models: A Profiling and Learning Approach. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Madrid, Spain). IEEE, 11 pages.

[76] Mark Weiser. 1981. Program Slicing. In *Proc. Int'l Conf. Software Engineering (ICSE)* (San Diego, CA, USA). IEEE, Piscataway, NJ, USA, 439–449.

[77] Claas Wilke, Sebastian Richly, Sebastian Götz, Christian Piechnick, and Uwe Amann. 2013. Energy Consumption and Efficiency in Mobile Applications: A User Feedback Study. In *Proc. Int'l Conf. Green Computing and Communications*. IEEE, 134–141.

[78] Chu-Pan Wong, Jens Meinicke, Lukas Lazarek, and Christian Kästner. 2018. Faster Variational Execution with Transparent Bytecode Transformation. *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* 2, Article 117 (Oct. 2018), 30 pages.

[79] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes* 30, 2 (2005), 1–36.

[80] 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 *Proc. Conf.Operating Systems Design and Implementation (OSDI)* (Savannah, GA, USA). USENIX Association, Berkeley, CA, USA, 619–634.

[81] 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 *Proc. Symp. Operating Systems Principles* (Farminton, PA, USA). ACM, New York, NY, USA, 244–259.

[82] Tingting Yu and Michael Pradel. 2016. SyncProf: Detecting, Localizing, and Optimizing Synchronization Bottlenecks. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)* (Saarbrücken, Germany). ACM, New York, NY, USA, 389–400.

[83] Tingting Yu and Michael Pradel. 2018. Pinpointing and Repairing Performance Bottlenecks in Concurrent Programs. *Empirical Softw. Eng.* 23, 5 (Oct. 2018), 3034–3071.

[84] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why? *SIGSOFT Softw. Eng. Notes* 24, 6 (Oct. 1999), 253–267.

[85] Andreas Zeller. 2009. *Why programs fail: a guide to systematic debugging*. Elsevier.

[86] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Salt Lake City, UT, USA). ACM, New York, NY, USA, 687–700.

[87] Sai Zhang and Michael D. Ernst. 2014. Which Configuration Option Should I Change?. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Hyderabad, India). ACM, New York, NY, USA, 152–163.

[88] Sai Zhang and Michael D. Ernst. 2015. Proactive Detection of Inadequate Diagnostic Messages for Software Configuration Errors. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)* (Baltimore, MD, USA). ACM, New York, NY, USA, 12–23.

[89] 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 *Proc. Symposium Cloud Computing (SoCC)* (Santa Clara, CA, USA). ACM, New York, NY, USA, 338–350.