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

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.

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 [8, 11, 26, 30, 32, 33, 55, 58, 59, 86]. This struggle is quite common when maintaining configurable software systems. Some empirical studies find that 59 percent of performance issues are related to configuration errors, 88 percent of these issues require fixing the code [26, 27], of which 61 percent take an average of 5 weeks to fix [41], and that 50 percent of patches in open-source cloud systems and 30 percent of questions in forums are related to configurations [75]. 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 [26, 29, 30, 32, 44, 68, 78].

When performance issues occur, developers typically use profilers to identify the locations of performance bottlenecks [10, 12, 13, 21, 84]. 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 of this system with 50 configuration options, which results in an unexpected 20x 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 as intended. To
determine the cause of the potentially 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 such 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 [15, 53, 74], developers could use more targeted profiling techniques [4, 10, 13, 83, 84], visualize performance behavior [2, 6, 12, 21, 62, 70], search for inefficient coding patterns [7, 46, 54, 56, 68], use information-flow analyses [43, 45, 48, 69, 79, 81, 88], or model the performance of the systems in terms of its options and interactions [24, 38, 64, 71, 72, 76]. Likewise, developers could use established program debugging techniques, such as delta debugging [85], program slicing [3, 39, 77], and statistical debugging [5, 67] 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 [59]. 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 [16, 51] 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 cause-effect chain, how influencing options are used in the implementation to directly and indirectly affect the performance of option hotspots. Subsequently, we design and implement information providers to support developers’ needs, adapting and tailoring global and local performance-influence models. CPU profiling, and program slicing, 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 global and local performance-influence models, CPU profiling, and program slicing, 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 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., 25, 29, 46, 56, 67]. 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 [8, 11, 27, 55]. For this task, in addition to using off-the-shelf profilers [15, 53, 74], some researchers suggest using more targeted profiling techniques [10, 12, 13, 21, 84] and visualizations [2, 6, 12, 21, 62, 70] to identify and analyze the locations of performance bottlenecks. Alternatively, some researchers suggest using techniques to search for inefficient coding patterns [7, 13, 46, 54, 56, 68]. 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 [85], program slicing [3, 39, 77], and statistical debugging [5, 67], 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 [9, 37, 42, 60]. For example, Whyline [37] 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 some also in terms of usability with 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 user-selected configuration is inconsistent or does not match the intended behavior [82, 87, 89]. By contrast, a software bug is a programming error by a developer that degrades a system’s behavior or functionality [5, 85, 86].

Research has repeatedly found that configuration-related performance issues are common and complex to fix in software systems [26, 27, 29, 41, 75]. For example, Han and Yu [26] found that 59 percent of performance issues concern configurations, 72 percent of which involved one option, and 88 percent require fixing the code. Similarly, Krishna et al. [41] found that 61 percent of performance issues take an average of 5 weeks to fix, and Wang et al. [75] found that 50 percent of patches in open-source cloud systems and 30 percent 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 [43, 45, 48, 69, 79, 81, 88]. For example, Lotrack [45] 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 [24, 38, 64, 71, 72, 76], as the models describe a system’s performance in terms of its options and their interactions. For example, the model $8 \cdot A \cdot B \cdot C + 5 \cdot C$ explains the influence of the options A, B, and C, and their interactions on the performance of a system; in this example, 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 [28, 71, 72, 76], 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 [36, 64–66, 71, 72] and optimizing performance [22, 52, 57, 90], they have not been evaluated in terms of usability; in particular, to support developers’ needs when debugging the performance of configurable software systems.

Contributions: We take a human-centered approach [16, 51] 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 [86], 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 [31] and a Wizard of Oz approach [14], 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, such as 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 [47, 49, 50, 59] 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, with 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 [1, 26, 32] (the system was spending a long time to transform and output a JPEG image). Participants were asked 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 [61]. The first author conducted the study and coded the sessions using open and descriptive coding, summarizing observations, discussions, and trends [73]. All authors met weekly to discuss the codes and observations. When codes were updated, previously analyzed sessions were reanalyzed to update the coding.

Threats to Validity and Reliability. 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 [26]. 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 recognizedly demonstrated 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 methods under 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.

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, nine 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.

| 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                           |
|                       |                                                                             |           | Inspect the option hotspots call stacks                                   | 12/19     | Program Slicing                         |
|                       |                                                                             |           | 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 influence the performance of option hotspots | 19/19     |                                         |
We aim to support developers in identifying influencing options, which can assist developers to debug the performance of configurations. Transactions to help developers identify the influencing options that cause an unexpected performance behavior. 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 [73], which can assist developers to debug the performance of configurable software systems. Table 1 shows which information needs are supported by the information sources global and local performance-influence models, CPU profiling, and program slicing that we adapt for designing information providers.

### 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 [73], which can assist developers to debug the performance of configurable software systems. Table 1 shows which information needs are supported by the information sources global and local performance-influence models, CPU profiling, and program slicing 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 [24, 38, 64, 71, 72, 76], which describe a system's performance in terms of its options and their interactions. For instance, the model 4.6 + 54.7 \( \cdot \) Duplicates \( \cdot \) Transactions + 8.9 \( \cdot \) Evict + 3.5 \( \cdot \) Temporary explains the influence of Duplicates, Transactions, Evict, and Temporary, and their interactions on the performance of Berkeley DB. 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 problemmatic 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. 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 (\( \downarrow \)) another configuration.

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 (\( \downarrow \)) another configuration.

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 [71, 72, 76].

*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. 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 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 [71, 72, 76]. 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 [71, 72, 76]. For instance, the local model of a method 0.9 + 42.9 \( \cdot \) Duplicates \( \cdot \) 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 another 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

---

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.
between configurations (e.g., a problematic and a non-problematic configuration) and (b) the influence of the changes on each method’s performance.

Local performance-influence models are usually built with white-box approaches as by-products of global models [71, 72, 76]. 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 duplicate 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 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 another 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.

CPU profiles can be collected with most off-the-shelf profilers. White-box approaches that build global and local performance-influence models collect these profiles [71, 72, 76]. 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 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.

help developers understand how the influencing options are used in the implementation to affect the option hotspots’ performance.

4.3.2 Program Slicing. We select program slicing [40, 77, 80, 86]; an approach to compute relevant fragments of a system based on a criterion. Several debugging tools have been implemented on top of program slicers [17, 37, 42, 80] to help developers narrow down relevant parts of a system where developers should focus their debugging efforts.

We adapt this information source to design another 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.

Example. Figure 6 shows a screenshot of our tool, which helps developers trace the cause-effect chain by showing a method-level
We evaluate the usefulness of our designed information providers 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 [34, 63]), 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 EVALUATING 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 Validating Usefulness of Information Providers

We implemented the information providers in a Visual Studio Code extension called GLIMPS [73]. Our prototype adapts global and local performance-influence models, CPU profiles, and a program slicer. 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 [72] to build the global and local performance-influence models and uses JProfiler [15] to collect the CPU profiles.

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 [19], as it is the most mature option. 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 [18]. For scalability and to reduce approximations, we modified JOANA to consider code coverage under the problematic and a non-problematic configurations.

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 [31] 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 asked to identify and explain which and how options caused the unexpected performance.
behavior. 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 were required 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 [61]. The first author conducted the study and 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 who debug the unexpected behavior without our information providers in 50 minutes, when they struggled to find relevant information.

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.”

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

Similar to our prior study, 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.

As in our prior study, we conducted a brief semi-structured interview, after the task, 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 asked to identify and explain which and how options caused the unexpected performance behavior. 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 for 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.

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 [61]. The first author conducted the study and 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.
Influencing Options | Option Hotspots | Cause-Effect Chain
---|---|---
Timeout | Did not find information | Did not find information
Treatment - Used GLIMPS | Did not find information | Did not find information
Control - Used simple plugin | Found information | Found information
System: Berkeley DB

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

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.2

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 global and local performance-influence models, CPU profiling, and program slicing, 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 Hertbyle, 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 (Award 2007202, 2038080, and 2107463), 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 80NSSC20K6660 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] Iago Abal, Jean Melo, Ştefan Stănciulescu, Marco Ribiero, and Andrej 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.
[2] 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.
[3] Hiralal Agrawal and Joseph R. Horgan. 1996. Dynamic Program Slicing. ACM SIGPLAN Notices 26, 6 (1996), 246–256.
[4] Mohammad Mejbah ul Alam, Tongping Liu, Guangming Zeng, and Abdallah Mzaliid. 2017. SynePerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs. In Proc. European Conference on Computer Systems (EuroSys) (Belgrade, Serbia). ACM, New York, NY, USA, 298–313.
[5] David Andrzejewski, Anne Mulhern, Ben Lliblit, and Xiaoju Zhe. 2007. Statistical Debugging Using Latent Topic Models. In Proc. European Conf. Machine Learning (Warsaw, Poland). Springer-Verlag, Berlin, Heidelberg, 6–17.
[6] Cor-Paul Bezemer, J.A. Pousweze, and Brendan Gregg. 2015. Understanding Software Performance Regressions Using Differential Flame Graphs. In Int’l Conf. Software Analysis, Evolution, and Reengineering (SANER) (Montreal, Canada). IEEE, Los Alamitos, CA, USA, 535–539.
[7] James Bornholt and Emına Törlok. 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.
[8] Silvia Breu, Rahul Premraj, Jonathan Silitos, 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.
[9] 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, 673–684.
[10] 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. Arch. Code Optim. (TACO) 12, 1, Article 6 (April 2015), 24 pages.
[11] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andriam 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.
On Debugging the Performance of Configurable Software Systems: Developer Needs and Tailored Tool Support  
ICSE ’22, May 21–29, 2022, Pittsburgh, PA, USA