Detecting Performance-Relevant Changes in Configurable Software Systems

SEBASTIAN BÖHM, Saarland University, Germany
FLORIAN SATTLER, Saarland University, Germany
NORBERT SIEGMUND, Leipzig University, Germany
SVEN APEL, Saarland University, Germany

Performance is a volatile property of a software system and, as the system evolves, its performance behavior may change, too. As a consequence, frequent performance profiling is required to keep the knowledge about a software system's performance behavior up to date. Repeating all performance measurements after every revision is a cost-intensive task, though. This is even more challenging in the presence of configurability (e.g., by user-selectable software features), where one has to measure multiple configurations of a given software system to obtain a comprehensive picture. Configuration sampling is a common approach to control the measurement cost. However, sampling cannot guarantee completeness and might miss performance regressions, especially if they only affect a small portion of the configuration space.

As an alternative to solve the cost reduction problem, we present ConfFLARE: ConfFLARE estimates whether a change potentially impacts performance by identifying data-flow interactions with performance-relevant code. It then extracts the configurable software features that participate in such interactions, as these features are highly likely to influence how the performance-relevant code is executed. Based on the identified features, we can select a subset of relevant configurations to focus performance profiling efforts on.

To evaluate the effectiveness of ConfFLARE in a performance regression setting, we conduct a study on both synthetic and real-world software systems. In particular, we have address three questions: How effective is ConfFLARE in detecting performance-relevant changes (RQ_1) and in identifying relevant features for performance regression analysis (RQ_2)? And how much performance measurement effort can we potentially save with this information (RQ_3)? We find that ConfFLARE correctly detects performance regressions in almost all cases and identifies the relevant features in all but two cases. With this information, we are able to reduce the number of configurations to be tested on average by 79% for synthetic and by 70% for real-world regression scenarios saving hours of performance testing time.

1 Introduction

The performance behavior of a software system is crucial for its success. The challenge is that it is a volatile property that changes frequently as the system in question evolves. Software evolution can easily lead to performance regressions that negatively affect user experience or operating costs [24].

Performance regression testing aims at revealing performance issues early such that they can be fixed before deployment [18]. Since running performance tests for every revision is often a cost-intensive task, it is desirable to estimate whether and how incoming changes affect the system's performance behavior. Such an estimation prior to actual testing becomes even more important in the presence of configurability: If a software system is configurable—and most systems are configurable—it is not sufficient to test only multiple revisions, but also multiple configurations need to be tested as well to obtain a complete picture. This causes an exponential blowup of the number of performance measurement runs. As a result, continuous performance testing on every revision is infeasible for many configurable software systems such that only a sample of all configurations are tested instead [41]. The problem is that the success of configuration sampling relies on the chance that regressing configurations are contained in the selected sample. As a

Authors' Contact Information: Sebastian Böhm, boehmseb@cs.uni-saarland.de, Saarland University, Saarbrücken, Germany; Florian Sattler, sattlerf@cs.uni-saarland.de, Saarland University, Saarbrücken, Germany; Norbert Siegmund, norbert.siegmund@uni-leipzig.de, Leipzig University, Leipzig, Germany; Sven Apel, apel@cs.uni-saarland.de, Saarland University, Saarbrücken, Germany.

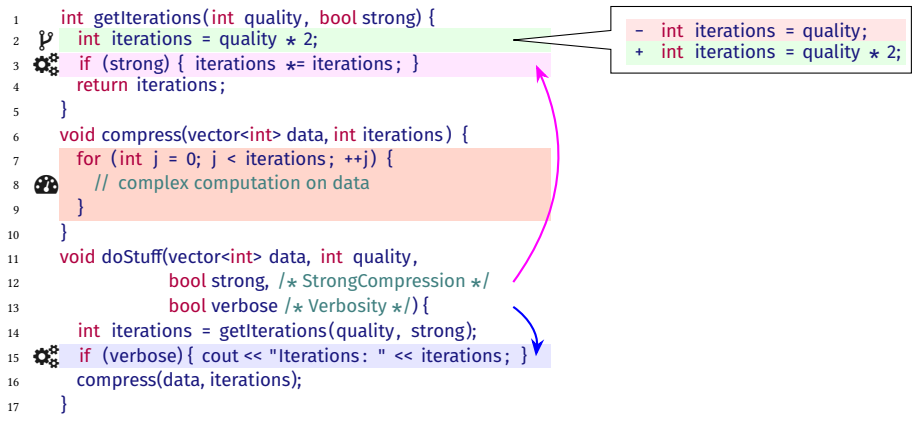


Fig. 1. Example of a performance-relevant code change. Function compress contains performance-relevant code (). The callout points to a change that was made to Line 2 (). This change has an influence on the performance behavior of region since it changes how the loop counter is calculated. The example contains two features () named *StrongCompression* and *Verbosity*, controlled by the parameters strong and verbosity, respectively. ConfFLARE reveals that only *StrongCompression* affects the example's performance by influencing the value of iterations.

consequence, regressions can easily be missed, especially if they affect only a small fraction of the overall configuration space [41]. Many sampling strategies have been devised that offer different trade-offs between sample size and coverage [25, 37]. These strategies typically only consider properties of the configuration space, that is, which configurable software features can be combined in which way, but they do not take advantage of the regression setting because they lack knowledge about which configurations are affected by a change.

Studying performance bugs in highly configurable software systems, Han and Yu [23] suggest to use static analysis to determine which configurations are impacted by performance-relevant code changes so that performance testing efforts can be focused on impacted configurations. In this work, we present ConfFLARE, an approach that achieves exactly that: In a first step, we employ a data-flow analysis to determine whether a change to the code base of a configurable software system interacts with performance-relevant code in the system. Only if such an interaction exists, we consider the change for performance analysis. In that case, we localize in a second step the part of the configuration space that is potentially affected by that change so that fewer configurations need to be analyzed. As with sampling, ConfFLARE is a heuristic approach that trades completeness for scalability (a change may affect performance-relevant code even without data flow). However, it is a heuristic that is informed by the program's semantics and, thus, can more precisely select relevant configurations than traditional configuration sampling strategies.

For illustration, consider a typical example of a code change that introduces a performance regression in Figure 1, which we use throughout the paper to explain how ConfFLARE works. Here, we briefly outline our idea on solving this problem. First, it is important to note that we focus here on performance regressions that are caused due to a change in a program's execution. We do not consider performance regressions that are caused by other external factors or side effects (e.g., I/O, cache effects, etc.). The example models a simple configurable compression tool that undergoes a change. It consists of a function doStuff that takes a vector of data elements as well as some additional parameters and compresses the data vector in two steps: First, getIterations

computes the number of iterations that is later used by the compression algorithm. Second, the data get compressed by function compress. This function mainly consists of a loop that is executed for the previously calculated number of iterations. As the program evolves, a new revision changes how variable iterations is computed (🔀). To judge whether this could lead to a performance regression, we need to decide whether the changes of the new revision influence performance-relevant code, that is code that is central to the program's performance behavior. ConfFLARE does so by determining whether a change interacts with performance-relevant code via a flow of data from the changed code to performance-relevant code. We call such data-flow interactions *performance-relevant interactions*. If we do not detect such interactions, we conclude (heuristically) that the change is likely not performance-relevant and, thus, no performance tests are run. By contrast, if ConfFLARE does find a performance-relevant change, it localizes the part of the configuration space that is potentially affected by that change. It does so by determining which configurable software features are involved in performance-relevant interactions. The idea is that only features that affect data flowing from changed code to performance-relevant code can also affect its performance behavior. Given that a configuration can be characterized by which features are enabled or disabled, configurations involving such features are of special interest for performance testing. In the example, the configuration space is made up by two features (⚙) *StrongCompression* and *Verbosity*. Only *StrongCompression* affects performance-relevant code by influencing the value of iterations. Feature *Verbosity* only reads data and, thus, has no influence on performance. As a result, we only test one configuration for each possible value of *StrongCompression*, reducing the number of tests to be executed (from 4 to 2).

To evaluate the effectiveness of ConfFLARE in a performance regression scenario, we have conducted an empirical study using both synthetic and real-world regression scenarios. In particular, we have addressed three questions: How effective is ConfFLARE in detecting performance-relevant changes (RQ_1) and in identifying relevant features for performance regression analysis (RQ_2)? How much performance measurement effort can we potentially save with this information (RQ_3)? We found that ConfFLARE correctly detects performance regressions in almost all cases and identifies the relevant features in all but two cases. With this information, we are able to reduce the number of configurations to be tested on average by 79% for synthetic and by 70% for real-world regression scenarios. In most cases, this leads to speedups of 2 to 32 in time spent for performance measurements compared to measuring all configurations, sometimes saving hours of performance testing time.

In summary, we make the following contributions:

- We present ConfFLARE, an approach that determines whether a change to a configurable software system interacts with performance-relevant code and—if so—computes which part of the configuration-space is affected.
- We provide an implementation of our approach in the VaRA framework [46].
- We evaluate ConfFLARE using synthetic and real-world software systems investigating its applicability and effectiveness.

All results and a replication package are available online [1].

2 Background

Before we present ConfFLARE in detail in Section 3, we introduce fundamental concepts from previous research.

[1] https://anonymous.4open.science/r/ConfFLARE-Supplements

2.1 Code Regions

Code regions are a uniform abstraction to facilitate the common analysis of different kinds of variability [45]. The idea of code regions is to attach high-level domain-specific information to individual program elements, so that results provided by a low-level analysis can later be contextualized in a high-level, domain-specific manner. For example, we can attach to a piece of code the information by which commit it has been introduced. A program analysis can then compute the data flows between these code regions to identify commits that interact with each other [45, 46].

Notably, the notion of code regions allows us to represent the three different concepts relevant to ConfFLARE—change, performance-relevant code, and features—in a uniform way. As a result, we can employ existing code-region-based analyses, as we illustrate in Section 2.2. Specifically, we use two existing instances of code regions: (1) a *commit region* comprises code introduced by a commit and (2) a *feature region* denotes code that belongs to a certain feature. Such a feature region is usually guarded by configuration switches (e.g., feature flags or toggles). Additionally, we introduce a new kind of code region to model performance-relevant code.

In what follows, we give a formal introduction of code regions, as defined by Sattler [45], and introduce commit and feature regions as relevant instances. We model a program p as a set of functions $f_1, \ldots, f_n \in p$, where each function is represented as a control-flow graph (CFG). We use an instruction-based representation of functions instead of a basic-block-based representation because it allows for a more fine-grained definition of code regions. Each function f is represented as a triple $\big(entry_f, exit_f, insts(f)\big)$, where $insts(f)$ is the set of instructions of f and $entry_f, exit_f \in insts(f)$ are the function's unique entry and exit instructions. The ordering of instructions inside a function is defined by two functions $pred$ and $next$. In our implementation (see Section 3), we use LLVM-IR as a program representation, such that $pred$ and $next$ are specified by the semantics of LLVM-IR.

Since we investigate the evolution of programs, we need to refer to a program revision at a certain point in time. We use a revision specifier p^v to refer to a program p at revision v. Note that we omit the revision specifier if it is irrelevant in a given context.

To extract code regions from this representation, a tagging function $tags : I \rightarrow \mathcal{P}(T)$ associates each instruction in the program's set of instructions I with a set of tags from T. The tags represent the higher-level concepts (e.g., commit identifiers) that we want to attach to an instruction. A code region cr is then a connected subgraph of some function $f \in p$ such that all instructions in the code region are tagged with the same tags. Here, *connected* means that, for any instruction i that belongs to cr, there exists another instruction $i' \neq i$ in cr such that $i' \in next(i) \vee i' \in prev(i)$. In other words, a code region is a *contiguous* set of instructions of a function that belong to the same high-level domain-specific concept labelled by a tag. The set CR contains all code regions. Given a code region $cr \in CR$, we can retrieve the tags associated with that code region using function $regionTags(cr)$ and the set of instructions using $insts(cr)$. For brevity, we use the notation $i \in cr$ instead of $i \in insts(cr)$.

Commit Regions. Commit regions comprise instructions that have been introduced by a certain revision in a version control system Sattler et al. [46]. The tagging function for commit regions annotates each instruction with the commit that last modified the source code that the instruction is generated from (e.g., using git-blame[2]). We denote the set of all commit regions in a program with CR_C. We use commit regions to determine which code belongs to a (potentially performance-relevant) change.

[2]Git-blame is a version control system (VCS) mechanism that annotates each line in a file with the commit that last modified it.

Feature Regions. To model which instructions belong to a certain feature, we use feature regions [45]. We denote the set of all feature regions with CR_F. The tagging function for feature regions associates each instruction with the name(s) of the feature(s) that specify(ies) whether this instruction is executed. That is, an instruction i is tagged as belonging to feature f if whether i is executed or not depends on the selection or deselection of f. There are different approaches on how to obtain this information depending on the mechanism that is used for implementing features. For example, there is a variety of approaches to handle configurability expressed with the C preprocessor [19, 27, 31, 32, 51] and C++ templates [20]. For load-time configuration options, a static taint analysis can be used to track feature choices that control decisions in the program [33, 51, 52]. We focus on load-time configuration options since this is a very common and practically relevant means to express configurability [6].

2.2 Region-Based Interaction Analysis

As outlined in Section 1, we aim at detecting performance-relevant changes by identifying data-flow interactions between changes and performance-relevant code. That is, as we model code changes and performance-relevant code as code regions, ConfFLARE requires an analysis that is able to identify data-flow interactions between different code regions. For this purpose, we use SEAL [46], which uses a state-of-the-art data-flow analysis to compute data-flow interactions between all instructions in a program. Based on the commit tags attached to the instructions, it infers code from which commits interacts with each other via data flow.

This kind of interaction analysis can be generalized to work with arbitrary code regions. The idea is that analyses should be decoupled from the domain-specific information associated with concrete code regions, this way, making the analysis reusable for different analysis scenarios.

An interaction between two code regions is based on a given relation $\blacklozenge \subseteq CR \times CR$. A code region $r_1 \in CR$ interacts with another code region $r_2 \in CR$ if and only if $r_1 \neq r_2$ and $r_1 \blacklozenge r_2$. The abstract relation $\blacklozenge$ links region information with the analysis providing a domain-specific view on the general analysis semantics. To apply this interaction definition to data-flow interactions, we instantiate the abstract relation $\blacklozenge$ with a concrete data-flow relation $\rightsquigarrow$. That is, $r_1 \rightsquigarrow r_2$ iff data flows from some instruction in r_1 to some instruction in r_2.

Since the concept of region interaction analysis is designed to be independent of the concrete kinds of code regions and only requires appropriate tagging functions, we can use the existing analysis from SEAL with some adaptions to trace data-flow interactions between a change and performance-relevant code. In the remaining section, we explain our instantiation of this analysis in more detail.

2.3 Dataflow Analysis and Exploded Supergraph

The analysis described in Section 2.2 uses an existing taint analysis based on *Interprocedural Distributive Environments* (IDE) [44], an algorithmic framework for implementing data-flow analyses. For more details on the concrete analysis implementation, we refer the reader to Sattler et al. [46]. One important aspect of IDE is the main data structure used to solve data-flow problems—the *exploded super-graph (ESG)*. Since we use the ESG later on to find features related to a performance-relevant change, we provide a brief introduction on how an ESG is constructed.

An ESG is a graph representation of a program that is used to efficiently solve data-flow problems such as our interaction analysis [44]. The nodes in an ESG can be written as tuples (i, d), where i is an instruction and d is a data-flow fact. In our case, d can be a program variable or an instruction that uses another variable. We consider instructions as data-flow facts to correctly model the static

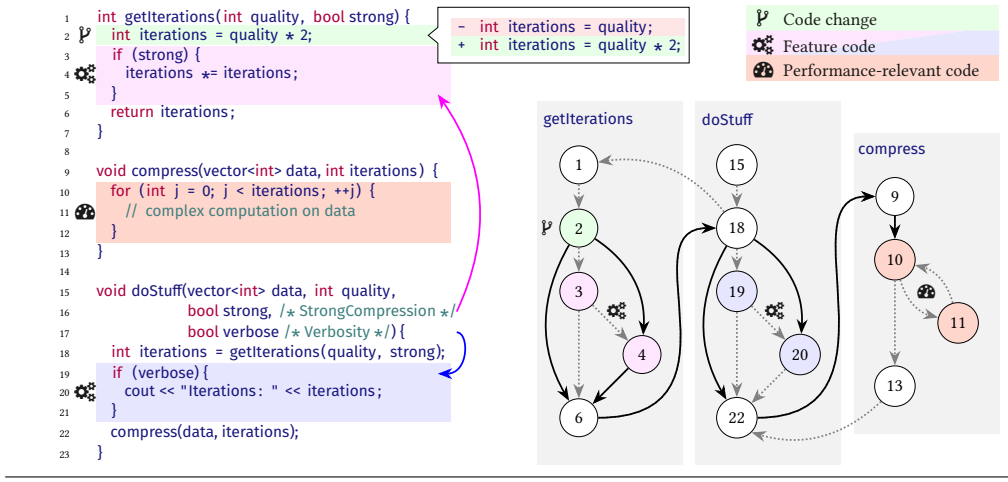


Fig. 2. Continuation of our running example from Figure 1. The code on the left is identical to Figure 1. On the right is the corresponding control-flow graph with line numbers as nodes and dashed arrows showing control-flow. The thick black arrows denote data flow between the change and performance-relevant code passing through *StrongCompression* and *Verbosity*.

single assignment form of LLVM-IR.[3] Figure 3 shows a simplified ESG of the program from our example in Figure 1 next to the corresponding CFGs. For each CFG node (each represents roughly one instruction), we show the most important ESG nodes next to it. Each column stands for a different data-flow fact. The edges between ESG nodes visualize how data-flow facts propagate throughout the program. For example, the edge from $(1, \text{quality})$ to $(2, \text{iterations})$ reflects how quality is used to calculate iterations in Line 2.

3 Introducing ConfFLARE

In this section, we introduce and explain ConfFLARE, our approach to detect performance-relevant changes and the corresponding affected part of the configuration-space. First, we give an overview of ConfFLARE using our running example. Then, we define what we consider performance-relevant code and how we model it as a specific type of code region. Finally, we describe how ConfFLARE computes performance-relevant interactions and how we pin down features that participate in a performance-relevant interaction.

3.1 Overview

To motivate and explain the idea behind ConfFLARE, we continue our running example in Figure 2.

Performance-Relevant Interactions. Let's assume that the loop in function compress dominates the performance behavior of this system (). We call this *performance-relevant code* because any change to its performance behavior affects the whole program's performance. Consequently, if changes from a revision *interact* with such code, then this revision potentially affects performance. There are many possibilities of how such an interaction could look like. For example, an interaction could be structural (e.g., the revision directly changes performance-relevant code). Or an interaction

[3]In *static single assignment form*, every variable is assigned exactly once. To fulfill this constraint, LLVM-IR generates temporary variables that are represented by the instructions they stem from. Note that these temporary variables are also data-flow facts.

could arise from a function call (e.g., performance-relevant code calls code that has been changed in a revision). These are rather simple examples that are easy to pin down. We focus on a more complex kind of interaction, *data-flow interactions*, that occur if data flows from changed code to performance-relevant code. Data-flow interactions are interesting for two reasons: (1) they can be indirect, meaning that there is no obvious relationship between changed code and performance-relevant code [46] and (2) they are hard to detect with state-of-the art approaches, because this requires a sophisticated data-flow analysis in combination with domain-specific information about program revisions as well as performance-relevant code, which is exactly what we provide with ConfFLARE.

The change in our example interacts with performance-relevant code via data flow by modifying how variable iterations is calculated. This variable controls the number of iterations the loop in the performance-relevant code region executes. In Figure 2, the data-flow paths causing this interaction are denoted by thick black arrows. The existence of this data-flow interaction between the change and the performance-relevant code indicates that the performance behavior of the system may be affected by that change. We call such a data-flow interaction a *performance-relevant interaction* and the involved change a *performance-relevant change*. We use this notion of a performance-relevant interaction to judge whether a given change potentially influences the performance behavior of a system. In our example, the change causes the loop in compress to be executed twice as much as previously. So it can be expected that it takes about twice as long to run the program (assuming that the time spent in the rest of the code is negligible).

Restricting the Configuration Space. A system's performance behavior often depends on its configuration [26]. Therefore, the entire configuration space must be measured to obtain a comprehensive picture of the performance behavior. However, this can be very expensive or even intractable for practical systems that provide thousands of configuration options [6]. Sampling is a common approach to control this cost [25, 37]. The problem is that, if a performance regression only affects a small portion of the configuration space, it might be easily missed by sampling [26, 41]. If we are able to identify which part of the configuration space is affected by a performance-relevant change, we can focus measurement efforts only on the affected part, reducing overall measurement costs.

ConfFLARE can identify which part of the configuration space is affected by a performance-relevant change by computing which features are *related* to a performance-relevant change. We deem a feature related if its code participates in data flow from a performance-relevant change to a performance-relevant code region. Note that ConfFLARE does not tell us whether a performance effect occurs when selecting or de-selecting the feature, or for both conditions. Computing the exact conditions (selections and de-selections of features) under which an effect occurs requires more expensive techniques such as symbolic execution. Consequently, ConfFLARE cannot give any guarantees regarding completeness—similar to sampling. However, by reducing the configuration space to a part that we know is related to a performance-relevant interaction, ConfFLARE can be used to increases the chance that a performance regression is detected, for example by guiding a sampling procedure to focus on this part of the configuration space.

In our example, if we follow the data-flow paths of the interaction, they pass through a node that belongs to the feature *StrongCompression* (i.e., Line 4). We can also see data flowing from the change to a node related to the feature *Verbosity* (Line 20). However, these data are not passed further on to performance-relevant code. Therefore, we conclude that feature *StrongCompression* is potentially affected by the performance-relevant change, but *Verbosity* is not.

3.2 Performance-Relevant Code

There are various reasons for why the performance behavior of a certain piece of code might be important to a stakeholder. For example, one might want to ensure that the implementation of a computation-heavy algorithm does not regress over time. In security applications, it is often desirable that certain algorithms take a fixed amount of time to mitigate timing attacks [5]. In safety-critical systems, parts of the code may have to fulfill real-time requirements that could be violated by a change [38]. In other words, what constitutes performance-relevant code is application-specific such that the user has to define what code they want to consider. So, we do not restrict ConfFLARE to only one kind of performance-relevant code. Naturally, one might want to automate this declaration for common scenarios. So, for our evaluation, we chose hot code—code that contributes disproportionately to the overall runtime compared to its size—as performance-relevant code. This is reasonable in a performance regression setting in which we want to detect performance change-points that arise from changes introduced by new program revisions [40].

In ConfFLARE, we model performance-relevant code with its own type of code region:

Definition 1. A *performance region* $cr_P \in CR_P$ is a code region that contains only instructions belonging to performance-relevant code. The tagging function for performance regions tags all instructions that belong to the class of performance-relevant code we are interested in, no additional data are attached:

$$
\begin{array}{lcll}
tags_P & : & I \rightarrow \mathcal{P}(\mathbb{B}) & \\
tags_P(i) & = & \{true\} & \text{if } i \text{ is performance-relevant code} \\
 & & \emptyset & \text{Otherwise}
\end{array}
$$

3.3 Performance-Relevant Interactions

We identify potentially performance-relevant interactions by computing how data flow between commit regions associated with changed code and performance regions. For this purpose, ConfFLARE builds on the region-based interaction analysis framework presented in Section 2. This way, we are able to connect the high-level performance regression scenario with the low-level concept of data flow via the concept of code regions. The changes made to the program by new revisions map to commit regions, whereas performance-relevant code is represented by performance regions.

To determine which commit regions belong to a certain change, we introduce a projection of the set of all commit regions CR_C to include only code regions tagged with certain revisions.

Definition 2. Given a set of revisions V, the set $CR_C^V \subseteq CR_C$ contains exactly the commit regions that are tagged with a revision from V:

$$CR_C^V = \left\{ cr \;\middle|\; cr \in CR_C \;\wedge\; regionTags(cr) \cap V \neq \emptyset \right\}$$

For a regression scenario in which every new program revision v is analyzed (e.g., in a CI/CD pipeline), one would choose $V = \{v\}$.

ConfFLARE allows us to use the region-based interaction analysis presented in Section 2.2 with multiple different kinds of code regions by slightly adapting the interaction relation $\rightsquigarrow$. Our version of $\rightsquigarrow$ restricts the relation such that it only captures data flows from the commit regions in question to performance regions:

Definition 3. Given a set of revisions V, the relation $\rightsquigarrow_{\text{🐞}} \subseteq CR_C^V \times CR_P$ represents data-flow interactions between commit regions that are tagged with revisions from V and performance regions.

This relation captures only interactions between changes from the desired revisions and performance-relevant code. Based on this, we can define set of performance-relevant interactions.

Definition 4. A *performance-relevant interaction* is a tuple consisting of a performance region $cr_P \in CR_P$ and a set of commit regions that interact with cr_p. PI denotes the set of all performance-relevant interactions.

$$PI = \left\{ \left(cr_P, interactCR(cr_P)\right) \;\middle|\; cr_P \in CR_P \;\wedge\; \left|interactCR(cr_P)\right| > 0 \right\}$$

where

$$interactCR(cr_P) = \left\{ cr_C \;\middle|\; cr_C \in CR_C^V \;\wedge\; cr_C \rightsquigarrow_{\text{🚗}} cr_P \right\}$$

A performance-relevant interaction defines which exact parts of a change interact with a certain performance region via data flow. With this representation, it is easy to see which changes are performance relevant (i.e., interact with performance-relevant code), and it provides the necessary information on which features are related to the interaction, which we describe next.

3.4 Extracting Feature Information

Based on the performance-relevant interactions detected in the previous step, we identify a subset of features of a software system that are related to a performance-relevant interaction. These features are potentially affected by a performance-relevant change and, thus, should be considered when selecting configurations during in-depth performance analysis. With this goal in mind, we define the criteria that a feature must fulfill to be considered related to a performance-relevant interaction.

Definition 5. A feature $f \in F$ is *related to a performance-relevant change* if, at least, one instruction i of a feature region cr_f associated with f fulfills the following criteria:

Let $cr_C \in CR_C^V,\ cr_P \in CR_P,\ cr_C \rightsquigarrow_{\text{🚗}} cr_P$.

(1) i is part of a program path π that gives rise to a performance-relevant interaction:

$$i \in \pi, \quad \pi = i_1, i_2, \cdots, i_n, \quad i_1 \in cr_C,\ i_n \in cr_P$$

(2) i uses data that flow from the change to the performance-relevant code region:

$$\exists\, i_{\text{🚗}}.\ i_{\text{🚗}} \rightsquigarrow i \;\wedge\; i_{\text{🚗}} \in cr_C$$

We denote the set of all features that fall under this definition with $\hat{F}$. To identify all features $f \in \hat{F}$, we extract the feature names from all feature regions that contain, at least, one instruction that fulfills both criteria.

Checking criterion (1), we perform a backwards search in the ESG constructed by the interaction analysis (see. Section 2.3). We formalize this procedure in Algorithm 1. To collect all instructions that are part of a program path that gives rise to a performance-relevant interaction, we start the traversal at nodes (i, i) belonging to instructions $i \in cr_P$ (Line 2). These nodes represent performance-relevant code that is affected by a change. In our example, (10c, 10c) is the only such node. For each start node, `reconstructPath` progresses backwards along the ESG's edges until it reaches an edge that leads out of a commit region that is part of the performance-relevant change under investigation (Lines 3, 16). In our example, this happens when we reach the node (2, iterations). In addition, we prune all sub-paths where the search terminates without ever encountering a node that is part of such a commit region (Line 11). This process, which is an instantiation of program slicing [57], collects the instructions that fulfill criterion (1).

Algorithm 1: Reconstruct data-flow paths for a performance-relevant interaction.

```
1  fun reconstructPaths(cr_P)
2      startNodes := {(i, i) | i ∈ cr_P}
3      endNodes := {(i, d) | ∃ cr_C. cr_C ∈ interactCR(cr_P) ∧ i ∈ cr_C}
4      insts := ∅
5      for node ∈ startNodes do
6          result := reconstructPath(node, endNodes)
7          if result = (Just path) then
8              insts := insts ∪ path
9      return insts
10 fun reconstructPath(node, endNodes)
11     if ¬(node ∈ endNodes) ∧ pred(node) = ∅ then
12         return Nothing
13     prunePath := true
14     insts := {node}
15     for pred ∈ pred(node) do
16         if node ∈ endNodes ∧ ¬pred ∈ endNodes then
17             continue
18         result := reconstructPath(pred, endNodes)
19         if result = (Just predInsts) then
20             prunePath := false
21             insts := insts ∪ predInsts
22     if prunePath then
23         return Nothing
24     return Just insts
```

Criterion (2) further restricts the set of instructions obtained in the previous step to include only those instructions that *use* data that originate from the change and eventually arrive at the performance-relevant code region. Note that, without any additional filter, the result from Algorithm 1 still includes instructions that load data, but neither modify nor use them for control-flow decisions. These instructions might bring additional features into consideration. This can be useful in certain scenarios, for example, in security applications where side-channel attacks are a concern. However, for our use case of performance regression analysis, such read-only instructions are not expected to play a considerable role in a program's performance behavior. So, we opt for keeping the configuration space as small as possible by excluding them.

We can observe the effects of applying the additional filters in our example: By applying criterion (1), we receive the following set of instructions:

$$\{2, 3, 4, 6, 9, 10\text{d}, 10\text{c}, 10\text{i}, 11, 18\text{r}, 19, 20, 22\text{c}\}$$

From this set, only instructions 4 and 20 are associated with a feature, and they both interact with instruction 2, meaning that we would report the features *StrongCompression* and *Verbosity* as related to the performance-relevant interaction. By excluding the read-only interaction of instruction 20, we can omit the feature *Verbosity*, since it has no influence on the value that is propagated to the performance-relevant region. With this extra filter, ConfFLARE can be fine-tuned with regard to how aggressively it should restrict the configuration space depending on the use case.

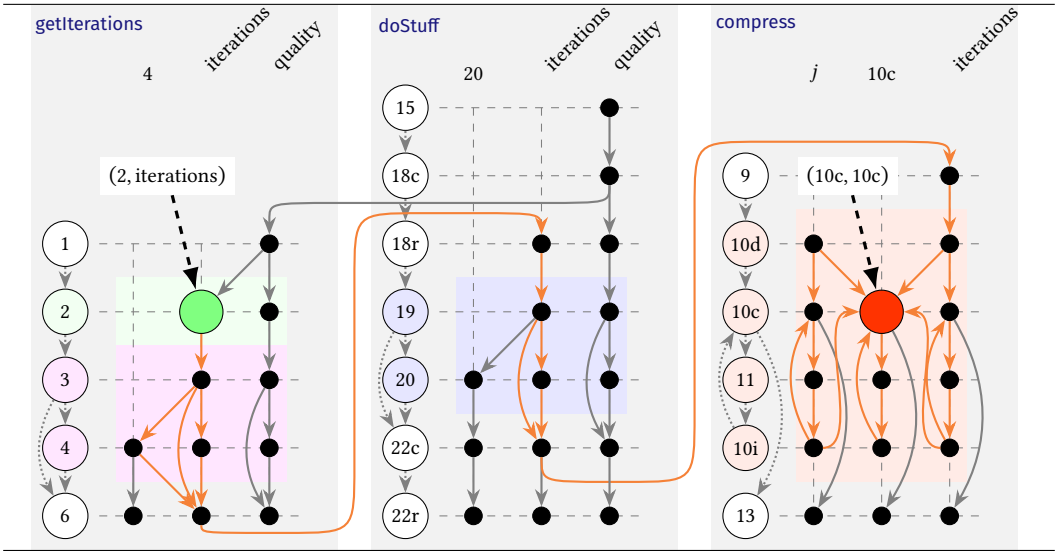


Fig. 3. Simplified ESG for our running example. For each function, we show its CFG with line numbers as nodes on the left and a simplified version of its ESG on the right that only contains nodes relevant for this example. Function calls are split into separate call and return nodes (suffixes *c* and *r*). The loop header in Line 10 is split into three nodes: variable declaration (10d), loop condition (10c), and increment (10i). The highlighted arrows show the result of the path reconstruction for the performance-relevant interaction between the change in Line 2 and the performance-relevant code region in Lines 10 and 11.

Finally, we can use $\hat{F}$ to select a subset $\hat{C}$ from the set of configurations of a software system C. One possible way of creating such a subset is to cover all possible combinations of the features in $\hat{F}$. This is a very conservative approach to creating $\hat{C}$ that considers many configurations, but serves as a good reference point for discussing how much the information provided by ConfFLARE helps reduce performance testing costs.

Note that, in practice, other means of creating $\hat{C}$ can be chosen. For example, $\hat{F}$ could be used to guide a sampling strategy to explore a certain part of the configuration space.

4 Empirical Study

To evaluate the applicability and capabilities of ConfFLARE in a performance regression setting, we conduct a study using both synthetic and real-world software systems. In particular, we investigate how effective our approach is in detecting performance-relevant changes and in identifying relevant features for performance regression analysis and how much this information helps reduce the cost of such analysis.

4.1 Research Questions

Our goal with ConfFLARE is to reduce the costs of performance regression analysis of configurable software systems by identifying changes and configurations that potentially cause performance regressions. To evaluate to which extent we achieve this goal, we address three research questions. First, we look into whether our approach selects appropriate changes for performance testing:

> ***RQ$_1$:*** *How well does ConfFLARE identify changes that incur a performance regressions?*

Here, we focus on the first step of ConfFLARE, which identifies whether a change is potentially performance-relevant. To be effective, it should find performance-relevant interactions for all changes that cause a performance regression.

Second, we investigate whether ConfFLARE provides useful information for selecting appropriate configurations for performance testing:

> ***RQ$_2$:*** *How accurately does ConfFLARE identify features related to performance regressions?*

This question focuses on the second step of ConfFLARE, which identifies features related to a performance-relevant change. Correctly identifying such features is important as any missed feature might cause configurations that are necessary to detect a performance regression to be omitted from performance tests. To check whether all relevant features were indeed identified, we select a subset of configurations based on the relevant features and check whether we can still detect the performance regression using only this set of configurations.

Finally, we take a look at the outcome of ConfFLARE in terms of cost savings for performance regression analysis:

> ***RQ$_3$:*** *How much performance measurement effort can be potentially saved with the information provided by ConfFLARE?*

Our overall goal is to reduce the costs of performance regression analysis of configurable software systems. This cost can be measured in terms of both, the number of analyzed configurations and time spent for performance measurements. ConfFLARE helps reduce this cost in two ways—classifying changes as not performance-relevant and identifying relevant features for performance testing. With this research question, we study how large these savings are.

4.2 Subject Systems

Our evaluation rests on synthetic, seeded, and real-world regression scenarios. Synthetic scenarios allow us to control the setting and confounding factors, explore edge cases, and provide a qualitative view on the results, increasing internal validity. Real-world scenarios allow us to evaluate ConfFLARE in a more realistic setting, increasing both ecological and external validity. As a middle ground between synthetic and real-world, we also consider seeded regression scenarios based on eight different GNU Coreutils tools. Table 1 lists the subject systems we use in our evaluation along with the number of analyzed regression scenarios and configurations. A *regression scenario* is a pair of revisions $(v_{\text{old}}, v_{\text{new}})$ of a system. Revision v_{old} refers to the state of the system before a change and v_{new} refers to the state after a change. For the synthetic and seeded regression scenarios, we obtain v_{new} by applying a hand-crafted patch to a revision v_{old}. For the real-world scenarios, we select pairs of revisions from the system's revision history. For our methodology, we aim at achieving high internal validity (instead of external validity) [35]: We selected the subject systems to provide a proof of concept (internal validity), rather than to make comprehensive statements on generalizability (external validity).

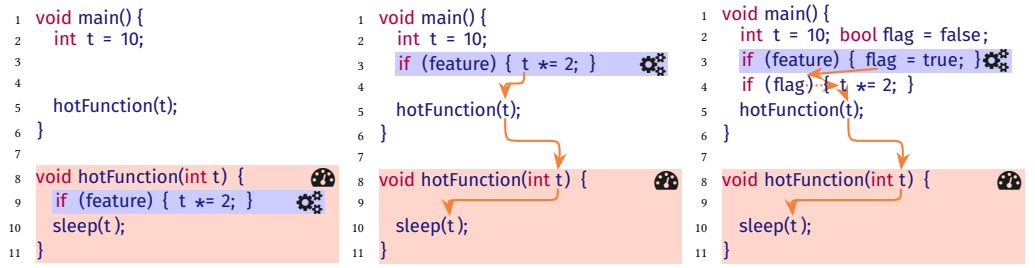
Synthetic Regression Scenarios. We use synthetic scenarios to systematically and qualitatively assess and discuss the capabilities of ConfFLARE. Specifically, we consider different mechanisms of how features can interact with performance-relevant code (Inter), various patterns of how changes in performance behavior contribute to a regression (Func), and how the structure of a configuration space and constraints between features affects ConfFLARE (Deg). The code for all synthetic scenarios is contained in our replication package.

Table 1. Overview of our subject systems

	Name	Domain	Scenarios	$\|F\|$	$\|C\|$
Synthetic	INTER$_{\text{STRUC}}$	Interaction pattern	1	1	2
	INTER$_{\text{DF}}$	Interaction pattern	1	1	2
	INTER$_{\text{IMPL}}$	Interaction pattern	1	1	2
	FUNC$_{\text{SINGLE}}$	Regression behavior	1	3	8
	FUNC$_{\text{ACCUM}}$	Regression behavior	3	3	8
	FUNC$_{\text{MULTI}}$	Regression behavior	3	3	8
	DEG$_{\text{LOW}}$	Configuration space	1	10	1024
	DEG$_{\text{HIGH}}$	Configuration space	1	10	1024
	DEG$_{\text{COMPLEX}}$	Configuration space	1	10	320
Seeded	BASENC	GNU Coreutils	5	10	84
	CKSUM	GNU Coreutils	5	12	36
	DD	GNU Coreutils	5	18	1404
	OD	GNU Coreutils	5	16	480
	PR	GNU Coreutils	5	12	1920
	SORT	GNU Coreutils	5	8	128
	UNIQ	GNU Coreutils	5	14	384
	WC	GNU Coreutils	5	5	32
RW	BZIP2	Compression	19	9	576
	GREP	Unix tool	7	10	288
	PICOSAT	SAT-solver	12	5	32

The INTER scenarios cover three common mechanisms of how features can interact with performance-relevant code: structural interactions (INTER$_{\text{STRUC}}$), data-flow interactions (INTER$_{\text{DF}}$), and interactions via implicit data-flow (INTER$_{\text{IMPL}}$). With these scenarios, we study the capabilities of CONFFLARE to detect which features are releated to a performance regression. Figure 4 shows simplified code examples of these scenarios: The regression scenario we investigate changes the initial value of variable `t` that influences how long `hotFunction` takes to execute. The only difference between the individual scenarios is *how* feature code interacts with performance-relevant code: In INTER$_{\text{STRUC}}$, feature code interacts *structurally* with performance-relevant code, i.e., the feature is part of the performance-relevant code. In INTER$_{\text{DF}}$, feature code modifies the value of variable `t`, which in turn is passed as a parameter to `hotFunction`. That is, the feature interacts with the performance-relevant code via *data flow*. In INTER$_{\text{IMPL}}$, feature code influences a control-flow decision (via variable `flag`) that once again impacts performance-relevant code because it modifies variable `t`. Such data flow that is caused by a control-flow decision is called *implicit flow* [29] and requires special handling in a data-flow analysis. Each of these scenarios requires different capabilities to identify the relevant feature. While INTER$_{\text{STRUC}}$ is easy to detect due to the structural overlap, INTER$_{\text{DF}}$ and INTER$_{\text{IMPL}}$ can only be detected by considering data flow—highlighting the benefit of CONFFLARE over simpler strategies.

The FUNC scenarios capture different patterns of how individual changes in performance behavior contribute towards a regression. Each of these scenarios consists of three performance-relevant functions. In FUNC$_{\text{SINGLE}}$, a change impacting a single function with performance-relevant code triggers a regression. In FUNC$_{\text{ACCUM}}$, we use three regression scenarios that successively impact more functions (first one, then two, and finally all three functions). Each impacted function exhibits a change in its performance behavior, but falls below the selected threshold to trigger a regression



(a) InterStruc (b) InterDF (c) InterImpl

Fig. 4. Simplified code examples of the Inter scenarios demonstrating different interaction patterns between features and performance-relevant code. Function hotFunction is performance-relevant (🏎) and code controlled by variable feature is feature code (⚙). Solid lines represent data flow, the dotted line represents implicit flow.

on its own. Only if all three are affected at the same time, the individual changes accumulate and the regression becomes noticeable. This scenario is interesting because static analysis has no information about the magnitude of the performance regression and, thus, should report each individual interaction as a potential regression. In $\text{Func}_{\text{multi}}$ the regression manifests only if two specific functions are impacted at the same time. There is no change in performance behavior at all if only individual functions are affected. This scenario poses another challenge to any static analysis tool because the exact conditions under which this regression manifests are not visible considering only data flow (see Section 3.1). Consequently, we expect that ConfFLARE will report potential regressions here more often than necessary.

The Deg scenarios target how the structure of a configuration space and constraints between features affect ConfFLARE. Deg_{low} and Deg_{high} share the same configuration space, meaning that they have the same features and configurations. The difference is the number of configurations for which the introduced regression can be observed. Specifically, the configuration space consist of ten optional and independent features (i.e., 1024 different configurations). Deg_{low} requires only very few features to be selected simultaneously to trigger the regression. That is, the feature interaction necessary to reveal the regression has a *low interaction degree*. Such a regression is easy to detect as it can be observed in many configurations. Deg_{high}, however, requires all ten features to be selected simultaneously in order to trigger the regression. That is, the regression requires a feature interaction with a *high interaction degree* and can only be observed in a single configuration, making it very hard to detect. The configuration space of $\text{Deg}_{\text{complex}}$ contains more complex constraints among features. Such a constrained configuration space is more realistic, because many constraints between features exist in real-world configurable software systems [8].

Seeded Regression Scenarios in Real-World Systems. To evaluate ConfFLARE in a more realistic setting, we consider seeded regression scenarios based on eight different tools from GNU Coreutils. We chose to use tools from GNU Coreutils as a basis for the seeded regression scenarios because they provide a good compromise between complexity and size. Their size allows us to understand these systems enough to seed additional regression scenarios and so that we are able to qualitatively assess the study results. At the same time, they are complex enough to provide a realistic setting for our experiments.

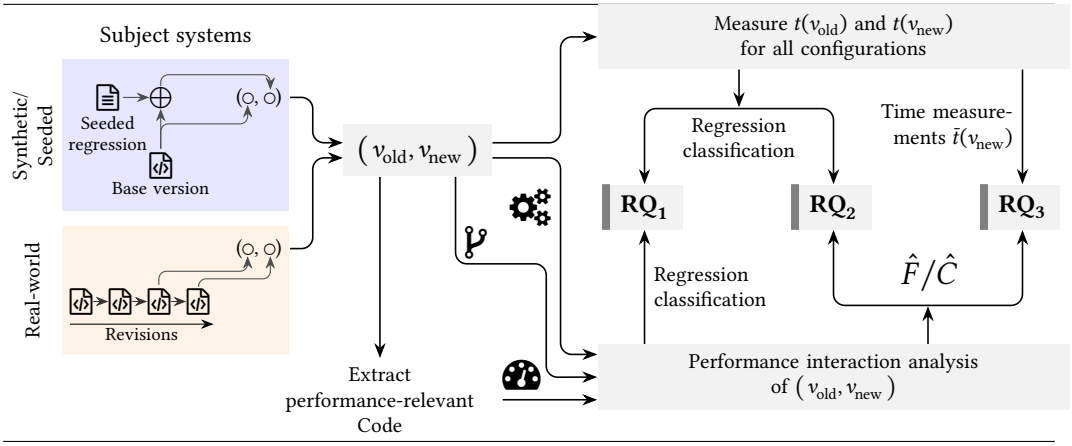


Fig. 5. Pairs of consecutive revisions of each subject system constitute the set of regression scenarios. For each scenario, we establish a ground truth by performing performance measurements for all configurations. Aditionally, we obtain a regression classification and the related features/configurations from our performance interaction analysis. We answer our research questions by comparing these results to the ground truth.

For each of the eight tools, we consider a set of five seeded regression scenarios that we have created for this purpose. Each scenario consists of two patches: one patch simulates a performance-relevant change, and the other patch simulates the actual regression. That way, we ensure that the seeded regression has a measurable impact on the performance behavior of the system.

Real-World Regression Scenarios. For the real-world scenarios, we select three open-source projects from different domains. We limit ourselves to three systems due to the high cost of our experiments measuring performance of the full configuration space for multiple revisions of each system. We focus on internal validity, and this set of three systems is sufficient and provides the necessary level of control and detail. For each system, we consider several real-world regression scenarios. Therefore, we select pairs of revisions that contain significant changes. Since many of our systems are quite mature and mostly experience very small maintenance changes, these revisions are not necessarily consecutive. For PICOSAT, we selected revisions tagged as releases. For BZIP2 and GREP, we manually examined the revision history and selected commits based on their diff. Since building a ground truth requires analyzing the full configuration space for all selected revisions of each subject system, we limit the feature model of each system to a subset of features. That way, we reduce the costs for our experiments to a feasible level while preserving their meaningfulness, as the exact number of considered features and configurations is not the focus in this study.

4.3 Operationalization

Figure 5 provides an overview of our experiments. For each subject system—synthetic and real-world—we consider a set of regression scenarios, that is, pairs of successive revisions $(v_{\text{old}}, v_{\text{new}})$. In a regression scenario, revision v_{old} is the current state of the system, for which we already have knowledge about its performance behavior and performance-relevant code regions. Revision v_{new} contains the code change that we want to examine for potential performance regressions.

To establish a proper ground truth for comparison with CONFFLARE, we measure the execution times for each configuration of v_{old} and v_{new} with 10 repetitions to account for measurement noise. Given a program revision v and configuration c, we denote the set of measurements (i.e., the 10

repetitions) of v^c with $t(v^c)$ and their mean as $\bar{t}(v^c)$. Based on these measurements, we classify a performance regression as follows:

Definition 6. Given a threshold ε and a significance level k, a pair of revisions $(v_{\text{old}}, v_{\text{new}})$ gives rise to a performance regression iff:

$$\exists\, c \in C.\ \left|\, \bar{t}(v^c_{\text{new}}) - \bar{t}(v^c_{\text{old}}) \,\right| \geq max\Big(\varepsilon \cdot \bar{t}(v^c_{\text{old}}), k \cdot \sigma'\Big)$$

with $\sigma' = max\Big(\sigma\big(t(v^c_{\text{old}})\big), \sigma\big(t(v^c_{\text{new}})\big)\Big)$, denoting the maximum of the standard deviations of the measurements.

The threshold parameter ε controls the sensitivity of the regression detection, and k controls the significance level. For example, setting $\varepsilon = 0.1$ and $k = 3$ means that a configuration of v_{new} needs to be, at least, 10% slower or faster than v_{old} to be considered regressing, and the probability that such a regression is just measurement noise is $< 0.3\%$.[4] For our experiments, we use $\varepsilon = 0.1$ and $k = 3$. These values give us high confidence that we observe real regressions and not just measurement noise.

Our performance interaction analysis requires information about changes (), features (), and performance-relevant code () to find performance-relevant interactions (see Section 3). The change information is already encoded in the diff between v_{old} and v_{new}. To provide feature information to the analysis, we follow the approach outlined in Section 2. As for performance-relevant code, we focus on hot code (see Section 3). To obtain hot code regions, we use two different approaches: (1) For the synthetic scenarios, we manually annotate hot code regions. This is possible because we specifically designed them so that we know the locations of hot code. (2) For the GNU Coreutils tools and the real-world systems, we extract hot code regions through performance profiles. Specifically, we create sample-based performance profiles[5] for all configurations of v_{old} and extract all functions that were attributed with, at least, 5% of the samples. We use this percentage as a proxy for execution time spent inside the function's code itself (i.e., self-time).

With this setup, we address our three research questions as follows.

Operationalization of RQ₁. To answer RQ_1, we use our performance interaction analysis as a classifier for potential performance regressions and compare its results to the baseline obtained from the performance measurements (see Definition 6). As ConfFLARE, by construction, can detect only regressions involving hot code, we consider only regressions that can be attributed to hot code (i.e., if the regression is also classified as such when considering only hot code). If we find performance-related interactions, we classify the change as a potential regression. We compare this classification to the ground truth using a precision–recall analysis. Since our goal is to reduce the costs of running performance measurements, our focus lies primarily on recall. We aim for a high recall because we consider missed regressions worse than false positives. Note that the synthetic and seeded regression scenarios do not contain any false positives, as we designed them so that all changes are performance-relevant. For the real-world scenarios, we do not expect a high precision since any data flow between a change and hot code will result in a performance-relevant interaction, even if such a data flow does not measurably influence the performance behavior. However, we do not consider this a problem because the information provided by ConfFLARE can still be useful for limiting the number of configurations that have to be analyzed in these cases (see RQ_3).

[4] We assume that the measurement noise is normally distributed and apply the empirical rule.
[5] We use Linux perf as a sampling profiler and sample at a rate of 997 Hz and 10 repetitions.

Operationalization of RQ_2. The information provided by ConfFLARE must be sufficient for identifying performance regressions. To verify this, applying Definition 6 to $\hat{C}$ instead of C should yield the same classification. For this purpose, we compare the classification of the configurations in $\hat{C}$ to the ground truth using an additional precision–recall analysis. Note that we consider only regression scenarios that were classified as a regression by ConfFLARE, because $\hat{C}$ does not exist for the other cases. Since the configurations in $\hat{C}$ provide less information ($\hat{C} \subseteq C$), there can never occur any false-positives (i.e., regression scenarios that are classified as non-regressing using C will never be classified as regressing using $\hat{C}$). As a consequence, the precision will always be 1. Instead, we are interested in the recall: a recall < 1 indicates that $\hat{C}$ did not include configurations necessary to detect a regression.

Operationalization of RQ_3. First, we are interested in the potential savings in terms of analyzed configurations, that is, how much smaller $\hat{C}$ is compared to C. Since for this experiment $\hat{C}$ has to exist as well, we again consider only experiment runs that are classified by ConfFLARE as potential regressions. We report the relative size $|\hat{C}|/|C|$ as well as $|\hat{F}|/|F|$ to see how the number of features flagged as related to a performance-relevant change affect the size of $\hat{C}$. Then, we report how much measurement time would have been saved in our experiments by relating the time it takes to execute all configurations $T_C = \sum_{c \in C} \bar{t}(v^c_{\text{new}})$ to $T_{\hat{C}} = \sum_{c \in \hat{C}} \bar{t}(v^c_{\text{new}})$—the time required for executing only configurations from $\hat{C}$. Last, we use the speedup $T_C / T_{\hat{C}}$ to put these savings into perspective.

5 Study Results

In this section, we report our results for the synthetic and real-world systems separately so that we can account for their individual premises in our discussion.

5.1 Synthetic Regression Scenarios

Results for RQ_1. Here, we evaluate how well ConfFLARE can detect performance-relevant changes in various regression scenarios. Table 2 summarizes the results for RQ_1. It shows for each scenario the key metrics of the precision–recall analysis. Overall, ConfFLARE detects all true regressions for all our synthetic systems resulting in a recall value of 1. This demonstrates that ConfFLARE is capable of detecting performance-relevant changes with different characteristics and independent of the size or structure of the configuration space. The scenarios $\text{Func}_{\text{accum}}$ and $\text{Func}_{\text{multi}}$ are noteworthy. Here, ConfFLARE classifies all three regression scenarios as potential regressions although only one of them is a true regression. This can be explained by the fact that, whether a regression occurs in these scenarios, depends on properties not visible to a data-flow analysis as discussed in Section 4.2. Still, the results of ConfFLARE are an over-approximation of the set of true performance regressions meaning that no regressions are missed.

Results for RQ_2. Next, we are concerned with whether performance regressions can be detected using only configurations from $\hat{C}$ (which is the ultimate goal to save effort and costs). Again, Table 2 summarizes the results for the synthetic regression scenarios. Since this experiment can never yield false positives (as $\hat{C} \subseteq C$, there can't be any new regressions; see Section 4.3), we do not report true positives or precision values. Any recall value smaller than 1 indicates a regression scenario where a true regression cannot be detected using $\hat{C}$, meaning that ConfFLARE did not provide the correct information required to identify regressing configurations. The only system where we can observe this issue is $\text{Inter}_{\text{impl}}$. Here, the feature triggering the regression does so via *implicit flow* (see Section 4.2). ConfFLARE fails to detect this interaction between feature and hot code because implicit flows are currently not supported by the underlying data-flow analysis. Note that

Table 2. Synthetic systems results of the precision–recall analysis for RQ_1 and RQ_2 (P = positives, PP = predicted positives, TP = true positives).

	RQ_1						RQ_2			
Project	Scenarios	P	PP	TP	Precision	Recall	Scenarios	P	PP	Recall
$\text{Inter}_{\text{struc}}$	1	1	1	1	1.00	1.00	1	1	1	1.00
Inter_{DF}	1	1	1	1	1.00	1.00	1	1	1	1.00
$\text{Inter}_{\text{impl}}$	1	1	1	1	1.00	1.00	1	1	0	0.00
$\text{Func}_{\text{single}}$	1	1	1	1	1.00	1.00	1	1	1	1.00
$\text{Func}_{\text{accum}}$	3	1	3	1	0.33	1.00	3	1	1	1.00
$\text{Func}_{\text{multi}}$	3	1	3	1	0.33	1.00	3	1	1	1.00
Deg_{low}	1	1	1	1	1.00	1.00	1	1	1	1.00
Deg_{high}	1	1	1	1	1.00	1.00	1	1	1	1.00
$\text{Deg}_{\text{complex}}$	1	1	1	1	1.00	1.00	1	1	1	1.00

this is not a conceptual limitation of our approach but follows from the choice of the particular data-flow analysis. However, in all other scenarios, ConfFLARE provides the necessary information to detect the regression using only configurations from $\hat{C}$, which shows that it accurately identifies relevant features.

Results for RQ3. Here, we investigate how much testing effort can potentially be saved using ConfFLARE. The results for the synthetic systems are shown in Table 3. We can see that the amount of saved performance testing effort, both, in terms of tested regressions and execution time, varies significantly between different scenarios. This can best be observed looking at the results for the Deg scenarios: On the one hand, for Deg_{low} $\hat{F}$ contains only a single feature leading to a severe reduction in performance testing effort (only two out of 1024 configurations need to be tested). On the other hand, all features are marked relevant for Deg_{high}, meaning that all 1024 configurations need to be considered during performance testing, even though a regression occurs only for a single configuration in this specific scenario. Lastly, for $\text{Deg}_{\text{complex}}$, $\hat{C}$ contains only 6% of all configurations despite 60% of features being reported as relevant. Here, the small size of $\hat{C}$ can be explained by the fact that the structure of the configuration space constrains which features can be selected together. From these results, we can infer that the potential cost savings depend on the structure of the configuration space as well as which features are declared relevant. But, as the synthetic scenarios can only show some representative examples, we investigate RQ_3 in more detail for the seeded and real-world scenarios later.

5.2 Seeded Regression Scenarios in Real-World Systems

Results for RQ1. For the seeded regressions, the results for RQ_1 and RQ_2 are summarized in Table 4. We can see that ConfFLARE correctly identifies all scenarios as performance-relevant changes except for two cases. For cksum, ConfFLARE does not detect any performance-relevant change. The reason for this is that all functions marked as hot code are only used via function pointers and the employed data-flow analysis cannot track data flow through function pointers with sufficient precision. As for the missed case in dd, the used data-flow analysis did not report any interaction between the changed code and performance-relevant code, but we were unable to determine the exact reason for this behavior. For all other systems, ConfFLARE correctly identifies all regressions

[6]Execution time of the selected configuration was rounded to 0.00 by our measurement setup. Hence, no speedup can be calculated.

Table 3. Synthetic regression scenarios results for RQ_3. Grouped rows show different regression scenarios for the same system. A dash (–) indicates no saved performance testing effort.

Project	$\lvert F\rvert$	$\lvert\hat{F}\rvert$	$\lvert C\rvert$	$\lvert\hat{C}\rvert$	$\lvert\hat{F}\rvert/\lvert F\rvert$	$\lvert\hat{C}\rvert/\lvert C\rvert$	T_C (s)	$T_{\hat{C}}$ (s)	$T_C/T_{\hat{C}}$
INTER_{DF}	1	1	2	2	1.00	1.00	14.50	14.50	1.00
$\text{INTER}_{\text{STRUC}}$	1	1	2	2	1.00	1.00	14.50	14.50	1.00
$\text{INTER}_{\text{IMPL}}$	1	0	2	1	0.00	0.50	14.50	6.50	2.23
$\text{FUNC}_{\text{SINGLE}}$	3	1	8	2	0.33	0.25	18.00	2.50	7.20
$\text{FUNC}_{\text{ACCUM}}$	3	1	8	2	0.33	0.25	78.00	13.50	5.78
	3	2	8	4	0.66	0.50	80.00	34.00	2.35
	3	3	8	8	1.00	1.00	82.00	82.00	1.00
$\text{FUNC}_{\text{MULTI}}$	3	2	8	4	0.66	0.50	8.00	2.00	4.00
	3	0	8	1	0.00	0.13	8.00	0.00[6]	–
	3	2	8	4	0.66	0.50	9.00	2.50	3.60
DEG_{LOW}	10	1	1024	2	0.10	0.0020	3328.00	6.50	512.00
DEG_{HIGH}	10	10	1024	1024	1.00	1.00	2561.50	2561.50	1.00
$\text{DEG}_{\text{COMPLEX}}$	10	6	320	20	0.60	0.063	800.00	50.00	16.00

Table 4. Seeded regressions results of the precision–recall analysis for RQ_1 and RQ_2 (P = positives, PP = predicted positives, TP = true positives).

	RQ_1						RQ_2			
Project	Scenarios	P	PP	TP	Precision	Recall	Scenarios	P	PP	Recall
BASENC	5	5	5	5	1.00	1.00	5	5	5	1.00
CKSUM	5	5	0	0	-	0.00	-	-	-	-
DD	5	5	4	4	1.00	0.80	4	4	4	1.00
OD	5	5	5	5	1.00	1.00	5	5	4	0.80
PR	5	5	5	5	1.00	1.00	5	5	5	1.00
SORT	5	5	5	5	1.00	1.00	5	5	5	1.00
UNIQ	5	5	5	5	1.00	1.00	5	5	5	1.00
WC	5	5	5	5	1.00	1.00	5	5	5	1.00

resulting in a recall of 1, demonstrating that it is effective in detecting performance-relevant changes also in real-world code.

Results for RQ_2. Notably, the information provided by ConfFLARE was sufficient to detect all but one of the considered regressions. The one missed scenario is actually due to the first step of ConfFLARE: It detects a performance-relevant interaction with a hot function, but the taint analysis does not reach the relevant part of that function. In this concrete case, there is feature code within this function that influences the performance behavior, but that part of the function is not tainted.[7] Therefore, the second step correctly omits that feature from $\hat{F}$ and, thus, $\hat{C}$ contains no regressing configuration. In all other cases, ConfFLARE provides the necessary information to detect the regression using only configurations from $\hat{C}$, which shows that the provided information is sufficiently precise such that no regressions are missed.

[7]We reported this issue to the developers of the data-flow analysis, and they confirmed that this happens due to the analysis producing unsound results in this specific case.

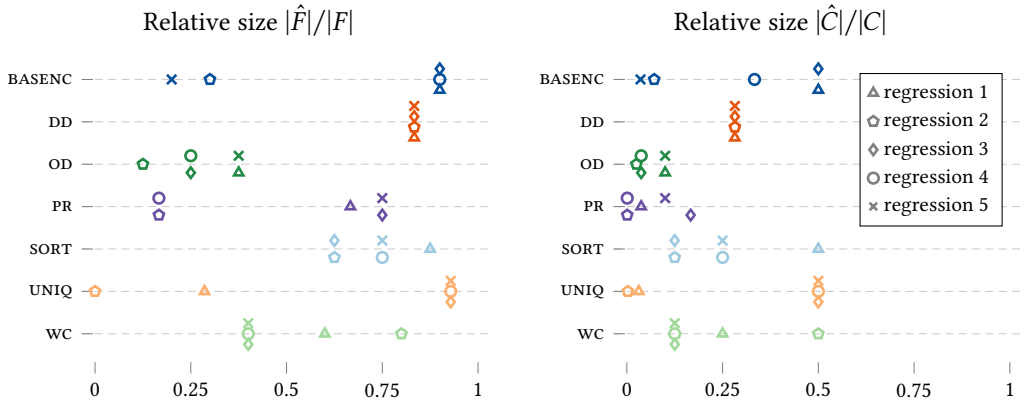


Fig. 6. Seeded regressions results for RQ_3. For each considered coreutils tool, we show the relative sizes $|\hat{F}|/|F|$ (left) and $|\hat{C}|/|C|$ (right). We use different markers to enable tracking of individual regression scenarios between plots: the same marker in the same color corresponds to the same regression scenario. Note that there are only four markers for DD as only four regression scenarios were detected as performance-relevant (see Table 4).

Results for RQ_3. We first focus on the results on cost savings in terms of the size of $\hat{C}$ depicted in Figure 6. In this figure, we show for each system the relative sizes of $\hat{F}$ and $\hat{C}$ compared to F and C, respectively. Overall, we can see that the number of features in $\hat{F}$ (and thus, the number of configurations in $\hat{C}$) varies significantly between different systems and regression scenarios. Still, at least one feature is not considered related for all regression scenarios. As a result, $\hat{C}$ is always at most half the size of C, meaning that we can always save, at least, half of the performance testing effort. For most scenarios (22 out of 34) this reduction is even more pronounced, saving 75% of configurations to test or more. Additionally, Figure 6 visualizes the relationship between the sizes of $\hat{F}$ and $\hat{C}$: the more features are marked as relevant, the more configurations need to be tested as the number of possible combinations of these features increases exponentially. Figure 7 shows the actual execution times for executing all configurations in C compared to only executing configurations from $\hat{C}$. As the different systems have very different execution times and different numbers of configurations, the sums of execution times T_C range from minutes to several hours. To better accommodate this wide range, we use a logarithmic scale for both axes. We can see that the speedups gained by only executing configurations from $\hat{C}$ is roughly 2 or greater for all regression scenarios. This makes intuitively sense, as we know from Figure 6 that we need to execute, at most, half of the configurations. For many scenarios, the speedup is even significantly higher. We observe the highest speedup of ≈ 2100 for two scenarios of PR reducing the execution times from around 13 hours to less than 22 seconds. This shows that the information provided by ConfFLARE can lead to significant cost savings in performance testing.

5.3 Real-World Regression Scenarios

Results for RQ_1. Finally, we present the results for the real-world regression scenarios. Table 5 summarizes the results for RQ_1 (and RQ_2). The first observation is that most scenarios do not contain any performance regression. This is a case that we could not simulate well with seeded regression scenarios, but is common in real-world systems. Therefore, it is interesting to see how well ConfFLARE can avoid false positives in these scenarios. For BZIP2, none of the 18 scenarios is an actual regression. ConfFLARE has flagged 8 of them as potential regressions, suggesting

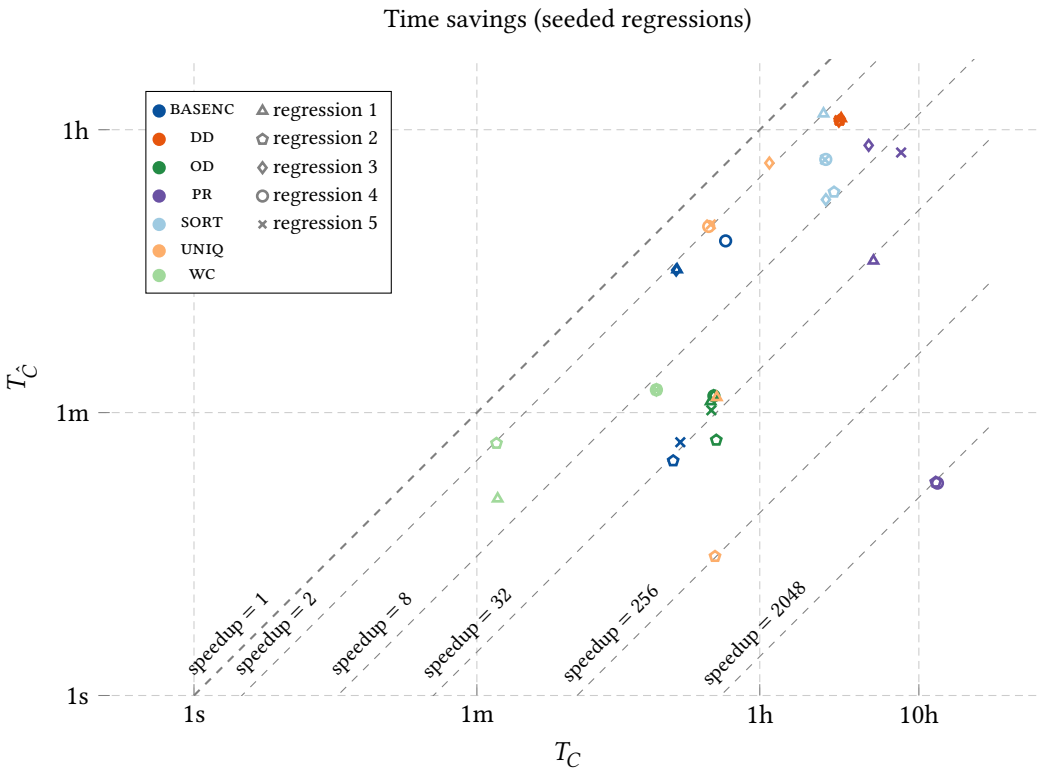


Fig. 7. Seeded regressions execution time results for RQ_3. Here, we plot the time in seconds for executing all configurations (x-axis) to the time for executing only configurations from $\hat{C}$ (y-axis). Note that this plot uses a logarithmic scale for both axes to better accomodate the execution times of all tested systems. There are no values to the north-west of the diagonal $x = y$ because $\hat{C} \subseteq C$. Diagonal lines indicate different speedup factors $T_C/T_{\hat{C}}$.

that further performance testing should be performed in less than half of the scenarios. For GREP only two out of seven scenarios are actual regressions. Here, CONFFLARE classifies four regressing scenarios as potential regressions, but detects only one of the two true regressions. In the missed regression scenario, the introduced change consists of exactly one deleted line of code, meaning that CR_C^V is empty for this scenario. Therefore, no performance-relevant interactions exist according to Definition 4. Only for PICOSAT, most of the selected scenarios (7 out of 11) contain a regression. Here, CONFFLARE classifies 9 scenarios as potential regressions, including all 7 true regressions. These results suggest that CONFFLARE is effective in detecting performance-relevant changes also in real-world systems as long as a change doesn't solely consist of deletions.

Results for RQ_2. From Table 5, we can see that information provided by ConfFLARE was sufficient to detect regressions. As no true regression exists for BZIP2, the choice of $\hat{C}$ does not matter here, but it will still be interesting for RQ_3 later. For GREP, CONFFLARE provides sufficient information to detect the one true regression. One interesting case is regression scenario (`ffc6e407e3`, `192e59903c`) of GREP, where the path reconstruction step did not terminate within a reasonable time frame. As a consequence, no feature information is available and $\hat{C}$ contains only one configuration. However, as this scenario is not a true regression, this does not lead to any missed regressions. For PICOSAT,

Table 5. Real-world systems results of the precision–recall analysis for RQ_1 and RQ_2 (P = positives, PP = predicted positives, TP = true positives).

	RQ_1						RQ_2			
Project	Scenarios	P	PP	TP	Precision	Recall	Scenarios	P	PP	Recall
BZIP2	18	0	8	0	0.00	–	8	0	0	–
GREP	7	2	4	1	0.25	0.50	4	1	1	1.00
PICOSAT	11	7	9	7	0.78	1.00	9	7	7	1.00

Fig. 8. Real-world regressions results for RQ_3. For each system, we show the relative sizes $|\hat{F}|/|F|$ (left) and $|\hat{C}|/|C|$ (right).

all regressions can be detected using only configurations from $\hat{C}$. From this, we can conclude that ConfFLARE provides sufficiently precise information to avoid missing regressions also in real-world systems.

Results for RQ_3. For the real-world regression scenarios, we show the cost savings in terms of the size of $\hat{C}$ in Figure 8. In contrast to the seeded regression scenarios, $\hat{F}$ contains more than half of the features in most cases. In four cases (one for BZIP2 and three for PICOSAT), all features are marked as relevant. Consequently, $\hat{C}$ contains all configurations for these regression scenarios as well. Still, for most scenarios, we only need to test 25% of all configurations. For the scenario of GREP where the path reconstruction did not terminate, $\hat{F}$ is empty and, thus, $\hat{C}$ contains only a single configuration. In Figure 9, we show the execution times for the real-world systems in the same way as for the seeded regression scenarios. For BZIP2, the execution times are almost identical between the different regression scenarios confirming the fact that there are no regressions. Also, the observed speedup values are in line with the relative sizes of $\hat{C}$. A similar picture emerges for GREP. Only the speedup factor for the scenario where we only test one configuration is smaller than expected, suggesting that the selected configuration is slower than average. PICOSAT shows the most interesting results here: First, almost all regression scenarios have very different execution times, supporting our claim that there exist real regressions. The three regression scenarios where $\hat{C} = C$ unsurprisingly show no speedup at all. However, four of the five scenarios where $\hat{C}$ contains 25% of configurations, we observe a speedup of 1.6. This means that we still spend 62.5% of the time while executing only 25% of the configurations. This shows that the savings in terms of number of configurations do not necessarily translate directly to savings in execution time. The same holds true for the remaining two regression scenarios of PICOSAT but with different speedup factors. These results demonstrate that, while the information provided by ConfFLARE always helps reduce the cost for performance testing, the magnitude of the reduction highly depends on the characteristics of the regression scenario in practice.

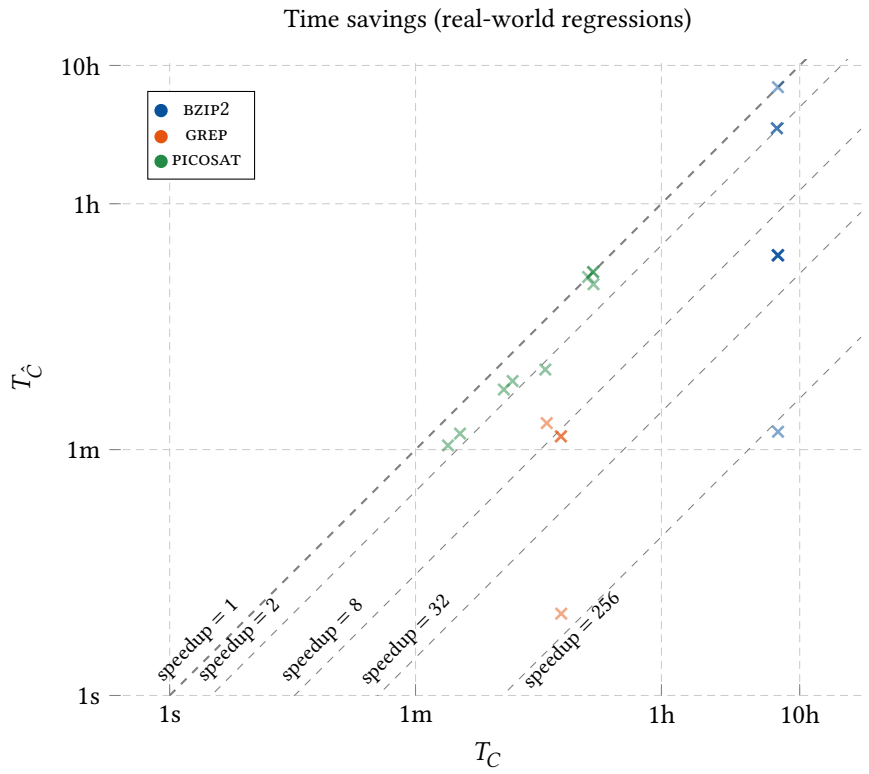


Fig. 9. Real-world regressions execution time results for RQ_3. We plot the time in seconds for executing all configurations (x-axis) to the time for executing only configurations from $\hat{C}$ (y-axis). For a more detailed description of the plot, see Figure 7.

5.4 Discussion

After presenting the results of our empirical study, we now discuss and interpret them with respect to our research questions. Furthermore, we elaborate on how ConfFLARE advances the status quo for performance testing and how it integrates in the wider landscape of techniques for analyzing configurable software systems.

Identifiyng Performance Regressions and Related Features (RQ_1 and RQ_2). With our first two research questions, we intended to evaluate how well ConfFLARE can identify changes in configurable software systems that cause performance regressions and how accurately it can identify which features and configurations are impacted by these changes. In our experiments, ConfFLARE correctly identified the majority of regressions demonstrating its ability to reliably detect performance regressions in a diverse set of circumstances. The only case where it conceptually failed is an edge case where the change consists only of deletions, which makes it impossible to attribute the change to any code location in the analyzed version—which is one of the prerequisites for our approach to be applicable. Other failures can be attributed to limitations of the underlying data-flow analysis, which is part of a cost-benefit trade-off that we discuss later in this section. Furthermore, the results for the real-world regression scenarios show that ConfFLARE often correctly identifies non-regressing changes as such, which greatly helps save performance testing effort.

RQ₁	ConfFLARE can reliably identify changes with performance regressions in a diverse set of scenarios.

The main contribution of this work is to provide an automatic way of identifying which features are related to a performance-relevant change. This information was previously only obtainable from domain experts, tedious manual work, or expensive trial-and-error testing. To measure whether ConfFLARE provides the correct features required to observe existing performance regressions, we selected a subset of configurations to be tested based on these features. If a regression can be detected using only the selected configurations, we consider the provided information as sufficient. In our experiments, this was the case for all but two regression scenarios, both of which arise from limitations of the underlying data-flow analysis. But overall, these results demonstrate that ConfFLARE is capable of accurately identifying which features are related to a performance-relevant change, which is highly valuable information for guiding performance testing.

RQ₂	ConfFLARE can accurately identify which features are related to a performance-relevant change in most cases. The accurracy and completeness of this information is subject to a cost-benefit trade-off of the underlying data-flow analysis.

Cost Saving Potential (RQ_3). Our main motivation for this work is to make performance testing of configurable software systems more efficient by reducing the number of regression scenarios and configurations that need to be tested. With our experiments, we demonstrated that ConfFLARE helps reduce performance testing effort in both regards: Its first step identifies whether a regression scenario should be considered for performance testing at all. In our experiments with real-world regression scenarios, 15 out of 27 non-regressing scenarios were correctly classified as such, meaning that all performance testing effort can be saved in more than half of these cases. The information provided by ConfFLARE's second step is useful for limiting the number of configurations to be tested. In our experiments, the fraction of configurations to be tested is on average 21% for the synthetic regression scenarios and 30% for the real-world regression scenarios. Only testing configurations from $\hat{C}$ instead of C resulted in speedup factors ranging from 2 to 32 in most cases with some outliers reaching even higher, sometimes saving hours of performance testing time. Note that our way of constructing $\hat{C}$ from $\hat{F}$ is intentionally conservative to provide a lower bound estimate. In practice, configurations could be selected much more intelligently, e.g., by informing sampling techniques about which features are relevant, further increasing the potential for saving measurement effort.

RQ₃	ConfFLARE helps significantly reduce the performance testing effort required to detect performance regressions. The magnitude of the savings depends on the characteristics of the regression scenario (magnitude of the regression; number of involved features) and the structure of the configuration space.

Analysis Cost-Benefit Trade-Off. As ConfFLARE makes heavy use of data-flow analysis, the quality of its results and its cost are closely tied to the precision of the underlying analysis. This precision enables our approach to find regressions that are hard to detect with other techniques. However, more precise analyses are usually also more expensive to execute. This leads to a trade-off between analysis cost and the benefit gained its results. For example, the data-flow analysis we used in our experiments cannot track implicit flows, has limitations when tracking function pointers, and sometimes underapproximates points-to information. Consequently, we missed some regressions in our experiments that, in theory, could have been detected with a more precise analysis, but at much higher cost. In contrast, it is able to process real-world C/C++ code, which is a rare ability.

It is important to note that this trade-off is conceptually separate from ConfFLARE itself and, depending on the use case, different choices can be made here.

Integration Into Tooling Landscape. While ConfFLARE and the information it provides is useful on its own, it also complements and integrates well with existing techniques used for tackling the challenges arising from performance regression-testing of configurable software systems. In Figure 10, we illustrate how ConfFLARE fits into the wider landscape of such techniques.

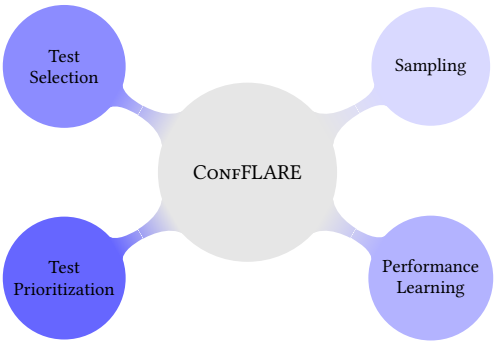


Fig. 10. Overview of how ConfFLARE integrates with existing techniques for performance testing of configurable software systems.

We already mentioned before how information about relevant features provided by ConfFLARE can be used to guide sampling or other configuration selection strategies towards configurations that are more likely to exhibit performance regressions. For performance modeling or learning approaches, one challenge is to decide which features or combinations thereof should be included in a model. Information about relevant features provided by ConfFLARE can help here as well as a starting point for building such models or to selectively update them after a change to the system. The same information could also be used to select or prioritize test cases, e.g., to favor test that exercise relevant features or configurations. The change-impact aspect of ConfFLARE complements aforementioned strategies well by skipping performance testing for entire revisions. ConfFLARE can also complement other aspects of performance testing that are not directly related to configurations, but the execution environment. For example, we found that one of the regression scenarios for sort only contains a real regression if a certain locale[8] is set on the system. Therefore, it is easy to miss if performance measurements are only performed on a system with the wrong locale. By looking at the code attributed to the extracted features it was easy to spot the dependence on the system's locale.

5.5 Threats to Validity

Internal Validity. There are several factors that are important to consider for the internal validity of our study. The most relevant one is certainly our definition of what we consider a performance regression. The choice of the parameters ε and k can have an influence on the results of all research questions. Our choice of $\varepsilon = 0.1$ and $k = 3$ ensures that only regressions with a substantial difference in execution time are considered and avoids false positives caused by measurement noise. To further reduce the impact of measurement noise, we performed all measurements on identical hardware with an identical environment and repeated the measurements 10 times. Our definition of regression

[8]The *locale* specifies, amongst other things, what language and character encoding the system should use.

is also quite strict in that it flags a revision as regressing if even a single configuration regresses. This strictness is necessary as regressions may affect only a small number of configurations.

Next is the choice of what to consider performance-relevant code. In our study, we focus on hot code, that is, code that contributes disproportionately to the overall runtime. We consider hot code because it can be measured automatically. Using a sampling profiler to do so is less precise than using instrumentation measuring the exact execution times of functions. However, we found that the overhead introduced by instrumentation is so high that sampling is the only viable choice. For our real-world subject systems, we consider all functions that contribute more that 5 percent to the overall execution time. We chose that threshold because we observed a jump in function execution times around that value with most functions taking considerably less time.

External Validity. As our study is the first to evaluate a novel approach for improving performance regression testing in configurable software systems, we focus on achieving high internal validity, instead of external validity [35]. This enables us to study the ConfFLARE in a controlled setting and assess its capabilities in detail. Further studies should then focus on how our initial results generalize (external validity) and how ConfFLARE can be best applied in practice (ecological validity). Still, both, our seeded and real-world regression scenarios, use real-world software demonstrating that ConfFLARE can be applied to practical systems and not only toy examples. Thus, we argue that our study provides a solid foundation for future research on how ConfFLARE can be utilized to its best.

6 Related Work

Performance Evolution of Configurable Software Systems. Considerable effort has been spent on understanding the performance behavior of configurable software systems. Several approaches have been devised to learn models for the performance behavior of configurable software systems [1, 13, 14, 21, 22, 47, 48, 52, 53, 56]. However, these approaches typically target individual revisions of a system.

More recently, the performance evolution of configurable software systems has gained increased attention. In an empirical study, Kaltenecker et al. [26] find that performance changes frequently in configurable software systems and most performance changes affect only a subset of configurations. Mühlbauer et al. [40] combine an iterative approach with Gaussian Process models to approximate the performance behavior of a software systems across its entire history. They later extend this work to identify configuration-dependent performance changes across both revisions and configurations of a configurable software system [41], although their approach works retrospectively and thus cannot be applied in a regression setting. Martin et al. [36] use transfer learning to reduce the number of required new training samples when transferring an existing model to new revisions. In their study, they only evaluate the effectiveness of transfer learning across configurations and revisions using a model for the binary size of the Linux kernel and leave other questions, such as applying their approach to performance models, for future work.

Analyzing the prevalence of configuration-related performance bugs, Han and Yu [23] highlight the lack of configuration-aware regression testing suitable for performance. They suggest that static analysis could be used to find which configurations are affected by a performance-critical change. This is exactly the goal of ConfFLARE.

Static Analysis of Configurable Software Systems. Dealing with configurability is a long-lasting challenge for static analysis. A common technique is to address configurability in static analysis is *family-based analysis*, which considers all possible configurations of a software system at once [49]. In a large-scale study, von Rhein et al. [54] have shown that family-based analyses outperform most sampling based techniques in both efficiency and effectiveness. Such family-based analyses

can be automatically be constructed by lifting existing analyses [12]. This can be achieved by encoding variability as choices, e.g., of types or data structures, between labeled alternatives [55]. For example, Brabrand et al. [10] present five different ways how a existing data-flow analysis can be adapted to analyze all configurations of a system. Bodden et al. [9] use *Interprocedural Distributive Environments* (IDE) to lift existing *Interprocedural Finite Distributive Subset* (IFDS) analyses to family-based analyses. Midtgaard et al. [39] introduce a systematic method for compositional derivation of sound family-based analyses based on abstract interpretation. Lanna et al. [30] developed a family-based strategy for reliability analysis of configurable software systems. More recently, Dimovski et al. [16] proposed a parameterized lifted analysis domain based on decision trees for analyzing numerical program families and later extended their work to deal with runtime configurable systems [15]. ConfFLARE can also be classified as a family-based approach, as our focus on runtime configurable systems allows our data-flow analysis to reason about all configurations at once.

Regression Testing. Regression testing has been an active research area for many years. In this area, a lot of focus has been put on selecting [28] and prioritizing [34] test cases and minimizing test suites [50]. Testing configurable software systems largely focuses on combinatorial interaction testing of features for which many configuration selection strategies have been proposed [17]. Apart from that, more specialized techniques exist for selecting and prioritizing configurations [2–4, 43], as well as generating and optimizing test suites [7, 11]. However, these techniques are mostly designed for single revision testing and do not take advantage of a regression testing setting. Closest to our work is the work by Qu et al. [42]: Similar to ConfFLARE, they use a static change impact analysis to determine which features are affected by a change. Then, they select a set of configurations that achieves pair-wise interaction coverage for the affected features. In contrast to our work, they only track change impact at the less precise function level (i.e., along the call graph). Also, they do not consider performance-relevant code, since their focus lies on traditional software (fault) testing. Therefore, we provide a more efficient way to detect performance regressions in configurable software systems by focusing on the configurations that are affected by a change.

7 Conclusion

Continuous performance regression testing of configurable software systems remains a challenging task due to the complex interplay between the system's configuration, its evolution, and other external factors. In this work, we presented ConfFLARE, an approach that aims at reducing the cost for performance regression testing of configurable software systems by focusing efforts on changes and features that are likely to affect the system's performance behavior. ConfFLARE employs a data-flow analysis to detect performance-relevant interactions between code changes and performance-relevant code and extracts feature information from interacting program paths to localize the part of the configuration space that is potentially affected by such changes. In an empirical study using both synthetic and real-world software systems, we demonstrate that ConfFLARE is effective in detecting performance-relevant changes and in identifying relevant features for performance regression analysis. While useful on its own, ConfFLARE can also be combined with complementary techniques, such as sampling, performance learning, or test suite optimization, to further improve the efficiency of performance regression testing in configurable software systems. How exactly such combinations can be realized and how they perform in practice is an interesting avenue for future research.

References

[1] Mathieu Acher, Hugo Martin, Luc Lesoil, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi, Olivier Barais, and Juliana Alves Pereira. 2022. Feature Subset Selection for Learning Huge Configuration Spaces: The Case of Linux Kernel Size. In *Proc. Int. Software Product Line Conf. (SPLC)*. ACM.

[2] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. 2016. IncLing: Efficient Product-Line Testing Using Incremental Pairwise Sampling. In *Proc. Int. Conf. Generative Programming: Concepts and Experiences*. ACM.

[3] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. 2019. Effective Product-Line Testing Using Similarity-Based Product Prioritization. *Softw. Syst. Model* 18, 1 (2019), 499–521.

[4] Mustafa Al-Hajjaji, Thomas Thüm, Jens Meinicke, Malte Lochau, and Gunter Saake. 2014. Similarity-Based Prioritization in Software Product-Line Testing. In *Proc. Int. Software Product Line Conf. (SPLC)*. ACM.

[5] José B. Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2016. Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC. In *Proc. Int. Conf./ Fast Software Encryption (FSE)*. Springer-Verlag.

[6] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer-Verlag.

[7] Hauke Baller and Malte Lochau. 2014. Towards Incremental Test Suite Optimization for Software Product Lines. In *Proc. Int. Workshop Feature-Oriented Software Development*. ACM.

[8] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wasowski. 2013. A survey of variability modeling in industrial practice. In *Proc. Int. Working Conf. Variability Modelling of Software-Intensive Systems (VAMOS)*. ACM.

[9] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. SPL$^{\mathrm{LIFT}}$: Statically Analyzing Software Product Lines in Minutes Instead of Years. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*. ACM.

[10] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. 2013. Intraprocedural Dataflow Analysis for Software Product Lines. *LNCS Trans. Aspect Oriented Softw. Dev.* 10 (2013), 73–108.

[11] Johannes Bürdek, Malte Lochau, Stefan Bauregger, Andreas Holzer, Alexander von Rhein, Sven Apel, and Dirk Beyer. 2015. Facilitating Reuse in Multi-goal Test-Suite Generation for Software Product Lines. In *Proc. Int. Conf. Fundamental Approaches to Software Engineering*. Springer-Verlag.

[12] Thiago M. Castro, Leopoldo Teixeira, Vander Alves, Sven Apel, Maxime Cordy, and Rohit Gheyi. 2021. A Formal Framework of Software Product Line Analyses. *ACM Trans. Softw. Eng. Methodol.* 30, 3 (2021), 34:1–34:37.

[13] Jiezhu Cheng, Cuiyun Gao, and Zibin Zheng. 2023. HINNPerf: Hierarchical Interaction Neural Network for Performance Prediction of Configurable Systems. *ACM Trans. Softw. Eng. Methodol.* 32, 2 (2023), 46:1–46:30.

[14] Marcin Copik, Alexandru Calotoiu, Tobias Grosser, Nicolas Wicki, Felix Wolf, and Torsten Hoefler. 2021. Extracting Clean Performance Models from Tainted Programs. In *Proc. Symp. Principles and Practice of Parallel Programming (PPoPP)*. ACM.

[15] Aleksandar S. Dimovski and Sven Apel. 2021. Lifted Static Analysis of Dynamic Program Families by Abstract Interpretation. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[16] Aleksandar S. Dimovski, Sven Apel, and Axel Legay. 2022. Several Lifted Abstract Domains for Static Analysis of Numerical Program Families. *Sci. Comput. Program.* 213 (2022), 102725.

[17] Ivan do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana de Almeida. 2014. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *Inf. Softw. Technol.* 56, 10 (2014), 1183–1199.

[18] Luciana Brasil Rebelo dos Santos, Érica Ferreira de Souza, André Takeshi Endo, Catia Trubiani, Riccardo Pinciroli, and Nandamudi Lankalapalli Vijaykumar. 2025. Performance Regression Testing Initiatives: A Systematic Mapping. *Inf. Softw. Technol.* 179 (2025), 107641.

[19] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*. ACM.

[20] Alexander Grebhahn, Christian Kaltenecker, Christian Engwer, Norbert Siegmund, and Sven Apel. 2021. Lightweight, Semi-Automatic Variability Extraction: A Case Study on Scientific Computing. *Empir. Softw. Eng.* 26, 2 (2021), 23.

[21] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wasowski, and Huiqun Yu. 2018. Data-Efficient Performance Learning for Configurable Systems. *Empir. Softw. Eng.* 23, 3 (2018), 1826–1867.

[22] Huong Ha and Hongyu Zhang. 2019. DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network. In *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE / ACM.

[23] Xue Han and Tingting Yu. 2016. An Empirical Study on Performance Bugs for Highly Configurable Software Systems. In *Proc. Int. Symp. Empirical Software Engineering and Measurement (ESEM)*. ACM.

[24] Capers Jones and Olivier Bonsignour. 2012. *The Economics of Software Quality*. Addison-Wesley.

[25] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. 2020. The Interplay of Sampling and Machine Learning for Software Performance Prediction. *IEEE Softw.* 37, 4 (2020), 58–66.

[26] Christian Kaltenecker, Stefan Mühlbauer, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. 2023. Performance Evolution of Configurable Software systems: an Empirical Study. *Empir. Softw. Eng.* 28, 6 (2023), 152.

[27] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proc. Int. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM.

[28] Rafaqut Kazmi, Dayang N. A. Jawawi, Radziah Mohamad, and Imran Ghani. 2017. Effective Regression Test Case Selection: A Systematic Literature Review. *ACM Comput. Surv.* 50, 2 (2017), 29:1–29:32.

[29] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. 2008. Implicit Flows: Can't Live with 'Em, Can't Live without 'Em. In *Proc. Int. Conf. Information Systems Security (ICISS)*. Springer-Verlag.

[30] André Lanna, Thiago M. Castro, Vander Alves, Genaína Nunes Rodrigues, Pierre-Yves Schobbens, and Sven Apel. 2018. Feature-family-based reliability analysis of software product lines. *Inf. Softw. Technol.* 94 (2018), 59–81.

[31] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. Int. Conf. Software Engineering (ICSE)*. ACM.

[32] Jörg Liebig, Andreas Janker, Florian Garbe, Sven Apel, and Christian Lengauer. 2015. Morpheus: Variability-Aware Refactoring in the Wild. In *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE.

[33] Max Lillack, Christian Kästner, and Eric Bodden. 2014. Tracking Load-Time Configuration Options. In *Proc. Int. Conf. Automated Software Engineering (ASE)*. ACM.

[34] Yiling Lou, Junjie Chen, Lingming Zhang, and Dan Hao. 2019. A Survey on Regression Test-Case Prioritization. *Adv. Comput.* 113 (2019), 1–46.

[35] Alina Mailach, Janet Siegmund, Sven Apel, and Norbert Siegmund. 2026. Views on Internal and External Validity in Empirical Software Engineering: 10 Years Later and Beyond. In *Proc. Int. Conf. Software Engineering (ICSE)*. ACM.

[36] Hugo Martin, Mathieu Acher, Juliana Alves Pereira, Luc Lesoil, Jean-Marc Jézéquel, and Djamel Eddine Khelladi. 2022. Transfer Learning Across Variants and Versions: The Case of Linux Kernel Size. *IEEE Trans. Softw. Eng.* 48, 11 (2022), 4274–4290.

[37] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proc. Int. Conf. Software Engineering (ICSE)*. ACM.

[38] Ben Swarup Medikonda and Seetha Ramaiah Panchumarthy. 2009. A framework for software safety in safety-critical systems. *SIGSOFT Softw. Eng. Notes* 34, 2 (2009), 1–9.

[39] Jan Midtgaard, Claus Brabrand, and Andrzej Wasowski. 2014. Systematic Derivation of Static Analyses for Software Product Lines. In *Proc. Int. Conf. Modularity (MODULARITY)*. ACM.

[40] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. 2019. Accurate Modeling of Performance Histories for Evolving Software Systems. In *Proc. Int. Conf. Automated Software Engineering (ASE)*. IEEE.

[41] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. 2020. Identifying Software Performance Changes Across Variants and Versions. In *Proc. Int. Conf. Automated Software Engineering (ASE)*. ACM.

[42] Xiao Qu, Mithun Acharya, and Brian Robinson. 2012. Configuration Selection Using Code Change Impact Analysis for Regression Testing. In *Proc. Int. Conf. Software Maintenance (ICSM)*. IEEE.

[43] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. 2008. Configuration-Aware Regression Testing: An Empirical Study of Sampling and Prioritization. In *Proc. Int. Symp. Software Testing and Analysis (ISSTA)*. ACM.

[44] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comput. Sci.* 167, 1 (1996), 131–170.

[45] Florian Sattler. 2023. Understanding Variability in Space and Time: Analyzing Features and Revisions in Concert.

[46] Florian Sattler, Sebastian Böhm, Philipp Schubert, Norbert Siegmund, and Sven Apel. 2023. SEAL: Integrating Program Analysis and Repository Mining. *ACM Trans. Softw. Eng. Methodol.* 32, 5 (2023), 121:1–121:34.

[47] Larissa Schmid, Marcin Copik, Alexandru Calotoiu, Dominik Werle, Andreas Reiter, Michael Selzer, Anne Koziolek, and Torsten Hoefler. 2022. Performance-Detective: Automatic Deduction of Cheap and Accurate Performance models. In *Proc. Int. Conf. Supercomputing (ICS)*. ACM.

[48] Norbert Siegmund, Alexander Grebhahn, Sven Apl, and Christian Kästner. 2015. Performance-Influence Models for Highly Configurable Systems. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM.

[49] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1 (2014), 6:1–6:45.

[50] Saif ur Rehman Khan, Sai Peck Lee, Nadeem Javaid, and Wadood Abdul. 2018. A Systematic Review on Test Suite Reduction: Approaches, Experiment's Quality Evaluation, and Guidelines. *IEEE Access* 6 (2018), 11816–11841.

[51] Alexander v. Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. 2016. Variability Encoding: From Compile-Time to Load-Time Variability. *J. Log. Algebraic Methods Program.* 85, 1 (2016), 125–145.

[52] Miguel Velez, Pooyan Jamshidi, Florian Sattler, Norbert Siegmund, Sven Apel, and Christian Kästner. 2020. ConfigCrusher: Towards White-Box Performance Analysis for Configurable Systems. *Proc. Int. Conf. Automated Software Engineering (ASE)* 27, 3 (2020), 265–300.

[53] 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. Conf. Software Engineering (ICSE)*. IEEE.

[54] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *IEEE Trans. Softw. Eng.* 27, 4 (2018), 18:1–18:33.

[55] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. 2014. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *Proc. Symp. New Ideas in Programming and Reflections on Software (Onward!)*. ACM.

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

[57] Mark Weiser. 1984. Program Slicing. *IEEE Trans. Softw. Eng.* SE-10, 4 (1984), 352–357.