Noname manuscript No.
(will be inserted by the editor)

ConfigCrusher: Towards White-Box Performance Analysis for Configurable Systems

Miguel Velez · Pooyan Jamshidi · Florian Sattler · Norbert Siegmund · Sven Apel · Christian Kästner

Received: date / Accepted: date

Abstract Stakeholders of configurable systems are often interested in knowing how configuration options influence the performance of a system to facilitate, for example, the debugging and optimization processes of these systems. Several black-box approaches can be used to obtain this information, but they either sample a large number of configurations to make accurate predictions or miss important performance-influencing interactions when sampling few configurations. Furthermore, black-box approaches cannot pinpoint the parts of a system that are responsible for performance differences among configurations. This article proposes ConfigCrusher, a white-box performance analysis that inspects the implementation of a system to guide the performance analysis, exploiting several insights of configurable systems in the process. ConfigCrusher employs a static data-flow analysis to identify how configuration options may influence control-flow statements and instruments code regions, corresponding to these statements, to dynamically analyze the influence of configuration options on the regions' performance. Our evaluation on 10 configurable systems shows the feasibility of our white-box approach to more efficiently build performance-influence models that are similar to or more accurate than current state of the art approaches. Overall, we showcase the benefits of white-box

Miguel Velez
Carnegie Mellon University

Pooyan Jamshidi
University of South Carolina

Florian Sattler
Saarland University

Norbert Siegmund
Leipzig University

Sven Apel
Saarland University, Saarland Informatics Campus

Christian Kästner
Carnegie Mellon University

performance analyses and their potential to outperform black-box approaches and provide additional information for analyzing configurable systems.

Keywords configurable systems · performance analysis · static analysis · dynamic analysis

1 Introduction

Most of today's software systems, such as databases, Web servers, processing libraries, and compilers, provide configuration options to satisfy a large variety of requirements (Apel et al., 2013; Jamshidi et al., 2017b; Kolesnikov et al., 2018; Siegmund et al., 2015; Xu et al., 2015). To this end, stakeholders select specific values for each option to obtain the desired functional properties and quality attributes in the system. However, this configuration process is often a difficult task, especially when lacking knowledge of how the configuration options influence the functionality and qualities of the system (Apel et al., 2013; Siegmund et al., 2015; Xu et al., 2013). For this reason, users, developers, and administrations typically resort to default configurations or change individual options in a trial-and-error fashion without understanding the resulting effect (Hubaux et al., 2012; Jamshidi and Casale, 2016; Jin et al., 2014; Xu et al., 2015).

Performance is one of the many interesting qualities of such systems. Understanding how individual configuration options and their combinations influence the performance of the system would facilitate the reasoning, debugging, adaptation, and optimization processes of these systems (Han and Yu, 2016; Han et al., 2018; Kolesnikov et al., 2018; Siegmund et al., 2015; Wang et al., 2018; Xu et al., 2013; Zhu et al., 2017). For example, users can find the configuration that performs an execution the fastest, and developers can find configuration options that cause excessive execution time when debugging the system.

One research area has focused on understanding the influence of options and their interactions on the performance of a configurable system by building a *performance-influence model* (Siegmund et al., 2015), which describes the performance of a system in terms of its configuration options for a specific workload and in a specific environment (e.g., on given hardware). Most prior work on deriving performance-influence models uses *black-box approaches* (Hutter et al., 2011; Olaechea et al., 2014; Siegmund et al., 2012b, 2015), which consider the system as a black box and measure for how long the system executes in different configurations. These approaches sample a subset of the configurations of a system and extrapolate a model based on the corresponding end-to-end measurements. The model's accuracy and cost depend on the approaches' sampling strategy (i.e., which configurations to measure) and the algorithm used for learning (Kolesnikov et al., 2018). Sampling is particularly important: The accuracy of a model might be low if the sample set does not capture performance-influencing interactions among options. Several different

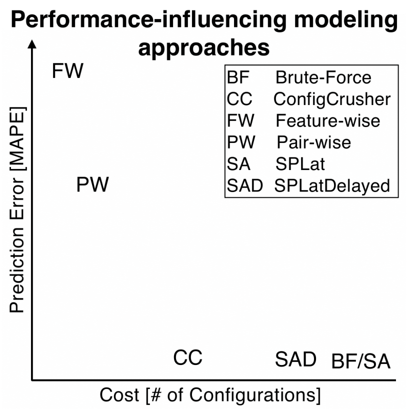


Fig. 1: Our conjecture on cost and prediction error comparison between state of the art approaches.

sampling strategies with different cost-accuracy tradeoffs have been explored (cf. Fig. 1).

We argue that a *white-box approach* that analyzes the system's source code can provide additional insights and guide the performance analysis to relevant options and interactions. Where black-box approaches are blind to the internals of an implementation, white-box approaches can, for example, identify options interacting in control-flow statements, and thus focus measurements on fewer, but more relevant configurations, thus promising to build more accurate models at lower cost (cf. Fig. 1). In addition, analyzing the implementation and performing measurements with regard to regions of the system rather than black-box end-to-end measurements allows us to pinpoint which regions of a system (e.g., set of statements influenced by the same set of options) are responsible for performance differences among configurations (i.e., more informative models), which can further help in performance debugging and optimization.

In this article, we introduce a novel white-box performance analysis approach for configurable systems, named ConfigCrusher. It combines static and dynamic analyses to identify and efficiently measure configurations that are relevant for accurate performance modeling. Specifically, ConfigCrusher (1) uses *static data-flow analysis* to trace the effect that configuration options may directly or indirectly have on a system's control-flow statements (including loops) and (2) *instruments* the system at configuration-relevant control-flow statements to measure the performance per region in concrete executions in a small set of selected configurations. One key benefit of measuring performance per region is that, in a single executed configuration, we independently measure the influence of multiple options on different regions, a process we call *compression.* Overall, ConfigCrusher exploits multiple insights about configurable systems, established in several prior studies (Jamshidi et al., 2017a; Kim et al., 2011, 2013; Kolesnikov et al., 2018; Lillack et al., 2018; Meinicke et al., 2016; Nguyen et al., 2016; Reisner et al., 2010; Siegmund et al., 2012a, 2013):

(a) *Irrelevance*: Not all options influence the performance of a system on a given workload. ConfigCrusher's data-flow analysis identifies options that do not influence the execution, reducing the number of configurations to sample.

(b) *Orthogonality*: Not all options interact with each other. ConfigCrusher's data-flow analysis identifies options that are orthogonal and can thus be measured together in a single execution, reducing the number of configurations to sample.

(c) *Low Interaction Degree*: Considering interactions is essential for accurate performance-influence models, but most options tend to interact only with few other options. ConfigCrusher's analysis identifies which interactions can occur, focusing the sampling towards performance-relevant configurations.

Compared to state of the art black-box approaches, ConfigCrusher *reduces the cost* of performance modeling while *preserving or increasing the accuracy* of the resulting models. Guided by program analysis, it will often measure fewer performance-relevant configurations, and, with instrumentation and compression, each measurement can provide information about multiple options and interactions. Furthermore, in addition to traditional performance-influence models, ConfigCrusher builds a *local performance-influence model* for each region of a system. These models can provide additional fine-grained information and insights to developers, for enhanced debugging and understanding of the performance behavior of a system.

To demonstrate the potential of our white-box approach, we implemented ConfigCrusher for Java systems. using the static taint-analysis engine *FlowDroid* (Arzt et al., 2014) for tracking configuration options. We show that our white-box approach outperforms existing state of the art approaches (i.e., more accurate models at lower cost) on 10 configurable systems. Due to known scalability issues with precise and accurate static data-flow analyses (Arzt et al., 2014; Avdiienko et al., 2015; Bodden, 2018; Do et al., 2017; Lerch et al., 2015; Pauck et al., 2018; Qiu et al., 2018; Wang et al., 2016), we limit our evaluation to relatively small, but still real-world open-source applications from different domains, including command-line programs, processing libraries, databases, and software product lines. Nevertheless, our evaluation still provides evidence for the feasibility of the underlying ideas. More generally, we show the potential benefits of analyzing the system structure to help in the understanding, debugging, and optimization of configurable systems and provide a foundation for future research to scale white-box performance analyses.

In summary, we make the following contributions:

– A white-box program analysis, combining data-flow and control-flow analysis and dynamic instrumentation for fine-grained performance measurement to identify how options affect the execution of configurable systems, exploiting insights about common characteristics of such systems (Sec 3).
– A compression technique that allows us to independently infer the influence of options and their interactions on multiple regions of a system's execu-

tion and a corresponding method to build accurate performance-influence models (Sec 3.2).

- An optimization to reduce the overhead of the instrumented systems that we analyze (Sec 3.3).
- An empirical evaluation of ConfigCrusher on 10 systems demonstrating the feasibility and potential of a white-box approach to reduce the cost and increase the accuracy of performance modeling compared to state of the art performance modeling approaches (Sec 5).
- A public open-source implementation of ConfigCrusher and reimplementations and improvements of prior state of the art approaches for Java systems (Velez et al., 2019).
- A replication package with technical information of the systems analyzed, environmental setup for experiments, analysis scripts, and data of several months of measurements (Velez et al., 2019).

2 Performance Modeling of Configurable Systems

A system's performance and, often directly correlated, energy consumption are important concerns for many software systems. While it is possible to design and optimize a system's implementation for a specific task, much of our modern software is built on top of reusable (open-source) infrastructure software, such as databases, Web servers, video encoders, and so forth. However, a single reusable infrastructure component will rarely ever satisfy all stakeholders equally ("no one size fits all"); for example, developers needing a data-storage component for a write-heavy scenario would likely make different implementation decisions than for a read-heavy scenario, if they would implement such component from scratch. To resolve this tension between a single reusable component and custom implementations for different stakeholders, reusable components often have a large number of configuration options that defer design decisions and allow users to choose between different implementations and different resulting functionality and quality tradeoffs to meet their needs (e.g., workloads and environments).

When reusing such infrastructure components with many options, making suitable configuration decisions can be challenging (Apel et al., 2013; Xu et al., 2015). Users are often unaware of the impact on options on various qualities or have only vague intuitions (Hubaux et al., 2012; Xu et al., 2013), and options may interact, producing surprising behavior (Siegmund et al., 2015). For example, a developer considering whether to enable encryption, would likely expect that encryption may slow down system performance, but may need to perform experiments to identify the severity of the effect, given the concrete system, workload, and other configuration decisions.

In this context, *performance-influence models* that explain how configuration options and their interactions influence the performance of a system, in a certain context (e.g., requirements and needs), can be helpful when making deliberate configuration decisions (e.g., optimizing performance for given workload or debugging the system's performance behavior). That is, performance-influence models have a *fundamentally different approach and goal* than traditional performance models, which typically model and analyze (e.g., using Queuing networks, Petri Nets, and Stochastic Process Algebras) the performance of a system's architecture under different workloads in the design stage of a project (Harchol-Balter, 2013; Serazzri et al., 2006), possibly also modeling design decisions as configuration options (Becker et al., 2009; Esfahani et al., 2013). In contrast, performance-influence models describe the performance behavior *of a given system implementation, with a given workload and environment*, in terms of configuration decisions and they are typically *learned from observing a specific system execution* under different configurations.

Performance-influence models are typically learned by fitting a model to explain the performance in terms of configuration options (Guo et al., 2013; Jamshidi et al., 2017a, 2018; Siegmund et al., 2015; Valov et al., 2017). The models can be used for performance debugging (Siegmund et al., 2015; Wang et al., 2018; Xu et al., 2013), optimization (Guo et al., 2013; Oh et al., 2017; Zhu et al., 2017), and adaptation (Jamshidi et al., 2017b, 2018; Wang et al., 2018; Zhu et al., 2017).

As a running example, the performance-influence model $\Pi = 1 + 3\mathtt{A} + 3\mathtt{AB} + 3\mathtt{AC}$ for the system in Figure 2 (Lines 1–15) suggests that the measured execution, with a given workload and environment, takes 1 second by default (Line 12), but 3 seconds longer if option `A` (compression) is selected (Lines 4 + 16) and another 3 seconds extra, each, if option `B` (page-size, Line 21) or `C` (encryption, Line 3) are selected together with `A`. Such model can help with several maintenance and understanding tasks. For example, users can optimize the execution time (e.g., deselect `A` in this example) or make an informed tradeoff decision, (e.g., whether the functionality of `A` is worth the 3 second overhead). Likewise, such models can generally help with system understanding such as identifying the interaction of `A` with `B` and `C` or recognizing that `A` might be a performance bottleneck. Similarly, we can use this model in a planning algorithm to adapt (i.e., dynamically reconfigure) the system, in response to changing requirements or environment conditions, such as high load or low battery (Aldrich et al., 2019; Weisenburger et al., 2017).

Performance-influence modeling is different from *performance tuning* approaches that attempt to change a system's configuration to optimize performance (possibly also considering multiple qualities with a suitable fitness function) for a given workload and environment (Hutter et al., 2011; Olaechea et al., 2014). These approaches search for a good configuration rather than building models of the entire configuration space. Performance tuning approaches for configurable systems conduct some sort of search in the configuration space, with and without building intermediate models to support the search (Jamshidi and Casale, 2016; Oh et al., 2017; Siegmund et al., 2012b), typically measuring the system execution under different configurations. For performance tuning, search is typically more efficient than building performance-influence models, because the focus is only on finding the fastest configuration, not on explaining why it is fast, not modeling the performance

```
 1 def foo(boolean x)
 2     // Begin region R1
 3     if(x) ... // execution: 4s          $\Pi_{R_1}$ = 1A + 3AC
 4     else ... // execution 1s
 5     // End region R1
 6 def main(List workload)
 7     a = getOpt("A"); b = getOpt("B");
 8     c = getOpt("C"); d = getOpt("D");
 9     e = getOpt("E"); f = getOpt("F");
10     g = getOpt("G"); h = getOpt("H");
11     i = getOpt("I"); j = getOpt("J");
12     ... // execution: 1s
13     boolean x = false;
14     // Begin region R2
15     if(a) // variable depends on option A
16         ... // execution: 2s
17         foo(c); // variable depends on option B
18         x = true;                        $\Pi_{R_2}$ = 2A
19     // End region R2
20     // Begin region R3
21     if(b && x) ... // execution: 3s    $\Pi_{R_3}$ = 3AB
22     // End region R3
23     if(d && e && f) ... // execution: 5s
24     if(a) ... // execution: 0.1s
25     if(b) ... // execution: 0.2s
26     if(c) ... // execution: 0.3s
27     if(d) ... // execution: 0.4s
28     if(e) ... // execution: 0.5s
29     if(f) ... // execution: 0.6s
30     if(g) ... // execution: 0.7s
31     if(h) ... // execution: 0.8s
32     if(i) ... // execution: 0.9s
```

Fig. 2: Running example system with three regions, indicated between red comments, influenced by configuration options. The comments indicate the execution time of the branch of each control-flow statement for a given workload, input size, and underlying hardware. For simplicity, we ignore the regions in Lines 23–32 through Sec. 4. Region 1 is influenced by a control-flow interaction and Region 3 by a data-flow interaction. The local performance-influence models are shown to the right of each region.

of slower configurations, and not characterizing the influence of options and interactions. In contrast, performance-influence models, the focus of this article, are more suited for tasks related to understanding, debugging, prediction, adaptation, and automated reasoning.

2.1 State of the art of building performance-influence models

Performance-influence models are typically created by selecting a set of configurations, measuring the performance of a system for each configuration (given a fixed workload and environment), and then fitting a model (e.g., a linear model) to explain the system's performance behavior in terms of its configuration options. The accuracy of performance-influence models is measured in terms of how well the models predict the performance of all configurations. The main cost driver for building performance-influence models is the need to execute and measure a potentially large number of sampled configurations – there is typically a tradeoff between accuracy and cost in that models trained on fewer samples are less expensive to build, but also less accurate (e.g., if the sampled configurations do not cover all relevant execution paths, cf. Fig. 1).

Essentially, all existing approaches for building performance-influence models are black box in that they do not consider the implementation of the system. The key differentiators are how configurations are sampled and how (and what kind of) models are fitted. Many combinations have been explored (Guo et al., 2013; Sarkar et al., 2015; Siegmund et al., 2012a,b, 2013, 2015), with different tradeoffs among applicability, cost, and accuracy (Kolesnikov et al., 2018).

By contrast, we propose to analyze the system's implementation (white-box) to guide the sampling to relevant configurations and to instrument the system to perform more fine-grained measurements, mapping performance influence of options to specific code regions. While there has been work on program analysis to identify how configuration options affect the execution of a system (Angerer et al., 2015; Kim et al., 2013; Lillack et al., 2018), usually for testing or system comprehension, beyond our own prior work with very limiting assumptions (e.g., no data-flow interactions, and exclusive to compile-time variability) (Siegmund et al., 2013), we are not aware of any work on using program analysis to inform performance-influence modeling. We conjecture that white-box strategies can achieve higher accuracy at lower cost, if the analysis identifies relevant options and interactions.

The insights on which we build this work are *Irrelevance* (not all options influence the performance of a system on a given workload), *Orthogonality* (not all options interact with each other), and *Low Interaction Degree* (most options tend to interact only with few other options) (Jamshidi et al., 2017a; Kim et al., 2011, 2013; Kolesnikov et al., 2018; Lillack et al., 2018; Meinicke et al., 2016; Nguyen et al., 2016; Reisner et al., 2010; Siegmund et al., 2012a, 2013). Other insights, such as prefix sharing and variational execution (Meinicke et al., 2016), could be exploited to efficiently build accurate models. However, these insights require special dynamic analysis techniques with excessive overhead, even for small systems, which is why we do not consider them in this work.

In the following paragraphs, we describe the state of the art approaches for building performance-influence models in terms of their sampling, measuring, and learning techniques, and to what degree they exploit the insights that we consider in this work. Table 1 summarizes the approaches and Table 2 compares the number of executions and accuracy of each approach when analyzing our running example (Fig. 2, Lines 1–21).

Brute Force is a black-box approach that samples *all* configurations of a system and measures the execution time of the system for a given workload *end-to-end*. It is rarely used in practice due to its obvious scalability issues for all but the smallest configuration spaces. Learning a performance-influence model from Brute Force executions is not necessary as we know the execution time of all configurations, although a simplified model could be learned for

Table 1: Comparison of the state of the art approaches.

Approach	Sampling	Measuring	Learning
Brute Force	Exhaustive	End-to-end	Not needed
SPLat	Distinct execution paths	End-to-end	Not needed
Sampling and Learning	Strategy based	End-to-end	Algorithm based
Family-Based	One configuration	Region-level	Not needed
ConfigCrusher	Static analysis-based	Region-level	Not needed

Table 2: Comparison of the cost and accuracy of the state of the art approaches when analyzing the running example in Fig. 2.

	Insights			Quality	
Approach	Irrelevance	Orthogonality	LID	Cost	Accurarcy
Brute Force	✗	✗	✗	1024	High
SPLat	✓	✗	✗	6	High
Sampling and Learning	✗	✓	✓	—[1]	—[1]
Family-Based	✗	✓	✗	1	—[2]
ConfigCrusher	✓	✓	✓	4	High

LID = Low Interaction Degree.

[1] Depends on sampling strategy, such as t-wise sampling (Medeiros et al., 2016)

[2] High accuracy in the absence of data-flow interactions

understanding (Kolesnikov et al., 2018). In our running example, among other inefficiencies, it will execute irrelevant configurations (e.g., all configurations that explore all values of options `D` – `J`).

SPLat (Kim et al., 2013) is a white-box testing approach that we repurpose for performance analysis.[1] By instrumenting the system, it dynamically tracks the configurations that produce distinct execution paths. It reexecutes the system until all configurations with distinct paths are explored. While it ignores irrelevant options, since they do not produce different paths, it explores all combinations of options that it encounters during execution; each time an option is reached in a new path, it explores both values for that option. Essentially, it is an *improved version* of the Brute Force approach. SPLat does infer from control-flow interactions that some options are only reachable when specific values are selected. In our running example, it will explore option `C` only when option `A` is enabled. Despite this benefit, it still often produces very large sets of configurations to sample, which can lead to scalability issues. Similar to the Brute Force approach, it measures the execution time *end-to-end* and learning a performance-influence model is not necessary.

Sampling and Learning approaches are combinations of a sampling technique; such as random sampling, feature-wise, pair-wise (Medeiros et al., 2016), design of experiments (Montgomery, 2006), or combinatorial sampling (Al-Hajjaji et al., 2016; Halin et al., 2018; Hervieu et al., 2011, 2016; Nie and Leung, 2011), to measure the *end-to-end* execution time of a *subset of the configuration space*, and a learning technique; such as regression, classification

[1] Though SPLat was designed for unit testing software product lines, the algorithm can be used to reduce the number of configurations to sample.

and regression trees (Guo et al., 2013; Sarkar et al., 2015; Siegmund et al., 2012a,b, 2015), or Gaussian Processes (Jamshidi et al., 2017b), to *extrapolate* a performance-influence model. The number of samples and accuracy of the learned model depend on the sampling strategy and learning algorithm. Although some sampling strategies rely on a coverage criteria to sample specific interaction degrees, such as t-wise sampling (Medeiros et al., 2016; Nie and Leung, 2011), they might miss important interactions leading to inaccurate models. In addition, due to their lack of insight of the internals of the system, none of these approaches recognizes irrelevant options.

Family-Based Performance Measurement (Siegmund et al., 2013), our own prior work, is currently the only white-box performance-influence modeling approach we are aware of. It uses a static mapping between options to code regions and instruments the system to *measure the execution time spent in the regions.* Subsequently, it *executes the system once* with all options enabled, tracking how much each option contributes to the execution time. The approach works well when all options only contribute extra behavior, but do not interact. Current implementations, however, derive the static map from compile-time variability mechanisms (preprocessor directives) (Siegmund et al., 2013) and could not handle our running example with load-time variability (i.e., loading and processing options in variables at runtime). Furthermore, the static map only covers direct control-flow interactions from nested preprocessor directives, and can lead to inaccurate models when data-flow interactions occur. In our running example (Fig. 2), data-flow analysis is needed to detect that the second if statement indirectly depends on option `A` (with implicit data-flow through variable `x`), leading to inaccurate performance-influence models otherwise.

All the surveyed approaches build performance-influence models with different levels of applicability, cost, and accuracy, but they either overapproximate or underapproximate the interactions in a system and configurations that need to be executed to build an accurate performance-influence model. Furthermore, none of the approaches can associate the resulting performance-influence model with regions in the source code, which can help to understand and debug individual components of a system. The only exception is the family-based approach, but it has severe limitations and assumptions of the systems it can analyze.

2.2 ConfigCrusher

We introduce **ConfigCrusher**, a new white-box approach that exploits the insights of *Irrelevance*, *Independence*, and *Low Interaction Degree*, which leads to a reduction in the cost to measure performance while also generating accurate and informative performance-influence models. Our approach improves upon the state of the art of performance-influencing modeling by using a *static analysis* to identify how *load-time configuration options* may influence regions in the system through *control-flow and data-flow dependencies.* Then, it derives a set of relevant configurations to *measure the execution time of regions*

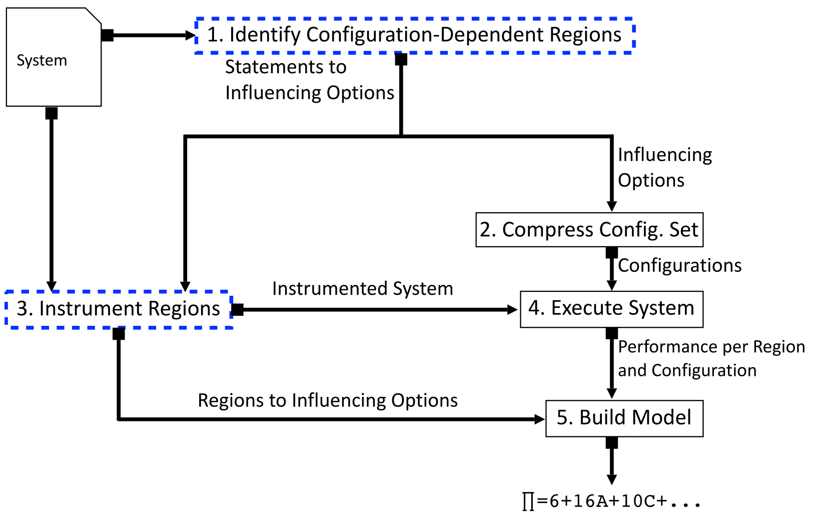


Fig. 3: Overview of ConfigCrusher's modular components. The components represented with solid boxes can be reused for analyzing systems implemented in any programming language. The dashed blue boxes indicate components that have to target specific programming languages.

and builds *local performance-influence models* that describe how options influence the execution time of those regions. Subsequently, the local models are aggregated to obtain a global performance-influence model for the system.

In our running example, ConfigCrusher will identify 3 *regions affected by configuration options* (Fig. 2) and use the options that influence the regions to *compress* the configuration space into a set of 4 *configurations* to be sampled (Table 3). Next, it will *instrument* the system's regions, reducing the instrumentation overhead through additional optimization (Sec. 3.3). Finally, it will *build* local performance-influence models (Sec. 3.5) for each region based on the performance observed when executing the instrumented system with the compressed set of configurations (Table 3) and, subsequently, aggregate them to produce an accurate global *performance-influence model*.

3 ConfigCrusher

The general idea of ConfigCrusher is to identify the regions (sets of statements influenced by a set of options from control-flow and data-flow dependencies) in the system that depend on configuration options, and use these options to generate a compressed set of configurations. The set is then used to measure the regions' performance to build an accurate performance-influence model. We proceed in five steps:

– *Identify Configuration-Dependent Regions* (Sec. 3.1): We perform a data-flow analysis that identifies the control-flow statements that depend on configuration options and the code regions affected by these statements.

– *Compress Configuration Set* (Sec. 3.2): We identify the smallest set of configurations that cover all relevant executions of all regions.
– *Instrument Regions with Optimizations* (Sec. 3.3): We instrument the regions in the system to track their execution time in different configurations and optimize the instrumentation to reduce measurement overhead.
– *Execute the Instrumented System* (Sec. 3.4): We execute the instrumented system to measure the performance of regions.
– *Build the Performance-Influence Model* (Sec. 3.5): We individually build local performance-influence models based on the measured code regions' performance and, subsequently, aggregate them to obtain a global performance-influence model.

3.1 Identifying Configuration-Dependent Regions

As first step, we identify the control-flow statements that depend on configuration options and the regions affected by these statements. To this end, we create the *statement influence map* SI from statements S to the set of options O that influence the execution of these statements ($SI : S \rightarrow \mathcal{P}(O)$). We use this map later to compress a set of configurations (Sec. 3.2) and to instrument the system (Sec. 3.3).

To obtain the statement influence map, we use a data-flow analysis to track how options are used in control-flow statements. That is, we track variables at API calls that load configuration options and then propagate them along control-flow and data-flow dependencies (including implicit flows). By tracking how each option flows through the system, we can identify, for each control-flow statement, the set of options that may influence this statement. Finally, we produce the map, mapping all statements in the branches of a control-flow statement to all influencing options.

Example: The options in our running example in Fig. 2 (Lines 1–21) are the fields $\texttt{A} - \texttt{J}$. Lines 6–13 are not influenced by any options, Lines 3–4 are influenced by the set of options $\{\texttt{A}, \texttt{C}\}$, Lines 15–18 by $\{\texttt{A}\}$, and Line 21 by $\{\texttt{A}, \texttt{B}\}$.

We can reason about *Irrelevance*, *Orthogonality*, and *Low Interaction Degree* with this data-flow analysis: Options that influence no control-flow statements are irrelevant and never appear in the resulting map. Likewise, we can identify which set of options interact on which control-flow statements and detect both orthogonality and low interaction degree. For example, in our running example, we learn that option `A` interacts with `B` and `C` separately, but not together, and that options $\texttt{D} - \texttt{J}$ are irrelevant in the system.

3.2 Compressing Configuration Set

Based on the statement influence map, we now calculate the *compressed set of configurations* ($CC \subseteq \mathcal{P}(O)$) that will be executed to measure the perfor-

Algorithm 1: Compression of configuration set

```
   Input: Influencing options $IO : \mathcal{P}(\mathcal{P}(O))$
   Output: Compressed set of configurations $CC : \mathcal{P}(C)$
1  Function compress_configuration_set($IO$)
2      $unique\_opts$ := unique_options($IO$) // Get unique sets of options
       // Remove subsets of other sets
3      $unique\_opts$ := remove_subsets($unique\_opts$)
4      $options\_to\_confs$ := new Map()
5      for $o \in unique\_opts$ do
6          $confs$ := configurations($o$) // Get all configurations
7          $options\_to\_conf$.put($o$, $confs$)
8      end
9      $o_1 := \varnothing$, $cs_1 := \varnothing$
10     for $o_2$, $cs_2 \in options\_to\_confs$ do
11         $CC := \varnothing$, $pivot := o_1 \cap o_2$
12         while $c \in cs_1 \wedge cs_1$.hasNext() $\wedge\ cs_2$.hasNext() do
13             $pv$ := pivot_value($pivot$, $c$) // Get value of pivot
14             $c_2$ := conf_with_pv($cs_2$) // Get conf. with value of pivot
15             $CC$.add($c_1 \cup c_2$)
16         end
17         $CC$.add_remaining($cs_1$), $CC$.add_remaining($cs_2$)
18         $o_1 := o_1 \cup o_2$, $cs_1 := CC$
19     end
20     return $CC$
21 end
```

mance of the regions in the system. We use the set of all interactions (image of *SI* from Sec. 3.1, $IO : \mathcal{P}(\mathcal{P}(O))$) to generate this set of configurations and use it later to execute the instrumented system (Sec. 3.4).

Intuitively, our goal is to execute the system such that *each region* is executed for *every combination of options* involved in that region, while minimizing the overall number of configurations to execute. Since different regions may be influenced by different options (orthogonality), we can execute them in the same configurations, in a process we call *compression.* The challenge is similar to finding covering arrays in combinatorial interaction testing, such as covering all combinations of pairs of options (Al-Hajjaji et al., 2016; Halin et al., 2018; Hervieu et al., 2011, 2016; Kuhn et al., 2013). However, we need to cover different interaction strengths for different sets of options depending on which combinations of options have been detected in our statement influence map.

We developed a heuristic compression algorithm (Algorithm 1) to find and compress a set of configurations that we use to measure the performance of the system. First, we select all unique sets of options that are not subsets of other sets (Lines 2–3) and calculate all combinations of each set (Lines 5–8)—these are the minimum combinations we need to cover. Next, we compress the set of configurations (Lines 10–19) by iteratively merging the partial configurations around the options that are common between two sets of options (i.e., the pivot).

Example: In our running example (Lines 1–15), the regions are influenced by the sets of options $\{A\}$, $\{A, B\}$, and $\{A, C\}$. That is, we need to cover two

Algorithm 2: Identify regions

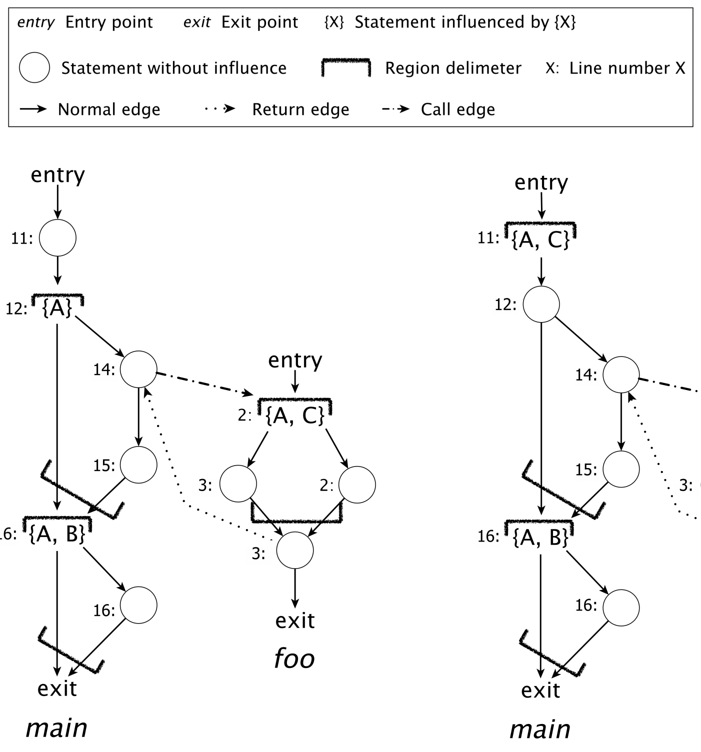
```
Input: Control-flow graph $CFG$, Statement-influence map $SI : S \rightarrow \mathcal{P}(O)$
Output: System with instrumented regions $R \rightarrow \mathcal{P}(E) \times \mathcal{P}(E)$, Regions to
        influencing options $R \rightarrow \mathcal{P}(O)$
1  Function identify_regions($CFG, SI$)
2      for each $stmt \in$ statements($CFG$)
3          $idom :=$ idom($stmt, CFG$) // Get immediate dominator
           // influence($s, SI$): $S \rightarrow \mathcal{P}(O)$
4          if influence($stmt, SI$) $\neq \varnothing \wedge$ influence($stmt, SI$) $\neq$ influence($idom,$
             $SI$) then
5              $r :=$ new Region()
               // Omit incoming edges from loops
6              for each $edge \in$ in($stmt, CFG$)
7                  start($r, edge, stmt$) // Map $r \rightarrow e$ and $r \rightarrow \mathcal{P}(\mathcal{P}(O))$
8              end
9              $pdom :=$ ipdom($stmt, CFG$) // Get immediate post-dominator
10             while influence($stmt, SI$) = influence($pdom, SI$) do
11                 $pdom :=$ ipdom($pdom, CFG$)
12             end
13             for each $edge \in$ in($pdom$)
14                 end($r, edge, stmt$) // Map $r \rightarrow e$ and $r \rightarrow \mathcal{P}(\mathcal{P}(O))$
15             end
16         end
17     end
18 end
```

combinations for $\{A\}$ (with `A` enabled and disabled), four combinations of $\{A, B\}$, and four combinations of $\{A, C\}$. The four combinations of $\{A, B\}$ already subsume the two configurations of $\{A\}$. Furthermore, based on the pivot $\{A\}$ of the remaining sets, we can create a merged compressed set of four configurations that still cover all interactions of `A` with `B` and `A` with `C`: $\{\{\}, \{B, C\}, \{A\}, \{A, B, C\}\}$.

Note how compression exploits *Irrelevance* and *Orthogonality*: It does not consider irrelevant options (e.g., `D`) and does not consider the combinations of options that do not interact (e.g., `B` and `C`). The size of the compressed set is dominated by the size of the largest interaction (at least 2^n configurations for an interaction among n options; $n = 2$ in our running example), which is often moderate due to *Low Interaction Degree*. At the same time, independent interactions of the same size can often be merged effectively.

3.3 Instrumenting Regions with Optimizations

Next, we instrument the system to measure its performance broken down by code regions. As part of the instrumentation, we identify and optimize the actual regions used for measurement, derived from the statement influence map (Sec. 3.1). We subsequently execute the instrumented system (Sec. 3.4) with the compressed set of configurations (Sec. 3.2) to build the performance-influence model (Sec. 3.5).



(a) Unoptimized (b) Optimized

Fig. 4: Unoptimized and optimized instrumented control-flow graphs of the methods of Fig. 2. For simplicity, statements within the regions and those before line 9 in method `main` are ignored.

A region is a set of statements influenced by the same set of options, identified by a set of control-flow edges that start the region and another set of edges that end it. Algorithm 2 calculates the regions and their start (Line 3) and end edges (Lines 4–16) in a method. A region starts before the first statement influenced by a set of options (indicated by the statement influence map) and ends after the last statement influenced by the same set of options. One task of the algorithm is to find the end of a region where all the paths originating from a control-flow statement meet again (i.e., the immediate post-dominator) (Lines 9–12). The algorithm obtains the immediate post-dominator and continuously searches for the next one until it finds the last statement with the same influence as the current control-flow statement.

After identifying all regions, we instrument the start and end edges of these regions with statements to log their execution time and measure their influence on performance. We also instrument the entry point of the system to measure the performance of code not influenced by any options. The result of executing an instrumented system is the total time spent in each region.

Example: Fig. 4a shows the three regions that we instrument in the control-flow graphs of our running example in Fig. 2. One region contains statements $\{12, 14, 15\}$ since statement 15 is influenced by $\{A\}$ and its last post-dominator without that influence is 21.

Optimization. Although Algorithm 2 can identify regions in a system, we observed excessive overhead in execution even in small systems (see Sec. 5.3). We found that the overhead arose from redundant, nested regions (i.e., regions with the same set of influencing options), and regions executed repeatedly in loops. Subsequently, we identified optimizations to reduce measurement overhead through instrumenting different regions without altering the performance-influence model that we produce. Specifically, we perform optimizations that preserve the following two invariants.

Invariant 1 (Expand regions): *Statements not influenced by options can be added to a region without altering the performance-influence model that is generated and without increasing measurement effort.* Statements not influenced by options contribute the same execution time to all configurations. Therefore, including these statements in a region increases the execution time of the region equally for all configurations, but does not affect the performance difference among configurations used to build the performance-influence model.

Example: Consider the statement in Line 12 and Region R_2 in our running example. The statement takes 1 second to execute and Region R_2 takes 2 seconds to execute when option `A` is enabled, from which we can derive the partial performance-influence model $\Pi_{R_2} = 1 + 2\texttt{A}$. Since the statement is not influenced by any options, we can include it in the region, now observing 1 or 3 seconds executions depending on whether `A` is enabled, preserving the same 2 seconds difference and resulting in the same model.

Invariant 2 (Merge regions): *$GI : \mathcal{P}(\mathcal{P}(O))$ is the set of all interactions in the system. Two consecutive regions or an outer and an inner region influenced by interactions $i_1 \in GI$ and $i_2 \in GI$ can be merged if $i_1 \cup i_2 \in GI$ without altering the performance-influence model that is generated and without increasing measurement effort.* Merging two consecutive regions or an outer and an inner region forms an interaction between the options that influence both regions. Therefore, we have to sample all combinations of the interaction to obtain their influence on the region. If that interaction is already present in the system, we already sample all these configurations anyway. Therefore, we can merge these regions into one that is influenced by the interaction of the two regions. As stated in invariant 1, merging does not affect the absolute performance difference used to build the performance-influence model. By merging regions, especially nested regions within loops, we *significantly* reduce the number of regions that are executed, which *significantly* reduces the overhead of measuring the instrumented system.

Algorithm 3: Propagate influence down

```
  Input: Statement stmt, Control-flow graph CFG, Statement-influence map
         $SI : S \to \mathcal{P}(O)$
  Output: Optimized statement-influence map $S \to \mathcal{P}(O)'$
1 Function propagate_down(stmt, CFG, SI)
2     ipdom := ipdom(stmt, CFG) // Get immediate post-dominator
      // Get set of statements in all paths
3     pstmts := paths_stmts(stmt, ipdom) − ipdom
4     for each $ps \in pstmts$
          // influence(ps, SI): $S \to \mathcal{P}(O)$
5         if influence(ps, SI) $\subset$ influence(stmt, SI) then
6             influence(ps, SI) := influence(stmt, SI)
7         end
8     end
9 end
```

Algorithm 4: Propagate regions up

```
  Input: Statement stmt, Control-flow graph CFG, Statement-influence map
         $SI : S \to \mathcal{P}(O)$, Set of all interactions in the system $GI : \mathcal{P}(O)$
  Output: Optimized statement-influence map $S \to \mathcal{P}(O)'$
1 Function propagate_up(stmt, CFG, SI, GI)
2     for each $pred \in$ preds(stmt, CFG)
          // influence(s, SI): $S \to \mathcal{P}(O)$
3         if influence(pred) $\cup$ influence(stmt) $\in GI \land$ influence(pred) $\neq$
            influence(pred) $\cup$ influence(stmt) then
4             influence(pred) := influence(stmt)
5         end
6     end
7 end
```

Example: Consider regions R_2 and R_3 in our running example. Region R_2 is influenced by $\{A\}$ and region R_3 by $\{A, B\}$. We sample all 4 combinations of `A` and `B` to conclude that Region R_2 takes 2 seconds to execute when option `A` is enabled and Region R_3 takes 3 seconds when both `A` and `B` are enabled, resulting in the partial performance-influence model $\Pi_{R_2+R_3} = 2\texttt{A} + 3\texttt{AB}$. Since we already sample all configurations for interaction $\{A, B\}$ and since $\{A\} \cup \{A, B\}$ does not create a new interaction, we can merge both regions into one that is influenced by interaction $\{A, B\}$ without having to sample more configurations. In this case, the merged region would take 2 seconds when `A` is enabled and 5 seconds when both `A` and `B` are enabled, resulting in the same performance-influence model when we calculate the actual influence of enabling both `A` and `B` (i.e., +3 seconds). With the same reasoning, we can also merge regions 1 and 2 in our running example.

We developed two algorithms (Algorithm 3 and Algorithm 4) that use the invariants to propagate the options that influence statements up and down a control-flow graph (i.e., intraprocedually), as well as across graphs (i.e., interprocedually), to expand, merge, and pull out regions. The propagation in Algorithm 3 merges consecutive regions (Lines 4–8) and expands where regions

Table 3: Configuration performance map of the optimized region of Fig. 2. For simplicity, the measurement noise was removed.

Configurations				Regions		
A	B	C	D	Base (s)	$OR_1 \equiv \{A, C\}$ (s)	$OR_2 \equiv \{A, B\}$ (s)
F	F	F	F	1	0	0
F	T	T	F	1	0	0
T	F	F	F	1	3	0
T	T	T	F	1	6	3

OR_1: Optimized Region 1; OR_2: Optimized Region 2.

end. The propagation in Algorithm 4 pulls out nested regions and expands where regions start. Obeying our invariants, both algorithms never create new interactions nor do they alter the performance-influence model that we generate, but significantly reduce the overhead of measuring the instrumented system. After propagation, we identify the regions and instrument them as before (Algorithm 2).

The propagation algorithms are non-deterministic (i.e., different results are obtained depending on the order in which regions are merged). In fact, different orderings can be used to optimize for different goals. Assuming that most of the overhead occurs in nested regions, especially those inside loops, we prioritize pulling regions out of loops. (We experimented with other orderings and the results were similar to Table 7). Fig. 4b presents an optimized instrumentation that prioritizes our goal, in which we pulled out the region in the callee.

3.4 Executing the Instrumented System

After instrumentation, we can now execute the system with the compressed set of configurations (CC, Sec. 3.2) and track execution times for each region. We produce a *configuration performance map* CP, which maps each region R in each executed configuration to a corresponding execution time T ($CP : CC \to (R \to T)$).

At the start and end of every region, we record the current time and log the difference as the execution time of the region. Since regions might be nested during execution, we also keep a stack of regions at runtime and subtract the time of nested regions from the time of outer regions. This additional step can become a source of overhead for deeply nested regions, which is what we observed in the unoptimized instrumented systems. We tried building a trace of regions and processing the execution times after the system finished executing. However, due to the large number of regions that were executed, the systems ran out of memory. Our evaluation shows that the dynamic processing incurs low overhead (Sec. 5.3).

Example: Table 3 presents a configuration performance map for the two optimized regions and compressed set of four configurations of our running example.

3.5 Building the Performance-Influence Model

Our final step is to build the *performance-influence model* Π that predicts the performance of each configuration ($\Pi : C \rightarrow T$), based on the configuration performance map ($CP : CC \rightarrow (R \rightarrow T)$, Sec. 3.4) and the region influence map ($RI : R \rightarrow \mathcal{P}(O)$, Sec. 3.3).

To build the global performance-influence model, we first build local models for each region separately and subsequently aggregate them. A local model contains performance terms for all combinations of options that are associated with the region (using RI), in the form $\Pi_r = t_1XY + t_2X\neg Y + t_3\neg XY + t_4\neg X\neg Y$ for a region r with options `X` and `Y` (or $\Pi_r = t_4 + (t_2 - t_4)X + (t_3 - t_4)Y + (t_1 - t_2 - t_3 + t_4)XY$ to highlight the influence of options and avoid negated terms) (Siegmund et al., 2012a). If a region has been executed multiple times for the same combination of options in one configuration, the execution time t_i should not differ beyond usual measurement noise (since other options should not influence the region), thus we average the execution time.

The global performance-influence model Π is obtained by aggregating all local performance-influence models. Note that local models can be useful for understanding and debugging the individual regions in the system.

Example: With the measurement times in Table 3, we build the local models of the base region as $\Pi_{\texttt{Base}} = \texttt{1}$ (averaged over 4 executions) and of the other two regions as $\Pi_{\texttt{OR}_1} = \texttt{3A}\neg\texttt{C + 6AC} = \texttt{3A + 3AC}$ and $\Pi_{\texttt{OR}_2} = \texttt{3AB}$, resulting in the overall model $\Pi = 1 + 3\texttt{A} + 3\texttt{AB} + 3\texttt{AC}$.

4 Implementation

To show the feasibility of our white-box approach, we implemented ConfigCrusher for Java systems and made it publicly available (Velez et al., 2019). Its modular design allows ConfigCrusher to analyze systems in any programming language; only the data-flow analysis (Sec. 3.1) and instrumentation (Sec. 3.3) components have to target the specific language (Fig. 3).

There are several strategies to track data-flow in a system: manual tracking, static analysis (Arzt et al., 2014; Dong et al., 2016; Enck et al., 2010; Lillack et al., 2018; Qiu et al., 2018; Rabkin and Katz, 2011), and dynamic analysis (Austin and Flanagan, 2009, 2012; Bell and Kaiser, 2014; Meinicke et al., 2016; Nguyen et al., 2016; Reisner et al., 2010; Yang et al., 2016). We used the state of the art (Do et al., 2017; Pauck et al., 2018; Qiu et al., 2018; Wang et al., 2016) object-, field-, context-, and flow-sensitive static taint analysis engine FlowDroid (Arzt et al., 2014). A taint analysis, typically used in the security domain, tracks what variables have been affected by selected inputs (sources) and are used in specific locations (sinks). We annotated the API calls to load configurations options as sources and control-flow statements as sinks.

We used the ASM library (Bruneton et al., 2002) to add bytecode instructions to measure the execution time of regions. We used the Soot framework (Vallée-Rai et al., 1999) to build the call graph of a system to optimize the instrumentation of regions.

Limitations. Our ConfigCrusher implementation is limited to analyzing single-threaded systems, as we dynamically process the execution time of regions, keeping a stack of regions at runtime to subtract the time of nested regions from the time of outer regions. Having one region executing at a time facilitates the processing and calculations that we perform. Other measurement strategies could be used, such as using a performance profiler, to measure the execution time of regions in multi-threaded systems.

At its core, the used static taint analysis is unsound and can lead to overtainting (Qiu et al., 2018; Wang et al., 2016), which can affect the results of our approach. For instance, at the one extreme, if the analysis *misses all interactions*, we will fail to produce a performance-influence model. At the other extreme, if the analysis indicates that *all options interact*, our approach results in the Brute Force approach. In addition, despite the high precision of FlowDroid (Arzt et al., 2014), the analysis is challenged by the size of the call graph, which restricts the size of the systems that our implementation of ConfigCrusher can analyze (Arzt et al., 2014; Avdiienko et al., 2015; Bodden, 2018; Do et al., 2017; Lerch et al., 2015; Pauck et al., 2018; Qiu et al., 2018; Wang et al., 2016). Similar to other approaches (Avdiienko et al., 2015; Lillack et al., 2018; Qiu et al., 2018), we reduced the precision of some FlowDroid specific settings (e.g., used an unexceptional control-flow graph) for a faster analysis (the analysis ran out of memory for systems 3 – 10 in Table 4 using the default settings). Despite running the static analysis on a server with 512 GB of RAM and 32 CPU cores, we were forced to exclude some systems from our evaluation since the server either used all of its memory or did not finish the analysis after 4 hours. Avdiienko et al. (Avdiienko et al., 2015) experienced similar results on a server with more RAM and CPU cores. Nevertheless, our evaluation demonstrates the feasibility of our implementation to produce accurate and informative performance-influence models, signifying that our approach is robust despite the levels of unsoundness and overtainting of FlowDroid. We expect that incorporating advancements in scaling taint analyses (Andreasen et al., 2017; Barros et al., 2015; Bodden, 2018; Christakis and Bird, 2016; Do et al., 2017; Garbervetsky et al., 2017; Lerch et al., 2015; Späth et al., 2017; Zhang and Su, 2017), the results in this article and the benefits of our approach will generalize to larger systems. We conjecture that similarly accurate results can be achieved with other taint analysis implementations.

5 Evaluation

To demonstrate the feasibility and potential of our white-box approach, we evaluate ConfigCrusher against state of the art approaches to build performance-influence models, for a specific workload, input size, and underlying hardware, in terms of the cost (i.e., number of configurations to sample and time to execute those configurations) to generate performance-influence models and their accuracy. Subsequently, we explore the usefulness of ConfigCrusher's local performance-influence models to identify the local influence of options on performance. Specifically, we address the following research questions:

RQ1: How does ConfigCrusher compare to other performance-influence modeling approaches in terms of cost and accuracy? We compare the effectiveness of ConfigCrusher regarding the cost of generating models and their accuracy to state of the art black-box and white-box approaches.

RQ2: How much overhead is induced by instrumentation? One of the goals of ConfigCrusher is to build performance-influence models efficiently. As discussed in Sec. 3.3, we observed an excessive amount of overhead when executing our unoptimized instrumented systems. We evaluate the effectiveness of our optimization by exploring how much overhead the instrumented regions induce and how it affects the performance that we measure.

RQ3: To how many regions can the influence of options on performance be localized? One of the benefits of ConfigCrusher over state of the art approaches is that it builds local performance-influence models, which indicate whether and how the options locally influence the performance of a system. In an exploratory analysis, we examine the local performance-influence models to determine the local influence of options on performance. Subsequently, we analyze the source code regions corresponding to the local models to further investigate how they are influenced by options. We conjecture that this type of information, derived from the local models, can provide insights to developers and maintainers for enhanced analysis of individual components of a system.

5.1 Subject Systems

The subject systems are summarized in Table 4. We selected a representative set of configurable systems that satisfy the following criteria: (a) systems from a variety of domains to increase external validity, (b) systems with at least 5 options (the Brute Force approach would produce results cheaply for systems with few options), (c) systems with characteristics representative of large-scale configurable systems (e.g., systems with binary and non-binary options), (d) single-threaded systems with deterministic execution time (we sampled each system multiple times with different approaches and observed execution times within usual measurement noise), and (e) systems for which the static taint analysis terminated. We included systems that have been used in previous studies, for comparability of results (systems 4, 7, 8) (Kim et al., 2013; Siegmund et al., 2013; Souto et al., 2017), and new configurable systems with a total of 860+ stars and 155+ forks on GitHub, and used by 160+ open-source projects, at the time of writing, to showcase the applicability of our approach (2, 3, 5, 6, 8, 9, 10). In the following sections, we consider the entire

Table 4: Configurable subject systems.

ID	Name	Domain	# SLOC	# Opt.	# Conf.	CID
1	Running Example	Example	69	10	1 024	—
2	Pngtastic Counter	Processing	1 250	5	32	`7f96382`
3	Pngtastic Optimizer	Optimization	2 553	5	32	`7f96382`
4	Elevator	SPL–Benchmark	575	6	20	—
5	Grep[1]	Command line	2152	7	128	`ef9eaa7`
6	Kanzi	Compression	20 537	7	128	`4dae29e`
7	Email	SPL–Benchmark	696	9	40	—
8	Prevayler	Database	1 328	9	512	`5be1ca4`
9	Sort[1]	Command line	2 163	12	4 096	`ef9eaa7`
10	Density Converter	Processing	1 359	22	2^{22}	`2ba8373`

CID = Commit ID.
[1] Java implementation of the Unix command.

system of Fig. 2 (Lines 1–32) as the Running Example (system 1 in Table 4) to showcase the potential of our approach.

Due to their novelty, white-box approaches impose strict limitations on the systems they can analyze (Kim et al., 2013; Siegmund et al., 2013; Souto et al., 2017). ConfigCrusher lifts some of these limitations; we consider data-flow interactions and do not limit the analysis to specific system implementations, which expands the types of systems that can be analyzed and increases the accuracy of the results. Still, the used implementation of static code analysis imposes limitations on the size of systems and their number of configuration options. We acknowledge the size of the real-world systems used in the evaluation and that black-box approaches can analyze larger systems. However, at this stage, we want to showcase the feasibility, benefits, and potential of white-box analyses and expect that, with improvements to the used data-flow analysis (Andreasen et al., 2017; Barros et al., 2015; Bodden, 2018; Christakis and Bird, 2016; Do et al., 2017; Garbervetsky et al., 2017; Lerch et al., 2015; Späth et al., 2017; Zhang and Su, 2017), our implementation will analyze larger systems. Nevertheless, we selected systems for our evaluation with representative characteristics of larger configurable systems (i.e., we observed the insights of *Irrelevance, Orthogonality, and Low-Interaction Degree* of configurable systems). Hence, the systems that we selected are suitable to answer our research questions and we conjecture that we can obtain similar results (Sec. 5.2) in larger systems with a more scalable implementation of the static taint analysis. Note the general trend in the results (Sec. 5.2): all other state of the art white-box approaches have the same scalability problem. Still, ConfigCrusher was able to analyze real-world systems which the other approaches could not; SPLat did not scale to Density Converter (system 10 in Table 4) and the family-based approach could not analyze any system besides the software product lines.

Although our static analysis correctly identified that all options interact in Elevator (4), since it was purposely built with such behavior (Kim et al., 2013; Meinicke et al., 2016; Souto et al., 2017) (i.e., our approach equals the Brute Force approach), we included the system in the evaluation since it is one of the two systems that the family-based approach can analyze.

5.2 RQ1: Comparison to State of the Art

With RQ1, we evaluate the cost and accuracy of the performance-influence models generated by ConfigCrusher and how it compares to state of the art black-box and white-box approaches. To answer this question, we measured the cost and prediction error of ConfigCrusher and all other approaches and compared them to the ground truth.

Procedure: We established ground truth by measuring the performance of the entire configuration space four times and averaged the performance of each configuration. Due to the high number of configurations and execution time of Sort (9) and Density Converter (10), we randomly sampled a large number of configurations each to act as the ground truth. We observed no variation in the errors of the results presented in Table 5 and Table 6 when using more than 1000 configurations.

Specifically, we compared ConfigCrusher to feature-wise sampling (i.e., enable one option at a time) and pair-wise sampling (i.e., cover all combinations of all pairs of options) (Medeiros et al., 2016) with stepwise linear regression (Sarkar et al., 2015; Siegmund et al., 2012a,b, 2015), Brute Force, SPLat (Kim et al., 2013), and the family-based approach (Siegmund et al., 2013). We excluded random sampling since research (Jamshidi et al., 2017a,b; Medeiros et al., 2016; Siegmund et al., 2015) has shown that it requires numerous samples to make accurate predictions and it is not clear how many configurations to sample for a specific system.

We conjectured that SPLat behaves essentially like the Brute Force approach in all but software product lines, since all configuration options are read at the start of the system. We included a SPLat variant, called SPLatDelayed (SAD), for which we modified the source code of the systems to delay the evaluation of options in control-flow statements (Saumont, 2017). The source code refactoring allowed us to evaluate how SPLat would operate in systems if it could detect when options are actually evaluated in control-flow decisions.

The static taint analysis and the performance measurements were executed on a 2.2 GHz Intel Core i7 MacBook Pro with 16 GB of RAM running OS X 10.13 (i.e., fixed underlying hardware). For each configuration, we initiated one VM invocation and ran the configuration (Georges et al., 2007). We used the JVM options `"-Xms10G -Xmx10G -XX:+UseConcMarkSweepGC"` to reduce the overhead of garbage collection. To control for measurement noise, we measured each configuration five times and averaged the performance of each configuration. For each system, we extracted the configuration options from the projects' documentation and executed a representative test scenario and workload provided by the system (i.e., fixed workload and input size) (Velez et al., 2019). Following the evaluation of state of the art approaches (Al-Hajjaji et al., 2016; Guo et al., 2013; Halin et al., 2018; Kim et al., 2013; Lillack et al., 2018; Medeiros et al., 2016; Meinicke et al., 2016; Sarkar et al., 2015; Siegmund et al., 2012a,b, 2013, 2015), we discretized the non-binary options (e.g., pick either the lowest or highest value) to reduce the number of samples to execute.

Table 5: Cost of building performance-influence models.

S	BF/SA	FW	PW	SAD	FB[1]	CC[2]
1	1024 [1.9h]	10 [16.6s]	56 [2.2m]	512 [56.9m]	N/A	8 [33.8s, 4.4s]
2	32 [2.9m]	5 [27.2s]	16 [1.5m]	24 [2.2m]	N/A	4 [21.9s, 7.8s]
3	32 [42.2m]	5 [1.6m]	16 [10.0m]	16 [21.0m]	N/A	10 [10.7m, 30.6s]
4	20 [10.8m]	3 [50.0s]	9 [3.3m]	20 [10.8m]	1 [49.5s]	64 [—]
5	128 [10.6m]	7 [22.1s]	29 [1.9m]	48 [3.5m]	N/A	64 [5.1m, 10.2s]
6	128 [1.2h]	7 [1.5m]	29 [8.8m]	64 [35.4m]	N/A	64 [35.4m, 12.6s]
7	40 [16.9m]	4 [23.5s]	11 [1.7m]	40 [16.9m]	1 [1.1m]	8 [1.5m, 12.8s]
8	512 [3.7h]	9 [2.7m]	46 [16.0m]	144 [1.5h]	N/A	32 [14.5m, 12.6s]
9	1298 [18.4h]	12 [13.1m]	79 [1.4h]	48 [42.8m]	N/A	256 [3.7h, 21.6s]
10	1414 [14.7h]	22 [21.3m]	254 [4.1h]	+24h[3]	N/A	256 [2.1h, 42.1s]

S = Subject system; FW = Feature-wise; PW = Pair-wise; BF = Brute Force; SA = SPLat; SAD = SPLat Delayed; FB = Family-Based; CC = ConfigCrusher.

A cell indicates approaches with the lowest costs.

[1] Not applicable to systems without static map derived from compile-time variability.
[2] Time includes the overhead of the static taint analysis.
[3] No data was collected due to timeout.

Table 6: MAPE comparison.

S	Feature-wise	Pair-wise	SPLat Delayed	Family-Based[1]	ConfigCrusher
1	56.9↑	6.2↑	0.2↑	N/A	0.1
2	0.8	2.0↑	1.3	N/A	1.1
3	19.7↑	0.9	1.0	N/A	1.1
4	51.1	1.5	∅	2.7	∅
5	32.1↑	114.7↑	1.9↓	N/A	3.6
6	1.9	1.3	1.21	N/A	2.7
7	100↑	44.2↑	∅	2.3↓	23.0
8	111.2↑	29.2↑	3.0↓	N/A	9.2
9	90.0↑	653.0↑	2.4↑	N/A	1.6
10	635.2↑	218.9↑	N/A[3]	N/A	4.3

A cell indicates approaches with statistically indistinguishable lowest errors. ↑ approach with statistically > error than ConfigCrusher. ↓ approach with statistically < error than ConfigCrusher. ∅ approach sampled all configurations, thus no performance to predict.

[1] Not applicable to systems without static map derived from compile-time variability.

Cost Metric. We measured the number of configurations and sampling time to generate a model. For ConfigCrusher, we also measured the one-time overhead of the static analysis.

Error Metric. We used the Mean Absolute Percentage Error (MAPE) to measure the mean difference between the values predicted by a model and the values actually observed (i.e., ground truth). For each approach, we calculate the prediction error on the configurations that the approach did not sample. We also calculated the error across all configurations (Velez et al., 2019). We used the multiple comparison $\tilde{\mathbf{T}}$-procedure (Konietschke et al., 2012) with 95% confidence to compare statistical differences between ConfigCrusher's prediction error to the prediction error of each of the other approaches.

Results: We show the cost results in Table 5 and the error results in Table 6. ConfigCrusher's prediction error is statistically indistinguishable or lower than other approaches. Furthermore, ConfigCrusher's high accuracy is usually achieved with lower cost compared to the other accurate approaches. Our results support our conjecture on the cost and prediction error comparison of Fig. 1.

Though feature-wise and pair-wise sampling tended to have lower costs than ConfigCrusher, when their errors are taken into account, we can conclude that more configurations had to be sampled to make accurate predictions. By comparison, for those systems, ConfigCrusher sampled more configurations, but attained significantly lower errors.

As we conjectured, SPLat behaves essentially like the Brute Force approach in all but software product lines. We also conjectured that SPLatDelayed would produce the lowest error since it uses a heuristic to perform a more efficient Brute Force approach. For the Running Example (1), Pngtastic Counter (2), Pngtastic Optimizer (3), and Sort (9), in which other approaches besides SPLatDelayed produced lower errors, but statistically indistinguishable, we can attribute the results to measurement noise. Interestingly, SPLat-Delayed did not finish analyzing Density Converter (10) within 24 hours. In this case, most options are read sequentially (similar to reading all options at the beginning of the system), thus indicating the limitations of the approach.

Only for Elevator (4) and Email (7), the family-based approach remains the most efficient and accurate approach, but, at the same time, the most limited one in terms of applicability.

Characteristics of interactions discussion: Thanks to ConfigCrusher, we observed and confirmed the insights of *Irrelevance*, *Orthogonality*, and *Low Interaction Degree* of configurable systems. In 9 out of 10 systems, it significantly reduced the configuration space to sample, thus reducing the cost to build accurate models. For example, our analysis identified the irrelevant options in our running example (1), Grep (5), and Sort (9), which is not leveraged by the black-box approaches before sampling. Similarly, ConfigCrusher identified orthogonal interactions and leveraged low interaction degree to sample fewer configurations, which was not exploited by the white-box approaches.

For Pngtastic Counter (2), Pngtastic Optimizer (3), and Kanzi (6), the black-box approaches produced accurate models with low cost. Upon inspection of the results, we discovered that (a) in Pngtastic Counter, the options did not affect the performance of the system; the execution time was essentially the same for all configurations, and (b) in Pngtastic Optimizer and Kanzi, the options did affect the performance, but the execution times were clustered in a few groups. For example, the performance of Kanzi under all configurations was either $\sim$4 seconds or $\sim$61 seconds. We consider these three systems as outliers since previous empirical studies (Apel et al., 2013; Jamshidi et al., 2017b; Kolesnikov et al., 2018; Siegmund et al., 2015) have shown that the performance of most configurable systems changes based on the selected configurations.

Source of prediction error discussion: Regarding ConfigCrusher's prediction error of Email (7), the system has a feature model (Apel et al., 2013) that describes its valid configurations. Since the invalid configurations were not executed, ConfigCrusher did not have all the information for each region to generate an accurate model. Despite missing information, ConfigCrusher was able to produce more accurate results than the other approaches, except the family-based approach. We hope to incorporate information from a feature model to produce more accurate models for this type of systems.

Regarding ConfigCrusher's prediction error of Prevayler (8), we observed that the execution time of certain regions differed beyond usual measurement noise. This behavior occurs when the correct interaction in the regions was not captured by the static analysis (a possible consequence of the unsoundness of the used taint analysis). We were unable to manually determine the correct interaction of the problematic regions. We conjecture that, since the system writes to disk, there might be some interactions in system calls, which we do not analyze. Despite this imprecision, ConfigCrusher was able to produce more accurate results than the other approaches. We hope to overcome this issue by analyzing system calls to obtain even more accurate results.

RQ1: ConfigCrusher's prediction error is statistically indistinguishable or lower than other approaches. ConfigCrusher's high accuracy is usually achieved with lower cost compared to the other accurate approaches.

5.3 RQ2: Instrumentation Overhead

As explained in Sec. 3.3, we observed excessive overhead when executing our instrumented systems. With RQ2, we investigate how much overhead is induced by instrumenting regions. To answer this question, we compared the instrumentation overhead and execution times of uninstrumented systems with systems instrumented with the unoptimized Algorithm 2 and with systems instrumented with the optimized Algorithm 2 and the propagation Algorithms 3 and 4.

Procedure: We used the execution time of the uninstrumented systems as ground truth and executed the configuration that triggered the most number of regions in the optimized systems. We executed the configuration with the highest execution time in case of multiple configurations with the same number of executed regions.

Static and Dynamic Overhead Metric. We measured the number of instrumented regions as the static overhead and the number of times the regions were entered and exited as the dynamic overhead.

Table 7: Static and dynamic comparison of instrumented regions before and after optimization for the configuration with the largest dynamic overhead.

	Original	Unoptimized			Optimized		
S	Time	R	RC	Time	R	RC	Time
1	12.14s	16	32	12.16s	10	22	12.13s
2	5.40s	36	$> 10^6$	>1h	13	18	5.54s
3	3.68m	397	$> 10^6$	>1h	7	88	3.77m
5	22.87s	46	$> 10^6$	>1h	1	2	21.87s
6	1.04m	128	$> 10^9$	>1h	23	4160	1.04m
7	26.08s	60	8530	25.50s	11	1204	25.49s
8	1.25m	147	$> 10^9$	>1h	28	$> 10^6$	1.27m
9	4.87m	166	$> 10^9$	>1h	1	2	4.89m
10	6.45m	202	2420	6.53m	10	78	6.45m

S = Subject system; R = # of instrumented regions; RC = # of executed regions.

Time Metric. We measured the execution times of the unoptimized and optimized instrumentations.

Results: Table 7 shows the results of our analysis. ConfigCrusher's optimized instrumentation with Algorithm 2, 3, and 4 reduced the number of regions and overhead by several orders of magnitude. By contrast, the unoptimized instrumentation, Algorithm 2, created an excessive amount of overhead, preventing running systems in a reasonable amount of time.

Only the systems Running Example (1), Email (7), and Density Converter (10) had a low unoptimized dynamic overhead and executed in similar time compared to the original systems. These systems do not have deeply nested regions, which does not increase the overhead of the dynamic analysis. Prevayler (8), which has similar structure to the previous systems, executed a large number of optimized instrumented regions with low overhead.

We can attribute the lower execution times of the instrumented systems compared to the original systems of the Running Example (1), Grep (5), and Email (7) to measurement noise.

RQ2: ConfigCrusher's optimized instrumentation incurs orders of magnitude less overhead compared to the unoptimized instrumentation.

5.4 RQ3: Performance Influence of Options in Regions

One of the benefits of ConfigCrusher over black-box approaches is that it builds local performance-influence models, which indicate whether and how options locally influence the performance of a system. With RQ3, we investigate to how many regions the influence of options on performance can be localized. To answer this question, we analyzed all local performance-influence models to determine the local influence of options on performance. Subsequently, we manually examined the source code regions corresponding to the local models

Table 8: Influence of options analysis of local performance-influence models.

	PNIO		Performance influenced by options			
	NEG	Non-NEG				
S	Regions	Regions	Regions	Min ID	Max ID	Structure
1	0	0	10	1	3	Sleep
2	13	2	0	N/A	N/A	Loop, I/O
3	3	1	3	3	3	Loop, I/O
5	0	0	1	6	6	Loop
6	19	2	2	6	6	Loop, I/O
7	7	0	4	2	8	Loop, Sleep
8	22	1	5	2	5	Loop, I/O
9	0	0	1	8	8	Loop
10	8	1	1	8	8	Loop, I/O

PNIO = Performance not influenced by options; NEG = Negligible execution time (region which contribute < 5% of the execution time of the system). ID = Interaction degree.

to further understand how they are influenced by options. We conjecture that this type of information, derived from the local models, can provide insights for enhanced analysis of individual components of a system.

Procedure: We classified all local performance-influence models into two categories according to how options influence their performance. Then, we manually analyzed all corresponding source code regions to understand how they are influenced by options. For example, we classified the following models from Pngtastic Optimizer (3):
- $\Pi_{\texttt{9cb}} = \texttt{0.0}$
- $\Pi_{\texttt{b15}} = \texttt{1.5}$
- $\Pi_{\texttt{b1a}} = \texttt{0.2 + 82.5Compress + 142.1CompressIter + ...}$,

as "Not influenced by options" (e.g., $\Pi_{\texttt{9cb}}$ and $\Pi_{\texttt{b15}}$) and "Influenced by options" (e.g., $\Pi_{\texttt{b1a}}$).

Results: Table 8 shows the results of our analysis. ConfigCrusher helped us to identify that the influence of options on performance can be localized to a few regions in a system. In these regions only subsets of all options interact. The local performance-influence models helped us to easily locate these regions in the source code to further analyze this performance behavior. In all such regions (e.g., Region `b1a` above), the options influenced a loop or a control-flow statement within a loop, which either manipulated data structures or performed I/O operations. These structures were sometimes located in the method where the region was instrumented, but other times we performed manual probing to find them in other methods called from the instrumented method.

Local models discussion. The local performance-influence models indicate the options that interact in the corresponding regions and whether and how they locally influence the performance of the system. The exploratory analysis of the corresponding source code regions yielded some interesting findings of how options are implemented in these systems, which cannot be found with black-box approaches.

In a few regions, the control-flow statements with non-negligible execution time depended on options, yet the same branch of the control-flow statement was executed for all configurations. This behavior was surprising since we selected, based on the systems' documentation, configuration values for each option that should behave differently. In fact, we discovered that two options of Pngtastic Counter (2) were not used in the source code as they were described in the documentation. For example, the valid range of an option was $0.0 - 1.0$, and we conjecture that the system would behave differently by picking different values. However, the control-flow statement where this option was used always executed the same branch if the value was > 0. Finding these inconsistensies, common in configurable systems (Cashman et al., 2018; Han and Yu, 2016; Rabkin and Katz, 2011; Xu et al., 2013), might be useful for developers and maintainers to debug these type of systems.

Interestingly, some options influenced only regions with negligible execution time. This behavior was surprising since we expected, based on the systems' documentation, that the options would influence the performance of the systems; for example, how the options that influence Region `9cb` above drastically change its execution time. We manually confirmed that they involve either a few statements or did not contain expensive loops nor calls. While black-box approaches also found that these options do not influence the performance, ConfigCrusher helped us to pinpoint the regions where these options are used to understand this potentially unexpected behavior.

We conjecture that developers can potentially discern similar findings in other configurable systems to make more informative decisions during debugging and optimization of these systems.

RQ3: ConfigCrusher helped us to identify that the influence of options on performance is localized to a few regions in a system. The options influence loops or control-flow statements within loops.

5.5 Threats to Validity

The primary threat to external validity is the selection of subject systems. As discussed in Sec. 5.1, we were limited by the overhead and precision of the static analysis. This limitation is due to the novelty of our approach and shared with other white-box approaches, though we lift some of their limitations. While we selected a set of widely-used open-source Java configurable systems from different domains, readers should be careful when generalizing results to other types of systems. For instance, we analyzed single-threaded systems with deterministic execution time. Additionally, we analyzed and built a performance-influence model for a single configurable system. Systems composed of numerous configurable systems, deployed in distributed environments, and implemented in different languages are beyond the scope of this article.

Another threat to validity is the subset of options that we selected for analysis in the subject systems. We selected options for which the systems' documentation or the options' functionality indicated that they would affect performance and observed a wide range of execution times for the configurations that we measured.

Another threat to validity is the selection of the data-flow analysis. As discussed in Sec. 4, we selected the state of the art static taint analysis, but reduced its precision in favor for an analysis that terminates. This strategy has been used in previous work and, as demonstrated in our evaluation, our approach is robust and produces accurate results with the settings that we selected.

Measurement noise cannot be excluded and may affect the results we obtained. We reduced this threat by repeating measurements several times on a dedicated machine and averaging the results.

6 Related Work

In Sec. 2, we described performance modeling in general and closely related state of the art black-box and white-box approach for performance-influence modeling and compared them to ConfigCrusher. In this section, we discuss additional research to position ConfigCrusher and white-box analyses in the context of prior work.

Analysis of configurable systems: Similar to our work, several researchers have leveraged some kind of program analysis to explore various properties of configuration options (Dong et al., 2016; Hoffmann et al., 2011; Meinicke et al., 2016; Nguyen et al., 2016; Rabkin and Katz, 2011; Reisner et al., 2010; Souto and d'Amorim, 2018; Wang et al., 2013). Thüm et al. (Thüm et al., 2014) presented a comprehensive survey of analyses for software product lines also applicable to configurable systems.

Similar to our approach, Lillack et al. (Lillack et al., 2018) used taint analysis to identify, for each code fragment, in which configurations it may be executed. However, they do not track information about individual options. Instead, our taint analysis tracks how options influence code fragments due to control-flow and data-flow interactions to track how options influence the performance of the system.

Hoffmann et al. (Hoffmann et al., 2011) used dynamic influence tracing to convert static parameters into dynamic control variables to adapt properties of an application. However, they do not consider interactions beyond two parameters. By contrast, our approach tracks control-flow and data-flow interactions among options, without a limit on the interaction degree, and how they influence the performance of the system.

Reisner et al. (Reisner et al., 2010) and Meinicke et al. (Meinicke et al., 2016) used symbolic execution and variational execution, respectively, to analyze the behavior of interactions in configurable systems and found the insights of Irrelevance, Independence, and Low Interaction Degree that we consider in this work . We leverage those insights to create a novel white-box performance analysis that efficiently generates accurate performance-influence models.

Testing configurable systems: Combinatorial Testing (Halin et al., 2018; Hervieu et al., 2011, 2016; Kuhn et al., 2013; Nie and Leung, 2011) is an approach to reduce the number of samples to test a system by satisfying a certain coverage criterion. Similarly, Souto et al. (Souto et al., 2017) improved SPLat (Kim et al., 2013) to use sampling heuristics (Medeiros et al., 2016) to select what configurations to sample. While both these approaches scale to large systems, they make assumptions on how options interact in the system and can potentially miss relevant interactions. Instead, our sampling is guided by white-box information on how options are used and interact in the systems.

Performance profiling: Several profiling techniques, including sampling and instrumentation, can be used to identify performance hot spots (Cito et al., 2018; Gregg, 2016; Mostafa et al., 2017; Yu and Pradel, 2018). For example, Castro et al. (Castro et al., 2015) used both techniques to identify hot spots that can be isolated and replayed as standalone systems for further performance analysis and optimization. Our approach is complementary to this line of work, assisting in potentially narrowing down the performance-intensive components for more comprehensive profiling.

Energy measurement: Modeling power or energy consumption is closely related to performance and employs similar techniques (Gui et al., 2016; Gupta et al., 2014; Jabbarvand et al., 2016). For example, Hao et al. (Hao et al., 2013) used program analysis to estimate the energy consumption of instructions of Android apps. This line of work, however, does not address configurability, but could benefit from our approach to understand how the configuration of the system influences its energy consumption.

7 Conclusion

This article presents ConfigCrusher, a white-box performance analysis approach for configurable systems. ConfigCrusher employs a data-flow analysis to identify how configuration options may influence control-flow statements and instruments regions corresponding to those statements for performance measurement. Our evaluation on 10 real-word systems shows the potential of our white-box approach to builds similar or more accurate performance-influence models than other approaches with lower cost. In contrast to state of the art approaches, our white-box approach provides additional information of the components of a system, which can aid stakeholders to analyze, optimize, and debug them. Our work can provide a foundation for future research on white-box analysis for configurable systems, such as understanding the performance of larger and distributed systems, explaining causes for performance differences, and combining white-box and black-box approaches for more accurate and efficient performance analysis.

8 Acknowledgments

This work has been supported in part by the NSF (awards 1318808, 1552944, and 1717022), AFRL, DARPA (FA8750-16-2-0042), and the German Research Foundation (AP 206/7-2, AP 206/11-1, SI 2171/2, SI 2171/3-1). We thank Chu-Pan Wong and Jens Meinicke for their comments during the development of this work. We thank the FOSD 2017 and 2018 meeting participants for their feedback on the central idea of this work. We thank Steven Artz for his help with FlowDroid.

References

Al-Hajjaji M, Krieter S, Thüm T, Lochau M, Saake G (2016) Incling: Efficient product-line testing using incremental pairwise sampling. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE), ACM, New York, NY, USA, pp 144–155

Aldrich J, Garlan D, Kaestner C, Le Goues C, Mohseni-Kabir A, Ruchkin I, Samuel S, Schmerl B, Timperley CS, Veloso M, Voysey I, Biswas J, Guha A, Holtz J, Camara J, Jamshidi P (2019) Model-based adaptation for robotics software. IEEE Software 36(2):83–90

Andreasen ES, Møller A, Nielsen BB (2017) Systematic approaches for increasing soundness and precision of static analyzers. In: Proc. Int'l Workshop State Of the Art in Program Analysis (SOAP), ACM, New York, NY, USA, pp 31–36, DOI 10.1145/3088515.3088521, URL http://doi.acm.org/10.1145/3088515.3088521

Angerer F, Grimmer A, Prähofer H, Grünbacher P (2015) Configuration-aware change impact analysis. In: Proc. Int'l Conf. Automated Software Engineering (ASE), IEEE Computer Society, Washington, DC, USA, pp 385–395

Apel S, Batory D, Kästner C, Saake G (2013) Feature-Oriented Software Product Lines: Concepts and Implementation. Springer-Verlag, Berlin/Heidelberg, Germany

Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Le Traon Y, Octeau D, McDaniel P (2014) Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proc. Conf. Programming Language Design and Implementation (PLDI), ACM, New York, NY, USA, pp 259–269

Austin TH, Flanagan C (2009) Efficient purely-dynamic information flow analysis. In: Proc. Workshop Programming Languages and Analysis for Security (PLAS), ACM, New York, NY, USA, pp 113–124

Austin TH, Flanagan C (2012) Multiple facets for dynamic information flow. In: Proc. Symp. Principles of Programming Languages (POPL), ACM, New York, NY, USA, pp 165–178

Avdiienko V, Kuznetsov K, Gorla A, Zeller A, Arzt S, Rasthofer S, Bodden E (2015) Mining apps for abnormal usage of sensitive data. In: Proc. Int'l

Conf. Software Engineering (ICSE), IEEE Press, Piscataway, NJ, USA, pp 426–436

Barros P, Just R, Millstein S, Vines P, Dietl W, dAmorim M, Ernst MD (2015) Static analysis of implicit control flow: Resolving java reflection and android intents (t). In: Proc. Int'l Conf. Automated Software Engineering (ASE), IEEE Computer Society, Washington, DC, USA, pp 669–679

Becker S, Koziolek H, Reussner R (2009) The palladio component model for model-driven performance prediction. J Syst Softw 82(1):3–22

Bell J, Kaiser G (2014) Phosphor: Illuminating dynamic data flow in commodity jvms. SIGPLAN Notices 49(10):83–101

Bodden E (2018) Self-adaptive static analysis. In: Proc. Int'l Conf. Software Engineering (ICSE): New Ideas and Emerging Results, ACM, New York, NY, USA, pp 45–48

Bruneton E, Lenglet R, Coupaye T (2002) Asm: a code manipulation tool to implement adaptable systems. Adaptable and extensible component systems 30(19)

Cashman M, Cohen MB, Ranjan P, Cottingham RW (2018) Navigating the maze: The impact of configurability in bioinformatics software. In: Proc. Int'l Conf. Automated Software Engineering (ASE), ACM, New York, NY, USA, pp 757–767

Castro PDO, Akel C, Petit E, Popov M, Jalby W (2015) Cere: Llvm-based codelet extractor and replayer for piecewise benchmarking and optimization. ACM Trans Archit Code Optim (TACO) 12(1):6:1–6:24

Christakis M, Bird C (2016) What developers want and need from program analysis: An empirical study. In: Proc. Int'l Conf. Automated Software Engineering (ASE), ACM, New York, NY, USA, pp 332–343

Cito J, Leitner P, Bosshard C, Knecht M, Mazlami G, Gall HC (2018) Performancehat: Augmenting source code with runtime performance traces in the ide. In: Proc. Int'l Conf. Software Engineering: Companion Proceeedings, ACM, New York, NY, USA, pp 41–44

Do LNQ, Ali K, Livshits B, Bodden E, Smith J, Murphy-Hill E (2017) Just-in-time static analysis. In: Proc. Int'l Symp. Software Testing and Analysis (ISSTA), ACM, New York, NY, USA, pp 307–317

Dong Z, Andrzejak A, Lo D, Costa D (2016) Orplocator: Identifying read points of configuration options via static analysis. In: Proc. Int'l Symposium Software Reliability Engineering (ISSRE), pp 185–195

Enck W, Gilbert P, Chun BG, Cox LP, Jung J, McDaniel P, Sheth AN (2010) Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: Proc. Conf. Operating Systems Design and Implementation (OSDI), USENIX Association, Berkeley, CA, USA, pp 393–407

Esfahani N, Elkhodary A, Malek S (2013) A learning-based framework for engineering feature-oriented self-adaptive software systems. IEEE Transactions on Software Engineering 39(11):1467–1493

Garbervetsky D, Zoppi E, Livshits B (2017) Toward full elasticity in distributed static analysis: The case of callgraph analysis. In: Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE), ACM, New York, NY, USA, pp 442–453, DOI 10.1145/3106237.3106261, URL http://doi.acm.org/10.1145/3106237.3106261

Georges A, Buytaert D, Eeckhout L (2007) Statistically rigorous java performance evaluation. SIGPLAN Notices 42(10):57–76

Gregg B (2016) The flame graph. Commun ACM 59(6):48–57

Gui J, Li D, Wan M, Halfond WGJ (2016) Lightweight measurement and estimation of mobile ad energy consumption. In: Proc. Int'l Workshop Green and Sustainable Software (GREENS), ACM, New York, NY, USA, pp 1–7

Guo J, Czarnecki K, Apel S, Siegmund N, Wąsowski A (2013) Variability-aware performance prediction: A statistical learning approach. In: Proc. Int'l Conf. Automated Software Engineering (ASE), IEEE Computer Society, ACM, New York, NY, USA, pp 301–311

Gupta A, Zimmermann T, Bird C, Nagappan N, Bhat T, Emran S (2014) Mining energy traces to aid in software development: An empirical case study. In: Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM), ACM, New York, NY, USA, pp 40:1–40:8

Halin A, Nuttinck A, Acher M, Devroey X, Perrouin G, Baudry B (2018) Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack. Empirical Software Engineering

Han X, Yu T (2016) An empirical study on performance bugs for highly configurable software systems. In: Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM), ACM, New York, NY, USA, pp 23:1–23:10

Han X, Yu T, Lo D (2018) Perflearner: Learning from bug reports to understand and generate performance test frames. In: Proc. Int'l Conf. Automated Software Engineering (ASE), ACM, New York, NY, USA, pp 17–28

Hao S, Li D, Halfond WGJ, Govindan R (2013) Estimating mobile application energy consumption using program analysis. In: Proc. Int'l Conf. Software Engineering (ICSE), IEEE Press, Piscataway, NJ, USA, pp 92–101

Harchol-Balter M (2013) Performance Modeling and Design of Computer Systems: Queueing Theory in Action, 1st edn. Cambridge University Press, New York, NY, USA

Hervieu A, Baudry B, Gotlieb A (2011) Pacogen: Automatic generation of pairwise test configurations from feature models. In: Int'l Symposium Software Reliability Engineering, pp 120–129

Hervieu A, Marijan D, Gotlieb A, Baudry B (2016) Optimal Minimisation of Pairwise-covering Test Configurations Using Constraint Programming. Information and Software Technology 71:129 – 146

Hoffmann H, Sidiroglou S, Carbin M, Misailovic S, Agarwal A, Rinard M (2011) Dynamic knobs for responsive power-aware computing. In: Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM, New York, NY, USA, pp 199–212

Hubaux A, Xiong Y, Czarnecki K (2012) A user survey of configuration challenges in linux and ecos. In: Proc. Workshop Variability Modeling of Software-Intensive Systems (VAMOS), ACM, New York, NY, USA, pp 149–155

Hutter F, Hoos HH, Leyton-Brown K (2011) Sequential model-based optimization for general algorithm configuration. In: Proc. Int'l Conf. Learning and Intelligent Optimization, Springer-Verlag, Berlin/Heidelberg, Germany, pp 507–523

Jabbarvand R, Sadeghi A, Bagheri H, Malek S (2016) Energy-aware test-suite minimization for android apps. In: Proc. Int'l Symp. Software Testing and Analysis (ISSTA), ACM, New York, NY, USA, pp 425–436

Jamshidi P, Casale G (2016) An uncertainty-aware approach to optimal configuration of stream processing systems. In: Int'l Symp.Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), pp 39–48

Jamshidi P, Siegmund N, Velez M, Kästner C, Patel A, Agarwal Y (2017a) Transfer learning for performance modeling of configurable systems: An exploratory analysis. In: Proc. Int'l Conf. Automated Software Engineering (ASE), ACM, New York, NY, USA

Jamshidi P, Velez M, Kästner C, Siegmund N, Kawthekar P (2017b) Transfer learning for improving model predictions in highly configurable software. In: Proc. Int'l Symp. Software Engineering for Adaptive and Self-Managing Systems (SEAMS), IEEE Computer Society, Los Alamitos, CA, USA, pp 31–41

Jamshidi P, Velez M, Kästner C, Siegmund N (2018) Learning to sample: Exploiting similarities across environments to learn performance models for configurable systems. In: Proc. Int'l Symp. Foundations of Software Engineering (FSE), ACM, New York, NY, USA, pp 71–82

Jin D, Qu X, Cohen MB, Robinson B (2014) Configurations everywhere: Implications for testing and debugging in practice. In: Companion Proc. Int'l Conf. Software Engineering, ACM, New York, NY, USA, pp 215–224

Kim CHP, Batory DS, Khurshid S (2011) Reducing combinatorics in testing product lines. In: Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD), ACM, New York, NY, USA, pp 57–68

Kim CHP, Marinov D, Khurshid S, Batory D, Souto S, Barros P, d'Amorim M (2013) Splat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In: Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE), ACM, New York, NY, USA, pp 257–267

Kolesnikov S, Siegmund N, Kästner C, Grebhahn A, Apel S (2018) Tradeoffs in modeling performance of highly configurable software systems. Software and System Modeling (SoSyM)

Konietschke F, Hothorn LA, Brunner E (2012) Rank-based multiple test procedures and simultaneous confidence intervals. Electronic Journal of Statistics 6:738–759

Kuhn DR, Kacker RN, Lei Y (2013) Introduction to Combinatorial Testing, 1st edn. Chapman & Hall/CRC

Lerch J, Späth J, Bodden E, Mezini M (2015) Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths (t). In: Proc. Int'l Conf. Automated Software Engineering (ASE), IEEE Computer

Society, Washington, DC, USA, pp 619–629

Lillack M, Kästner C, Bodden E (2018) Tracking load-time configuration options. IEEE Transactions on Software Engineering 44(12):1269–1291

Medeiros F, Kästner C, Ribeiro M, Gheyi R, Apel S (2016) A comparison of 10 sampling algorithms for configurable systems. In: Proc. Int'l Conf. Software Engineering (ICSE), ACM, New York, NY, USA, pp 643–654

Meinicke J, Wong CP, Kästner C, Thüm T, Saake G (2016) On essential configuration complexity: Measuring interactions in highly-configurable systems. In: Proc. Int'l Conf. Automated Software Engineering (ASE), ACM, New York, NY, USA, pp 483–494

Montgomery DC (2006) Design and Analysis of Experiments. John Wiley & Sons

Mostafa S, Wang X, Xie T (2017) Perfranker: Prioritization of performance regression tests for collection-intensive software. In: Proc. Int'l Symp. Software Testing and Analysis (ISSTA), ACM, New York, NY, USA, pp 23–34

Nguyen T, Koc T, Cheng J, Foster JS, Porter AA (2016) iGen dynamic interaction inference for configurable software. In: Proc. Int'l Symp. Foundations of Software Engineering (FSE), IEEE Computer Society, Los Alamitos, CA, USA

Nie C, Leung H (2011) A survey of combinatorial testing. ACM Comput Surv (CSUR) 43(2):11:1–11:29

Oh J, Batory D, Myers M, Siegmund N (2017) Finding near-optimal configurations in product lines by random sampling. In: Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE), ACM, New York, NY, USA, pp 61–71

Olaechea R, Rayside D, Guo J, Czarnecki K (2014) Comparison of exact and approximate multi-objective optimization for software product lines. In: Proc. Int'l Software Product Line Conference (SPLC), ACM, New York, NY, USA, pp 92–101

Pauck F, Bodden E, Wehrheim H (2018) Do android taint analysis tools keep their promises? In: Proc. Int'l Symp. Foundations of Software Engineering (FSE), ACM, New York, NY, USA, pp 331–341, DOI 10.1145/3236024.3236029, URL http://doi.acm.org/10.1145/3236024.3236029

Qiu L, Wang Y, Rubin J (2018) Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In: Proc. Int'l Symp. Software Testing and Analysis (ISSTA), ACM, New York, NY, USA, pp 176–186

Rabkin A, Katz R (2011) Static extraction of program configuration options. In: Proc. Int'l Conf. Software Engineering (ICSE), ACM, New York, NY, USA, pp 131–140

Reisner E, Song C, Ma KK, Foster JS, Porter A (2010) Using symbolic evaluation to understand behavior in configurable software systems. In: Proc. Int'l Conf. Software Engineering (ICSE), ACM, New York, NY, USA, pp 445–454

Sarkar A, Guo J, Siegmund N, Apel S, Czarnecki K (2015) Cost-efficient sampling for performance prediction of configurable systems. In: Proc. Int'l

Conf. Automated Software Engineering (ASE), IEEE Computer Society, Washington, DC, USA, pp 342–352

Saumont PY (2017) Lazy computations in java with a lazy type

Serazzri G, Casale G, Bertoli M, Serazzri G, Casale G, Bertoli M (2006) Java modelling tools: an open source suite for queueing network modelling andworkload analysis. In: Third International Conference on the Quantitative Evaluation of Systems - (QEST'06), pp 119–120

Siegmund N, Kolesnikov SS, Kästner C, Apel S, Batory D, Rosenmüller M, Saake G (2012a) Predicting performance via automated feature-interaction detection. In: Proc. Int'l Conf. Software Engineering (ICSE), IEEE Press, Piscataway, NJ, USA, pp 167–177

Siegmund N, Rosenmüller M, Kuhlemann M, Kästner C, Apel S, Saake G (2012b) Spl conqueror: Toward optimization of non-functional properties in software product lines. Software Quality Journal 20(3-4):487–517

Siegmund N, von Rhein A, Apel S (2013) Family-based performance measurement. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE), ACM, New York, NY, USA, pp 95–104

Siegmund N, Grebhahn A, Apel S, Kästner C (2015) Performance-influence models for highly configurable systems. In: Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE), ACM, New York, NY, USA, pp 284–294

Souto S, d'Amorim M (2018) Time-space efficient regression testing for configurable systems. Journal of Systems and Software

Souto S, d'Amorim M, Gheyi R (2017) Balancing soundness and efficiency for practical testing of configurable systems. In: Proc. Int'l Conf. Software Engineering (ICSE), IEEE Press, Piscataway, NJ, USA, pp 632–642

Späth J, Ali K, Bodden E (2017) Ideal: Efficient and precise alias-aware dataflow analysis. Proc ACM Program Lang 1(OOPSLA):99:1–99:27, DOI 10.1145/3133923, URL http://doi.acm.org/10.1145/3133923

Thüm T, Apel S, Kästner C, Schaefer I, Saake G (2014) A classification and survey of analysis strategies for software product lines. ACM Comput Surv (CSUR) 47(1):6:1–6:45

Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V (1999) Soot - a java bytecode optimization framework. In: Proc. Conf. Centre for Advanced Studies on Collaborative Research (CASCON), IBM Press, pp 13–

Valov P, Petkovich JC, Guo J, Fischmeister S, Czarnecki K (2017) Transferring performance prediction models across different hardware platforms. In: Proc. Int'l Conf. on Performance Engineering (ICPE), ACM, New York, NY, USA, pp 39–50

Velez M, Jamshidi P, Sattler F, Siegmund N, Apel S, Kästner C (2019) Configcrusher: Towards white-box performance analysis for configurable systems - Supplementary Material - https://bit.ly/3diKZmK

Wang B, Passos L, Xiong Y, Czarnecki K, Zhao H, Zhang W (2013) Smartfixer: Fixing software configurations based on dynamic priorities. In: Proc. Int'l Software Product Line Conference (SPLC), ACM, New York, NY, USA, pp 82–90, DOI 10.1145/2491627.2491640, URL http://doi.acm.org/10.1145/2491627.2491640

Wang S, Li C, Hoffmann H, Lu S, Sentosa W, Kistijantoro AI (2018) Understanding and auto-adjusting performance-sensitive configurations. In: Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM, New York, NY, USA, pp 154–168

Wang Y, Zhang H, Rountev A (2016) On the unsoundness of static analysis for android guis. In: Proc. Int'l Workshop State Of the Art in Program Analysis (SOAP), ACM, New York, NY, USA, pp 18–23

Weisenburger P, Luthra M, Koldehofe B, Salvaneschi G (2017) Quality-aware runtime adaptation in complex event processing. In: Proc. Int'l Symp. Software Engineering for Adaptive and Self-Managing Systems (SEAMS), IEEE Computer Society, Los Alamitos, CA, USA, pp 140–151

Xu T, Zhang J, Huang P, Zheng J, Sheng T, Yuan D, Zhou Y, Pasupathy S (2013) Do not blame users for misconfigurations. In: Proc. Symp. Operating Systems Principles, ACM, New York, NY, USA, pp 244–259

Xu T, Jin L, Fan X, Zhou Y, Pasupathy S, Talwadker R (2015) Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In: Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE), ACM, New York, NY, USA, pp 307–319

Yang J, Hance T, Austin TH, Solar-Lezama A, Flanagan C, Chong S (2016) Precise, dynamic information flow for database-backed applications. In: Proc. Conf. Programming Language Design and Implementation (PLDI), ACM, New York, NY, USA, pp 631–647

Yu T, Pradel M (2018) Pinpointing and repairing performance bottlenecks in concurrent programs. Empirical Softw Eng 23(5):3034–3071

Zhang Q, Su Z (2017) Context-sensitive data-dependence analysis via linear conjunctive language reachability. In: Proc. Symp. Principles of Programming Languages (POPL), ACM, New York, NY, USA, pp 344–358, DOI 10.1145/3009837.3009848, URL http://doi.acm.org/10.1145/3009837.3009848

Zhu Y, Liu J, Guo M, Bao Y, Ma W, Liu Z, Song K, Yang Y (2017) Bestconfig: Tapping the performance potential of systems via automatic configuration tuning. In: Proc. Symposium Cloud Computing (SoCC), ACM, New York, NY, USA, pp 338–350