ConfigCrusher: White-Box Performance Analysis for Configurable Systems

Miguel Velez
Carnegie Mellon University
USA

Pooyan Jamshidi
University of South Carolina
USA

Florian Sattler
Saarland University
Germany

Norbert Siegmund
Bauhaus-University Weimar
Germany

Sven Apel
Saarland University
Germany

Christian Kästner
Carnegie Mellon University
USA

ABSTRACT

In configurable software systems, stakeholders 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. There are several black-box approaches to obtain this information, but they usually require a large number of samples to make accurate predictions, whereas the few existing white-box approaches impose limitations on the systems that they can analyze. This paper proposes ConfigCrusher, a white-box performance analysis that exploits several insights of configurable systems. ConfigCrusher employs a static data-flow analysis to identify how configuration options may influence control-flow decisions and instruments code regions corresponding to these decisions to dynamically analyze the influence of configuration options on the regions' performance. Our evaluation using 10 real-world configurable systems shows that ConfigCrusher is more efficient at building performance models that are similar to or more accurate than current state-of-the-art black-box and white-box approaches. Overall, this paper showcases the benefits and potential of white-box performance analyses to outperform black-box approaches and provide additional information for analyzing configurable systems.

CCS CONCEPTS

• **Configurable software**; • **Performance modeling**; • **Static analysis**; • **Dynamic analysis**;

1 INTRODUCTION

Most of today's software systems, such as databases, Web servers, and compilers, provide configuration options to satisfy a large variety of requirements [4, 36, 40, 57, 67]. Satisfying specific requirements consists of selecting values for each option to obtain the desired functional and non-functional properties of the system.

However, this configuration process is often a difficult task, especially when there is a lack of knowledge of how the configuration options influence the functionality and properties of the system [4, 57, 68]. 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 [31, 33, 37, 67].

Performance is one of the many interesting properties of such systems. Having an understanding of how individual configuration options and their combinations influence the performance of the
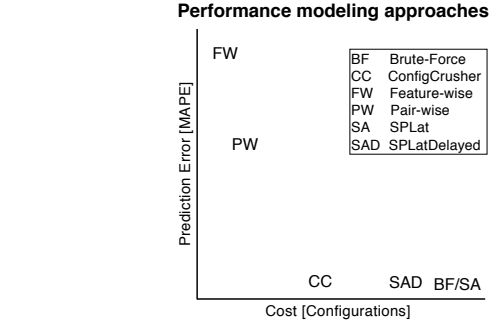

Figure 1: Our conjecture on cost and prediction error comparison between state-of-the-art approaches.

system would facilitate the reasoning, debugging, adaptation, and optimization processes of these systems.

Approaches to understand the performance of a configurable system build a *performance-influence model* to describe the influence of options and their interactions on performance. Most prior studies have focused on *black-box* approaches [23, 34–36, 40, 55, 57–59], which sample a subset of the configurations of a system and extrapolate a performance model based on the corresponding measurements. The performance model's accuracy and cost depend on the approaches' tradeoff between the sampling strategy and the algorithm used for learning [40] (Fig. 1). For example, the accuracy of a model might be low if the sample set does not capture important interactions among options. However, studies have neglected the area of *white-box* performance analyses, which analyze the source code and, with the information that is obtained, measure the system targeting a dedicated subset of the configurations to build a performance model [60]. Their cost depends on the precision of the static or dynamic program analyses, but they can generate more accurate models (Fig. 1) since they can pinpoint interactions in the code (e.g., at control-flow level). In addition, white-box approaches have the potential to pinpoint which regions of a program are responsible for performance differences among configurations, thus providing richer models. Despite these and other benefits yet to be explored and exploited [34, 38–40, 44, 46, 49, 54, 58, 60], only a couple of approaches exist [39, 60], and they make strong assumptions (e.g., no data-flow interactions, and exclusive to software product lines), which limit their accuracy and applicability.

In this paper, we introduce an approach in the hardly explored area of white-box performance analysis for configurable systems,

named ConfigCrusher. It combines static and dynamic analyses to identify and efficiently measure configurations that are relevant for accurate performance modeling (Fig. 1). Specifically, ConfigCrusher uses *static data-flow analysis* to trace the effect that configuration options may directly or indirectly have on control-flow decisions in the program, and subsequently *instruments regions* corresponding to these decisions for performance measurement. One key insight of our approach is that, in a single executed configuration, we independently measure the influence of multiple options on different regions, a process we call *compression*. That is, ConfigCrusher captures the performance characteristics of several options in a single execution, finally *building a performance model* from the performance measurements of the individual regions. Specifically, ConfigCrusher exploits the following insights about configurable systems, established throughout several prior studies [34, 38–40, 44, 46, 49, 54, 58, 60]:

(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 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 the state-of-the-art, 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 and more relevant configurations, and, due to instrumentation and compression, each measurement can provide information about multiple options and interactions. Furthermore, ConfigCrusher builds a *local performance model* for each region of a program. These models indicate the options that interact in the corresponding regions and whether and how they locally influence the performance of the system. This *local information*, missing in black-box approaches, can provide insights for enhanced debugging and understanding of the performance behavior of a system.

We implemented ConfigCrusher for Java programs, and we evaluated it against 3 black-box and 2 white-box state-of-the-art approaches on performance modeling. Across 10 subject systems in different domains, including command line programs, processing libraries, databases, and software product lines, we show that ConfigCrusher is more efficient at building accurate performance models than other approaches.

In summary, we make the following contributions:

- A white-box program analysis, combining data-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.


```
 1  void foo(boolean c)
R1:{A,C}  2    if(c) ... // 4s else ... // 1s  T_R1=1A+3AC
 3  void main()
 4    boolean a = Opt.A; boolean b = Opt.B;
 5    boolean c = Opt.C; boolean d = Opt.D;
 6    boolean e = Opt.E; boolean f = Opt.F;
 7    boolean g = Opt.G; boolean h = Opt.H;
 8    boolean i = Opt.I; boolean j = Opt.J;
 9    ... // 1s
10    boolean x = false;
11    if(a)
R2:{A} 12      ... // 2s                        T_R2=2A
13      foo(c);
14      x = true;
R3:{A,B} 15    if(b && x) ... // 3s              T_R3=3AB
16    if(d && e && f) ... // 5s
17    if(a) ... // 0.1s  if(b) ... // 0.2s
18    if(c) ... // 0.3s  if(d) ... // 0.4s
19    if(e) ... // 0.5s  if(f) ... // 0.6s
20    if(g) ... // 0.7s  if(h) ... // 0.8s
21    if(i) ... // 0.9s
```

Figure 2: Running example for a given workload with 3 regions influenced by configuration options. For simplicity, we ignore the regions in Lines $16 - 21$ through Sec. 4. Region 1 is influenced by a control-flow interaction and Region 3 by a data-flow interaction. The local performance models are shown at the right of each region.

- A compression technique that allows us to accurately infer the influence of options and their interactions on independent regions of a program's execution and a corresponding method to build accurate performance models.
- An optimization to reduce the overhead of the instrumented programs that we analyze.
- An empirical evaluation of ConfigCrusher on 10 systems showing the reduction in cost and increase in accuracy of performance modeling compared to state-of-the-art black-box and white-box approaches.
- Open source implementation of ConfigCrusher and reimplementations and improvements of prior state-of-the-art approaches for Java programs [1].
- A replication package with technical information of the systems analyzed, environmental setup for experiments, analysis scripts, and data of several months of measurements [1].

2 PERF. MODELING OF CONFIG. SYSTEMS

Our goal is to efficiently build accurate performance models for configurable systems. That is, we want to identify the minimum number of configurations to measure to build a performance model that accurately predicts the performance of all other configurations. This goal lowers the costs for performance modeling, which is useful in the development process, for example, to understand how the system works, identify options that are affecting the performance unexpectedly, and optimize system configurations.

Performance-influence models. A performance-influence model describes the performance of a system or region for a given workload and input size in terms of its configuration options [57]. Note the contrast of these models to *performance models* that also incorporate how workload, input size, and hardware variability influence the performance of a system. Our work, similar to the state-of-the-art approaches discussed in this paper [23, 55, 57–60], builds *performance-influence models*. For simplicity, we refer to performance-influence models as performance models in the rest

of the paper. For example, we can describe how options and interactions influence the performance of our running example in Fig. 2 (Lines 1–15) with the following model:

$$T = 1 + 3A + 3AB + 3AC \qquad (1)$$

This model indicates that the system executes in 1 second if all options are disabled (i.e., set to `false`), takes 3 additional seconds if option A is enabled (i.e., set to `true`), 3 additional seconds if options A and B are enabled together, and 3 additional seconds if options A and C are enabled. Notice how the model describes the influence of individual options (A) as well as the influence of interactions of options ({A,B} and {A,C}).

A performance model, such as the one above, can be used to assess if the performance values comply with the system's requirements, to pick optimal configurations, or to make informed trade-off decisions between performance and other properties. As we will detail in Sec. 3.5, performance models built with our white-box approach can associate terms of the model with individual code regions to describe which options interact in each region and whether and how they influence its performance. For example, ConfigCrusher will indicate that the terms 3A + 3AC of the performance model describe the performance of regions R1 ($T_{R1} = 1A + 3AC$) and R2 ($T_{R2} = 2A$). This fine-grained information can help in the understanding and debugging of individual components of a system.

State of the art of building performance-influence models. There are different strategies for building global performance models for configurable systems [23, 55, 57–60], with different tradeoffs among applicability, cost, and accuracy [40]. *Black-box approaches*, which have been studied extensively, sample some part of the configuration space and extrapolate a performance model for the remaining configurations using machine learning [23, 55, 57–59]. However, sampling a few configurations to avoid scalability issues might sacrifice accuracy (e.g., if the sampled configurations do not cover all relevant execution paths). *White-box approaches*, mostly neglected by previous work, analyze the system (statically, dynamically, or both) to build a performance model, often using the analysis (i.e., what options interact) to inform a sampling strategy [39, 60]. Existing white-box approaches can achieve higher accuracy (e.g., if the analysis identifies all relevant execution paths), but the few existing ones have strong assumptions (e.g., no data-flow interactions, and exclusive to software product lines), which threaten generality [39, 60].

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) [34, 38–40, 44, 46, 49, 54, 58, 60]. There are other insights that can be exploited to efficiently build accurate models (e.g., prefix sharing and variational execution [46]), but they are expensive and do not scale to large systems, which is why we do not consider them in this work.

In the following paragraphs, we describe the state-of-the-art approaches and to what degree they exploit the insights that we consider in this work. Table 1 summarizes the approaches, including the number of executions and accuracy for our running example (Lines 1–15).

Brute Force is a black-box approach that measures all configurations of a system. It is rarely used in practice due to its obvious

Table 1: Comparison of the state-of-the-art approaches. The cost and accuracy values correspond to applying each approach to the running example in Fig. 2.

Approach	Type	Insights			Quality	
		I	O	LID	Cost	Accurracy
Brute Force	Black-box	✗	✗	✗	1024	High
Sampling	Black-box	✗	✓	✓	—[1]	—[1]
SPLat	White-box	✓	✗	✗	6	High
Family-Based	White-box	✗	✓	✗	1	—[2]
ConfigCrusher	White-box	✓	✓	✓	4	High

I = Irrelevance. O = Orthogonality. LID = Low Interaction Degree.

[1] Depends on sampling strategy (e.g., t-wise sampling [45])

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

scalability issues for all but the smallest configuration spaces. In our running example, among other inefficiencies, it will execute irrelevant configurations (e.g., all configurations that explore all values of options D − J).

Sampling and learning approaches are black-box techniques that sample a subset of the configuration space (e.g., random sampling, feature-wise, pair-wise [45], design of experiments [47], or combinatorial sampling [2, 25, 28, 29, 50]) and use a learning algorithm (e.g., regression, classification and regression trees [23, 55, 57–59], or Gaussian Processes [36]) to extrapolate a performance 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 (e.g., t-wise sampling [45, 50]), they might miss important interactions leading to inaccurate models. In addition, due to their lack of insight of the internals of the program, none of these approaches recognizes irrelevant options.

SPLat [39] is a white-box approach that dynamically tracks the configurations that produce distinct execution paths. It reexecutes the program until all configurations with distinct paths are explored. While it ignores irrelevant options, since they do not produce different paths, it is technically a brute-force approach on the options that it encounters during execution; each time an option is reached in a new path, it explores both values for that option. 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, its brute-force nature can lead to scalability issues.

Family-Based Performance Measurement [60] is a white-box approach that uses a static mapping between options to code regions and instruments the program to measure the execution time spent in the regions. Subsequently, it executes the program 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. Current implementations, however, derive the static map from compile-time variability mechanisms (preprocessor directives) [4, 60] 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 control-flow interactions, 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 models otherwise.

All the surveyed approaches build performance 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 model. Furthermore, none of the approaches, except for the family-based approach, but with severe limitations, can associate the resulting performance model with regions in the source code, which can be helpful to understand and debug individual components of a system.

Our approach. We introduce **ConfigCrusher**, a novel 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 models. 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 executed (Table 2). Next, it will *instrument* the program's regions, reducing the instrumentation overhead through additional optimization (Sec. 3.3). Finally, it will *build* local performance models (Sec. 3.5) for each region based on the performance observed when executing the instrumented program with the compressed set of configurations (Table 2) and, subsequently, aggregate them to produce an accurate global *performance model* (Equation 1).

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 program 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 model. We proceed in five steps: *Identifying Configuration-Dependent Regions* (Sec. 3.1): We perform a data-flow analysis that identifies the control-flow decisions that depend on configuration options and the code regions affected by these decisions. *Compressing Configuration Set* (Sec. 3.2): We identify the smallest set of configurations that cover all relevant executions of all regions. *Instrumenting Regions* (Sec. 3.3): We instrument the regions in the program to track their execution time in different configurations and optimize the instrumentation to reduce measurement overhead. *Executing the Instrumented Program* (Sec. 3.4): We execute the instrumented program to measure performance. *Building the Performance Model* (Sec. 3.5): We build local performance models based on the measured code regions' performance and, subsequently, aggregate them to produce a performance model.

3.1 Identifying Config.-Dependent Regions

As first step, we identify the control-flow decisions that depend on configuration options and the regions affected by these decisions. 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 program (Sec. 3.3).

To obtain the statement influence map, we use a data-flow analysis (Sec. 4) to track how options are used in control-flow decisions. 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 program, we can identify, for each control-flow decision, the set of options that may influence this decision. Finally, we produce the map, mapping all statements that are control-dependent on a decision to all influencing options.

Example: The options in our running example in Fig. 2 (Lines 1–15) are the fields A – J. Lines 4–10 are not influenced by any options, Line 2 is influenced by the set of options $\{A, C\}$, Lines 11–14 by $\{A\}$, and Line 15 by $\{A, B\}$.

With this data-flow analysis, we can reason about *Irrelevance*, *Orthogonality*, and *Low Interaction Degree*: Options that influence no control-flow decisions are irrelevant and never appear in the resulting map. Likewise, we can identify which set of options interact on which control-flow decisions 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 D – J are irrelevant in the program.

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 performance of the program. We use the set of all interactions (codomain 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 program (Sec. 3.4).

Intuitively, our goal is to execute the program 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 refer to 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 (e.g., such that they cover all combinations of pairs of options [2, 25, 28, 29, 42]). 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 and calculate all combinations of each set—these are the minimum combinations we need to cover. Next, we compress the set of configurations 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 associated with the sets of options $\{A\}$, $\{A, B\}$, and $\{A, C\}$. That is, we need to cover 2 combinations for $\{A\}$ (with A enabled and disabled), 4 combinations of $\{A, B\}$, and 4 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

Algorithm 1: Configuration compression

Input: $IO : \mathcal{P}(\mathcal{P}(O))$
Output: $CC : \mathcal{P}(C)$

1 Function compress_configurations
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

Algorithm 2: Identify regions

Input: CFG, $S \rightarrow \mathcal{P}(O)$
Output: $R \rightarrow \mathcal{P}(E) \times \mathcal{P}(E)$, $R \rightarrow \mathcal{P}(O)$

1 Function identify_regions
2   **for** each $stmt \in$ statements(CFG) **do**
3 $idom :=$ idom($stmt$, CFG) ▷ Get immediate dominator
            ▷ influence(s): $S \rightarrow \mathcal{P}(O)$
4   **if** influence($stmt$) $\neq \varnothing$ $\wedge$ influence($stmt$) $\neq$ influence($idom$) **then**
5 $r :=$ new Region()
                ▷ Omit incoming edges from loops
6   **for** each $edge \in$ in($stmt$, CFG) **do**
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($stmt$) $=$ influence($pdom$) **do**
11 $pdom :=$ ipdom($pdom$, CFG)
12   **end**
13   **for** each $edge \in$ in($pdom$) **do**
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

---

remaining sets, we can create a merged compressed set of 4 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

Next, we instrument the program 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 program (Sec. 3.4) with the compressed set of configurations (Sec. 3.2) to build the performance model (Sec. 3.5).

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 and end edges in a method. A region starts before the first statement influenced by a set of options (indicated by the statement influence map; Sec. 3.1) 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 decision meet again (i.e., the immediate post-dominator). 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 decision.

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 program to measure the performance of

code not influenced by any options. The result of executing an instrumented program is the total time spent in each region.

Example: Fig. 3a shows the 3 regions that we instrument in the control-flow graphs of our running example in Fig. 2. One region contains statements $\{11, 13, 14\}$ since statement 11 is influenced by $\{A\}$ and its last post-dominator without that influence is 15.

Optimization. Although Algorithm 2 can correctly identify regions in a program, we observed excessive overhead in execution even in small programs (see Sec. 5.3). We found that the overhead arose from redundant, nested regions, and regions executed repeatedly in loops, and we identified optimizations to reduce measurement overhead through instrumenting different regions without altering the performance 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 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 model.

Example: Consider the statement in Line 9 and Region 2 in our running example. The statement takes 1 second to execute and Region 2 takes 2 seconds to execute when option A is enabled, from which we can derive the partial performance model T = 1 + 2A. 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 program. 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 model that is generated and without increasing measurement effort.* Merging two
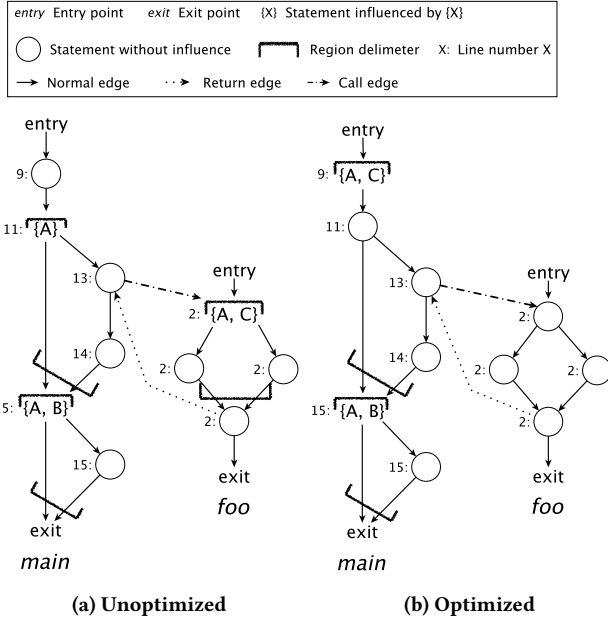
Figure 3: Unoptimized and optimized instrumented control-flow graphs of the methods of Fig. 2. For simplicity, statements within the regions and those before line 9 in method `main` are ignored.

Algorithm 3: Propagate influence down

Input: *stmt*, *CFG*, $S \rightarrow \mathcal{P}(O)$
Output: $S \rightarrow \mathcal{P}(O)'$
1 **Function** *propagate_down*
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* ∈ *pstmts* **do**
         ▷ influence(*ps*): $S \rightarrow \mathcal{P}(O)$
5   **if** influence(*ps*) ⊂ influence(*stmt*) **then**
6   influence(*ps*) := influence(*stmt*)
7   **end**
8   **end**
9 **end**

Algorithm 4: Propagate regions up

Input: *stmt*, *CFG*, $S \rightarrow \mathcal{P}(O)$, $GI : \mathcal{P}(O)$
Output: $S \rightarrow \mathcal{P}(O)'$
1 **Function** *propagate_up*
2   **for** each *pred* ∈ preds(*stmt*, *CFG*) **do**
         ▷ influence(*s*): $S \rightarrow \mathcal{P}(O)$
3   **if** influence(*pred*) ∪ influence(*stmt*) ∈ *GI* ∧
         influence(*pred*) ≠ influence(*pred*) ∪ influence(*stmt*)
         then
4   influence(*pred*) := influence(*stmt*)
5   **end**
6   **end**
7 **end**

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 program, 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 model. By merging and pulling out 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 program.

Example: Consider regions 2 and 3 in our running example. Region 2 is influenced by $\{A\}$ and region 3 by $\{A, B\}$. We sample all 4 combinations of A and B to conclude that Region 2 takes 2 seconds to execute when option A is enabled and Region 3 takes 3 seconds when both A and B are enabled, resulting in the partial performance model T = 2A + 3AB. 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 model when we calculate the actual influence of enabling both A and B (i.e., +3 seconds) (Sec. 3.5). 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., intraprocedurally), as well as across graphs (i.e., interprocedurally), to expand, merge,

and pull out regions. The propagation in Algorithm 3 merges consecutive regions and expands where regions 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 model that we generate, but significantly reduce the overhead of measuring the instrumented program. 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 5). Fig. 3b presents an optimized instrumentation that prioritizes our goal, in which we pulled out the region in the callee.

3.4 Executing the Instrumented Program

After instrumentation, we can now execute the program 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 in each executed configuration to a corresponding execution time T ($CP : CC \rightarrow (R \rightarrow T)$).

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 programs. We tried building a trace

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

Configurations				Regions		
A	B	C	D	Base	OR1 ≡ {A, C}	OR2 ≡ {A, B}
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

of regions and processing the execution times after the program finished executing. However, due to the large number of regions that were executed, the programs ran out of memory. Our evaluation shows that the dynamic processing incurs low overhead (Sec. 5.3).

Example: Table 2 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 Model

Our final step is to build the *performance 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 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 $T_r = t_1 XY + t_2 X \neg Y + t_3 \neg XY + t_4 \neg X \neg Y$ for a region r with options X and Y (or $T_r = t_4 + (t_2 - t_4)X + (t_3 - t_4)Y + (t_1 - t_2 - t_3 + t_4)XY$ to highlight the influence of options and avoid negated terms) [58]. If a region has been executed multiple times for the same combination of options, 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 model Π is obtained by aggregating all local performance models. Note that local models can be useful for understanding and debugging the individual regions in the program.

Example: With the measurement times in Table 2, we build the local models of the base region as $T_{Base} = 1$ (averaged over 4 executions) and of the other two regions as $T_{OR1} = 3A \neg C + 6AC = 3A + 3AC$ and $T_{OR2} = 3AB$, resulting in the overall model $\Pi = 1 + 3A + 3AB + 3AC$.

4 IMPLEMENTATION

We implemented ConfigCrusher for Java programs and made it publicly available [1]. 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.

Limitations. Following various state-of-the-art approaches [2, 23, 25, 39, 44–46, 55, 57–60], we implemented ConfigCrusher with support for Boolean options, which required discretizing numeric options, and assume determinism in the subject system's execution time.

There are several strategies to track data-flow in a program (Sec. 3.1): manual tracking [44], static analysis [5, 17, 18, 44, 52, 53], and dynamic analysis [6, 7, 10, 46, 49, 54, 69]. We used the state-of-the-art [16, 51, 52, 66] object-, field-, context-, and flow-sensitive

static taint analysis provided by FlowDroid [5]. 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 decisions as sinks.

At its core, the used static taint analysis is unsound and can lead to overtainting [52, 66], which can affect the results of our approach. In addition, despite the high precision of FlowDroid [5], the analysis is challenged by the size of the call graph, which restricts the size of the programs that can be analyzed [5, 8, 11, 16, 43, 51, 52, 66]. Similar to other approaches [8, 44, 52], we reduced the precision of some of the FlowDroid specific settings (e.g., used an unexceptional control-flow graph) [1] for a faster analysis (the analysis ran out of memory for systems 3 – 10 in Table 3 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 programs from our evaluation since the server either used all of its memory or did not finish the analysis after 4 hours. Avdiienko et al. [8] experienced similar results on a server with more RAM and CPU cores. Nevertheless, our evaluation (Sec. 5) demonstrates that our implementation produced accurate and informative performance models, signifying that our approach is robust despite the levels of unsoundness and overtainting of FlowDroid. With advancements in scaling taint analyses [3, 9, 11, 14, 16, 19, 43, 63, 71], we expect that our results and benefits will generalize to larger systems. We conjecture that similarly accurate results can be achieved with other taint analysis implementations.

5 EVALUATION

We first evaluate ConfigCrusher against state-of-the-art approaches for performance modeling in terms of the cost to generate performance models and their accuracy. Subsequently, we explore the usefulness of ConfigCrusher' local performance 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 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 models efficiently. As discussed in Sec. 3.3, we observed an excessive amount of overhead when executing our unoptimized instrumented programs. 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 black-box approaches is that it builds local performance models, which indicate whether and how the options locally influence the performance of a system. In an exploratory analysis, we examine the local performance 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

Table 3: Configurable subject systems.

ID	Name	Domain	# SLOC	# Opt.	# Conf.
1	Running Example	Example	69	10	1024
2	Pngtastic Counter	Processing	1250	5	32
3	Pngtastic Optimizer	Optimization	2553	5	32
4	Elevator	SPL – Simulator	575	6	20
5	grep	Command line	2152	7	128
6	Kanzi	Compression	21K	7	128
7	Email	SPL – Simulator	696	9	40
8	Prevayler	Database	1328	9	512
9	sort	Command line	2163	12	4096
10	Density Converter	Processing	1359	22	2^{22}

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 3. We selected a representative set of 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), and (c) systems with deterministic execution time (we sampled each system multiple times with different approaches and observed execution times within usual measurement noise). We included systems that have been used in previous studies for comparability of results (4, 7, 8) [39, 60, 62] and new configurable systems with 800+ stars and 130+ forks combined at the time of writing (2, 3, 5, 6, 8, 9, 10). In the following sections, we consider the entire program of Fig. 2 (Lines 1−21) as the Running Example (1) to showcase the potential of our approach.

Due to their novelty, white-box approaches impose strict limitations on the systems they can analyze [39, 60, 62] (Sec. 2). ConfigCrusher lifts some of these limitations; we consider data-flow interactions and do not limit the analysis to specific program implementations, which expands the types of programs 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 programs and their number of configuration options. We acknowledge the size of the real-world systems that we evaluate and that black-box approaches are able to analyze larger systems. However, at this stage, we want to showcase the benefits and potential of white-box analyses and expect that, with improvements to the used data-flow analysis [3, 9, 11, 14, 16, 19, 43, 63, 71], our implementation will analyze larger systems (Sec. 4). Nevertheless, we conjecture that the selected systems are representative of larger systems since we observed the insights of *Irrelevance, Orthogonality, and Low-Interaction Degree* in configurable systems that we exploit. Hence, we expect to obtain similar results (Sec. 5.2) in larger systems with our approach with a more scalable implementation of the static taint analysis. Note the general trend (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 programs which the other approaches could not; SPLat did not scale to Density Converter (10) and the family-based approach could not analyze any system besides the software product lines.

From an initial sample of 12 systems, for which the static analysis terminated, we excluded 2 systems since the analysis indicated that all options interact (i.e., our approach equals the Brute Force

approach). A third system, Elevator (4), purposely built to have all options interact [39, 46, 62], was included since it is one of the two systems that the family-based approach can analyze. That is, potentially in 3 out of 12 systems, our insight of low interaction degree does not hold. Even if these results were similar with better static analyses that reduce overtainting, our results confirmed that not all options interact in most real-world systems [34, 39, 40, 46, 49, 60].

For the systems that we analyzed, we extracted the configuration options from the projects' documentation and executed a representative test scenario and workload provided by the system [1].

5.2 RQ1: Comparison to State-of-the-Art

With RQ1, we evaluate the cost and accuracy of the performance 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 in Table 4 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) [45] with stepwise linear regression [55, 57–59], Brute-Force, SPLat [39], and the family-based approach [60]. We excluded random sampling since research [34, 36, 45, 57] 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 program (Sec. 2). 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 decisions [56]. 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 laptop with 16 GB of RAM running OS X 10.13. For each configuration, we initiated one VM invocation and ran the configuration [20]. 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.

Metric – Cost. 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.

Metric – Error. 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

Table 4: Cost and error comparison.

	Cost (Configurations [Time])						Prediction Error (MAPE) compared to GT					
S	BF/SA	FW	PW	SAD	FB[1]	CC[2]	BF/SA	FW	PW	SAD	FB[1]	CC
1	1024 [1.90h]	10 [16.64s]	56 [2.15m]	512 [56.86m]	N/A	8 [33.75s + 4.37s]	∅	56.91↑	6.22↑	0.18↑	N/A	0.07
2	32 [2.91m]	5 [27.15s]	16 [1.45m]	24 [2.18m]	N/A	4 [21.88s + 7.80s]	∅	0.80	1.94↑	1.33	N/A	1.10
3	32 [42.23m]	5 [1.60m]	16 [9.97m]	16 [21.04m]	N/A	10 [10.73m + 30.60s]	∅	19.67↑	0.91	0.99	N/A	1.07
4	20 [10.80m]	3 [50.03s]	9 [3.26m]	20 [10.80m]	1 [49.50s]	64 [—]	∅	51.09	1.48	∅	2.72	∅
5	128 [10.56m]	7 [22.09s]	29 [1.85m]	48 [3.49m]	N/A	64 [5.14m + 10.20s]	∅	32.14↑	114.74↑	1.94↓	N/A	3.58
6	128 [1.18h]	7 [1.46m]	29 [8.76m]	64 [35.39m]	N/A	64 [35.39m + 12.60s]	∅	1.86	1.29	1.21	N/A	2.66
7	40 [16.85m]	4 [23.46s]	11 [1.68m]	40 [16.85m]	1 [1.08m]	8 [1.47m + 12.81s]	∅	100↑	44.23↑	∅	2.34↓	23.02
8	512 [3.69h]	9 [2.69m]	46 [15.96m]	144 [1.51h]	N/A	32 [14.49m + 12.60s]	∅	111.21↑	29.23↑	2.95↓	N/A	9.23
9	1298 [18.41h]	12 [13.10m]	79 [1.43h]	48 [42.84m]	N/A	256 [3.67h + 21.58s]	∅	89.96↑	653.03↑	2.38↑	N/A	1.57
10	1414 [14.70h]	22 [21.26m]	254 [4.12h]	+24h[3]	N/A	256 [2.13h + 42.13s]	∅	635.24↑	218.86↑	N/A[3]	N/A	4.32

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

[1] Not applicable to systems without static map derived from compile-time variability (Sec. 2).

[2] Time includes the overhead of the static taint analysis (Sec. 3.1).

[3] No data was collected due to timeout (Sec. 5.2).

configurations that the approach did not sample. We also calculated the error across all configurations [1]. We used the multiple comparison $\tilde{T}$-procedure [41] 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 and error results in Table 4. 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. Table 4 illustrates 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 (Sec. 2). 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, SPLatDelayed 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 program), thus indicating the limitations of the approach.

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.

Discussion - Characteristics of Interactions: ConfigCrusher allowed us to observe and confirm 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 program; 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 ~4 seconds or ~61 seconds. We consider these three systems as outliers since previous empirical studies [4, 36, 40, 57] have shown that the performance of most configurable systems changes based on the selected configuration.

Discussion - Source of prediction error: Regarding ConfigCrusher's prediction error of EMAIL (7), the system has a feature model [4] 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, see Sec. 4). We were unable to manually determine the correct interaction of the problematic regions. We conjecture that, since the program 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.

Table 5: 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. R = # of instrumented regions. RC = # of executed regions.

In summary, 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 programs. 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 the unoptimized (Algorithm 2) and optimized (Algorithm 2 + the two propagation algorithms [1]) instrumented programs to the ground truth.

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

Metric - Static and dynamic overhead. 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.

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

Results: Table 5 shows the results of our analysis. ConfigCrusher's optimized instrumentation (Algorithms 2 + the two propagation algorithms [1]) 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 programs in a reasonable amount of time.

Only the Running Example (1), Email (7), and Density Converter (10) had a low unoptimized dynamic overhead and executed in similar time compared to the original programs. These systems do not have deeply nested regions, which does not increase the overhead of the dynamic analysis (Sec. 3.4). 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 programs compared to the original programs of the Running Example (1), Grep (5), and Email (7) to measurement noise.

In summary, ConfigCrusher's optimized instrumentation incurs lower overhead compared to the unoptimized instrumentation.

Table 6: Influence of options analysis of local performance models.

	PNIO		PIO			
	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. PIO = Performance influenced by options. NEG = Negligible execution time (region which contribute < 5% of the execution time of the system). ID = Interaction degree.

5.4 RQ3: Perf. Influence of Options in Regions

One of the benefits of ConfigCrusher over black-box approaches is that it builds local performance models, which indicate whether and how the 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 models to determine the local influence of options on performance. Subsequently, we examined the source code regions corresponding to the local models 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 models, such as the following models from Pngtastic Optimizer (3):

- T_{9cb} = 0.0
- T_{b15} = 1.5
- T_{b1a} = 0.2 + 82.5Compress + 142.1CompressIter + ...,

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.

Results: Table 6 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 program. Note that in these regions only subsets of all options interact. The local performance 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 decision within a loop, which either manipulated data structures or performed I/O operations (Table 6). 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.

Discussion - Local models. The local performance 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 decisions with non-negligible execution time depended on options (Table 6), yet the same branch of the decision 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 decision where this option was used always executed the same branch if the value was > 0. Finding these inconsistensies, common in configurable systems [12, 26, 53, 68], might be useful for developers and maintainers to debug these type of systems.

Interestingly, some options influenced only regions with negligible execution time (Table 6). This behavior was surprising since we expected, based on the systems' documentation, that the options would influence the performance of the systems (e.g., the options that influence Region 9cb above [1]). 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.

In summary, ConfigCrusher helped us to identify that the influence of options on performance is localized to a few regions in a program. The options influence loops or control-flow decisions 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. Nevertheless, we argue that we selected a representative sample of configurable systems from different domains that showcase the benefits of our approach.

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 (Sec. 5), our approach is robust and produces accurate results with the settings that we selected [1].

6 RELATED WORK

In Sec. 2, we described closely related state-of-the-art black-box and white-box approaches for performance modeling and compared them to ConfigCrusher. In this section, we discuss additional research to position ConfigCrusher 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 [17, 30, 46, 49, 53, 54, 61, 65]. Thüm et al. [64] presented a comprehensive survey of analyses for software product lines also applicable to configurable systems.

Similar to our approach, Lillack et al. [44] 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.

Testing configurable systems: Combinatorial Testing [25, 28, 29, 29, 42, 50] is an approach to reduce the number of samples to test a program by satisfying a certain coverage criterion. Similarly, Souto et al. [62] improved SPLat [39] to use sampling heuristics [45] to select what configurations to sample. While both these approaches scale to large systems, they make assumptions on how options interact in the program 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, have been implemented to identify performance hot spots [15, 21, 48, 70]. For example, Castro et al. [13] used both techniques to identify hot spots that can be isolated and replayed as standalone programs 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 [22, 24, 32]. For example, Hao et al. [27] 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 by considering how energy consumption varies with the configuration of the system.

7 CONCLUSION

This paper 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 decisions and instruments regions corresponding to those decisions for performance measurement. Our evaluation on 10 real-word systems shows that ConfigCrusher builds similar or more accurate performance models than other approaches with lower cost. In contrast to black-box approaches, ConfigCrusher provides additional information of the components of a system, which can aid stakeholders to analyze, optimize, and debug them.

REFERENCES

[1] [n. d.]. https://bit.ly/2ARTc0H.
[2] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. 2016. IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. ACM, New York, NY, USA, 144–155.

[3] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. 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, 31–36. https://doi.org/10.1145/3088515.3088521

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer-Verlag, Berlin/Heidelberg, Germany.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 259–269.

[6] Thomas H. Austin and Cormac Flanagan. 2009. Efficient Purely-dynamic Information Flow Analysis. In *Proc. Workshop Programming Languages and Analysis for Security (PLAS)*. ACM, New York, NY, USA, 113–124.

[7] Thomas H. Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. In *Proc. Symp. Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 165–178.

[8] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining Apps for Abnormal Usage of Sensitive Data. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Press, Piscataway, NJ, USA, 426–436.

[9] Paulo Barros, Rene Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo dAmorim, and Michael D. Ernst. 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, 669–679.

[10] Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating Dynamic Data Flow in Commodity Jvms. *SIGPLAN Notices* 49, 10 (Oct. 2014), 83–101.

[11] Eric Bodden. 2018. Self-adaptive Static Analysis. In *Proc. Int'l Conf. Software Engineering (ICSE): New Ideas and Emerging Results*. ACM, New York, NY, USA, 45–48.

[12] Mikaela Cashman, Myra B. Cohen, Priya Ranjan, and Robert W. Cottingham. 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, 757–767.

[13] Pablo De Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. 2015. CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization. *ACM Trans. Archit. Code Optim. (TACO)* 12, 1, Article 6 (April 2015), 24 pages.

[14] Maria Christakis and Christian Bird. 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, 332–343.

[15] Jürgen Cito, Philipp Leitner, Christian Bosshard, Markus Knecht, Genc Mazlami, and Harald C. Gall. 2018. PerformanceHat: Augmenting Source Code with Runtime Performance Traces in the IDE. In *Proc. Int'l Conf. Software Engineering: Companion Proceeedings*. ACM, New York, NY, USA, 41–44.

[16] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. 2017. Just-in-time Static Analysis. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)*. ACM, 307–317.

[17] Z. Dong, A. Andrzejak, D. Lo, and D. Costa. 2016. ORPLocator: Identifying Read Points of Configuration Options via Static Analysis. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*. 185–195.

[18] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 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, 393–407.

[19] Diego Garbervetsky, Edgardo Zoppi, and Benjamin Livshits. 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, 442–453. https://doi.org/10.1145/3106237.3106261

[20] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. *SIGPLAN Notices* 42, 10 (Oct. 2007), 57–76.

[21] Brendan Gregg. 2016. The Flame Graph. 59, 6 (May 2016), 48–57.

[22] Jiaping Gui, Ding Li, Mian Wan, and William G. J. Halfond. 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, 1–7.

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

[24] Ashish Gupta, Thomas Zimmermann, Christian Bird, Nachiappan Nagappan, Thirumalesh Bhat, and Syed Emran. 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, Article 40, 8 pages.

[25] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2018. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. *Empirical Software Engineering* (July 2018).

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

[27] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. 2013. Estimating Mobile Application Energy Consumption Using Program Analysis. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Press, Piscataway, NJ, USA, 92–101.

[28] A. Hervieu, B. Baudry, and A. Gotlieb. 2011. PACOGEN: Automatic Generation of Pairwise Test Configurations from Feature Models. In *Int'l Symposium Software Reliability Engineering*. 120–129.

[29] Aymeric Hervieu, Dusica Marijan, Arnaud Gotlieb, and Benoit Baudry. 2016. Optimal Minimisation of Pairwise-covering Test Configurations Using Constraint Programming. *Information and Software Technology* 71 (March 2016), 129 – 146.

[30] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. 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, 199–212.

[31] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. 2012. A User Survey of Configuration Challenges in Linux and eCos. In *Proc. Workshop Variability Modeling of Software-Intensive Systems (VAMOS)*. ACM, 149–155. https://doi.org/10.1145/2110147.2110164

[32] Reyhaneh Jabbarvand, Alireza Sadeghi, Hamid Bagheri, and Sam Malek. 2016. Energy-aware Test-suite Minimization for Android Apps. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)*. ACM, New York, NY, USA, 425–436.

[33] Pooyan Jamshidi and Giuliano Casale. 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)*. 39–48.

[34] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. 2017. Transfer Learning for Performance Modeling of Configurable Systems: An Exploratory Analysis. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. ACM, New York, NY, USA, 13.

[35] Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. 2018. Learning to Sample: Exploiting Similarities Across Environments to Learn Performance Models for Configurable Systems. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)*. ACM, New York, NY, USA, 12.

[36] Pooyan Jamshidi, Miguel Velez, Christian Kästner, Norbert Siegmund, and Prasad Kawthekar. 2017. 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, 31–41.

[37] Dongpu Jin, Xiao Qu, Myra B. Cohen, and Brian Robinson. 2014. Configurations Everywhere: Implications for Testing and Debugging in Practice. In *Companion Proc. Int'l Conf. Software Engineering*. ACM, New York, NY, USA, 215–224.

[38] Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. 2011. Reducing Combinatorics in Testing Product Lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*. ACM, New York, NY, USA, 57–68.

[39] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo d'Amorim. 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, 257–267.

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

[41] Frank Konietschke, Ludwig A Hothorn, and Edgar Brunner. 2012. Rank-based multiple test procedures and simultaneous confidence intervals. *Electronic Journal of Statistics* 6 (2012), 738–759.

[42] D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. 2013. *Introduction to Combinatorial Testing* (1st ed.). Chapman & Hall/CRC.

[43] J. Lerch, J. Späth, E. Bodden, and M. Mezini. 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, 619–629.

[44] Max Lillack, Christian Kästner, and Eric Bodden. 2018. Tracking Load-time Configuration Options. *IEEE Transactions on Software Engineering* 44, 12 (12 2018), 1269–1291. https://doi.org/10.1109/TSE.2017.2756048

[45] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, New York, NY, USA, 643–654.

[46] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity: Measuring Interactions in Highly-configurable Systems. In *Proc. Int'l Conf. Automated Software Engineering*

(ASE). ACM, New York, NY, USA, 483–494.

[47] Douglas C. Montgomery. 2006. *Design and Analysis of Experiments*. John Wiley & Sons.

[48] Shaikh Mostafa, Xiaoyin Wang, and Tao Xie. 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, 23–34.

[49] Thanhvu Nguyen, Thanhvu Koc, Javran Cheng, Jeffrey S. Foster, and Adam A. Porter. 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.

[50] Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *ACM Comput. Surv. (CSUR)* 43, 2, Article 11 (Feb. 2011), 29 pages.

[51] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android Taint Analysis Tools Keep Their Promises?. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)*. ACM, New York, NY, USA, 331–341. https://doi.org/10.1145/3236024.3236029

[52] Lina Qiu, Yingying Wang, and Julia Rubin. 2018. Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)*. ACM, New York, NY, USA, 176–186.

[53] Ariel Rabkin and Randy Katz. 2011. Static Extraction of Program Configuration Options. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, New York, NY, USA, 131–140.

[54] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. 2010. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, New York, NY, USA, 445–454.

[55] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 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, 342–352.

[56] Pierre-Yves Saumont. 2017. Lazy Computations in Java with a Lazy Type.

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

[58] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. 2012. Predicting Performance via Automated Feature-interaction Detection. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Press, Piscataway, NJ, USA, 167–177.

[59] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012. SPL Conqueror: Toward Optimization of Non-functional Properties in Software Product Lines. *Software Quality Journal*

20, 3-4 (Sept. 2012), 487–517.

[60] Norbert Siegmund, Alexander von Rhein, and Sven Apel. 2013. Family-Based Performance Measurement. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. ACM, New York, NY, USA, 95–104.

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

[62] Sabrina Souto, Marcelo d'Amorim, and Rohit Gheyi. 2017. Balancing Soundness and Efficiency for Practical Testing of Configurable Systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Press, Piscataway, NJ, USA, 632–642.

[63] Johannes Späth, Karim Ali, and Eric Bodden. 2017. IDEal: Efficient and Precise Alias-aware Dataflow Analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 99 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133923

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

[65] Bo Wang, Leonardo Passos, Yingfei Xiong, Krzysztof Czarnecki, Haiyan Zhao, and Wei Zhang. 2013. SmartFixer: Fixing Software Configurations Based on Dynamic Priorities. In *Proc. Int'l Software Product Line Conference (SPLC)*. ACM, New York, NY, USA, 82–90. https://doi.org/10.1145/2491627.2491640

[66] Yan Wang, Hailong Zhang, and Atanas Rountev. 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, 18–23.

[67] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 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, 307–319.

[68] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do Not Blame Users for Misconfigurations. In *Proc. Symp. Operating Systems Principles*. ACM, New York, NY, USA, 244–259.

[69] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, Dynamic Information Flow for Database-backed Applications. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 631–647.

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

[71] Qirun Zhang and Zhendong Su. 2017. Context-sensitive Data-dependence Analysis via Linear Conjunctive Language Reachability. In *Proc. Symp. Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 344–358. https://doi.org/10.1145/3009837.3009848