White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems

Miguel Velez
Carnegie Mellon University

Pooyan Jamshidi
University of South Carolina

Norbert Siegmund
Leipzig University

Sven Apel
Saarland Informatics Campus
Saarland University

Christian Kästner
Carnegie Mellon University

Abstract—**Performance-influence models can help stakeholders understand how and where configuration options and their interactions influence the performance of a system. With this understanding, stakeholders can debug performance behavior and make deliberate configuration decisions. Current black-box techniques to build such models combine various sampling and learning strategies, resulting in tradeoffs between measurement effort, accuracy, and interpretability. We present Comprex, a white-box approach to build performance-influence models for configurable systems, combining insights of local measurements, dynamic taint analysis to track options in the implementation, compositionality, and compression of the configuration space, without relying on machine learning to extrapolate incomplete samples. Our evaluation on 4 widely-used, open-source projects demonstrates that Comprex builds similarly accurate performance-influence models to the most accurate and expensive black-box approach, but at a reduced cost and with additional benefits from interpretable and local models.**

I. INTRODUCTION

Configuring software systems is often challenging. In practice, many users execute systems with inefficient configurations in terms of performance and, often directly correlated, energy consumption [20, 21, 31, 54]. While users can adjust configuration options to tradeoff between performance and the system's functionality, this configuration task can be overwhelming; many systems, such as databases, Web servers, and video encoders, have numerous configuration options that may interact, possibly producing unexpected and undesired behavior. For this reason, understanding how options and their interactions affect the system's performance and making suitable configuration decisions can be difficult [2, 78]; users often have, at most, vague intuitions [25, 63, 80]. We aim at efficiently building performance-influence models, without machine learning, that characterize how options influence the system's performance. We intent our models to be easy to inspect and interpret by developers and users to ease performance debugging and to make informed configuration decisions.

Good performance-influence models help users understand *how*, *where*, and *why* options and their interactions influence the performance of the system in a specific way [37, 63]. In the situations where these models are useful, simple manual experiments are often not practical due to a system's exponentially large configuration space. For example, a developer considering whether to enable encryption would likely expect that encryption would slow down the system. However, the
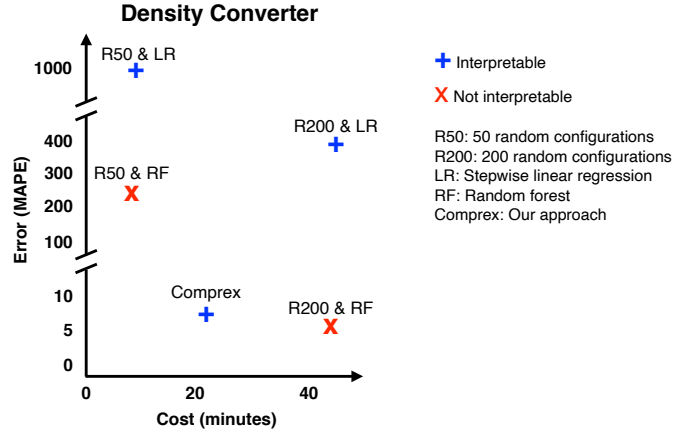

Fig. 1: Our white-box approach, Comprex, builds similarly accurate (low error) performance-influence models to the most accurate and expensive sampling and machine learning approaches, but can often be built more efficiently (low cost) and are interpretable. Additionally, our models are significantly more accurate than other sampling and machine learning approaches that produce interpretable linear models.

developer may need to perform *numerous* experiments to quantify the effect of the option itself and of its interactions with other configuration decisions in a given environment and workload. To understand why and where encryption causes the slow down, a developer would have to carefully profile or debug the system in different configurations.

Most existing techniques build global performance-influence models and treat the system as a *black box*, measuring the system's execution in an environment with a given workload for a subset of all configurations, and learning a model from these observations. The sampling (i.e., selecting which configurations to measure) and learning techniques used [13, 15, 34, 35, 48, 60, 63–65] result in tradeoffs among the cost to build the models and the accuracy and interpretability of the models [13, 34, 37]. For example, larger samples are more expensive, but usually lead to more accurate models; random forests, with large enough samples, tend to learn more accurate models than those built with linear regression, but the models are harder to interpret when users want to understand performance or debug their systems [13, 34, 48] (see Fig. 1).

In this paper, we introduce Comprex, a *white-box* approach to build *accurate*, *interpretable*, and *local* performance-

influence models at low cost. Our approach departs drastically from traditional black-box approaches: It analyzes and instruments the source code to accurately capture configuration-specific performance behavior, without relying on machine learning to extrapolate incomplete samples. We reduce measurement cost by *simultaneously analyzing and measuring* multiple code regions of the system, building a local linear performance-influence model per region with a few configurations (an insight we call *compression*). Subsequently, we *compose* the local models into a global model for the entire system. We use an *iterative dynamic taint analysis* to identify *where* and *how* load-time configuration options influence control-flow statements in the system, through *control-flow and data-flow dependencies*.

Our empirical evaluation on 4 medium- to large-scale, widely-used, open-source projects demonstrates that Comprex builds similarly accurate performance-influence models to those learned with the most accurate and expensive black-box approach, but at a reduced cost. Furthermore, Comprex generates *interpretable* and *local* models, which quantify the influence of individual options and interactions on performance, and can even map the influence to code regions.

In summary, we make the following contributions:

- An approach that combines *compression* and *composition* to accurately infer the influence of options on the performance of numerous independent regions of a system with a few configurations, composing the influence in a full performance model.
- An iterative dynamic taint analysis to identify how configurations influence the performance of independent regions and a conjecture to reduce the cost of running the analysis to build accurate performance-influence models.
- An empirical evaluation on 4 medium- to large-scale, open-source systems demonstrating that our white-box approach builds interpretable, but similarly accurate models with less cost compared to the state of the art.
- A replication package with subject systems, experimental setup, and data of several months of measurements [74].

II. PERFORMANCE MODELS FOR CONFIG. SYSTEMS

There is substantial literature on modeling the performance of software systems [e.g., 13, 34, 37, 75]. Performance-influence models solve a specific problem: Explaining how options and their interactions influence a system's performance for a given workload and environment, designed to help users understand performance and make deliberate configuration decisions.

Traditional performance models are typically used by designers and developers to model and analyze the performance of a system's architecture (e.g., using Queuing networks, Petri Nets, and Stochastic Process Algebras) and workload in the *design stage* of a project [22, 38, 62]. At this stage, design decisions are usually modeled as configuration options [5, 11]. By contrast, performance-influence models describe the performance behavior of a *system implementation*

```
1  def main(List workload)
2      a = getOpt("A"); b = getOpt("B");
3      c = getOpt("C"); d = getOpt("D");
4      ...          // execution: 1s
5      int i = 0;                T_main = 3 - 1A
6      if(a)        // variable depends on option A
7          ...      // execution: 1s
8          foo(b);  // variable depends on option B
9          i = 20;
10     else
11         ...      // execution: 2s
12         i = 5;
13     while(i > 0)
14         bar(c);  // variable depends on option C
15         i--;
16 def foo(boolean x)           T_foo = 1A + 3AB
17     if(x) ...    // execution: 4s
18     else ...     // execution: 1s
19 def bar(boolean x)           T_bar = 5 + 15A + 10C + 30AC
20     if(x) ...    // execution: 3s
21     else ...     // execution: 1s
```

Fig. 2: Example system with three colored regions (methods) influenced by configuration options. Local performance-influence models for a specific workload are shown per region.

in a given workload and environment, and are typically *learned* by observing the behavior of the *system implementation*.

In the context of configurable systems, performance-influence models *predict* a system's performance in terms of configuration options and their interactions [15, 17, 18, 28, 29, 29, 30, 37, 63, 72, 73]. Some models are explicitly designed to *explain* how specific options and their interactions influence a system's performance [15, 37, 63, 64, 66, 73]. For example, the sparse linear model $8+15A+10C+3AB+30AC$ captures the execution time of the system in Fig. 2, which predicts the performance of arbitrary configurations, and explains *how* the options A, B, and C and their interactions influence the system's performance. The model can be inspected by developers, for example, to determine whether the large performance impact when combining A and C is an unexpected performance bug.

Building performance-influence models: Performance-influence models are typically built by measuring a system's execution time with the target workload in the target environment under different configurations [63]. Almost all existing approaches are *black-box* in nature: They do not analyze the system's implementation and measure the end-to-end execution time of the system.

The simplest approach is to observe the execution of all configurations in a *brute-force* approach. The approach obviously does not scale, as the number of configurations grows exponentially with the number of options.

In practice, most approaches measure executions only for a *sampled* subset of all configurations and extrapolate performance behavior for the rest of the configuration space using machine learning [13], which we collectively refer to as *sampling and learning* approaches. Specific approaches differ in how they sample, learn, and represent models: Common sampling techniques include uniform random, feature-wise, and pair-wise sampling [46], design of experiments [49], and combinatorial sampling [1, 19, 23, 24, 52]. Common learning techniques include linear regression [35, 63–65], regression

trees [13, 15, 16, 60], Fourier Learning [17], and neural networks [18]. Different sampling and learning techniques yield different tradeoffs between measurement effort, prediction accuracy, and interpretability of the learned models [13, 34, 37].

Goals of performance-influence modeling: Performance-influence models can be used for different tasks in different scenarios, which benefit from different model characteristics.

In the simplest case, a user wants to *optimize* a system's performance by selecting the fastest configuration for their workload and in the target environment. Performance-influence models have been used for optimization [15, 50, 56, 82], though metaheuristic search (e.g., hill climbing) is often more effective at pure optimization problems [26, 27, 56, 57], as they do not need to understand the entire configuration space.

In other scenarios, users want to predict the performance of individual configurations. Scenarios include *automatic reconfiguration* and *runtime adaptation*, where there is no human in the loop and online search is impractical. For example, when dynamically deciding during a robot's mission which options to change to react to low battery levels [29, 30, 76, 82]. In these scenarios, the model's prediction accuracy is important across the entire configuration space, but understanding the structure of the model is not important. In this context, deep regression trees [15, 16, 60], Fourier Learning [17], and neural networks [18] are commonly used, which build accurate models, but are not easy to interpret by humans [13, 34, 37, 48, 63].

When performance-influence models are used by users to make deliberate configuration decisions [13, 34, 37, 63, 76, 80] (e.g., whether to accept the performance overhead of encryption), interpretability regarding *how* options and interactions influence performance becomes paramount. In these settings, researchers usually suggest sparse linear models, such as $8 + 15A + 10C + 3AB + 30AC$ above, typically learned with stepwise linear regression or similar variations [35, 63–65]. In such models, users can directly recognize the influence of an option or an interaction.

In addition to end users who configure a system, developers who maintain the system can also benefit from performance-influence models to understand and debug the performance behavior of their systems [37, 39, 73]. For example, when presenting performance-influence models to developers in high-performance computing, Kolesnikov et al. [37] reported that a developer *"was surprised to see that [an option] had only a small influence on system performance"*, indicating a potential bug. In such setting, understanding *how* individual options and interactions influence performance is again paramount, favoring interpretable models. Ideally, models would also indicate *where* the influence occurs in the implementation. For example, in prior work [73], we shared experience of how mapping performance influence of options to specific code regions was useful in identifying several inconsistencies between documentation and implementation.

In this paper, we build performance models that can be used in all of these scenarios. The models are *accurate* for optimization and prediction, they are interpretable explaining *how* options and interactions influence performance, and they
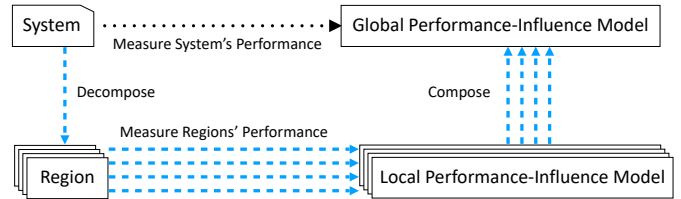

Fig. 3: Building performance-influence models is compositional: Instead of building a single model for the entire system (dotted black arrow), we can simultaneously build a local model per region and compose those models (dashed blue arrows).

map performance influence to code regions explaining *where* performance influence manifests. Our white-box approach, Comprex, builds these models by analyzing and observing system internals, without using machine learning.

III. COMPREX

We introduce **Comprex**, a *white-box* approach to efficiently and accurately generate performance-influence models, without using traditional sampling and learning techniques.

Similar to black-box approaches, we build performance-influence models by observing the execution of a system in different configurations, but we guide the exploration with a white-box analysis of the internals of the system. For a given input, a system with a set of Boolean options O can exhibit up to $2^{|O|}$ distinct execution paths, one per configuration.[1] If we measure the execution time of each distinct path, we can build a performance-influence model that accurately maps performance differences to options and their interactions, without any approximation through machine learning. The resulting models can be expressed as familiar linear models.

Our approach to efficiently build performance-influence models relies on two insights: (1) Performance-influence models can be built *compositionally*, by composing models built independently for smaller regions of the code (cf. Fig. 3). (2) Multiple performance-influence models for smaller regions can be built simultaneously by observing a system's executions often with only a few configurations, which we call *compression*.

First, building performance-influence models is compositional: We can measure the performance of smaller regions in the system (e.g., considering each method as a region) and build a performance-influence model per region separately, which describes each region's performance behavior in terms of options. Subsequently, we can compose the local models to describe the performance of the entire system, computed

---

[1]For simplicity, we describe Comprex in terms of Boolean options, but other option types can be encoded and discretized. The distinction between inputs and options is subjective and domain specific. We consider options as special inputs with a small finite domain (e.g., Boolean) that a user might explore to change functionality or influence quality attributes. We consider fixed values for other inputs. Note that a user might fix some configuration options and consider alternative values for inputs (e.g., use an option for different workloads). However, we explore the performance influence of options with finite domains, assuming all other inputs are fixed at specific values. This setting results in a finite, but typically very large configuration space.
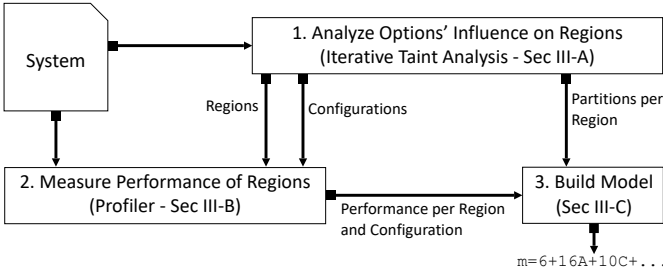
Fig. 4: Overview of **Comprex**'s three components.

as the sum of the individual influences in each model (e.g., composing $5 + 4A$ and $1 - 1A + 2B$ to $6 + 3A + 2B$).

Compositionality helps reduce the cost of our approach, as many smaller regions of a system are often influenced only by a subset of all options, confirmed by prior empirical research [47, 51]. Hence, the number of distinct paths to observe in a region is usually much smaller than the number of distinct paths in the entire system. If we have an analysis that can identify the subset of options that directly and indirectly influence the smaller region (see Sec. III-A), we can build a local performance-influence model by observing all distinct paths in a region often with only a few configurations.

Second, compression makes our approach scale without relying on machine learning approximations: When executing a single configuration, we can *simultaneously* observe the execution of multiple regions. In addition, if the regions are influenced by different options, a common case confirmed by prior research [47, 51], then we can separately measure, *in one configuration*, the influence of different options in different regions in the system, instead of exploring all combinations of all options. For example, three independent regions `if(a){} if(b){} if(c){}` influenced by options A, B, and C, respectively, each have two distinct paths. Instead of exploring all 8 combinations of the three options, we can explore all distinct paths in each region with only 2 configurations, as long as each option is enabled in one configuration and disabled in the other configuration.

Our approach combines compositionality and compression to build accurate performance-influence models, without traditional sampling or machine-learning techniques. The resulting models can be presented in an interpretable format and even be mapped to individual code regions. Key to our approach is the property that not all options interact in the same region, instead influencing different parts of the system independently; a pattern observed empirically in configurable systems [47, 51].

To operationalize compression and compositionality for building accurate and interpretable performance-influence models with low cost, we need three technical components, as illustrated in Fig. 4: First, we identify which regions are influenced by which options to select configurations to explore all paths per region and to map measured execution times to options and their interactions (Sec. III-A). Second, we execute the system to measure the performance of all regions (Sec. III-B). Third, we build local performance-

influence models per region and compose them into one global model for the entire system (Sec. III-C).

A. Analyzing Options' Influence on Regions

As a first step, we identify which options (directly or indirectly) influence control-flow decisions in which code regions. We use this information to select configurations to explore all paths per region and map measured performance differences to options and their interactions (Sec. III-B).[2] To this end, we track *information flow* from configuration options (sources) to *control-flow decisions* (sinks) in each region. If a configuration option flows, directly or indirectly (including implicit flows), into a control-flow decision in a region, this implies that selecting or deselecting the option may lead to different execution paths within the region. Thus, we should observe at least one execution with a configuration in which the option is selected and another execution in which the option is not selected.

More specifically, we conservatively partition the configuration space per region into subspaces, such that every configuration in each subspace takes the same path through the control-flow decisions within a region, and that all distinct paths are explored when taking one configuration from each subspace. Formally, a set of Boolean options O forms a configuration space $C = \mathcal{P}(O)$ where each configuration $c \in C$ is represented by the set of selected options. A *partition* of the configuration space ($p \subseteq \mathcal{P}(C)$) is a grouping of configurations into nonempty subsets, which we call *subspaces*, such that each configuration is part of exactly one subspace. For notational convenience, we describe subspaces using propositional formulas over options. For example, $[\![A \wedge \neg B]\!]$ describes the subspace of all configurations in which option A is selected and option B is deselected.

To track information flow between options and control-flow decisions in regions, we employ a *dynamic taint analysis* and *iteratively* collect information through repeated executions of the system in different configurations, until we have explored all distinct paths in each region. A taint analysis, typically used in the security domain [3, 6], tracks how values are affected by selected inputs (sources) and are used in specific locations (sinks). During each execution, we track how API calls load configuration options (Lines 2–3 in Fig. 2) (sources) and propagate them along data-flow and control-flow dependencies, including implicit flows, to the decisions of control-flow statements (sinks). We use a dynamic rather than a static analysis [44, 73], since it scales to larger systems and avoids some overtainting problems by tracking actual executions [6].

Incrementally partitioning the configuration space: Alg. 1 describes how we partition the configuration space per region, based on incremental updates from our dynamic taint analysis. Intuitively, we execute the system in a configuration

---

[2]We focus on different execution paths caused by configuration changes, fixing all other inputs. We focus on configuration changes in control-flow statements, as a system's execution time changes in those statements, depending on which branch is executed and how many times it is executed, confirmed by empirical research [21, 31, 53, 66, 73]. Execution differences caused by nondeterminism are orthogonal and must be handled in conventional ways (e.g., averaging multiple observations or controlling the environment).

Algorithm 1: Iterative Dynamic Taint Analysis

Input: configurable system p
Output: partition for each region $R \to \mathcal{P}(\mathcal{P}(C))$

1   **Function** partition_all_regions(p)
2     $partitions := R \to \{C\}$; $executed\_configs := \emptyset$
3   **until** explored_all_subspaces($executed\_configs$, $partitions$) **do**
4       $cc :=$ get_next_config($executed\_configs$, $partitions$)
5       $executed\_configs := executed\_configs \cup cc$
6   **during** execute_taint_analysis(p, cc) **when** reaching a control-flow decision with taints t_d and t_c in region r:
7         $partitions[r] := partitions[r] \times$ get_part(t_d, t_c, cc)
8   **end**
9   **end**
10   **return** $partitions$
11 **end**

Input: executed configurations $ec : \mathcal{P}(C)$, $partitions : R \to \mathcal{P}(\mathcal{P}(C))$
Output: true or false

12 **Function** explored_all_subspaces(ec, $partitions$)
13    $all\_subspaces := \bigcup$ image($partitions$)
14   **return** $\forall s \in all\_subspaces. \exists c \in ec. c \in s$
15 **end**

Input: data-flow taints t_d, control-flow taints t_c, current configuration cc
Output: partition $p : \mathcal{P}(\mathcal{P}(C))$ for the current decision

16 **Function** get_part(t_d, t_c, cc)
17    $s_{reach} := \{c \in C \mid \forall o \in t_c.\ o \in c \Leftrightarrow o \in cc\}$
18    $p := \{C \setminus s_{reach}\}$ // subspace of config. that might not reach decision
    // add one subspace for every combination of options in data-flow taints
19   **for** $a \in \mathcal{P}(t_d)$ **do**
20      $s := \{c \in C \mid \forall o \in t_d.\ o \in c \Leftrightarrow o \in a\}$
21      $p := p \cup \{s \cap s_{reach}\}$
22   **end**
23   **return** $p \setminus \emptyset$
24 **end**

---

and observe when data-flow and control-flow taints from options reach each control-flow decision in each region, and subsequently update each region's partition: Whenever we reach a control-flow statement during execution, we identify, based on taints that reach the condition of the statement, the sets of configurations that would possibly make different decisions, thus updating the partition that represents different paths for this region (Line 7). Since a dynamic taint analysis can only track information flow in the current execution, but not for alternative executions (i.e., for paths not taken), we repeat the process with new configurations, selected from the partitions identified in prior executions, updating partitions until we have explored one configuration from each subspace of each partition (main loop, Line 3); that is, until we have observed each distinct path in each region at least once. Note that some subspaces in the region might make the same control-flow decision as other subspaces, but we do not know which subspace will make which decision until we actually execute those configurations.

Updating partitions works as follows: When we reach a control-flow statement in a region with data-flow taints t_d, this indicates that the options in t_d affect the control-flow decision, *but other options do not*. Thus, we know that all configurations that share the same selection for all options in t_d will result in the same control-flow decision, while configurations with different selections of these options may result in different decisions. Since the taint analysis tells us only that the options in t_d may somehow (directly or indirectly) affect the decision's condition, but not how, we will need to explore at least one configuration for every possible assignment to these options, even though multiple or even all may end up taking

the same branch. Therefore, we partition the configuration space *at this decision* corresponding to all combinations of the options in t_d (Lines 19–20). For example, for a decision influenced by options A and B, we partition the configuration space into four subspaces: all configurations in which A and B are selected together, all configurations in which A is selected but not B, all configurations in which B is selected but not A, and all configurations in which neither A nor B are selected. Finally, we update the *region's* partition with the partition derived for the decision by computing their cross product ($\times$, Line 7). This operation reflects that, to explore all paths among multiple control-flow decisions in a region, including multiple executions of the same control-flow statement, we need to explore all combinations of the individual paths in the region.

Distinguishing data-flow taints from control-flow taints allows us to optimize the exploration of nested decisions (e.g., `if(a){ if(b) ... }`). Control-flow taints specify which options (directly or indirectly) influenced outer control-flow decisions, which indicates that different assignments to options in the control-flow taints *may* lead to paths where the current decision is not reached in the first place. Hence, we do not necessarily need to explore all interactions of options affecting outer and inner decisions. Instead of exploring combinations for all options of data-flow and control-flow taints, we first split the configuration space into (1) those configurations for which we know that they will reach the current decision, as they share the assignments of options in control-flow taints (s_{reach}, Line 17), and (2) the remaining configurations which may not reach the current decision ($C \setminus s_{reach}$, Line 18). Then, we only create subspaces for interactions of options in data-flow taints within s_{reach} (Lines 19–21) and consider the entire set of configurations outside s_{reach} as a single subspace (Line 18). The iterative nature of our analysis ensures that at least one of the configurations outside s_{reach} will be explored, and, if the configuration also reaches the same decision, the region's partition will be further divided.

The iterative analysis executes the system in different configurations until one configuration from each subspace of each partition in each region has been explored. That is, we start by executing any configuration (e.g., the default configuration), which reveals the subspaces per regions that could make different decisions. The algorithm then selects the next configuration to explore unseen subspaces in the regions (Line 4), which may further update the regions' partitions. To select the next configuration, we use a greedy algorithm to pick a configuration that explores the most unseen subspaces across all regions.[3]

Example: We exemplify running the iterative analysis in our running example from Fig. 2, considering each method as a region. If we execute the configuration $\{A, D\}$, in which options A and D are selected and the other options are

---

[3] To avoid enumerating an exponential number of configurations, we use a greedy algorithm that picks a random subspace and incrementally intersects it with other non-disjoint subspaces, which seems sufficiently effective in practice. The problem can also be encoded as a MAXSAT problem, representing subspaces as propositional formulas, to find the configuration that satisfies the formula with the most subspaces.

deselected, the taint analysis will indicate that the value of the variable `a` is tainted by option `A` (from Line 2) when reaching the `if` statement in Line 6 (region `main`). Thus, all configurations in which `A` is selected result in the same control-flow decision and all configurations in which `A` is not selected result potentially in the same or a different decision. Hence, we derive the initial partition $\{\llbracket A \rrbracket, \llbracket \neg A \rrbracket\}$ for this method.

Continuing the execution, we next reach the `if` statement in Line 16 (region `foo`), where the value of the variable `x` is tainted with control-flow taint `A` (from Line 6) and data-flow taint `B` (from variable `b`). Thus, all configurations in which `A` and `B` are selected result in the same control-flow decision, all configurations in which `A` is selected and `B` is not selected may result in a different control-flow decision, and all configurations in which `A` is not selected may not reach this decision. Hence, we derive the partition $\{\llbracket A \wedge \neg B \rrbracket, \llbracket \neg A \rrbracket, \llbracket A \wedge B \rrbracket\}$ for this region. Note how we explore this nested `if` statement with 3 instead of 4 subspaces by separately tracking data-flow and control-flow taints.

Further in the execution, the decision in the `while` statement (Line 13) depends on the tainted value of the variable `i` (implicit flow), in each loop iteration, resulting in $\{\llbracket A \rrbracket, \llbracket \neg A \rrbracket\}$, which is consistent with `main`'s existing partition. Hence, the cross product does not change the partition. Similarly, the decision in Line 19 (region `bar`) repeatedly depends on data-flow taint `C` and control-flow taint `A`, resulting in the partition $\{\llbracket A \wedge \neg C \rrbracket, \llbracket \neg A \rrbracket, \llbracket A \wedge C \rrbracket\}$.

After this first execution, we identified six distinct subspaces among the partitions of the three regions ($\llbracket A \rrbracket, \llbracket \neg A \rrbracket, \llbracket A \wedge B \rrbracket, \llbracket A \wedge \neg B \rrbracket, \llbracket A \wedge C \rrbracket$, and $\llbracket A \wedge \neg C \rrbracket$), of which $\llbracket A \rrbracket, \llbracket A \wedge \neg B \rrbracket$, and $\llbracket A \wedge \neg C \rrbracket$ were explored with the initial configuration. In the next iteration, we select a configuration, for example $\{A,B,C\}$, to explore unseen subspaces in the regions and update partitions. In this case, however, nothing changes. We continue executing new configurations to explore unseen subspaces, possibly updating the regions' partitions, until we have explored all subspaces in the regions. After executing only 4 out of 16 configurations, for example $\{A,D\}, \{A,B,C\}, \{\}$, and $\{C\}$, we have explored at least one configuration from each subspace of each partition in each region, and the iterative analysis terminates. The subspaces derived for the three regions's partitions are $\llbracket A \rrbracket$, $\llbracket \neg A \rrbracket, \llbracket A \wedge \neg B \rrbracket, \llbracket A \wedge B \rrbracket, \llbracket \neg A \wedge \neg C \rrbracket, \llbracket \neg A \wedge C \rrbracket, \llbracket A \wedge \neg C \rrbracket, \llbracket A \wedge C \rrbracket$.

Discussion: Note how the iterative analysis explores regions independently and does not explore paths for options that do not influence that region (e.g., we do not explore the interaction of `B` and `C` and never explore configurations specifically for `D`). Also, note how the taint analysis tracks both direct and indirect dependencies interprocedurally.

The iterative analysis is guaranteed to terminate, as it explores new configurations during each iteration. In the worst case, all configurations in the system (finite set) will be executed, but in practice often much fewer executions are needed.

Our algorithm will produce the same partitions independent of the order in which configurations are executed. All subspaces that are derived during any execution of the taint anal-

ysis will be derived at some point, because (1) we eventually explore all paths in each region and (2) we update the partition of each region with the commutative cross-product operation.

Theoretically, we can measure performance while running the taint analysis. In practice though, running a dynamic taint analysis with control-flow tracking imposes significant overhead, which significantly alters measurement results. A second reason for separating performance measurement from taint tracking is an optimization to perform taint tracking on a different (smaller) workload than the one used for performance measurement, which we discuss in Sec. IV.

B. Measuring Performance of Regions

We measure the execution time of each region when executing a configuration, resulting in performance measurements for each pair of configuration and region. To this end, we use an off-the-shelf profiler. We measure *self-time* per region to track the time spent in the region itself, which excludes the time of calls to execute code from other regions.

Since we measure performance in a separate step than the iterative analysis, we can pick a new and possibly smaller set of configurations to explore all paths per region. Ideally, we want to find a minimal set of configurations, such that we have at least one configuration per subspace of each region's partition. Since finding the optimal solution is NP-hard[4], and existing heuristics from combinatorial interaction testing [1, 23, 24, 40] are expensive, we developed our own simple greedy algorithm: Incrementally intersecting subspaces that overlap in at least one configuration, until no further such intersections are possible. Then, we simply pick one configuration from each subspace. However, we never found a smaller set of configurations than the one used in the iterative analysis. Thus, we use the same set of configurations to measure performance.

Example: Four configurations cover all subspaces for the three regions of our example, e.g., $\{\}, \{A\}, \{C\}, \{A,B,C\}$.

C. Building Performance-Influence Models

In the final step, we build performance-influence models for each region based on (1) the partitions identified per region and (2) the performance measured per region and observed configuration. Then, we compose the local models into a performance-influence model for the entire program.

Since we collect at least one measurement per distinct path through a region, building models is straightforward. For a region with a partition and a set of configurations with corresponding performance measurements, we associate each measurement with the subspace of the partition to which the configuration belongs. If multiple measured configurations belong to the same subspace of the region's partition, we expect the same performance behavior for that region (modulo measurement noise) and average the measured results. As a result, we can map each subspace of a region's partition to

---

[4]The problem can be reduced to the set cover problem, in which the union of a collection of subsets (all subspaces) equals a set of elements called "the universe" (the union of all subspaces). The goal is to identify the smallest sub-collection whose union equals the universe.

a performance measurement. For instance, for region `main` in our example, we report an execution time of 2 seconds for configurations in $[\![A]\!]$ and 3 seconds for configurations in $[\![\neg A]\!]$.

For interpretability, to highlight the influence of options and avoid sets or propositional formulas, we write linear models in terms of options and interactions, for example $m_{main} = 3 - 1A$.

The global performance-influence model is obtained simply by aggregating all local models; we add the individual influences of each model. Note that local models can be useful for understanding and debugging individual regions, as they describe the performance behavior of each region.

Example: In our running example, we derive the local models $m_{main} = 3 - 1A$, $m_{foo} = 1A + 3AB$, and $m_{bar} = 5 + 15A + 10C + 30AC$, which can be composed into the global performance-influence model $m = m_{main} + m_{foo} + m_{bar} = 8 + 15A + 10C + 3AB + 30AC$.

IV. Design, Optimizations, and Implementation

Dynamic Taint Analysis Overhead: Comprex executes a dynamic taint analysis with control-flow tracking, which imposes significant overhead in the system's execution. For instance, one execution of our subject system Berkeley DB takes about 1 hour with the taint analysis, whereas about 300 configurations can be executed in the same time! In general, we observe $26\times$ to $300\times$ overhead from taint tracking, which varies widely between systems. To reduce cost, we separate the iterative analysis and performance measurement in two steps and perform the former with a *drastically* reduced workload size.

This optimization is feasible when the workload is *repetitive* and repetitions of operations are affected similarly by options, which we conjecture to be common in practice. Many performance benchmarks execute many operations, which are similarly affected by configuration options. For instance, Berkeley DB's benchmark populates a database, where options determine, for example, whether duplicates are allowed and the durability characteristics of a transaction. The benchmark can be scaled by a parameter that controls the number of entries to insert, but does not affect which *operations* are performed. In our evaluation, we show that Comprex generates accurate models using a significantly smaller workload in the iterative analysis.

We conduct the actual performance measurements with the original workload to record realistic performance.

Granularity of Regions: Our approach can be applied to different levels of granularity, but with different tradeoffs. On one extreme, we could consider the entire system as a single region, but would not benefit from compression. At the other extreme, we could consider each control-flow statement as the start of its own region, ending with its immediate post-dominator, which allows maximum compression, but results in excessive measurement cost. Note the latter is analogous to using an *instrumentation profiler*, but instead of focusing on few locations of interest, as usually recommended [41], one would add instrumentation throughout the entire system at control-flow statements. In our evaluation, we show that considering each method as a region is a practical compromise.

When considering methods as regions, we may lose some compression potential compared to more fine-grained regions, if multiple control-flow statements within a method are influenced by distinct options. On the other hand, we can use off-the-shelf *sampling profilers* that accurately capture performance with low overhead, and simply map the performance of methods to the closest regions on the calling stack. Our empirical evaluation shows that method level regions do not require more configurations compared to more fine-grained regions, but significantly reduce the measurement overhead.

Implementation: We implemented Comprex for Java systems [74]. We used Phosphor, a state of the art tool for dynamic taint analysis [6] to track options. We annotated the APIs to load options as sources and control-flow statements as sinks. We encoded subspaces as propositional formulas and use SAT4J [61] for operations on those formulas. We used JProfiler 10.1 [33] to measure performance at method level.

V. Evaluation

To evaluate the efficiency and effectiveness of our approach, we compare Comprex to state of the art performance-influence modeling approaches for configurable systems, in terms of the cost to generate the models and their accuracy, and discuss their interpretability. We evaluate the efficiency of compression by choosing regions at different granularities. Specifically, we address the following research questions:

RQ1: How does Comprex compare to state of the art performance modeling approaches in terms of cost, accuracy, and interpretability?

RQ2: How efficient is compression at different granularities to reduce the cost to build models?

A. Experiment Setup

Subject Systems: We selected 4 configurable, widely-used, open-source Java systems that satisfy the following criteria (common in our domain): (a) medium- to large-scale systems with over 40K SLOC and over 20 options, (b) systems with binary and non-binary options, (c) systems with fairly stable execution time (despite nondeterminism and concurrency, we observed execution times within usual measurement noise for repeated execution of the same configuration), and (d) systems with different performance behaviors (see Fig. 5). Table I provides an overview of all subject systems.

We focus on a large subset of all options that are potentially relevant for performance. We considered options for which the systems' documentation indicated that they would affect performance, but excluded options that might not influence performance, (e.g., `--help`). This selection is representative of common use cases where users are interested in the performance behavior of many, but not all options. Following the evaluation of state of the art approaches [16, 19, 35, 44, 46, 60, 63–66], we selected, for non-binary options, two different values and encoded the values as a binary option.

We executed a long-running benchmark shipped with the system, representative of a user analyzing the system's per-
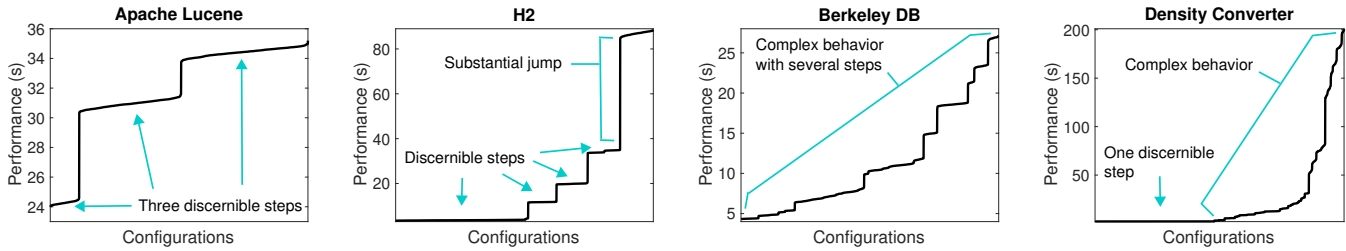
Fig. 5: Performance behaviors of our subject systems, from simple to complex. We randomly selected 2000 configurations and sorted their execution time from fastest to slowest.

TABLE I: Subject systems.

System	Domain	#KLOC	#Opt.	#Conf.	V/ID
Apache Lucene	Index/Search	396	17	131K	7.4.0
H2	Database	142	16	65K	1.4.201
Berkeley DB	Database	164	16	65K	7.5.11
Density Converter	Image processor	49[1]	22	4.9M	110c4

Opt: Options; Conf: Configurations; V/ID: Version/Commit ID; [1]: Interface to several libraries for processing images with 1.5K SLOC, included in the evaluation, as the system has a large configuration space.

TABLE II: Cost comparison.

Sample	Apache Lucene	H2	Berkeley DB	Density Conv.
BF	2^{17} [~48.4d]	2^{16} [~16.0d]	2^{16} [~8.6d]	2^{22} [~3.6y]
R50	50 [26.7m]	50 [16.4m]	50 [9.1m]	50 [10.1m]
R200	200 [1.8h]	200 [1.1h]	200 [36.4m]	200 [40.4m]
FW	17 [8.6m]	16 [2.4m]	16 [4.8m]	22 [8.2m]
PW	154 [1.3h]	137 [21.1m]	137 [39.4m]	254 [1.8h]
Comprex	26 [14.9m]	64 [22.6m]	144 [30.2m]	88 [16.6m]

The time to measure configurations for brute-force (BF) is extrapolated from 2000 randomly selected configurations.

(a) Cost of sampling configurations.

Approach	Apache Lucene	H2	Berkeley DB	Density Conv.
R50 & LR	8.9s	6.6s	5.7s	14.9s
R200 & LR	6.8m	4.6m	4.9m	1.6m
FW & LR	9.4s	4.3s	7.7s	19.8s
PW & LR	1.7m	44.8s	3.6m	5.5m
* & RF	≤0.2s	≤0.2s	≤ 0.3s	≤0.2s
Comprex	28.9m	9.3m	11.2m	8.5m

(b) Learning/Analysis time.

BF: Brute Force; R50: 50 random configurations; R200: 200 random configurations; FW: Feature-wise; PW: Pair-wise; LR: Stepwise linear regression; RF: Random forest; *: Results stable across all samples.

formance under different configurations. Detailed information about the benchmarks is included in our appendix [74].

For each system, we changed the workload parameter to run the iterative dynamic taint analysis with a smaller workload by factors ranging from 20 to 50000, depending on the system. The smaller workload reduced the iterative analysis time for a single configuration, for example, on average, from 1 hour to 5 seconds for Berkeley DB. Information about the changes in the workload can be found in our appendix [74].

Hardware: Analyses and measurements were executed on an Ubuntu 18.04 LTS desktop, with a 3.4 GHz 8-core Intel Core i7 processor, 16 GB of RAM, and Java HotSpot[TM] 64-bit Server VM (v1.8.0_202).

Performance Measurement: We established a large evaluation set for quantifying accuracy by measuring the performance of 2000 randomly selected configurations, as the configuration spaces of the subject systems are intractably large. We executed each configuration five times and used the median to reduce the effects of measurement noise. We initiated one VM invocation per configuration, thus all measures include startup time [12]. For Comprex, we profiled each system with JProfiler's default sampling rate of 5ms.

B. RQ1: Comparison to the State of the Art

With RQ1, we evaluate the cost to generate performance-influence models with Comprex, at method granularity, asses their accuracy and interpretability, and compare them to state of the art approaches. Specifically, we compare Comprex to numerous combinations of sampling and learning approaches. For learners, we evaluate variations of linear regressions [63–65], decision trees and random forests [13, 15, 16, 60], and a neural network. For sampling, we evaluate uniform random sampling with 50 and 200 configurations, 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) [46]. We selected 50 and 200 random configurations to use more configurations than other sampling strategies and use sampling sets comparable to ones used in related research. Due to space restrictions, we report results only for stepwise linear regression and random forest. The other results can be found in our appendix [74].

Note that we do not compare against approaches for selecting the fastest configuration [35, 56, 82], as those approaches solve a pure optimization problem where modeling the entire configuration space is not necessary.

We do not evaluate existing white-box approaches due to their limitations [66, 73]. None of the approaches could analyze any of our subject systems, except for a subset of Density Converter's code base, which we discuss in Sec. VI.

Cost Metric: We report the number of configurations executed to generate a model and time to measure configurations. For the learning approaches, we report the learning time. For Comprex, we report the time to execute the iterative analysis.

Accuracy Metric: We report the Mean Absolute Percentage Error (MAPE), which measures the mean difference between the values predicted by a model and the values actually observed (i.e., the baseline), lower is better.

As profiling the performance of regions with our approach

TABLE III: MAPE comparison (lower is better).

Approach	Apache Lucene	H2	Berkeley DB	Density Conv.
R50 & LR	**4.5**	124.1	19.7	1037.2
R200 & LR	**2.9**	93.9	14.9	434.5
FW & LR	**7.9**	129.3	768.7	1596.0
PW & LR	**4.7**	113.3	34.2	1596.0
R50 & RF	**0.4**	**1.1**	16.4	59.9
R200 & RF	**0.3**	**0.7**	**1.1**	**5.5**
FW & RF	**8.7**	119.0	106.1	1185.9
PW & RF	**4.0**	124.6	46.9	27.3
Comprex (O)	**8.0**	**9.3**	**6.7**	**11.2**
Comprex (C)	**3.2±0.2**	**2.9±0.5**	**5.0±0.6**	**9.4±2.4**[1]

LR: Stepwise linear regression; RF: Random forest; R50: 50 random configurations; R200: 200 random configurations; FW: Feature-wise; PW: Pair-wise; O: Original model; C: Model corrected for profiler overhead, reporting mean and standard deviations over 5 corrections. **Bolded** values in cells indicate similarly low errors.

[1] Non-linear profiling overhead might have increased the error [77].

adds some overhead (about 8% in our subject systems), we expect our models to be systematically biased. We performed an optional linear correction to account for the average overhead in our subject systems, learned from measurements with and without profiling of 5 randomly selected configurations. We separately report these "corrected" MAPE values in our results.

Interpretability: We intend our models to also be used in understanding and debugging tasks (see Sec. II). Unfortunately, measuring interpretability of models is nontrivial and controversial. In the machine learning community, interpretability is an open research problem with an active community, but without a generally agreed measure or even definition for interpretability [10, 48].

Generally, interpretability captures the ability of humans to make predictions, understand predictions, or understand the decisions of a model [48]. When the model is complex (e.g., the model includes numerous decisions), humans have more difficulty understanding the model directly. Some simpler forms of models are usually considered *inherently interpretable*, because humans can inspect and understand the models directly. For example, scientists have decades of experience using and interpreting linear models (e.g., much of empirical software engineering research relies on interpreting linear model coefficients). Models with more complex structures and very large numbers of decisions (e.g., deep neural networks with millions of weights or random forests with hundreds of trees), exceed human capacity for directly understanding the model. With these more complex models, the trend is to develop *post-hoc explanations*, where tools provide explanations for specific aspects of the model (e.g., the reason for a given prediction) without having to understand the model's internals [45, 48, 58, 68]. The use of post-hoc explanations is, however, controversial, as the explanations usually are only approximations that may be unreliable or even misleading [59].

We do not attempt to quantify interpretability. Instead, we generally consider sparse linear models as inherently interpretable (assuming a moderate numbers of terms). This is supported by prior interviews that have shown that developers understand linear performance-influence models with a few dozen terms [37]. By contrast, we consider random forests and neural networks as not inherently interpretable, and do not evaluate the reliability or usefulness of post-hoc explanations.

To assure readers that the linear models that we produce are indeed *sparse* and, hence, likely interpretable by humans, we report the number of terms they contain. As our algorithm to build these models (Sec. III-C) does not include any machine learning and regularization, the algorithm detects and reports even minuscule amounts of measurement noise. Hence, we exclude all trivial terms that do not make meaningful contributions to the systems' performance. We report the number of terms (options or interactions) that contribute, at least, 0.3 seconds, which is approximately 1% of the execution time of the default configurations of our subject systems. Our appendix contains additional data at other thresholds [74].

Results: We report the main results in Table II (cost) and III (accuracy), and our appendix [74] includes results for additional learners (neural networks, decision trees). Overall, Comprex builds models that are similarly accurate to those learned by the most accurate and expensive black-box approach (random forests with 200 samples), but our models are interpretable and usually built more efficiently, despite the cost of the additional taint analysis step. Comprex outperforms other approaches that build linear models by a wide margin.

While random forest with 200 samples produced slightly more accurate models than Comprex, our approach was usually more efficient, in some cases building models in half the time, while also generating local and interpretable models (see Fig. 1). The efficiency originates from Comprex's white-box analysis to identify a small number of relevant configurations to capture the performance-relevant interactions. By contrast, as our results show, black-box approaches perform significantly worse on such small samples (e.g., compare R50 and R200 results).

The linear models produced by Comprex are moderate in size, with 19, 16, 32, and 72 performance-relevant terms (options or interactions) for Lucene, H2, Berkeley DB, and Density Converter respectively. Our models are similar in size to models learned with linear regression from samples (e.g., 26, 13, 28, and 30 terms using R200 & LR), but much more accurate. At this size, we argue that manual inspection of the models is still plausible. More importantly, the performance influences can be mapped to 24, 5, 15, and 9 specific regions in the code for the four systems respectively.

> RQ1: In summary, models produced with Comprex have comparable accuracy to the most accurate and expensive black-box approaches, but can often be built more efficiently. Additionally, the models significantly outperform other black-box approaches that produce (interpretable) linear models. The models are interpretable and can be mapped to specific code regions.

C. RQ2: Compression Efficiency

With RQ2, we explore the impact of choosing regions at different granularities on the efficiency of Comprex, both

in terms of the *number of configurations* to measure and the *overhead* to perform these measurements.

Procedure: For RQ1, we executed the taint analysis considering each method as a region. We additionally tracked partitions for control-flow statements and derived partitions for the entire program by combining the partitions of all methods.

Result – Number of Configurations: In Table IV, we report the size of the minimum set of configurations needed to cover each subspace of each region's partition for each granularity. When considering the entire program as a region, significantly more configurations need to be explored, as we do not benefit from compression. Interestingly though, while there are, as expected, fewer regions at the method level than at the control-flow statement level, the number of configurations needed is the same. These results show that compression at finer-grained levels than the method level do not yield additional benefits in our subject systems.

We found that the control-flow statement regions combined within a method are usually partitioned in the same way. Only in 3 out of the 2263 method-level regions, the method's partition had more subspaces than the corresponding control-flow statement regions (e.g., two `if` statements depending on different options). However, in all three cases, the additional subspaces were already explored in other parts of the program. Hence, no additional configurations needed to be explored.

We conclude that fined-grained compression is highly effective, but that control-flow granularity does not seem to offer significant compression benefits over method granularity.

Results – Measurement Overhead: Measuring performance at different granularities requires different strategies, each with vastly different amounts of measurement overhead. Measuring at the program level is cheap, as a single end-to-end measurement is sufficient to measure the entire program (e.g., Unix `time`). At finer granularities, multiple measurements of different parts of the program are required. Instrumenting the program at control-flow statements and corresponding post-dominators leads to significant measurement overhead, as the measurement instructions are executed frequently (similar to an instrumentation profiler). For instance, in the time to measure methods (e.g., Berkeley DB executed 144 configurations in 30.2 minutes), not a single configuration finished measuring control-flow statements.

By contrast, measuring the performance of numerous methods is inexpensive with a *sampling profiler* (cf. Comprex in Table IIa), for which we observed a mostly linear overhead of about 8% in our subject systems.

RQ2: In summary, compression at method and control-flow granularities is highly efficient to reduce measurement effort. Compression at method granularity provides a good compromise between compression potential and measurement overhead.

TABLE IV: Number of regions and configurations to measure with compression at different region granularities.

System	Control-flow		Method		Program	
	#Reg.	#Conf.	#Reg.	#Conf.	#Reg.	#Conf.
Lucene	1654	**26**	551	**26**	1	16384
H2	2483	**64**	932	**64**	1	256
Berkeley DB	2152	**144**	718	**144**	1	2048
Density Converter	190	**88**	62	**88**	1	4608

#Reg: Number of regions; #Conf: Number of configurations. **Bolded** values in cells indicate the minimum number of configurations to cover all partitions' subspaces.

D. Limitations and Threats to Validity

Limitations of the Taint Analysis: Our approach to compute partitions per region may produce inaccurate results due to two sources of inaccuracy: (1) A standard dynamic taint analysis cannot reason about *paths not taken* [4] and, thus, may miss some taints, and (2) our use of a small workload to reduce the cost of the iterative analysis may lead to some missed interactions. The former threat is somewhat mitigated by exploring multiple configurations and using the cross product when updating partitions; we see both branches of each control-flow decision. The latter issue depends on how the workload is shortened (see Sec. IV), but will likely have a low impact in highly repetitive workloads. Both threats can lead to generating inaccurate models.

Importantly, our results for RQ1 suggest that inaccuracies on the regions' partition caused by these threats resulted in, at most, minor accuracy degradation in our performance-influence models, given the consistently high accuracy achieved across all subject systems.

Furthermore, we used a debugging strategy to identify potential effects of inaccurate partitions. As discussed in Sec. III-C, multiple executions of configurations within the same subspace of a region's partition must have the same performance behavior. Significant differences, beyond normal measurement noise, indicate that the region's partition might be inaccurate and not capturing all relevant interactions.

Specifically, we analyzed the performance measurements of all regions, searching for regions with a significant performance influence (> 0.1ms in any observed configuration) and with a high variance among the execution times of the same subspace in a region (coefficient of variation of the execution times > 1.0). Among the 2263 method-level regions that we analyzed, we found only 8 of such regions, all in Lucene.

We executed the iterative analysis with the regular workload, which resulted in additional subpaces in 7 out of 8 regions, due to slightly different taints in the shorter workload. While the missing subspaces slightly decreased the MAPE from 8.0 to 7.4 (as a result of slight changes in the coefficients in the model, not from new options nor interactions becoming performance relevant), the time to run the taint analysis with the regular workload is *extremely* expensive; 11 hours instead of 29 minutes to run the same 26 configurations. In fact, the iterative analysis did no finish executing after 24 hours in the other subject systems!

We argue that the extremely high cost and inability to use

the regular workload does not outweigh a potential slight accuracy increase of the already highly accurate models that we generated. These results support our conjecture that inaccuracies on the regions' partition caused by using an unsound analysis with a small workload are only a minor issue in practice.

Threats to Validity: Beyond the limitations of the taint analysis, measurement noise cannot be excluded and may affect *all* results. We reduced this threat by repeating measurements on a dedicated machine and using the median.

The primary threat to external validity is the selection of subject systems. While we selected medium- to large-scale widely-used open-source Java configurable systems from different domains, readers should be careful when generalizing results. For instance, all analyzed systems are multi-threaded, but had mostly deterministic performance behavior. Additionally, we analyzed a single configurable system, whereas systems composed of numerous configurable systems, deployed in distributed environments, and implemented in different languages are beyond the scope of this paper.

Another threat is the selected subset of options, which might not affect performance at all, making modeling a trivial task. We selected options for which the systems' documentation or the options' functionality indicated that they would affect performance, and we observed a wide range of execution times for the configurations that we measured (see Fig. 5).

VI. RELATED WORK

In Sec. II, we discussed performance modeling in general and evaluated closely related state of the art black-box approaches for performance-influence modeling to Comprex. In this section, we discuss *additional* research to position Comprex in a broader context of prior work.

Only few researchers have explored white-box performance modeling of configurable systems in prior work [43, 66, 73]. Siegmund et al. [66] introduced the idea of white-box analysis, but assumed a specific programming style that provided a static mapping from options to regions and ignored data-flow between regions, severely limiting the systems that could be analyzed. In prior work, we developed ConfigCrusher [73], which used static data-flow analysis to map options to regions. ConfigCrusher also measures the performance of regions and builds local performance-influence models. ConfigCrusher is, however, limited by the overhead of the static analysis and was unable to scale beyond small systems (e.g., it never terminated when we tried it on our subject systems). By contrast, Comprex uses a dynamic data-flow analysis that scales to large systems, demonstrating the feasibility of white-box approaches to efficiently build accurate performance-influence models for large-scale configurable systems. For comparison, we evaluated both approaches on Density Converter, though Comprex analyzed all used libraries, resulting in comparable accuracies (4.3 with ConfigCrusher vs. 9.4 with Comprex in terms of MAPE), but measuring fewer configurations (256 with ConfigCrusher vs. 88 with Comprex).

In prior work [73], we also adapted *SPLat* [36] for performance modeling, originally designed for *testing* configurable systems. SPLat uses lightweight instrumentation to observe which options are accessed during execution, and in which order, to explore all combinations of options that it encounters during execution. This type of analysis is cheaper than our taint tracking, but explores many spurious interactions, resulting in SPLat essentially exploring all configurations.

Li et a. [43] used intraprocedural control-flow analysis to identify usage patterns of individual options in the source code for predicting performance properties based on the patterns. In contrast to our work, they do not measure performance, build performance-influence models, nor consider interactions.

While our line of work focuses on the influence of configuration options, which are easy to change by users without modifying the implementation, there is a active research field to identify bottlenecks and performance bugs, generally using different forms of static or dynamic analysis [e.g., 7, 8, 14, 20, 32, 42, 55, 81]. For example, Castro et al. [7] use both techniques to identify performance bottlenecks that can be analyzed and optimized in isolation. These kinds of approaches are focusing on different concerns than performance-influence models but are likely complementary for developers seeking to understand a system's performance behavior.

At the same time, researchers have used static and dynamic analyses to characterize and track options in configurable systems [9, 44, 47, 51, 67, 69–71, 79]. For example, Toman and Grossman [71] use dynamic taint analysis to identify the use of stale configuration data. Our work is inspired by insights from analyzing configurable systems and uses similar analysis strategies as foundations to map options to regions.

VII. CONCLUSION

We presented Comprex, a white-box approach that efficiently builds accurate and interpretable performance-influence models for configurable systems, without relying on traditional sampling and learning techniques. Comprex employs an iterative dynamic taint analysis to identify where options influence the system, building and composing local linear performance-influence models. Our empirical evaluation on 4 systems demonstrates the accuracy and efficiency of Comprex.

VIII. ACKNOWLEDGEMENTS

REFERENCES

[1] M. Al-Hajjaji, S. Krieter, T. Thüm, M. Lochau, and G. Saake, "Incling: Efficient product-line testing using incremental pairwise sampling," in *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. ACM, Oct. 2016, pp. 144–155.

[2] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer-Verlag, 2013.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "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, Jun. 2014, pp. 259–269.

[4] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *Proc. Workshop Programming Languages and Analysis for Security (PLAS)*. ACM, Jun. 2009, pp. 113–124.

[5] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *J. Syst. Softw.*, vol. 82, no. 1, pp. 3–22, Jan. 2009.

[6] J. Bell and G. Kaiser, "Phosphor: Illuminating dynamic data flow in commodity JVMs," *SIGPLAN Notices*, vol. 49, no. 10, pp. 83–101, Oct. 2014.

[7] P. D. O. Castro, C. Akel, E. Petit, M. Popov, and W. Jalby, "Cere: Llvm-based codelet extractor and replayer for piecewise benchmarking and optimization," *ACM Trans. Archit. Code Optim. (TACO)*, vol. 12, no. 1, pp. 6:1–6:24, Apr. 2015.

[8] J. Cito, P. Leitner, C. Bosshard, M. Knecht, G. Mazlami, and H. C. Gall, "PerformanceHat: Augmenting source code with runtime performance traces in the IDE," in *Proc. Int'l Conf. Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 41–44.

[9] Z. Dong, A. Andrzejak, D. Lo, and D. Costa, "ORPLocator: Identifying read points of configuration options via static analysis," in *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*. IEEE, Oct. 2016, pp. 185–195.

[10] F. Doshi-Velez and B. Kim, "Towards a rigorous science of interpretable machine learning," *arXiv preprint arXiv:1702.08608*, 2017.

[11] N. Esfahani, A. Elkhodary, and S. Malek, "A learning-based framework for engineering feature-oriented self-adaptive software systems," *IEEE Transactions on Software Engineering*, vol. 39, no. 11, pp. 1467–1493, Nov. 2013.

[12] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous Java performance evaluation," *SIGPLAN Notices*, vol. 42, no. 10, pp. 57–76, Oct. 2007.

[13] A. Grebhahn, N. Siegmund, and S. Apel, "Predicting performance of software configurations: There is no silver bullet," 2019.

[14] M. Grechanik, C. Fu, and Q. Xie, "Automatically finding performance problems with feedback-directed learning software testing," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, Jun. 2012, p. 156–166.

[15] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wąsowski, "Variability-aware performance prediction: A statistical learning approach," in *Proc. Int'l Conf. Automated Software Engineering (ASE)*. ACM, Nov. 2013, pp. 301–311.

[16] J. Guo, D. Yang, N. Siegmund, S. Apel, A. Sarkar, P. Valov, K. Czarnecki, A. Wasowski, and H. Yu, "Data-efficient performance learning for configurable systems," *Empirical Software Engineering*, vol. 23, pp. 1826–1867, 2017.

[17] H. Ha and H. Zhang, "Performance-influence model for highly configurable software with fourier learning and lasso regression," in *Proc. Int'l Conf. Software Maintance and Evolution (ICSME)*, Sep. 2019, pp. 470–480.

[18] H. Ha and H. Zhang, "DeepPerf: Performance prediction for configurable software with deep sparse neural network," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, May 2019, p. 1095–1106.

[19] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and B. Baudry, "Test them all, is it worth it? assessing configuration sampling on the JHipster web development stack," *Empirical Software Engineering*, Jul. 2018.

[20] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, Jun. 2012, pp. 145–155.

[21] X. Han and T. Yu, "An empirical study on performance bugs for highly configurable software systems," in *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*. ACM, Sep. 2016, pp. 23:1–23:10.

[22] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*, 1st ed. Cambridge University Press, 2013.

[23] A. Hervieu, B. Baudry, and A. Gotlieb, "PACOGEN: Automatic generation of pairwise test configurations from feature models," in *Proc. Int'l Symposium Software Reliability Engineering*, Nov. 2011, pp. 120–129.

[24] A. Hervieu, D. Marijan, A. Gotlieb, and B. Baudry, "Optimal Minimisation of Pairwise-covering Test Configurations Using Constraint Programming," *Information and Software Technology*, vol. 71, pp. 129 – 146, Mar. 2016.

[25] A. Hubaux, Y. Xiong, and K. Czarnecki, "A user survey of configuration challenges in Linux and eCos," in *Proc. Workshop Variability Modeling of Software-Intensive Systems (VAMOS)*. ACM, Jan. 2012, pp. 149–155.

[26] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Proc. Int'l Conf. Learning and Intelligent Optimization*. Springer-Verlag, Jan. 2011, pp. 507–523.

[27] P. Jamshidi and G. Casale, "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)*, Sep. 2016, pp. 39–48.

[28] P. Jamshidi, N. Siegmund, M. Velez, C. Kästner, A. Patel, and Y. Agarwal, "Transfer learning for performance modeling of configurable systems: An exploratory analysis," in *Proc. Int'l Conf. Automated Software Engineering (ASE)*. ACM, Oct. 2017.

[29] P. Jamshidi, M. Velez, C. Kästner, and N. Siegmund, "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, Nov. 2018, pp. 71–82.

[30] P. Jamshidi, M. Velez, C. Kästner, N. Siegmund, and P. Kawthekar, "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, May 2017, pp. 31–41.

[31] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Proc. Conf. Programming Language Design and Implementation (PLDI)*. ACM, Jun. 2012, pp. 77–88.

[32] M. Jovic, A. Adamoli, and M. Hauswirth, "Catch me if you can: Performance bug detection in the wild," in *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, Oct. 2011, p. 155–170.

[33] JProfiler. (2019) Jprofiler 10. [Online]. Available: https://www.ej-technologies.com/products/jprofiler/overview.html

[34] C. Kaltenecker, A. Grebhahn, N. Siegmund, and S. Apel, "The interplay of sampling and machine learning for software performance prediction," *IEEE Software*, 2020.

[35] C. Kaltenecker, A. Grebhahn, N. Siegmund, J. Guo, and S. Apel, "Distance-based sampling of software configuration spaces," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, May 2019.

[36] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. d'Amorim, "SPLat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems," in *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)*. ACM, Aug. 2013, pp. 257–267.

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

[38] S. Kounev, "Performance modeling and evaluation of distributed component-based systems using queueing Petri Nets," *IEEE Transactions on Software Engineering*, vol. 32, no. 7, pp. 486–502, 2006.

[39] R. Krishna, M. S. Iqbal, M. A. Javidian, B. Ray, and P. Jamshidi, "CADET: A systematic method for debugging misconfigurations using counterfactual reasoning," 2020.

[40] D. R. Kuhn, R. N. Kacker, and Y. Lei, *Introduction to Combinatorial Testing*, 1st ed. Chapman & Hall/CRC, 2013.

[41] F. Lange. (2011, Jan.) Measure Java performance – sampling or instrumentation?

[42] C. Lemieux, R. Padhye, K. Sen, and D. Song, "Perffuzz: Automatically generating pathological inputs," in *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)*. ACM, 2018, pp. 254–265.

[43] C. Li, S. Wang, H. Hoffmann, and S. Lu, "Statically inferring performance properties of software configurations," in *Proc. European Conf. Computer Systems (EuroSys)*. ACM, Apr. 2020.

[44] M. Lillack, C. Kästner, and E. Bodden, "Tracking load-time configuration options," *IEEE Transactions on Software Engineering*, vol. 44,

no. 12, pp. 1269–1291, 12 2018.

[45] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., 2017, pp. 4765–4774.

[46] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, May 2016, pp. 643–654.

[47] J. Meinicke, C.-P. Wong, C. Kästner, T. Thüm, and G. Saake, "On essential configuration complexity: Measuring interactions in highly-configurable systems," in *Proc. Int'l Conf. Automated Software Engineering (ASE)*. ACM, Sep. 2016, pp. 483–494.

[48] C. Molnar, *Interpretable Machine Learning*, 2019, https://christophm.github.io/interpretable-ml-book/.

[49] D. C. Montgomery, *Design and Analysis of Experiments*. John Wiley & Sons, 2006.

[50] V. Nair, T. Menzies, N. Siegmund, and S. Apel, "Using bad learners to find good configurations," in *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)*, ser. ESEC/FSE 2017. ACM, Aug. 2017, p. 257–267.

[51] T. Nguyen, T. Koc, J. Cheng, J. S. Foster, and A. A. Porter, "iGen dynamic interaction inference for configurable systems," in *Proc. Int'l Symp. Foundations of Software Engineering (FSE)*. IEEE, Nov. 2016.

[52] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv. (CSUR)*, vol. 43, no. 2, pp. 11:1–11:29, Feb. 2011.

[53] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, "Caramel: Detecting and fixing performance problems that have non-intrusive fixes," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, May 2015, pp. 902–912.

[54] A. Nistor, T. Jiang, and L. Tan, "Discovering, reporting, and fixing performance bugs," in *Proc. Int'l Conf. Mining Software Repositories*. IEEE, May 2013, pp. 237–246.

[55] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: Detecting performance problems via similar memory-access patterns," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 5 2013, pp. 562–571.

[56] J. Oh, D. Batory, M. Myers, and N. Siegmund, "Finding near-optimal configurations in product lines by random sampling," in *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)*. ACM, Sep. 2017, pp. 61–71.

[57] R. Olaechea, D. Rayside, J. Guo, and K. Czarnecki, "Comparison of exact and approximate multi-objective optimization for software product lines," in *Proc. Int'l Software Product Line Conference (SPLC)*. ACM, Sep. 2014, pp. 92–101.

[58] M. T. Ribeiro, S. Singh, and C. Guestrin, ""Why should I trust you?": Explaining the predictions of any classifier," in *Proc. Int'l Conf. Knowledge Discovery and Data Mining (KDD)*. ACM, Aug. 2016, pp. 1135–1144.

[59] C. Rudin, "Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead," *Nature Machine Intelligence*, no. 5, pp. 206–215, 2019.

[60] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-efficient sampling for performance prediction of configurable systems," in *Proc. Int'l Conf. Automated Software Engineering (ASE)*. IEEE, Nov. 2015, pp. 342–352.

[61] SAT4J. (2020) Sat4j. [Online]. Available: www.sat4j.org

[62] G. Serazzri, G. Casale, M. Bertoli, G. Serazzri, G. Casale, and M. Bertoli, "Java modelling tools: An open source suite for queueing network modelling andworkload analysis," in *Proc. Int'l Conf. the Quantitative Evaluation of Systems - (QEST'06)*, Sep. 2006, pp. 119–120.

[63] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proc. Europ.*

[66] N. Siegmund, A. von Rhein, and S. Apel, "Family-based performance measurement," in *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. ACM, Oct. 2013, pp. 95–104.

Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE). ACM, Aug. 2015, pp. 284–294.

[64] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, Jun. 2012, pp. 167–177.

[65] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake, "SPLConqueror: Toward optimization of non-functional properties in software product lines," *Software Quality Journal*, vol. 20, no. 3-4, pp. 487–517, Sep. 2012.

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

[68] E. Štrumbelj and I. Kononenko, "Explaining prediction models and individual predictions with feature contributions," *Knowledge and information systems*, vol. 41, no. 3, pp. 647–665, 2014.

[69] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Comput. Surv. (CSUR)*, vol. 47, no. 1, pp. 6:1–6:45, Jun. 2014.

[70] J. Toman and D. Grossman, "Legato: An at-most-once analysis with applications to dynamic configuration updates," in *European Conf. Object-Oriented Programming (ECOOP)*, 7 2016.

[71] ——, "Staccato: A bug finder for dynamic configuration updates," in *Proc. European Conf. Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Jul. 2016.

[72] P. Valov, J.-C. Petkovich, J. Guo, S. Fischmeister, and K. Czarnecki, "Transferring performance prediction models across different hardware platforms," in *Proc. Int'l Conf. on Performance Engineering (ICPE)*. ACM, Apr. 2017, pp. 39–50.

[73] M. Velez, P. Jamshidi, F. Sattler, N. Siegmund, S. Apel, and C. Kästner, "Configcrusher: Towards white-box performance analysis for configurable systems," *Autom Softw Eng*, 2020.

[74] M. Velez, P. Jamshidi, N. Siegmund, S. Apel, and C. Kästner, "White-box analysis over machine learning: Modeling performance of configurable systems - Supplementary Material - https://bit.ly/3bbbgG8," 2021.

[75] N. Viswanadham and Y. Narahari, *Performance modeling of automated systems*. PHI Learning Pvt. Ltd., 2015.

[76] S. Wang, C. Li, H. Hoffmann, S. Lu, W. Sentosa, and A. I. Kistijantoro, "Understanding and auto-adjusting performance-sensitive configurations," in *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Mar. 2018, pp. 154–168.

[77] M. Weber, S. Apel, and N. Siegmund, "White-box performance-influence models," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, May 2021.

[78] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "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, Aug. 2015, pp. 307–319.

[79] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early detection of configuration errors to reduce failure damage," in *Proc. Conf.Operating Systems Design and Implementation (OSDI)*. USENIX Association, Nov. 2016, pp. 619–634.

[80] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proc. Symp. Operating Systems Principles*. ACM, Nov. 2013, pp. 244–259.

[81] T. Yu and M. Pradel, "Pinpointing and repairing performance bottlenecks in concurrent programs," *Empirical Softw. Eng.*, vol. 23, no. 5, pp. 3034–3071, Oct. 2018.

[82] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "Bestconfig: Tapping the performance potential of systems via automatic configuration tuning," in *Proc. Symposium Cloud Computing (SoCC)*. ACM, 9 2017, pp. 338–350.