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 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 [22, 23, 33, 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 [31, 79]; users often have, at most, vague intuitions [27, 66, 78]. We aim at efficiently building performance-influence models, without machine learning, that characterize how options influence the system’s performance. We intend 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 [38, 56]. 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 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 [15, 17, 35, 36, 49, 61, 63, 64, 66] result in tradeoffs among the cost to build the models and the accuracy and interpretability of the models [15, 36, 38]. 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 \[15, 36, 49\] (see Fig. 1).

In this paper, we introduce \textsc{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 \textsc{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, \textsc{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., 15, 36, 38, 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 Queueing networks, Petri Nets, and Stochastic Process Algebras) and workload in the design stage of a project \[24, 39, 62\]. At this stage, design decisions are usually modeled as configuration options \[7\] [13]. By contrast, performance-influence models describe the performance behavior of a system implementation 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 \[17, 19, 20, 30–32, 38, 66, 72, 73\]. Some models are explicitly designed to explain how specific options and their interactions influence a system’s performance \[17, 38, 63, 65, 66, 73\]. 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 \[66\]. 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 \[15\], 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 [47], design of experiments [50], and combinatorial sampling [8, 21, 25, 26, 53]. Common learning techniques include linear regression [35, 63, 64, 66], regression trees [15, 17, 18, 61], Fourier Learning [20], and neural networks [19]. Different sampling and learning techniques yield different tradeoffs between measurement effort, prediction accuracy, and interpretability of the learned models [15, 36, 38].

**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 [17, 51, 57, 82], though metaheuristic search (e.g., hill climbing) is often more effective at pure optimization problems [28, 29, 57, 58], 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 [31, 32, 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 [17, 18, 61], Fourier Learning [20], and neural networks [19] are commonly used, which build accurate models, but are not easy to interpret by humans [15, 50, 58, 49, 66].

When performance-influence models are used by users to make deliberate configuration decisions [15, 36, 38, 65, 76, 78] (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, 64, 66]. 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 [38, 40, 73]. For example, when presenting performance-influence models to developers in high-performance computing, Kolessnikov et al. [38] 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 map performance influence to code regions explaining **where** performance influence manifests. Our white-box approach, **Compref**, builds these models by analyzing and observing system internals, without using machine learning.

### III. Compref

We introduce **Compref**, 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.

---

\(^1\)For simplicity, we describe **Compref** 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.
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 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 [48, 52]. 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 [48, 52], 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 $\text{if}(a)\{\text{if}((b))\text{if}(c)\}$ 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 [48, 52].

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). 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 ($\rho \subseteq \mathcal{P}(C)$) 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, $[A \land \neg B]$ 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 [5, 8], 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 [45, 73], since it scales to larger systems and avoids some overpainting problems by tracking actual executions [8].

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 [33, 59, 63, 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 \rightarrow \mathcal{P}(\mathcal{P}(C))

Function partition_all_regions(p)

\begin{align*}
\text{partitions} & := R \rightarrow \{C\}; \text{executed_configs} := \emptyset \\
\text{until} \text{explored_all_subspaces} \left(\text{executed_configs}, \text{partitions}\right) \text{do} \\
& \quad \text{cc} := \text{get_next_config} \left(\text{executed_configs}, \text{partitions}\right) \\
& \quad \text{executed_configs} := \text{executed_configs} \cup \text{cc} \\
& \quad \text{during execute_taint_analysis} \left(p, cc\right) \text{ when reaching a} \\
& \quad \text{control-flow decision with taints} t_d \text{ and } t_c \text{ in region } r: \\
& \quad \quad \text{partitions}[r] := \text{partitions}[r] \times \text{get_part} \left(t_d, t_c, cc\right) \\
& \quad \text{end} \\
& \text{end} \\
& \text{return partitions}
\end{align*}

Function explored_all_subspaces(cc, partitions)

\begin{align*}
\text{Input: executed configurations } cc : \mathcal{P}(C), \text{partitions} : R \rightarrow \mathcal{P}(\mathcal{P}(C)) \\
\text{Output: true or false} \\
\text{all_subspaces} := \bigcup \text{image} \left(\text{partitions}\right) \\
\text{return } s \in \text{all_subspaces}, 3e \in cc, e \in s \\
\text{end}
\end{align*}

Function get_part(t_d, t_c, cc)

\begin{align*}
\text{Input: data-flow taints } t_d, \text{control-flow taints } t_c, \text{current configuration } cc \\
\text{Output: partition } p : \mathcal{P}(\mathcal{P}(C)) \text{ for the current decision} \\
\text{if(a)} \{ \text{if(b)} \ldots \} \text{). 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 19–20) 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 3), 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 unexplored subspaces across all regions.\footnote{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.}

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 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

\begin{align*}
\text{Algorithm 1: Iterative Dynamic Taint Analysis}
\end{align*}
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 deselected, the taint analysis will indicate that the value of the variable \( x \) 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 \( \{[A], [-A]\} \) 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 \( B \) 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 \( \{[A \land \neg B], [-A], [A \land B]\} \) 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 \( \{[A], [-A]\} \), 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 \( \{[A \land \neg C], [-A], [A \land C]\} \).

After this first execution, we identified six distinct subspaces among the partitions of the three regions \( \{[A], [-A], [A \land B], [A \land \neg B], [A \land C], [A \land \neg C]\} \) of which \( [A], [A \land \neg B], [A \land \neg C] \) 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' partitions are \( \{[A], [-A], [A \land B], [A \land \neg B], [A \land C], [A \land \neg C]\} \).

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 analysis 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 and existing heuristics from combinatorial interaction testing 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 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., \( \{\}, \{A\}, \{C\}, \{A,B,C\} \).
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 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 `[-A]`.

For interpretability, to highlight the influence of options and avoid sets or propositional formulas, we write linear models in configurations in `2`. In our example, we report an execution time of `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, Optimization, 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× to 300× 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 postdominator, 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 [42], 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 [8] 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 SAT [1] for operations on those formulas. We used JProfiler 10.1 [2] 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 [1] 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 [18, 21, 35]
Comprex to numerous combinations of sampling and learning approaches. For learners, we evaluate variations of linear regressions [63, 64, 66], decision trees and random forests [15, 17, 18, 61], 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) [47]. 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, 57, 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 [65, 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.
TABLE III: MAPE comparison (lower is better).

<table>
<thead>
<tr>
<th>Approach</th>
<th>Apache Lucene</th>
<th>H2</th>
<th>Berkeley DB</th>
<th>Density Conv.</th>
</tr>
</thead>
<tbody>
<tr>
<td>R50 &amp; LR</td>
<td>4.5</td>
<td>19.7</td>
<td>1037.2</td>
<td></td>
</tr>
<tr>
<td>R200 &amp; LR</td>
<td>2.9</td>
<td>14.9</td>
<td>434.5</td>
<td></td>
</tr>
<tr>
<td>FW &amp; LR</td>
<td>7.9</td>
<td>768.7</td>
<td>1596.0</td>
<td></td>
</tr>
<tr>
<td>PW &amp; LR</td>
<td>4.7</td>
<td>34.2</td>
<td>1596.0</td>
<td></td>
</tr>
<tr>
<td>R50 &amp; RF</td>
<td>0.4</td>
<td>16.4</td>
<td>59.9</td>
<td></td>
</tr>
<tr>
<td>R200 &amp; RF</td>
<td>0.3</td>
<td>0.7</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>FW &amp; RF</td>
<td>8.7</td>
<td>106.1</td>
<td>1185.9</td>
<td></td>
</tr>
<tr>
<td>PW &amp; RF</td>
<td>4.0</td>
<td>46.9</td>
<td>27.3</td>
<td></td>
</tr>
<tr>
<td>Comprex (O)</td>
<td>8.0</td>
<td>9.3</td>
<td>6.7</td>
<td>11.2</td>
</tr>
<tr>
<td>Comprex (C)</td>
<td>3.2±0.2</td>
<td>2.9±0.5</td>
<td>5.0±0.6</td>
<td>9.4±2.4</td>
</tr>
</tbody>
</table>

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 [74].

**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 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. [11]). 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 [12] [49].

Generally, interpretability captures the ability of humans to make predictions, understand predictions, or understand the decisions of a model [49]. 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 [46] [49] [59] [68]. The use of post-hoc explanations is, however, controversial, as the explanations usually are only approximations that may be unreliable or even misleading [60].

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 [38]. 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 II), for which we observed a mostly linear overhead of about 8% in our subject systems.

<table>
<thead>
<tr>
<th>System</th>
<th>Control-flow</th>
<th>Method</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Reg.</td>
<td>#Conf.</td>
<td>#Reg.</td>
</tr>
<tr>
<td>Lucene</td>
<td>1654</td>
<td>26</td>
<td>551</td>
</tr>
<tr>
<td>H2</td>
<td>2483</td>
<td>64</td>
<td>932</td>
</tr>
<tr>
<td>Berkeley DB</td>
<td>2152</td>
<td>144</td>
<td>718</td>
</tr>
<tr>
<td>Density Converter</td>
<td>190</td>
<td>88</td>
<td>62</td>
</tr>
</tbody>
</table>

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

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.

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 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 subspaces 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 [44, 65, 73]. Siegmund et al. [65] 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 [37] 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 al. [44] used intra-procedural 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., 9, 10, 16, 22, 34, 43, 55, 81]. For example, Castro et al. [9] 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 [11, 45, 48, 52, 67, 69, 71, 80]. For example, Toman and Grossman [70] 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**

We specially want to thank Katherine Hough and Jonathan Bell for their indispensable help with Phosphor. We thank Chu-Pan Wong, Jens Meinicke, and Florian Sattler for their comments during the development of this work, the FOSD 2019 meeting participants for their feedback on the iterative dynamic taint analysis, and Claire Le Goues and Rohan Padhye for their feedback on earlier drafts of this paper. This work has been supported in part by the Software
Engineering Institute, NSF (Awards 1552944, 1717022, and 2007202), NASA (RASPBERRY-SI 80NSSC20K1720), the German Federal Ministry of Education and Research (BMBF, 01IS19059A and 01IS18026B), and the German Research Foundation (DFG, SI 2171/2, SI 2171/3-1, AP 206/11).

REFERENCES

[1] Sarj. URL: www.sarj.org


