Predicting Performance via Automated Feature-Interaction Detection

Norbert Siegmund,* Sergiy S. Kolesnikov,† Christian Kästner,‡ Sven Apel,†
Don Batory,§ Marko Rosenmüller,* and Gunter Saake*
* *University of Magdeburg, Germany*
† *University of Passau, Germany*
‡ *Philipps University Marburg, Germany*
§ *University of Texas at Austin, USA*

***Abstract*—Customizable programs and program families provide user-selectable features to allow users to tailor a program to an application scenario. Knowing in advance which feature selection yields the best performance is difficult because a direct measurement of all possible feature combinations is infeasible. Our work aims at predicting program performance based on selected features. However, when features interact, accurate predictions are challenging. An interaction occurs when a particular feature combination has an unexpected influence on performance. We present a method that automatically detects performance-relevant feature interactions to improve prediction accuracy. To this end, we propose three heuristics to reduce the number of measurements required to detect interactions. Our evaluation consists of six real-world case studies from varying domains (e.g., databases, encoding libraries, and web servers) using different configuration techniques (e.g., configuration files and preprocessor flags). Results show an average prediction accuracy of 95 %.**

I. Introduction

There are many ways to customize a program. Commonly, a program uses command-line parameters, configuration files, etc. [1]. Another way is to derive tailor-made programs at compile-time using *product-line* technology. In product-line engineering, stakeholders derive tailored programs by means of a program generator to satisfy their requirements [2]. The generation process is based on *features*, where a feature is a stakeholder-visible behavior or characteristic of a program [2]. By mapping features to implementation units, a generator produces a program based on a user's feature selection. In this paper, we use product-line terminology and call any customization option that stakeholders can select at compile-time or load-time a *feature* of a program.

Stakeholders are also interested in non-functional properties of a program. For example, a database management system is usually customized to achieve maximum performance when used on a server, but is customized differently for low energy consumption when deployed on a battery-supplied system (e.g., on a smartphone or sensor node). Besides the target platform, other factors influence non-functional properties of a program. Database performance depends on the workload, cache size, page size, disk speed, reliability and security features, and so forth. Non-functional properties can be customized by selecting a specific set of features, called a *configuration*, that yields a valid program. However, finding *the best* configuration efficiently is a hard task. There can be hundreds of features resulting in myriads of configurations: 33 optional and independent features yields a configuration for each human on the planet, and 320 optional features yields more configurations than there are estimated atoms in the universe. To find the configuration with the best performance for a specific workload requires an intelligent search; brute-force is infeasible.

We aim at *predicting* a configuration's non-functional properties for a specific workload based on the user-selected features [3][4]. That is, we aggregate the influence of each selected feature on a non-functional property to compute the properties of a specific configuration. Here, we concentrate on performance predictions only. Unfortunately, the accuracy of performance predictions may be low, because many factors influence performance. Usually, a property, such as performance, is program-wide: it emerges from the presence and interplay of multiple features. For example, database performance depends on whether a search index or encryption is used and how both features operate together. If we knew how the combined presence of two features influences performance, we could predict a configuration's performance more accurately. Two features interact if their simultaneous presence in a configuration leads to an unexpected behavior, whereas their individual presences do not [5][6].

Today, developers detect feature interactions by analyzing the program (e.g. source code or control flow) or specification of features [7]. These and similar approaches require substantial domain knowledge, exhaustive analysis capacities, or availability of source code to achieve the task. Furthermore, each implementation technique (e.g., configuration options, `#ifdef` statements, generators, components, and aspects) requires a specialized solution. To the best of our knowledge, there is no generally applicable approach that treats a customizable program as a black box and detects performance feature interactions automatically.

We improve the accuracy of predictions in two steps: (i) we detect which features interact and (ii) we measure to what extent they interact. In our approach, we aim at finding the sweet spot between prediction accuracy, generality in terms of a black-box approach, and measurement effort.

The distinguishing property of our approach is that we neither require domain knowledge, source code, nor complex program-analysis methods, and we are not restricted to special implementation techniques, programming languages, or domains. Overall, we make the following contributions:

- An approach for efficient (in terms of measurement complexity) automated detection and quantification of performance feature interactions to enable an accurate prediction of a configuration's performance.
- An improved tool, called *SPL Conqueror* [8], to measure performance, detect feature interactions, and predict performance in an automated manner.
- A demonstration of practicality and generality of our approach with six customizable programs and product lines from different domains, programming languages, and customization mechanism.
- A 95 percent prediction accuracy when feature interactions are included, which is a 15 percent improvement over an approach that takes no interactions into account.

In contrast to our previous work [3][8], we (1) do not rely on domain knowledge, (2) reduce the effort for pair-wise measurement, (3) measure and predict performance instead of footprint size, (4) incorporate higher-order feature interactions, and (5) evaluate our approach with additional industrial product lines.

II. A Model of Feature Interactions

Our work relies on a recent model of feature composition [9]. If program P consists of features a, b, and c, we write: $P = a \cdot b \cdot c$ where $\cdot$ denotes the associative and commutative composition of features. Evaluating $a \cdot b \cdot c$ generates P.[1]

Features interact: Features that perform one way in isolation may behave differently when other features are present; interactions may affect semantics as well as (in our case) performance of the overall system. A classic example is a flood-control (fc) sensor working with a fire-alarm (fa) sensor [10]. If only one of fc or fa is present, the behavior is unambiguous: Water is turned on when fire is detected and turned off when a flood is detected. When fc and fa are both present, there is an interaction $fc\#fa$ that turns water off after the fire sensor turned water on to prevent a fire. In code, we make this interaction explicit such that we can control this interaction with an appropriate behavior. Nevertheless, the interaction is present whether we handle it or not.

More generally, if a program P contains features a and b, it should also include the interaction $a\#b$. Basic mathematics encodes these ideas. When a stakeholder wants features a and b, (s)he also wants their interaction $a\#b$ (because $a\#b$ says how a and b are to work correctly together, e.g., keeping water on when fire and flood are detected). The associative and commutative operation $\times$ expands a given configuration to all feature terms and all feature-interaction terms:[2]

$$a \times b = a\#b \cdot a \cdot b \quad (1)$$

That is, a program does not only contain the behavior of each individual feature, but also the interaction behaviors among all features. Many of these feature interactions have no observable effect; only some of them are relevant. In this paper, we propose heuristics to detect only the relevant performance feature interactions.

To relate the above abstract model to performance prediction, we state that performance of a feature composition $\Pi(a \cdot b)$ be the sum of their individual performance values:[3]

$$\Pi(a \cdot b) = \Pi(a) + \Pi(b) \quad (2)$$

From (1) and (2), we estimate P's performance as follows:

$$\begin{aligned}\Pi(P) &= \Pi(a \times b \times c)\\ &= \Pi(a \cdot b \cdot c \cdot a\#b \cdot a\#c \cdot b\#c \cdot a\#b\#c)\\ &= \Pi(a) + \Pi(b) + \Pi(c) +\\ &\quad \Pi(a\#b) + \Pi(a\#c) + \Pi(b\#c) + \Pi(a\#b\#c)\end{aligned}$$

To improve prediction accuracy, we need to determine the influence of an interaction on performance. We use a basic result that follows from (1) and (2). If we can measure a performance value for $\Pi(a)$ and $\Pi(b)$, we certainly can measure the value of $\Pi(a \times b)$. We therefore know the value of $\Pi(a\#b)$:

$$\Pi(a\#b) = \Pi(a \times b) - \Pi(a) - \Pi(b) \quad (3)$$

Here is the challenge: a product of n features yields $O(2^n)$ terms. We cannot compute a value for each term, as this is infeasible for anything beyond programs with few features. Furthermore, (3) assumes that we can measure the performance influence of each feature in isolation. This is not always possible. We avoid both problems by composing multiple terms that cannot be separately measured as a single term, called a *delta*. Given a base configuration C, we compute the impact of a feature a on C's performance as the performance delta induced by feature a:

$$\Delta a_C = \Pi(a \times C) - \Pi(C) \quad (4)$$

From (4) and (1), an equivalent definition of Δa_C is:

$$\begin{aligned}\Delta a_C &= \Pi(a \times C) - \Pi(C) \quad // \ (4)\\ &= \Pi(a\#C) + \Pi(a) + \Pi(C) - \Pi(C) \quad // \ (1)\\ &= \Pi(a\#C) + \Pi(a) \quad (5)\end{aligned}$$

That is, Δa_C is the performance contribution of a by itself plus the performance contributions of a's interaction with all

[1] Henceforth, capital letters denote compositions of one or more terms, lowercase letters a are terms (features or feature interactions).

[2] Commutativity and other axioms of sequential, interaction, and product composition are spelled out in [9]; details beyond what is presented here are non-essential to this paper.

[3] As a limitation of this approach, we require additivity of performance measurements.

terms in C. (If C is the empty set, then $\Delta a_C = \Pi(a)$). If C is a product of i features, Δa_C is a sum of $O(2^i)$ terms.

As we demonstrate in subsequent sections, knowing Δa_C for some C is often sufficient to accurately predict the performance of programs that include a. We do not need to assign values to each of Δa_C's terms; we measure only two variants of (4) instead of 2^i terms. Herein lies the key to the efficiency and practicality of our approach.

III. Predicting Performance

We predict performance (and other non-functional properties) by measuring the influence of each feature, its delta, and summing the deltas for all relevant features. With few measurements (linear complexity in the number of features), we can predict performance of all configurations (exponential in the number of features). Although the approach is simple, it yields surprisingly good results.

The general concept of quantifying the influence of each feature on performance is as follows: For each feature a, we find a configuration $min(a)$ that is *minimal* in the number of features such that $min(a)$ does not contain a and both $min(a)$ and $a \times min(a)$ are valid configurations.[4] We determine each feature's delta as:

$$\Delta a_{min} \quad = \quad \Pi(a \times min(a)) \; - \; \Pi(min(a))$$

Consider the feature model in Figure 1, which has five features. The minimal configuration for each feature is:[5]

Feature	$min()$
b	{}
i	b
t	b
e	b
d	$b \times e$

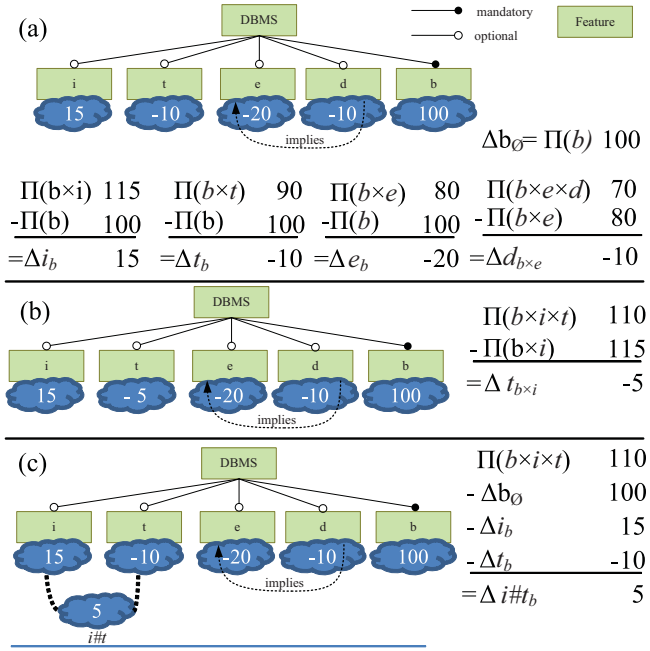
We need only five measurements to determine the influence of each feature (all values in our example are measured in transactions per second):

$$\begin{aligned}
\Delta b_{min} &= \Pi(b) - 0 &&= \quad 100 \\
\Delta i_{min} &= \Pi(b \times i) - \Pi(b) &&= \quad 15 \\
\Delta t_{min} &= \Pi(b \times t) - \Pi(b) &&= -10 \\
\Delta e_{min} &= \Pi(b \times e) - \Pi(b) &&= -20 \\
\Delta d_{min} &= \Pi(b \times e \times d) - \Pi(b \times e) &&= -10
\end{aligned}$$

[4]Features may not be independent, such that we cannot measure arbitrary configurations. We explored calculating deltas in the presence of complex domain dependencies previously [3]. It is outside the scope of this paper.

With constraints between features, in principle, there can be multiple minimal configurations (for example, in the presence of mutually exclusive features). In this case, we use any minimal configuration. Furthermore, we admit the empty or null program as a minimal configuration when determining the performance of a root feature.

[5]A feature model, a standard idea in product-line engineering [2], defines features and their relationships. Features are decomposed into a hierarchical structure and are marked as mandatory, optional, or mutually exclusive. To select a child feature, the parent feature must be selected. A configuration is valid if its feature selection fulfills all constraints (i.e., arbitrary propositional formulas) of the feature model.



i = Index, t = Transactions, e = Encryption, d = Decryption, b = Base; Measured values are transactions per second.

Figure 1. Measuring deltas for features and interactions.

To predict the performance of a configuration, we simply add the deltas of all relevant features. For example, for configuration $b \times t \times i$, we predict $\Delta b_{min} + \Delta t_{min} + \Delta i_{min} = 100 - 10 + 15 = 105$.

Unfortunately, this prediction scheme is inaccurate. As mentioned earlier, when measuring feature deltas, we might obtain very different results when using different configurations. Consider Figure 1b, which computes the delta for feature t for a different configuration. Our first value, computed above, was $\Delta t_{min} = -10$, whereas the newly computed value is $\Delta t_{b \times i} = -5$. Consequently, predictions for the same configuration $b \times t \times i$ will differ when using Δt_{min} (105) or $\Delta t_{b \times i}$ (110). The difference is due to feature interactions. Detecting and quantifying the influence of interactions allows us to overcome the differences among different deltas leading to consistent predictions. The question is: Which features interact that cause this discrepancy?

If we know that two features interact, we can improve our prediction by measuring the delta for their interaction. Suppose configuration C has both features a and b. The contribution of the interaction of a and b to C is:

$$\begin{aligned}
\Delta(a\#b)_C &= \Pi(a\#b \times C) - \Pi(C) \\
&= \Pi(a\#b\#C) + \; \Pi(a\#b) + \Pi(C) - \Pi(C) \\
&= \Pi(a\#b) + \Pi(a\#b\#C) \qquad (6)
\end{aligned}$$

Similar to the delta of a feature, the delta of interaction $a\#b$ includes the interaction $a\#b$ and all interaction terms of $a\#b$ with terms in C.

In Figure 1c, we illustrate such a measurement for

interaction $i\#t$. Knowing the interaction's delta improves our predictions: in our example, it patches the value of Δt_{min}. If more than two features interact (a.k.a., higher-order interactions [11]), we proceed in a similar way. The challenge is how to find interactions that actually contribute to performance out of an exponential number of potential interactions.

IV. Automated Detection of Feature Interactions

Our goal is to identify feature interactions automatically using a small number of measurements. Our approach consists of two steps: (1) identifying features that participate in some interactions (called *interacting features*) and (2) finding minimal combinations of features that actually cause a feature interaction. We use the setting from Figure 1 as our running example.

A. Detecting Interacting Features

Our first step is to identify features that interact. The rationale is to reduce our search space. For example, suppose a program has 16 features, in which 4 features interact, the rest do not. We have to look only at $2^4 = 16$ instead of $2^{16} = 65536$ configurations to detect interactions.

In the presence of interacting features, the delta for a feature a differs depending on which base configurations it was measured with. We say a is *not an interacting feature* if Δa_C is the same for all possible base configurations C (within some measurement accuracy). Conversely, if Δa_C changes with different configurations of C, we know that a is interacting. We express this as:

$$a \text{ interacts} \Leftrightarrow \exists\, C, D \mid C \neq D \wedge \Delta a_C \neq \Delta a_D$$

To avoid measuring Δa_C for a potentially exponential number of configurations of C, we use a heuristic. We determine the deltas of a that are most likely to differ, because it is affected by the largest number of feature interactions: We compare Δa_{min}, the delta for the minimal configuration, with Δa_{max}, a delta for a configuration with most features selected. Let $max(a)$ and $a \times max(a)$ be two valid configurations, such that $max(a)$ does not contain a and is a maximal set of features that could be composed with a. We call $max(a)$ a *maximal configuration*.[6] Δa_{max} is their performance difference:

$$\Delta a_{max} = \Pi(a \times max(a)) - \Pi(max(a))$$

The rationale of determining $max(a)$ is that it maximizes the number of features that could interact with a. Consequently, if Δa_{min} and Δa_{max} are similar, then a does not interact with the features that are present in $max(a)$ but not in $min(a)$. Otherwise, a interacts with those features (we do not know yet with which features and to what extent). Thus, with at most four measurements per feature (two for Δa_{min} using $\Pi(a \times Min)$ and $\Pi(Min)$, and two for Δa_{max} using $\Pi(a \times Max)$ and $\Pi(Max)$), we discover interacting features.[7]

In our running example, we determine the following maximal configurations and assume the following corresponding measurements:[8]

Feature	$max()$	$\Pi(max())$
i	$b \times t \times e \times d$	60
t	$b \times i \times e \times d$	85
e	$b \times i \times t$	110
d	$b \times i \times t \times e$	90

Note $max(e)$ does not include d, as d requires e for a valid configuration (Figure 1). With these additional measurements, we compute the additional deltas as follows with six measurements:

$$\begin{aligned}
\Delta i_{max} &= \Pi(i \times max(i)) - \Pi(max(i)) &&= 20 \\
\Delta t_{max} &= \Pi(t \times max(t)) - \Pi(max(t)) &&= -5 \\
\Delta e_{max} &= \Pi(e \times max(e)) - \Pi(max(e)) &&= -20 \\
\Delta d_{max} &= \Pi(d \times max(d)) - \Pi(max(d)) &&= -10
\end{aligned}$$

We conclude that features i and t are interacting:

$$\begin{aligned}
\Delta i_{min} \neq \Delta i_{max} \quad &since \quad 15 \neq 20 \\
\Delta t_{min} \neq \Delta t_{max} \quad &since \quad -10 \neq -5 \\
\Delta e_{min} = \Delta e_{max} \quad &since \quad -20 = -20 \\
\Delta d_{min} = \Delta d_{max} \quad &since \quad -10 = -10
\end{aligned}$$

We know that feature i interacts with a feature in the set $max(i) \backslash min(i)$. From these candidate features, we can exclude features b, e, and d, because their deltas do not change. Feature t remains the only candidate for interaction. The same conclusion is reached had we analyzed feature t (concluding feature i is the only possible interaction candidate). In this way, we found the feature combination that causes an interaction. Note that if we find more than two interacting features, we have no information which feature combination causes an interaction. This is the goal of the next step.

B. Identifying Feature Combinations Causing Interactions

After detecting all interacting features, we have to find the specific, valid combinations that actually have an influence on performance. Suppose we know that features a, b, and c are interacting. We have to identify which of the following interactions have an influence on performance: $a\#b$, $a\#c$, $b\#c$, or $a\#b\#c$. Again, we do not want to measure all combinations (whose number is exponential in the number of interacting features).

[6] We allow the empty set as a valid configuration. This is necessary to create a maximal configuration for mandatory features.

[7] Of course, there is an obvious situation that we can not detect: when two interactions cancel each other (e.g., one has influence $+4$ and another one -4), we will not detect them. We have no evidence that this situation is common, but we are aware of its existence.

[8] Surprisingly, $max(b)$ is an empty configuration, because feature b is mandatory; the only valid configuration without feature b is the empty set.

We use three heuristics. Each makes an assumption under which it can detect interactions (thus improving performance prediction) with a few additional measurements. Some heuristics are based on the experience we gained during the manual analysis of feature interactions (i.e, searching the source code for nested `#ifdef` statements, using domain knowledge, etc.) for the prediction of a program's binary footprint [3]. Other heuristics are based on assumptions we make due to analyses of source-code feature interactions and on related work (see Section VI). We explore in our evaluation whether our heuristics actually reduce measurement effort and improve accuracy of our predictions.

Auxiliary – Implication Graph: In all three heuristics, we reason about feature chains in an implication graph. An *implication graph* is a graph in which nodes represent features and directed edges denote implications between features. Using implications, we conclude that Δa_{min} always includes the influence of all interactions with features implied by a (i.e., all features in a's implication chain). For example, if feature a always requires the presence of feature b, then we have implicitly quantified the influence of interaction $a\#b$ when computing Δa_{min}. This mechanism reduces computation effort in all heuristics, especially, for hierarchically-deep feature models and for feature models with many constraints.

Heuristic 1 – Pair-Wise Interactions (PW): We assume that pair-wise (or first-order) interactions are the most common form of performance feature interactions.

We justify this assumption as follows: Related research often uses a similar approach: The software-test community often uses pair-wise testing to verify the correctness of programs [12][13]. Pair-wise testing was also applied successfully to test feature interactions in the communication domain [14] and to find bugs in product-line configurations [15]. Furthermore, analysis of variability in 40 large-scale programs showed that structural interactions are mostly between two features [16]; although structural interactions do not necessarily cause performance feature interactions, we assume that this distribution also holds for performance, because the additional code may have some affect on performance.

Within the set of interacting features, we use this heuristic to locate pair-wise interactions first (as they are the most common). We search for higher-order interactions with the remaining heuristics.

Heuristic 2 – Composition of Higher-Order Interactions (HO): We assume that second-order feature interactions (i.e., interactions among three features) can be predicted by analyzing already detected pair-wise interactions.

The rationale is, if three features interact pair-wise in any combination, they likely also participate in a triple-wise interaction. That is, if we know that two of these three interactions $\{a\#b, b\#c, a\#c\}$ are non-zero, then and only then will we check whether $a\#b\#c$ has an influence on performance. For example, if both $a\#b$ and $b\#c$ allocate 1 GB RAM, then it is likely that there is an interaction $a\#b\#c$ that results in a lower performance (because 2 GB RAM was allocated). We experienced this phenomenon in previous work on measuring and predicting footprint [3]. A different footprint may also indicate a possible impact on performance, because either functionality is added (increased footprint) or is removed (decreased footprint). This added or removed functionality can cause performance deviations.

We do not consider other higher-order interactions to save a huge number of measurements. Thus, we might miss some interactions in attempt to balance measurement effort and accuracy.

Heuristic 3 – Hot-Spot Features (HS): Finally, we assume the existence of *hot-spot* features. We experienced that there are usually a few features that interact with many features and there are many features that interact only with few features. High coupling between features or many dependencies can impact the performance of the whole system, because both features strongly interact with each other at the implementation level.

These observations are analogous to previous work on coupling in feature-oriented and object-oriented software [17][18], and footprint feature interaction [3]. We anticipate the same distribution for performance feature interactions, following a power law[18].

Using this heuristic, we perform additional measurements to locate interactions of hot-spot features with other interacting features. Specifically, we attempt to locate second-order interactions for hot-spot features, because they seem to represent a performance-critical functionality in a program. We do not identify interactions with an order higher than three, because this increases measurement effort substantially.

C. Realization

So far, we described a general approach to (1) detect interacting features and (2) to find feature combinations that cause interactions. Next, we detail how we implemented these techniques and heuristics in our tool *SPL Conqueror*: `http://fosd.de/SPLConqueror`

As an underlying data structure, we use an implication graph, as described earlier. We can easily generate this graph from a feature model using a SAT solver [19]. To locate pair-wise interactions (PW heuristic), we consider only pair-wise interactions between interacting features of different implication chains. We do not need to determine interactions of features belonging to the same implication chain, because the interaction is already included in Δa_{min}. Furthermore, the order of the measurements is crucial. Our algorithm starts from the top of one implication chain and determines the influence of interacting features with the interacting features of another chain, also starting from the top. Afterwards, we continue with the next chain. For example, in Figure 2, the order we use to detect pair-wise

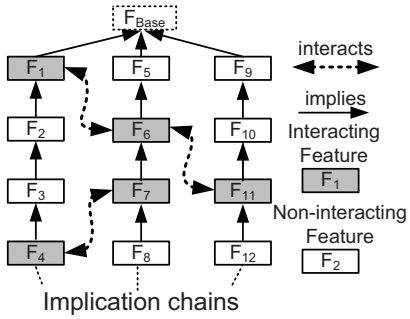


Figure 2. Implication chains with interacting features.

interactions is $F_1\#F_6$, $F_1\#F_7$, $F_4\#F_6$, $F_4\#F_7$, $F_6\#F_{11}$, $F_7\#F_{11}$,$F_1\#F_{11}$, $F_4\#F_{11}$.

To identify whether two features a and b interact, we compare the measured performance $\Pi(a \times b)$ with the performance *prediction* of the same configuration that includes all known feature interactions up to this time. If the result of $\Delta a\#b_C$ exceeds a threshold (e.g., we use the standard deviation of measurement bias as a threshold), we record it.

Next, we search for second-order interactions among features that interact in a pair-wise fashion (HO heuristic). Again, we perform additional measurements and compare them to the predicted results. For example, if we noticed that F_1 interacts with F_7 and F_7 interacts with F_{14}, we would examine whether interaction $F_1\#F_7\#F_{14}$ has an influence on performance.

Finally, we search for further second-order interactions involving hot-spot features (HS heuristic). We count the number of interactions per feature identified so far. Next, we compute the average number of interactions per feature. We classify all features that interact above the arithmetic mean as hot-spot features (other thresholds are possible, too). With hot-spot features, we search (with the usual mechanism: additional measurements, comparing deltas) for interactions involving (1) a hot-spot feature, (2) a feature that already interacts with this hot-spot feature, and (3) an interacting feature that does not interact pair-wise with the hot-spot feature.

V. Evaluation

Our approach to performance prediction is simple. But it is the simplicity that makes it practical. We demonstrate this with six real-world case studies.

The goal of our evaluation is to judge prediction accuracy and the utility of our heuristics. That is, we analyze how we detect performance feature interactions with additional measurements and how detected interactions improve prediction accuracy. To that end, we compare predictions with actual performance measurements.

A. Experimental Setting

We selected six existing real-world programs (i.e., three customizable programs and three product lines) with different characteristics to cover a broad spectrum of scenarios (see Table I). They are of different sizes (45 thousand to 300 thousand lines of code, 192 to millions of configurations), implemented in different languages (C, C++, and Java), and configurable with varying mechanisms (such as conditional compilation, configuration files, and command-line options).

Project	Domain	Lang.	LOC	Features	Configs
Berkeley DB CE	Database	C	219,811	18	2560
Berkeley DB JE	Database	Java	42,596	32	400
Apache	Web Server	C	230,277	9	192
SQLite	Database	C	312,625	39	3,932,160
LLVM	Compiler	C++	47,549	11	1024
x264	Video Enc.	C	45,743	16	1152

Table I
Overview of sample programs used in the evaluation

The programs we selected have usually under 3 000 configurations. The reason is that, this way, we can actually measure *all* configurations of these programs in a reasonable time. Hence, even though it required over 60 days of measurement with multiple computers, we could actually perform the brute-force approach and determine accuracy of our prediction over *all* configurations.

1) Setup: We measure *all* configurations of all programs that affect performance (i.e., that are invoked by a benchmark). The exception is SQLite in which we measure only the configurations needed to detect interactions and additionally 100 random configurations to evaluate the accuracy of predictions. We identified features in each case study and created a feature model describing their dependencies. All feature models and measurement results are available online at the tool's website.

We automated the process of generating programs according to specific configurations and running the benchmark. Since Berkeley DB C and Java and SQLite use compile-time configuration, we compiled a new program for each configuration that includes only the relevant features. For Apache, LLVM, and x264, we mapped the configuration to command-line parameters. We used five standard desktop computers for the measurements.[9]

We repeated each measurement between 5 to 20 times depending on the measurement bias. It is known that measurement bias can cause false interpretations and are difficult to control [20], especially for performance [21]. The width of the 95 % confidence interval is smaller than 10 % of the according means. We used a range between 2 to 10 % to specify the threshold for detecting performance feature interactions. We use the mean of all measurements of a single configuration C as $\Pi(C)$.

2) Benchmarks: We use standard benchmarks either delivered by the vendor or used in the community of the respective

[9]Intel Core 2 Quad CPU 2.66 GHZ, 4GB RAM, Vista 64Bit; AMD Athlon64 2.2GHz, 2GB RAM, Debian GNU/Linux 7; AMD Athlon64 Dual Core @2.0GHz, 2GB RAM, Debian GNU/Linux 7; Intel Core2 Quad @2.4GHz, 8GB RAM, Debian GNU/Linux 7. Each program was benchmarked on an individual systems.

application. We did not develop our own benchmark to avoid bias and uncommon performance behavior caused by flaws in benchmark designs.

Since performance predictions are especially important in the database domain, we list three database product lines: Berkeley DB's Java and C version (which differ significantly in their implementation and provided functionality) and SQLite. For each program, we use the benchmark delivered by the vendor. For example, we use Oracle's standard benchmark to measure the performance of Berkeley DB. The workload produced by the benchmarks is a typical sequence of database operations.

Furthermore, we selected the Apache Web server to measure its performance in different configurations. We used the tools *autobench* and *httperf* to produce the following workload: For each server configuration, we send 810 requests per second to a static HTML page (2 KB) provided by the server. After 60 seconds, we increase the request rate by 30 until 2700 requests per seconds are reached. After this process, we analyzed at which request rate the Web server could no longer respond or produced connection errors.

LLVM is a modular compiler infrastructure. For our benchmarks, we use the *opt-tool* that provides different compile-time optimizations. We measure the time LLVM needs to compile its standard test suite (i.e., with different optimizations, such as inline functions and combine redundant instructions enabled). In this case, the workload is the program code from the LLVM test suite that has to be compiled with the enabled optimizations.

x264 is a command line tool to encode video streams into H.264 and MPEG-4 AVC format. The tool provides several options, such as parallel encoding on multiple processors. We measured the time needed to encode the video trailer *Sintel* (735 MB). This trailer is used by different video-encoding projects as a standard benchmark for encoders.

B. Results

We compute a fault rate of our prediction as the relative difference between predicted and actual performance: $\frac{|actual-predicted|}{actual}$ and accuracy as 1-fault rate in percent. As said, we measure each program several times. From these measurements, we compute the average performance (i.e., arithmetic mean) and the standard deviation. We use the average performance to compute the delta of a feature. We use the standard deviation to set the threshold at which we identify a feature interaction, because we consider every unexpected performance behavior above the measurement error as an interaction.

1) Accuracy and Effort: In Table II, we show the results of our six case studies: For each approach, we depict the required number of measurements, the time needed for these measurements, and the number of identified interactions. Furthermore, we show the distribution of the fault rate of our predictions with box plots. Finally, we show for each approach the mean fault rate of all predictions including the standard deviation. Note that, when adding a new heuristic, we keep the previous heuristic working, because they are successively applied to a program.

The feature-wise (FW) approach does not use a heuristic and does not account for feature interactions. We achieve good predictions for programs in which interactions have no substantial influence on performance. For example, our predictions have an average error rate of less than 8 % for all LLVM configurations. In contrast, we usually have a high fault rate (e.g., over 44 % for BerkeleyDB C version) when no interactions are considerd. The average accuracy of performance prediction is 79.7 %.

Using the pair-wise heuristic (PW) usually improves predictions significantly (91 %, on average), because the majority of interactions are pair-wise. The benefit of implication chains compared to the common pair-wise measurement is that it reduces the number of measurements. For example, we require 81 measurements to detect first-order interactions for x264 (see Table II), which is 82 less than 163, which would be needed to measure all pairs of features.

With the higher-order (HO) heuristic, we achieve an average accuracy of 93.7 % for all case studies. Interestingly, for LLVM, we could not find a feature combination that satisfies our preconditions to search for higher-order interactions. It is important to note that this heuristic usually doubles the number of measurements. For Apache the fault rate increases, because measurement bias over the determined threshold lead to a false detection of interactions. We detected these false positives when we search for third-order feature interactions, as we do with the hot-spot heuristic.

Finally, the hot-spot heuristic (HS) (including the other two heuristics) improves accuracy again to 95.4 % on average. Considering that the measurement bias for a single measurement of the case studies Apache, LLVM, and x264 is 5 %, for SQLite it is 1 %, and for Berkeley C and Java version it is 2 % our predictions are as accurate as the bias of a single measurement.

2) Influence of Heuristics: Since all our heuristics are consecutively applied, we can visualize the trade-off between additional measurements and error rate of predictions as in Figure 3. Dashed lines represent the average error rate of our predictions and straight lines depict the percentage of measurements, compared to the maximum number of measurements. As expected, with an increasing number of measurements the fault rate decreases. The results show that the relative number of measurements strongly differ to achieve the same accuracy for different programs. Further note that we have to measure approximately 0.1 % of all variants of SQLite, which demonstrates the scalability of our approach.

Pair-Wise vs. Higher-Order Interactions: Look at the Apache case study (which is similar to others): A higher-order interaction usually improves predictions. We detected 18 first-

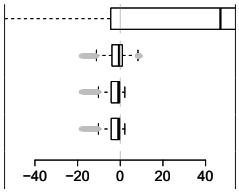
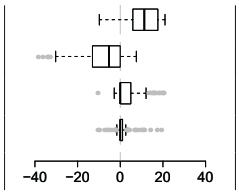
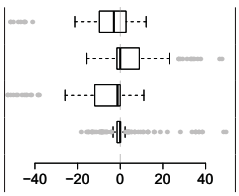
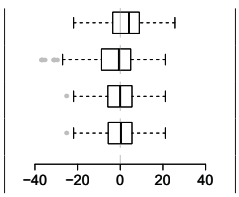
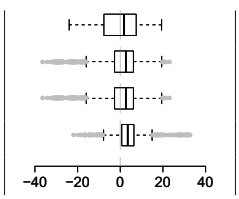
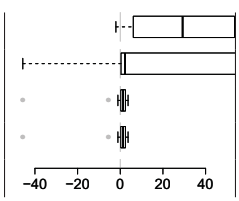
Program	Appr.	Effort			Fault Rate (in %)	
		Measurements	Time (in h)	Interactions	Distribution	Mean±Std
Berkeley CE	FW	15 (0.6 %)	3	0		44.1±42.3
	PW	139 (5.4 %)	23	14		3.9±5.3
	HO	160 (6.3 %)	27	22		2.8±3.7
	HS	164 (6.4 %)	27	22		2.8±3.7
	BF	2 560 (100 %)	426	-		___
Berkeley JE	FW	10 (3 %)	8.4	0		17.7±19.6
	PW	48 (12 %)	40	24		8.5±9.6
	HO	16 (4 %)	97	51		3.8±5.7
	HS	162 (40.5 %)	137	69		1.7±3.5
	BF	400 (100 %)	335	-		___
Apache	FW	9 (4.7 %)	10	0		14.9±24.8
	PW	29 (15.1 %)	32	18		7.7±11.2
	HO	80 (41.7 %)	89	44		11.6±22.7
	HS	143 (74.5 %)	159	73		5.3±10.8
	BF	192 (100 %)	213	-		___
SQLite	FW	26 (0 %)	2.1	0		7.8±9.2
	PW	566 (0 %)	47	2		9.3±12.5
	HO	566 (0 %)	47	3		7.1±9.1
	HS	569 (0 %)	47.4	3		7±9
	BF	3 932 160 (100 %)	327 680			___
LLVM	FW	11 (1.1 %)	2	0		7.8±9
	PW	62 (6.1 %)	12	27		7.4±10.2
	HO	62 (6.1 %)	12	27		7.4±10.2
	HS	88 (8.6 %)	17	38		5.7±7
	BF	1 024 (100 %)	202	-		___
x264	FW	12 (1 %)	2	0		29.6±22
	PW	81 (7 %)	16	13		17.9±27.2
	HO	89 (7.7 %)	17	17		5.1±15.1
	HS	89 (7.7 %)	17	17		5.1±15.1
	BF	1 152 (100 %)	224	-		___

Table II
EVALUATION RESULTS FOR SIX CASE STUDIES; APPROACHES (APPR.): FEATURE-WISE (FW), PAIR-WISE (PW), HIGHER-ORDER (HO), HOT-SPOT (HS), BRUTE FORCE (BF). MEAN: MEAN FAULT RATE OF PREDICTIONS, STD: STANDARD DEVIATION OF PREDICTIONS.

order interactions and 55 higher-order interactions. Using the PW heuristic, we have some features that interact with many other features (e.g., KeepAlive) and other features that interact only with one or two features (e.g., ExtendedStatus). Although without using the hot-spot heuristic, we observe that some features are more likely to interact. Additionally, we found two features (Base and InMemory) that do not interact which substantially decreases the search space.

C. Threats to Validity

Internal Validity: Regarding SQLite, we cannot measure all possible configurations in reasonable time. Hence, we sampled only 100 configurations to compare prediction and actual performance values. We are aware that this evaluation leaves room for outliers.

Also, we are aware that measurement bias can cause false interpretations [20]. Since we aim at predicting performance for a special workload, we do not have to vary benchmarks. Additionally, we determined the width of a 95 % confidence

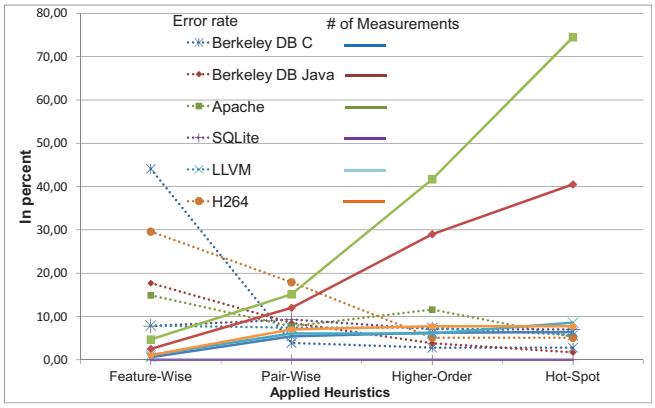


Figure 3. Comparing percentage of measurements (straight lines) with average error rates of predictions (dashed lines) for each heuristic.

interval of our measurements smaller than 10 % of the according means.

External Validity: We aimed at increasing external validity by choosing programs from different domains with different configuration mechanisms and implemented with different programming languages. Furthermore, we used programs that are deployed and used in the real-world. Nevertheless, we are aware that the results of our evaluations are not automatically transferable to all configurable programs. In addition to our sample program selection, the strong and exhaustive evaluation (over 60 days of measurement with 5 computers) indicate that our heuristics hold for many practical application scenarios.

D. Discussion

Although we use a simplistic performance model, we demonstrated that the approach is feasible. With an average accuracy of 95 %, we achieve predictions that even stay in the range of the observed measurement bias for the case studies. It is important to note that we experienced large differences in accuracy when we changed the threshold at which a performance feature interaction is detected. Having a too small threshold causes many false detections of interactions. The fault rate increases, because we sum the influence of measurement bias instead of the influence of interactions.

We observed that we need a relatively large number of measurements when many alternative features exist compared to independent features, because having many alternative features limits the number of valid configurations substantially. For example, we can only generate 400 configurations in Berkeley DB Java, though it has 32 features. This number is below quadratic. Hence, already the detection of interacting features require a relatively large number of measurements. However, having programs with a small number of valid configurations make a brute-force approach feasible, which is not our intended application scenario.

Furthermore, we do not consider performance behavior of a program independently of the workload. We can make accurate statements for any configuration given a *specific* workload. That is, we address end-users that have a certain application scenario in mind but do not know which configuration performs best. Measurements can be performed on a live system in a real environment, which produces more suitable performance predictions than standard benchmark results in a synthetic environment. For a new customer or a new workload, we have to repeat the measurements. We believe that many interactions still exist, though the values of the interactions will change. This, however, means that we may save additional measurements for new customers, since we already know which features interact.

We believe that our approach is not limited to the detection of non-functional feature interactions for the property performance, but also for other *quantifiable and additive* non-functional properties, such as binary footprint, memory footprint, and bandwidth.

VI. Related Work

A. Performance Prediction

There are several approaches that aim at predicting performance of a customizable program or a product line. Abdelaziz et al. provide an overview of component-based prediction approaches [22]. Typically, the approaches belong to one of three categories: model-based, measurement-based (as we use in this paper), and mixed.

Model-based: Model-based predictions are common [23][24]. For example, linear and multiple regression explore relationships between input parameters (features) and measurements. Based on such a regression model, different estimation methods (e.g., ordinary least squares) can be used to predict the performance for specific input parameters. Bayesian (or belief) networks are used to model dependencies between variables in the network [25]. They are often used to learn causal relationships and hence may be applicable to detect feature interactions. Furthermore, machine-learning approaches can be used to find the correlation between a configuration and a measurement (e.g., *canonical correlation analysis* [26]), which uses dataset pairs to identify those linear combinations of variables with the best correlation. *Principal component analysis* [27] finds dimensions of maximal variance in a dataset that can also be used to detect interactions. Ganapathi et al. provides an analysis for different machine-learning approaches in the context of performance prediction of database queries [28].

The feasibility of model-based approaches depends on the application scenario and program to be analyzed. Our work differs in that it offers a general way to produce accurate predictions independent of the application scenario, and it uses heuristics to significantly reduce measurement effort.

Krogmann et al. [29] combine monitoring data, genetic programming, and reverse engineering to reduce the number of measurements to create a platform-independent behavioral model of components. For a platform-specific prediction, they use bytecode-benchmark results of concrete systems to parameterize the behavior model. We predict the performance independently of the used programming language and availability of bytecode.

Happe et al. present a compositional reasoning approach, based on the *Palladio component model* [30]. The idea is that each component specifies its resource demands and predicted execution time in a global repository. The approach is applicable only to component-based programs, whereas we use a single approach for all customizable programs.

Also in this vein, MDE-based work uses feature models to customize or synthesize performance models (e.g. [31]). This line of research requires up-front and detailed knowledge of domain-specific performance modeling, where tuning predictions for accuracy can be difficult. Our approach avoids these problems by directly measuring performance.

Measurement-based: Sincero et al. [4] predict a configurations's non-functional properties based on a knowledge base consisting of measurements of already produced and measured configurations. They aim at finding a correlation between feature selection and measurement. This way, they can provide *qualitative* information about how a feature influences a non-functional property during configuration. In contrast to our approach, they do not actually predict a value quantitatively, and they do not provide means to detect feature interactions.

Chen et al. [32] use a combined benchmarking and profiling approach to predict the performance of component-based applications. Based on a benchmark and a Java profiling tool, a performance prediction model is constructed for application server components. In contrast, we correlate the measurements to the configuration, and measure only those configurations from which we expect to detect performance feature interactions.

Abdelaziz et al. argue that most measurement approaches lack generality [22], as they are applicable only to specific application scenarios or infrastructures [32][33]. Our work can be used for a broad range of applications of different domains, implementation techniques, etc.

B. Feature-Interaction Detection

There is a large body of research on automated detection of feature interactions (e.g., see Nhlabatsi et al. [6] and Calder et al. [34] for surveys). Many approaches aim at detecting feature interactions at the specification level. For example, Calder and Miller use a pair-wise measurement approach based on linear temporal logic to detect feature interactions [7]. They specify the behavior of a product line in Promela (a modeling language). Using a model checker, they generate for each pair-wise combination a model checking run to verify whether the defined properties are still valid. Other approaches use state charts to model and detect feature interactions [35]. For example, in [36] feature specifications are translated to a reachability graph. The authors use state transitions to detect whether a certain state is not exclusively reachable in isolation (i.e. a feature interaction occurs).

There are approaches that provide means to detect semantic feature interactions, i.e., feature interactions that change the functional behavior of a program. Some use model checking techniques to find semantic feature interactions [37][38]. Apel's work uses model-checking techniques to verify whether semantic constraints still hold in a particular feature combination [39][40]. Other approaches aim at investigating the code base to detect structural feature interactions. For example, Liu et al. [9][41] propose to model feature interactions explicitly using algebraic theory. In contrast to these approaches, we focus on performance feature interactions in a black-box fashion.

VII. Conclusion

We presented a method that allows stakeholders to accurately predict the performance of customized and generated programs. It detects interactions among configuration options or features and evaluates their influence on performance. We detect feature interactions in a step-wise manner. First, we find features that interact. Second, we detect the combinations of these features that cause a measurable interaction and quantify their impact on the performance of a configuration. The common weak spot of such an approach is the exhaustive number of measurements required to detect interactions. We solved this problem by means of three heuristics that reduce the number of measurements without sacrificing precision in predictions.

Our evaluations demonstrate that an accuracy of 95 % is possible, on average, when using our heuristics. We demonstrated generality by using applications of varying domains, implemented with different programming languages and techniques, and configured via configuration files or compilation options.

Acknowledgments

We are grateful to Janet Feigenspan for comments on an earlier draft of this paper. The work of Siegmund and Saake is supported by the German ministry of education and science (BMBF), number 01IM10002B. Rosenmüller's, Apel's, and Kolesnikov's work is supported by the German Research Foundation (SA 465/34-1, AP 206/2, AP 206/4, and LE 912/13). Kästner's work is supported by ERC grant #203099. Batory's work is funded by NSF's Science of Design Project CCF 0724979.

References

[1] A. Rabkin and R. Katz, "Static extraction of program configuration options," in *ICSE*. ACM, 2011, pp. 131–140.

[2] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[3] N. Siegmund, M. Rosenmüller, C. Kästner, P. Giarrusso, S. Apel, and S. Kolesnikov, "Scalable prediction of non-functional properties in software product lines," in *SPLC*. IEEE, 2011, pp. 160–169.

[4] J. Sincero, W. Schroder-Preikschat, and O. Spinczyk, "Approaching non-functional properties of software product lines: Learning from products," in *APSEC*. IEEE, 2010, pp. 147–155.

[5] M. Calder, M. Kolberg, E. Magill, and S. Reiff-Marganiec, "Feature interaction: A critical review and considered forecast," *Comput. Netw.*, vol. 41, no. 1, pp. 115–141, 2003.

[6] A. Nhlabatsi, R. Laney, and B. Nuseibeh, "Feature interaction: The security threat from within software systems," *Progress in Informatics*, no. 5, pp. 75–89, 2008.

[7] M. Calder and A. Miller, "Feature interaction detection by pairwise analysis of LTL properties: A case study," *Formal Methods System Design*, vol. 28, no. 3, pp. 213–261, 2006.

[8] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake, "SPL Conqueror: Toward optimization of non-functional properties in software product lines," *Software Quality Journal*, 2011, online first.

[9] D. Batory, P. Höfner, and J. Kim, "Feature interactions, products, and composition," in *GPCE*. ACM, 2011, pp. 13–22.

[10] J. Lee, K. Kang, and S. Kim, "A feature-based approach to product line production planning," in *Software Product Lines*, ser. LNCS. Springer, 2004, vol. 3154, pp. 137–140.

[11] C. Kim, C. Kästner, and D. Batory, "On the modularity of feature interactions," in *GPCE*. ACM, 2008, pp. 23–34.

[12] D. Cohen, S. Dalal, J. Parelius, and G. Patton, "The combinatorial design approach to automatic test generation," *IEEE Software*, vol. 13, no. 5, pp. 83–88, 1996.

[13] K.-C. Tai and Y. Lei, "A test generation strategy for pairwise testing," *IEEE TSE*, vol. 28, no. 1, pp. 109 –111, 2002.

[14] A. Williams, "Determination of test configurations for pairwise interaction coverage," in *TestComm*. Kluwer, 2000, pp. 59–74.

[15] S. Oster, F. Markert, and P. Ritter, "Automated incremental pairwise testing of software product lines," in *SPLC*, 2010, pp. 196–210.

[16] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *ICSE*. ACM, 2010, pp. 105–114.

[17] S. Apel and D. Beyer, "Feature cohesion in software product lines: An exploratory study," in *ICSE*. ACM, 2011, pp. 421–430.

[18] C. Taube-Schock, R. Walker, and I. Witten, "Can we avoid high coupling?" in *ECOOP*. Springer, 2011, pp. 204–228.

[19] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *ICSE*. ACM, 2011, pp. 461–470.

[20] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *ASPLOS*. ACM, 2009, pp. 265–276.

[21] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous Java performance evaluation," in *OOPSLA*. ACM, 2007, pp. 57–76.

[22] A. Abdelaziz, W. Kadir, and A. Osman, "Comparative analysis of software performance prediction approaches in context of component-based system," *IJCA*, vol. 23, no. 3, pp. 15–22, 2011.

[23] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: A survey," *IEEE TSE*, vol. 30, no. 5, pp. 295–310, 2004.

[24] I. Witten and E. Frank, *Data mining : Practical machine learning tools and techniques*, 2nd ed. Elsevier, Morgan Kaufman, 2005.

[25] F. Jensen and T. Nielsen, *Bayesian Networks and Decision Graphs*, 2nd ed. Springer, 2007.

[26] K. Mardia, J. Kent, and J. Bibby, *Multivariate Analysis (Probability and Mathematical Statistics)*, 1st ed. Academic Press, 1980.

[27] H. Hotelling, "Analysis of a complex of statistical variables into principal components," *Journal of Educational Psychology*, vol. 24, no. 6, pp. 417–441, 1933.

[28] A. Ganapathi, H. Kuno, U. Dayal, J. Wiener, A. Fox, M. Jordan, and D. Patterson, "Predicting multiple metrics for queries: Better decisions enabled by machine learning," in *ICDE*. IEEE, 2009, pp. 592–603.

[29] K. Krogmann, M. Kuperberg, and R. Reussner, "Using genetic search for reverse engineering of parametric behavior models for performance prediction," *IEEE TSE*, vol. 36, no. 6, pp. 865–877, 2010.

[30] J. Happe, H. Koziolek, and R. Reussner, "Facilitating performance predictions using software components," *IEEE Software*, vol. 28, no. 3, pp. 27 –33, 2011.

[31] R. Tawhid and D. Petriu, "Automatic derivation of a product performance model from a software product line model," in *SPLC*. IEEE, 2011, pp. 80–89.

[32] S. Chen, Y. Liu, I. Gorton, and A. Liu, "Performance prediction of component-based applications," *Journal of System Software*, vol. 74, no. 1, pp. 35–43, 2005.

[33] S. Yacoub, "Performance analysis of component-based applications," in *SPLC*. Springer, 2002, vol. 2379, pp. 1–5.

[34] M. Calder, M. Kolberg, E. Magill, and S. Reiff-Marganiec, "Feature interaction: A critical review and considered forecast," *Comp. Netw. and ISDN Systems*, vol. 41, pp. 115–141, 2003.

[35] C. Prehofer, "Plug-and-play composition of features and feature interactions with statechart diagrams," *Software and Systems Modeling*, vol. 3, no. 3, pp. 221–234, 2004.

[36] K. Pomakis and J. Atlee, "Reachability analysis of feature interactions: A progress report," in *International Symposium on Software Testing and Analysis*, vol. 21. ACM, 1996, pp. 216–223.

[37] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: Efficient verification of temporal properties in software product lines," in *ICSE*. ACM, 2010, pp. 335–344.

[38] K. Lauenroth, K. Pohl, and S. Toehning, "Model checking of domain artifacts in product line engineering," in *ASE*. IEEE, 2009, pp. 269–280.

[39] S. Apel, W. Scholz, C. Lengauer, and C. Kästner, "Detecting dependences and interactions in feature-oriented design," in *ISSRE*. IEEE, 2010, pp. 161–170.

[40] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer, "Detection of feature interactions using feature-aware verification," in *ASE*. IEEE, 2011, pp. 372–375.

[41] J. Liu, D. Batory, and S. Nedunuri, "Modeling interactions in feature-oriented designs," in *ICFI*. IOS Press, 2005, pp. 178–197.