A Mathematical Model of Performance-Relevant Feature Interactions

Yi Zhang
University of Waterloo
y825zhan@uwaterloo.ca

Jianmei Guo
East China University of Science and Technology
gjm@ecust.edu.cn

Eric Blais
University of Waterloo
eric.blais@uwaterloo.ca

Krzysztof Czarnecki
University of Waterloo
kczarnec@gsd.uwaterloo.ca

Huiqun Yu
East China University of Science and Technology
yhq@ecust.edu.cn

ABSTRACT

Modern software systems have grown significantly in their size and complexity, therefore understanding how software systems behave when there are many configuration options, also called features, is no longer a trivial task. This is primarily due to the potentially complex interactions among the features. In this paper, we propose a novel mathematical model for performance-relevant, or quantitative in general, feature interactions, based on the theory of Boolean functions. Moreover, we provide two algorithms for detecting all such interactions with little measurement effort and potentially guaranteed accuracy and confidence level. Empirical results on real-world configurable systems demonstrated the feasibility and effectiveness of our approach.

CCS Concepts

•**Software and its engineering** → **Software design techniques;** *Software development process management;* Language features; •**Mathematics of computing** → *Discrete mathematics;* •**Theory of computation** → Formalisms;

Keywords

feature interactions, performance, Boolean functions, Fourier transform

1. INTRODUCTION

Modern software systems have become highly customizable via their configuration options, which we refer to as *features*. Given a finite number of Boolean features that users can either turn on or off, any combination of these features could give rise to a particular *configuration* of the system, which in turn, given a fixed workload, produces a particular performance measure, for example execution time, of the system.

If all features of a software system are completely independent, the performance of any particular configuration of the software system would be easily predictable: it would be the "simple sum" of the performance of the configurations where only a single feature to be composed is selected. However, since modern software systems commonly and inevitably involve complex relationships among the parts of the source code responsible for different features, the actual performance of a configuration rarely fits such a simple formula.

Performance-relevant feature interactions (PRFIs) [18] are the configurations' significant deviations between their actual performance and their "simple sum" performance. In a real scenario of performance analysis, a straightforward way to understand a feature's performance influence is to measure the performance of the configuration activating the feature only. Normally, we aggregate each selected feature's performance influence to compute a particular configuration's performance. However, PRFIs would break such a "simple sum" prediction, because two or more features may interact if their simultaneous presence in a configuration leads to an unexpected performance result (that might be caused by unexpected behavior, shared resources, etc.), whereas their individual presences do not.

PRFIs are the key to performance prediction and tuning. Users can predict a configuration's performance more accurately if they understand how the combined presence of multiple features influences performance. Furthermore, they can focus on features involved in the most influential PRFIs when reconfiguring to improve performance or to avoid undesirable PRFIs. Traditional studies on feature interactions have been largely focusing on the aspect of functional behavior [1, 4, 6] and have proposed various detection methods through, for example, code analysis for better feature-oriented design and specification [3, 13]. Although there have been recent studies on predicting performance of configurable software systems [8, 18, 19], building a complete and rigorous understanding of the impact of features and feature interactions on software performance in addition to learning the performance values themselves is still important and needed.

A key challenge in the performance analysis of highly configurable systems is that, as the number of features in software systems increases and their complexities grow, the number of configurations could explode exponentially. Given that the cost of a single measurement may already be high (e.g. executing a complex benchmark), measuring the performance of *all* configurations of a software system becomes practically infeasible. Hence, this exponential growth presents a challenge for detecting PRFIs, namely the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPLC '16, September 16 - 23, 2016, Beijing, China

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4050-2/16/09...$15.00

DOI: http://dx.doi.org/10.1145/2934466.2934469

need to identify and quantify PRFIs from as few sample measurements of configurations as possible.

To formalize the problem, we consider a software system together with its performance, as an abstract Boolean function in the form of:

$$f : \{0,1\}^n \to \mathbb{R} \tag{1}$$

Take execution time as an example, and given four features (i.e., $n = 4$), the function f represents the performance of a particular configurable software system. The statement $f(1011) = 4.8$s means that the system with the first, third and fourth features turned on and the second feature off has an average execution time of 4.8 seconds when running a certain fixed benchmark for performance measurement. In this way, the performance aspect of a software system becomes equivalent to a Boolean function that maps any configuration to a real number representing its performance. We will hereafter refer to the function as a *performance function*, since we are concerned with the performance aspect of software systems.

Given the notion of a performance function, we propose a novel and rigorous approach that establishes the correspondence between theoretical properties of Boolean functions and PRFIs of software systems, such that we not only give a formal definition and interpretation of PRFIs that complies with and extends the notion proposed by Batory et al. [4], but also provide two algorithms that calculate *all* such feature interactions with little measurement effort. In brief, we give a precise definition of a PFRI as a *partial derivative* (of a mathematical function). First-order partial derivative is the contribution of one feature (all features turned off and one turned on). Higher-order partial derivatives are higher-order feature interactions. For example, the interaction between features $f1$ and $f2$ would be the performance change caused to the contribution of $f1$ by turning on $f2$, or vice versa; if there is no interaction between $f1$ and $f2$, the second-order partial derivative would be zero.

Furthermore, since our approach is based on the abstraction of Boolean functions, both the mathematical analysis and the detection algorithms generalize to any quantitative measure (a.k.a. quality attribute or non-functional property) of software systems that is associated with a configuration, such as required memory space or the number of bugs. Since we currently do not have available datasets to evaluate our approach with respect to different types of quantitative measures, we focus our analysis and evaluation on performance in this paper. We are confident that the proposed theory can be readily applied to other quantitative measures of configurable software systems.

To the best of our knowledge, our approach is the first to detect PRFIs with an upper bound on error. The main contributions of this paper include:

- a mathematical formulation of PRFIs in terms of Boolean functions;
- two algorithms for automatic detection of PRFIs with a small random sample of measured configurations.

The rest of this paper is organized as follows. Section 2 introduces the mathematical background of Boolean functions and their relationships to configurable software systems. Section 3 presents the two algorithms of detecting PRFIs. Sections 4 evaluates and discusses the experimental results of the algorithms on five real-world software systems. The paper concludes after outlining the related work.

2. MATHEMATICAL FORMALIZATION

This section introduces the notions of Fourier transform and derivatives of Boolean functions in the form of equation (1) as *alternative representations* of the performance function of a configurable software system. For coherence, we will always refer to the performance function as f. For completeness, we recall equations (2) to (8) from our previous work [20].

Table 1: Truth table of a Boolean function f

x_1	x_2	$\dots$	x_n	$f(x)$
0	0	$\dots$	0	y_1
1	0	$\dots$	0	y_2
0	1	$\dots$	0	y_3
$\vdots$	$\vdots$	$\vdots$	$\vdots$	y_i
1	1	$\dots$	1	y_{2^n}

2.1 Fourier Transform

Consider a Boolean function f of the form:

$$f : \{0,1\}^n \to \mathbb{R}$$
$$\text{where} \quad f(x) = f(x_1, x_2, \dots, x_n) = y \in \mathbb{R}$$

A natural way of visualizing f is through a "truth table" that specifies what $f(x)$ is on every input $x \in \{0,1\}^n$ as shown in Table 1.

However, f can be viewed from a different perspective through its Fourier transform. Given a function $f : \{0,1\}^n \to \mathbb{R}$, it can be rewritten as follows:

$$f(x) := \sum_{z \in \{0,1\}^n} \hat{f}(z)\chi_z(x) \tag{2}$$

where $\hat{f}(z) \in \mathbb{R}$ are the *Fourier coefficients* of f, and $\chi_z(x)$ are the *character functions* defined as:

$$\chi_z(x) := \begin{cases} +1 & \text{if } \sum_{i=1}^n z_i x_i = 0 \mod 2 \\ -1 & \text{if } \sum_{i=1}^n z_i x_i = 1 \mod 2 \end{cases} \tag{3}$$

This Fourier transform [15] essentially decomposes the function f into the sum of 2^n "basis functions", namely the character functions, indexed by the 2^n vectors z of length n, in the space $\{0,1\}^n$.

From (2), a function f is completely determined by its 2^n Fourier coefficients. In other words, the set of 2^n Fourier coefficients uniquely determines the function values $f(x)$ for all x. Perhaps less trivially, the converse is also true [15] as shown in (8), namely all Fourier coefficients can be uniquely calculated from the function f itself.

Given the usual inner product defined on Boolean functions $f, g : \{0,1\}^n \to \mathbb{R}$ as:

$$\langle f, g \rangle := \frac{1}{2^n} \sum_{x \in \{0,1\}^n} f(x)g(x), \tag{4}$$

the set of character functions as defined in equation (3) forms an *orthonormal basis* of the class of functions from $\{0,1\}^n$ to $\mathbb{R}$. Here, an orthonormal basis for an inner product space V with finite dimension is a basis for V whose vectors are all unit vectors and orthogonal to each other. Thus:

$$\langle \chi_{z_1}, \chi_{z_2} \rangle = \begin{cases} 1 & \text{if } z_1 = z_2 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Then, we derive the formula for calculating a given Fourier coef-

ficient $\hat{f}(z^*)$:

$$\begin{aligned}\langle f, \chi_{z^*}\rangle &= \left\langle \sum_{z\in\{0,1\}^n} \hat{f}(z)\chi_z, \chi_{z^*}\right\rangle \\ &= \sum_{z\in\{0,1\}^n} \hat{f}(z)\left\langle \chi_z, \chi_{z^*}\right\rangle \\ &= \hat{f}(z^*)\left\langle \chi_{z^*}, \chi_{z^*}\right\rangle \\ &= \hat{f}(z^*)\end{aligned} \tag{6}$$

by the orthonormality of the character functions.

In summary, the equivalence between the Fourier coefficients and the function values of f can be established as follows:

- For any point $x \in \{0,1\}^n$, its value can be calculated using the set of Fourier coefficients:

$$f(x) = \sum_{z\in\{0,1\}^n} \hat{f}(z)\chi_z(x) \tag{7}$$

- For any vector $z \in \{0,1\}^n$, its corresponding Fourier coefficient can be calculated using the function values:

$$\hat{f}(z) = \langle f, \chi_z\rangle = \frac{1}{2^n}\sum_{x\in\{0,1\}^n} f(x)\chi_z(x), \tag{8}$$

2.2 Partial Derivatives

Given a function $f : \{0,1\}^n \to \mathbb{R}$, treating the input as an n-dimensional vector, we can define the partial derivatives of f in the natural way:

$$\frac{\partial f}{\partial x_i}(x) := \frac{f(x_{i\to 1}) - f(x_{i\to 0})}{1-0} = f(x_{i\to 1}) - f(x_{i\to 0}) \tag{9}$$

where $x_{i\to 1}$ denotes the input x with the i-th coordinate forced to be 1, and similarly for $x_{i\to 0}$. Therefore, the derivative of f along a particular direction or coordinate i is the difference of f evaluated at x between when x_i is 1 and when it is 0.

From this definition of partial derivatives, we deduce the following properties:

1. $\frac{\partial f}{\partial x_i}(x)$ does not actually depend on x_i, therefore all second or higher order derivatives on the same coordinate are trivially 0. For all $x \in \{0,1\}^n$ and all i:

$$\begin{aligned}\frac{\partial^2 f}{\partial x_i \partial x_i}(x) =& \left(f(x_{i\to 1}) - f(x_{i\to 0})\right) - \\ & \left(f(x_{i\to 1}) - f(x_{i\to 0})\right) \\ =& 0\end{aligned} \tag{10}$$

2. The partial differentiation operator $\frac{\partial}{\partial x_i}$ here is a linear operator, and we have commutativity over it, i.e. for any coordinates i and j, we have:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial^2 f}{\partial x_j \partial x_i}. \tag{11}$$

3. Since taking derivatives along different directions is commutative, we use the shorthand notation $\frac{\partial f}{\partial x_\alpha}$ where $\alpha \in \{0,1\}^n$ for the derivative of f along all directions i where $\alpha_i = 1$. For example:

$$\frac{\partial f}{\partial x_{1011}} := \frac{\partial^3 f}{\partial x_1 \partial x_3 \partial x_4}. \tag{12}$$

With a well-defined notion of derivatives, we are able to define yet another representation of a Boolean function, through its *Taylor expansion*. Thus, given a Boolean function $f : \{0,1\}^n \to \mathbb{R}$, we have its Taylor expansion around the origin $\mathbf{0}$:

$$\begin{aligned}f(x) &= f(x_1, x_2, \ldots, x_n) \\ &= f(\mathbf{0}) + \sum_{i=1}^n \left.\frac{\partial f}{\partial x_i}\right|_{\mathbf{0}} \cdot x_i + \frac{1}{2}\sum_{i=1}^n\sum_{j=1}^n \left.\frac{\partial^2 f}{\partial x_i \partial x_j}\right|_{\mathbf{0}} \cdot x_i x_j + \cdots\end{aligned} \tag{13}$$

From equation (10), we know that any term involving either second- and higher-order derivatives or any $x_i = 0$ will vanish. Furthermore, $f(\mathbf{0})$ can be viewed as $\left.\frac{\partial f}{\partial x_{\mathbf{0}}}\right|_{\mathbf{0}}$. Therefore, the equation (13) can be simplified to:

$$\begin{aligned}f(x) &= f(x_1, x_2, \ldots, x_n) \\ &= f(\mathbf{0}) + \sum_{\alpha\in\{0,1\}^n} \frac{1}{|\alpha|!} \cdot |\alpha|!\left.\frac{\partial f}{\partial x_\alpha}\right|_{\mathbf{0}} \cdot \prod_{i:\ \alpha_i=1} x_i \\ &= \sum_{\substack{\alpha\in\{0,1\}^n \\ \alpha\le x}} \left.\frac{\partial f}{\partial x_\alpha}\right|_{\mathbf{0}}\end{aligned} \tag{14}$$

where $\alpha \le x$ means $\alpha_i \le x_i$ for all i, and $|\alpha|$ is the number of 1's in the vector α.

Intuitively, the equation (14) can be interpreted as a representation of $f(x)$ that consists of a "base value", $f(\mathbf{0})$, and a collection of increments from the appropriate directions or coordinates. For example, $f(1011)$ can be expressed as:

$$\begin{aligned}f(1011) = f(\mathbf{0}) &+ \left.\frac{\partial f}{\partial x_{1000}}\right|_{\mathbf{0}} + \left.\frac{\partial f}{\partial x_{0010}}\right|_{\mathbf{0}} + \left.\frac{\partial f}{\partial x_{0001}}\right|_{\mathbf{0}} \\ &+ \left.\frac{\partial f}{\partial x_{1010}}\right|_{\mathbf{0}} + \left.\frac{\partial f}{\partial x_{1001}}\right|_{\mathbf{0}} + \left.\frac{\partial f}{\partial x_{0011}}\right|_{\mathbf{0}} \\ &+ \left.\frac{\partial f}{\partial x_{1011}}\right|_{\mathbf{0}}\end{aligned} \tag{15}$$

The set of all 2^n derivatives $\left.\frac{\partial f}{\partial x_\alpha}\right|_{\mathbf{0}}$, indexed by all vectors in $\{0,1\}^n$, is a third equivalent representation of the function f, in the sense that all values of f can be constructed from the derivatives through Taylor expansion in equation (14) and all derivatives can be calculated from the function values using a combination of equations (9) and (11).

Furthermore, there is a direct relationship between the set of derivatives and the set of Fourier coefficients [15]. The derivatives evaluated at the origin can be calculated from the Fourier coefficients by:

$$\left.\frac{\partial f}{\partial x_i}\right|_{\mathbf{0}} = -2 \cdot \sum_{\substack{z\in\{0,1\}^n \\ z_i=1}} \hat{f}(z) \tag{16}$$

i.e. the derivative with respect to the i-th coordinate is directly proportional to the sum of all Fourier coefficients whose indices contain the i-th coordinate.

By the linearity of the differential operator and induction, derivatives evaluated at the origin with respect to multiple directions can be calculated by:

$$\left.\frac{\partial f}{\partial x_\alpha}\right|_{\mathbf{0}} = (-2)^{|\alpha|} \cdot \sum_{\substack{z\in\{0,1\}^n \\ z\ge\alpha}} \hat{f}(z) \tag{17}$$

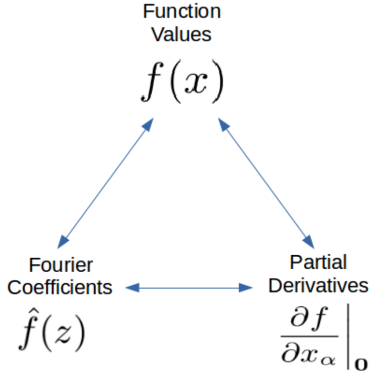

Figure 1: Three equivalent representations of f

where again $z \geq \alpha$ means $z_i \geq \alpha_i$ for all i, and $|\alpha|$ is the number of 1's in α.

2.3 Feature Interactions

We now have three different but equivalent ways of expressing the same Boolean function f, namely the function values, the Fourier coefficients, and the partial derivatives evaluated at the origin. These three representations encode the same information about f, and can be freely and easily transformed between one another, as illustrated in Figure 1.

In their corresponding software system terms, firstly, the original function values represent the performance of each configuration of the system.

Secondly, although the set of Fourier coefficients does not directly translate to a measurable quantity of software systems, each individual coefficient is a measure of disparities between the performance of particular types of configurations. Furthermore, the set of all Fourier coefficients is shown to be potentially structured and thus useful in testing and learning the function values [20].

Thirdly, the partial derivatives evaluated at the origin correspond to the degrees of feature interactions, or PRFIs particularly in the context of software performance.

DEFINITION 2.1. *Given a set of features (identified by their indices) $F = \{i$: feature i is selected$\}$, we say the vector $\alpha \in \{0,1\}^n$ where $\alpha_i = 1$ if and only if $i \in F$ is the* indicator vector *of the feature set F.*

As in our early example, the indicator vector for features one, three and four would be 1011.

DEFINITION 2.2. *Given a set of features (again, identified by their indices) $F = \{i$: feature i is selected$\}$, and α being its indicator vector, we define the* feature interaction (FI) *of F to be the partial derivative with respect to α evaluated at* **0**, *namely:*

$$FI(F) := \left.\frac{\partial f}{\partial x_\alpha}\right|_{\mathbf{0}} \tag{18}$$

Our definition of feature interactions can be understood more intuitively by a comparison between the example of (15) and the example given by Batory et al. [4], as shown below:

$$\texttt{F} \times \texttt{G} \times \texttt{H} = \texttt{F} \cdot \texttt{G} \cdot \texttt{H} \cdot \texttt{F\#G} \cdot \texttt{F\#H} \cdot \texttt{G\#H} \cdot \texttt{F\#G\#H} \tag{19}$$

In their notation, and also in the context of software performance, $\texttt{F} \times \texttt{G} \times \texttt{H}$ would be the performance of the system with the features F, G and H selected, the dot composition $\cdot$ is essentially independent addition and # denotes feature interactions. For example, F#G represents the pair-wise feature interaction between the features F and G, and similarly F#G#H represents the triple-wise interaction among the three.

As shown in the example of (15), our description of the performance of a system configuration through its Taylor expansion bears close similarities to the feature composition notation in [4]. The key difference is that, in our representation, we have explicitly carved out the abstract quantity $f(\mathbf{0})$ that can be intuitively interpreted as the "base value" or the "core measure" that the system produces with the minimal configuration, most likely when only mandatory features and no optional features are selected. If all features considered in a software system are configurable, the execution time of the system with no features selected is most likely to be 0, which then puts the two definitions of feature interactions essentially identical to each another.

Although our definition and Batory's notation [4] capture the same conceptual idea, our more mathematical representation better serves our purpose from two distinct angles. Firstly, it provides a more theoretical and rigorous insight of what PRFIs are in terms of their abstraction and a general framework, under which the feature interactions can be estimated via learning and the additional effects of single and multiple features are treated alike.

Secondly, our representation comes with two algorithms for estimating and detecting PRFIs, one through learning the function values, potentially by existing performance prediction methods via statistical machine learning such as [8], and the other through directly learning the Fourier coefficients, potentially as a byproduct of our previous work [20]. Both methods will estimate the degrees of PRFIs of *all* sets of features, namely all partial derivatives of the performance function, by measuring a small random sample of configurations.

3. THE ALGORITHMS

The two algorithms of detecting PRFIs come in similar flavors. They both start by taking a random sample of the software configurations and measure their performance on a fixed benchmark. Then, the algorithms use an oracle for estimating all function values or all Fourier coefficients respectively. Finally, they both reach the partial derivatives from their respective paths via the formulas introduced in Sections 2.1 and 2.2.

3.1 Preliminaries

In order to estimate the PRFIs of all sets of features, namely all 2^n partial derivatives, we first learn either the set of function values, or the set of Fourier coefficients, as the two routes of inferring the partial derivatives as shown in Figure 1.

The function values can be obtained from a small sample using, for instance, Classification And Regression Tree (CART) method as presented in [8]. CART gives a symbolic representation of the values as a decision tree. On the other hand, the set of Fourier coefficients can be learned, also from a random sample, via Fourier transform as shown in [20].

Of course, any algorithm that can approximate these sets of values can be readily plugged in and used as an oracle for our methods, such as the ones in [12, 14], which we will discuss in Section 5. Here, we use the two approaches mentioned above for our analysis and evaluation. The oracles are listed in Table 2.

3.2 PRFI Detection

Table 2: Oracles for PRFI Detection

Oracle 1	Obtaining function values
Input	Sample function values
Output	Estimated all 2^n function values $f(x)$

Oracle 2	Obtaining Fourier coefficients
Input	Sample function values
Output	Estimated all 2^n Fourier coefficients $\hat{f}(z)$

Algorithm 1 Feature Interaction Detection 1

1: **Input**: n: dimension of f.
2: **Output**: $\{FI(F) : F \subset \{1, 2, \ldots, n\}\}$.

3: **Initialization:**
4: Take random sample of configurations
5: Measure all **sample** configurations.

6: **Obtaining all function values:**
7: Use **Oracle 1** to obtain all function values:
8: $\{f(x) : x \in \{0, 1\}^n\}$

9: **Transformation:**
10: Use (9) and (11) to calculate all derivatives:
11: $\{FI(F) : F \subset \{1, 2, \ldots, n\}\}$

Algorithm 2 Feature Interaction Detection 2

1: **Input**: n: dimension of f.
2: **Output**: $\{FI(F) : F \subset \{1, 2, \ldots, n\}\}$.

3: **Initialization:**
4: Take random sample of configurations
5: Measure all **sample** configurations.

6: **Obtaining all Fourier coefficients:**
7: Use **Oracle 2** to obtain all Fourier coefficients:
8: $\{\hat{f}(z) : z \in \{0, 1\}^n\}$

9: **Transformation:**
10: Use (17) to calculate all derivatives:
11: $\{FI(F) : F \subset \{1, 2, \ldots, n\}\}$

With the oracles at hand, we present the PRFI detection algorithms in full to make them self-explanatory. Algorithm 1 first takes measurements of a random sample of configurations to construct an estimated performance function of the entire system using Oracle 1. Then, the performance values are transformed to the derivatives using equations (9) and (11). Similarly, Algorithm 2 obtains an estimation of the set of Fourier coefficients via Oracle 2, and transforms them to the derivatives via equation (17).

Besides being easy to understand and easy to implement, our algorithms have the following nice properties. Firstly, from a software-engineering point of view, the algorithms are *modular*, in that they are made of independent steps that can be readily substituted by other subroutines. With more developed methods for predicting function values or advanced techniques for learning Fourier coefficients, the PRFI detection algorithms can take full advantage of their speed, accuracy and efficiency. Secondly, previous performance prediction methods such as [8, 18, 16] did not have explicit guarantees on their prediction accuracies, whereas the Fourier learning algorithm [20] provides upper bounds on the error it exhibits when learning the Fourier coefficients. Therefore, based on the Fourier learning method, Algorithm 2 is the first algorithm that is able to detect PRFIs with theoretically bounded accuracies.

Table 3: Summary of Subject Software Systems

System	Domain	Lang.	Configs	n	m
Apache	Web Server	C	192	8	29
LLVM	Compiler	C++	1,024	10	62
x264	Encoder	C	1,152	13	81
Berkeley DB	Database	C	2,560	16	139
Berkeley DB	Database	Java	180	17	48

Lang. = Language of the software system.
Configs = Number of valid configurations.
n = Number of features after preprocessing.
m = Number of configurations taken for experiment.

4. EVALUATION

In this section, we present the experimental results of the two algorithms on five real-world software systems as well as evaluate our algorithms.

4.1 Subject Systems

To demonstrate our approach to PRFI detection, we ran both of our algorithms on the public datasets of five real-world software systems across different domains and written in different languages, as previously used in [8, 18, 20]. We have purposely left out one system from the original dataset, SQLite, due to its incomplete data. A summary of the systems is shown in Table 3.

We have pre-processed the datasets and eliminated all mandatory features in all systems, since they have no influence on their respective performance functions. When we hereafter refer to systems as “large” or “small”, it would be in terms of their number of features. We will also hereafter abbreviate the Berkeley DB systems to BDBC for the C version and BDBJ for the Java version respectively.

4.2 Experimental Setup

We ran both algorithms on the five subject systems. For Algorithm 1, we ran the experiment 10 times and took the average result. We used the same size m of random samples as introduced in [18] for performance prediction, an important parameter in their algorithm to make sure of a minimum feature coverage. The same size was also used in [8], so it makes our results comparable to the previous work.

For Algorithm 2, since these example systems are relatively small, the Fourier learning algorithm required a larger sample size than the total number of valid configurations [20]. Thus, we used the whole population to calculate the actual Fourier coefficients and performed the transformation to the derivatives. Given that the whole population has been used to learn the derivatives, the results from Algorithm 2 then serve as benchmarks for assessing the accuracies of Algorithm 1. Had we had access to datasets with larger numbers of features, we would have been able to evaluate the

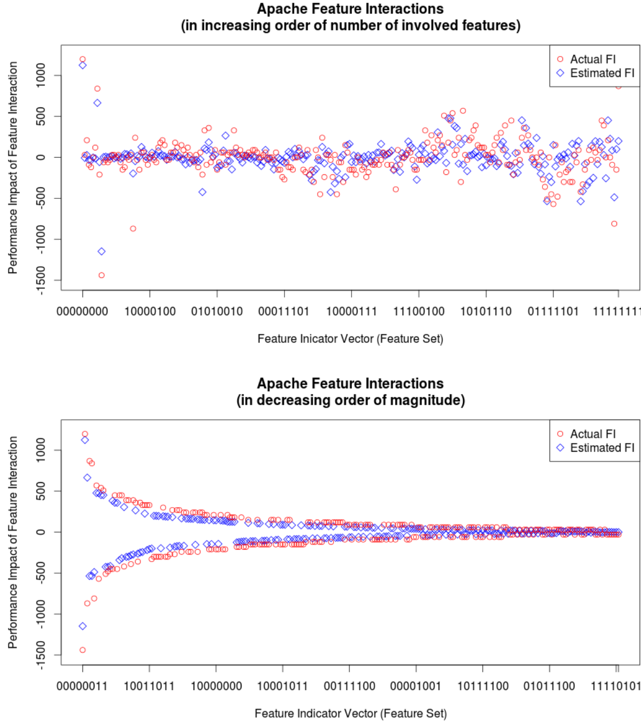

Figure 2: PRFIs of Apache

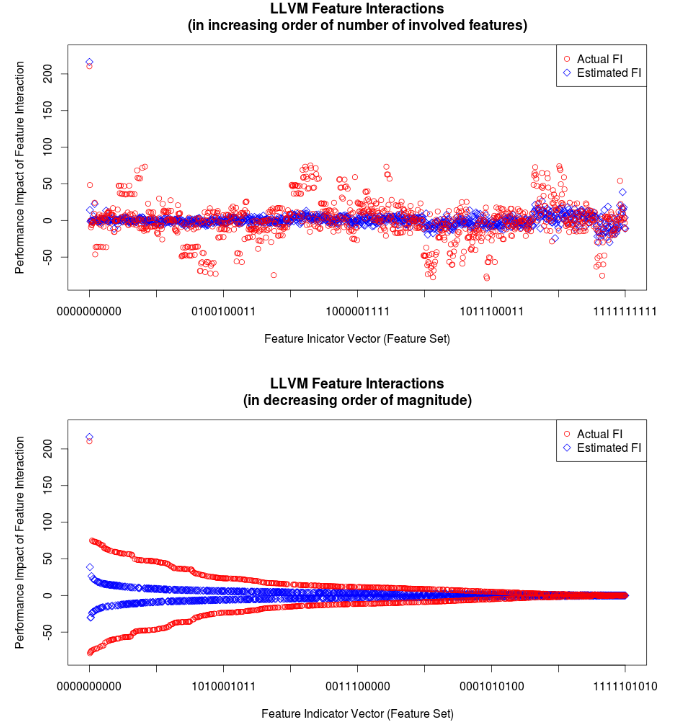

Figure 3: PRFIs of LLVM

accuracy of Algorithm 2 as well.

The algorithms were implemented in Java 7 and the experiments were carried out on a single Ubuntu 14.04 machine with Intel Core i7 CPU 2.2 GHz and 8 GB of RAM.

4.3 Detection Accuracy

We first present the detected PRFIs from all subject systems. The estimated and actual PRFIs from Algorithm 1 and 2 respectively are illustrated in Figures 2 to 6. The top graph in each figure shows all PRFIs in the corresponding software system, indexed by all sets of features and ranked in increasing order of the number of features involved, namely from none to all. The bottom graph reorders the most significant, namely the largest, PRFIs in decreasing order of their magnitude. Here, the impact or magnitude of a PRFI is the value of a partial derivative for a given feature indicator vector.

As shown in the figures, the estimated PRFIs, namely partial derivatives, are generally quite accurate, especially for larger systems like the Berkeley DB systems. From the top graph of each system, we can see that it is difficult to draw a decisive conclusion on whether small sets of features or large sets tend to interact more with each other. Most of PRFIs for x264, for instance, involve a relatively small number of features as shown towards the left side of the graph, whereas the Berkeley DB systems tend to display more of a periodic pattern.

Using the same normalized sum square error as in [20], namely if $g(\alpha)$ are the set of actual derivatives and $h(\alpha)$ are the estimated ones, the accuracy is given by:

$$accuracy = 1 - \frac{\sum_{\alpha \in \{0,1\}^n} \left(g(\alpha) - h(\alpha)\right)^2}{\sum_{\alpha \in \{0,1\}^n} g(\alpha)^2}, \qquad (20)$$

we summarized the detection accuracies in Table 4.

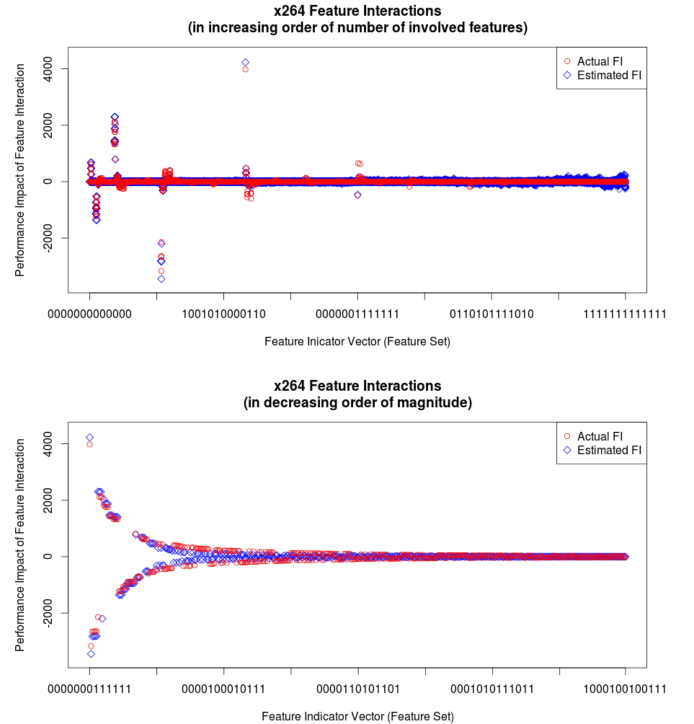

Figure 4: PRFIs of x264

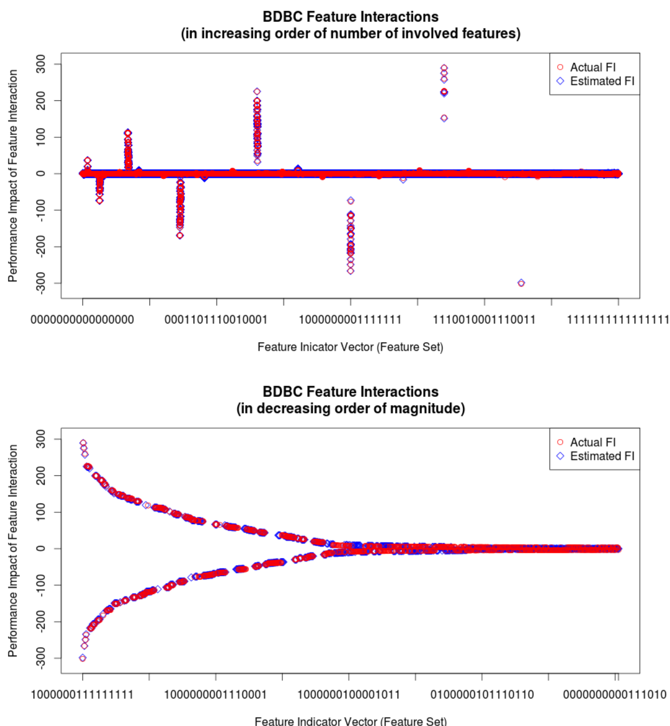

Figure 5: PRFIs of BDBC

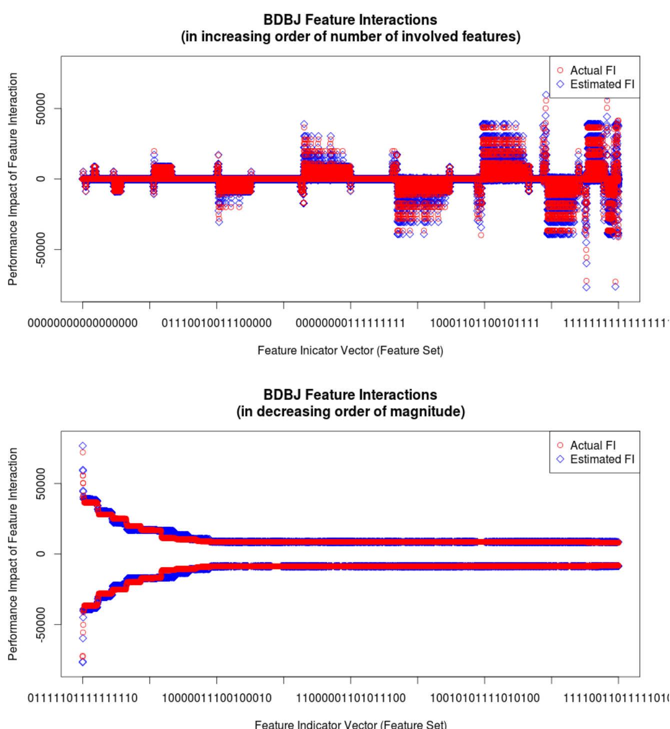

Figure 6: PRFIs of BDBJ

Table 4: PRFI Detection Accuracies

System	Detection Accuracy
Apache	33.3%
LLVM	21.3%
x264	90.9%
BDBC	99.7%
BDBJ	98.6%

Table 5: Significant PRFIs: BDBC

PRFIs	Feature Indicator Vector
−301.60	1000000111111111
290.67	1000000011111111
277.22	1000000101111111
−266.29	1000000001111111
260.49	1000000110111111
−249.55	1000000010111111
−236.11	1000000100111111
226.27	1000000111110111
226.20	1000000111111011
226.19	1000000111111110

Table 4 confirms the intuitive observations from the figures. Algorithm 1 works much better on relatively larger systems, and the two smallest systems produced the lowest accuracies, most likely due to the disproportionate sample size we have adopted from [18].

As presented in Algorithms 1 and 2, the accuracy of PRFIs we detect depends on the accuracy of the oracle we use for estimating the function values or Fourier coefficients. Tracing back to the performance prediction accuracies of CART in [8], we could see for example that it did not do particularly well on Apache, which also partially explains the low accuracy on Apache here.

4.4 Significant PRFIs

We chose two representative systems, BDBC and LLVM, one large and one relatively small, and we explicitly listed out the 10 most *significant* PRFIs and their corresponding feature sets represented by their feature indicator vectors in Tables 5 and 6 respectively. Here, we quantify the significance in terms of the absolute value of the PRFIs, i.e. the magnitude of the partial derivative for a given feature indicator vector. A positive PRFI number means extra execution time, thus a negative impact on the software performance, and vice versa.

From Tables 5 and 6, we observe that there are usually some *key* features involved in many significant interactions. For example the third and the last features in LLVM are present in all top 10 most significant interactions, so are the first, eleventh and twelfth features in BDBC. At the same time, there seem to be features that are more independent and do not interact with others intensively, for example the second through the sixth features in BDBC, and the second feature in LLVM. Furthermore, note the all 0 feature indicator vector in Table 6, representing a high base line of LLVM's execution time and relatively low impact of feature interactions comparing to the base line. In this way, our approach statistically recommends the potential features that are most or least relevant to performance.

Table 6: Significant PRFIs: LLVM

PRFIs	Feature Indicator Vector
210.44	0000000000
−78.80	1011001011
−77.53	0011100111
−76.54	1011000111
−75.29	1011011111
75.03	0011000111
−74.60	1011000001
74.07	1011001111
73.44	0011000001
−73.43	0010101111

Table 7: Detection Coverages of Software Systems

System	Detection Coverage
Apache	40%
LLVM	10%
x264	100%
BDBC	90%
BDBJ	90%

Note that not all derivatives are significant PRFIs in practice since most derivatives are negligible (close to zero). The threshold of significant PRFIs is up to the particular application context. Users can set a lower threshold and compute more derivatives to improve accuracy.

4.5 Detection Coverage

Detecting the PRFIs accurately is only one aspect of the requirement. In many real-life situations, identifying the particular feature sets where interactions are significant might be arguably more important than quantifying the interactions themselves. Table 7 summarizes the overlap between the estimated feature indicator vectors that have the top 10 most significant PRFIs from Algorithm 1 and the actual feature indicator vectors that have the 10 most significant PRFIs, namely the estimated and actual second column as in Tables 5 and 6 for each system. We refer to this quantity as the *detection coverage*. In other words, it measures whether the identified feature indicator vectors including significant PRFIs have real impact on the system's performance.

Given the relatively poor detection accuracies on the two smaller systems, Apache and LLVM, there is no surprise that the detection coverages on them are also lower. However in the Apache case, the fact that 4 of the top 10 detected feature sets are in fact significant is still quite helpful in understanding the performance impact of the features involved.

4.6 Execution Time

The transformation process itself completes within minutes even for a system as large as BDBJ, which has $131,072$ Fourier coefficients and the same number of derivatives.

For PRFI detection purposes, the execution time of our algorithms is typically not an important concern. Instead, the data gathering and measurement phase typically requires much more effort, hence making the number of required configurations for measurement a much more important factor, which again depends on the oracles we use.

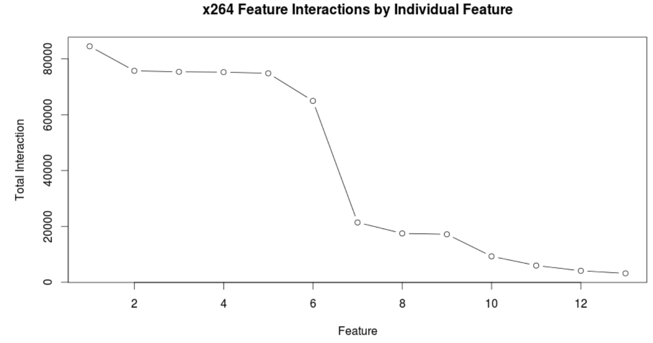

Figure 7: Sum of PRFIs involving a certain feature in x264

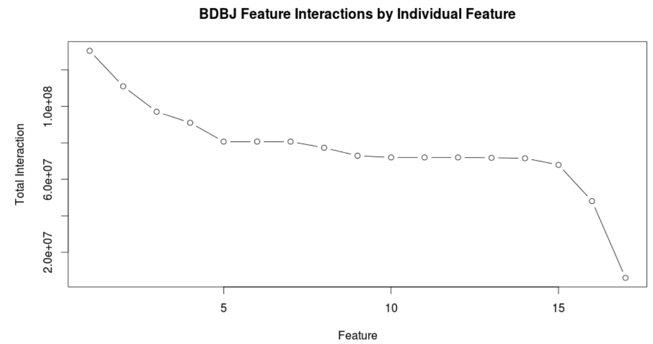

Figure 8: Sum of PRFIs involving a certain feature in BDBJ

4.7 Comparative Analysis

There has been few previous studies on PRFI detection, and the only published experimental results on the same subject systems are from Siegmund et al. [18] where a few heuristics of detecting frequent PRFIs were proposed. However, they measured the performance of a few configurations according to their heuristics for performance prediction purposes and did not systematically detect all potential feature interactions. Without access to their intermediate results, we are not able to make a direct comparison with them on the level of PRFIs. However, with our theoretical model and algorithms of calculating all PRFIs of all features, we could provide a deeper understanding of their heuristics and thus guide further use of them.

The first heuristic claims that pair-wise interactions, i.e., interactions involving two features are the most common and most significant. From the top graph of Figures 2, 3 and 5, it is evident that this is possibly *not* universal for all systems. To quantify it, the total magnitude of pair-wise interactions only account for 10.02%, 3.53%, 2.72%, 0.02%, 0.01% respectively of all PRFIs in the five systems. Thus, even though this heuristic might work well for some systems, it does not appear to be universal, especially on larger systems.

Another heuristic asserts the existence of "hot-spot features", namely a small number of features that interact with many features.

In other words, we can interpret this as the fact that a large proportion of interactions are concentrated on a small number of features. Tables 5 and 6 have given us a few candidates for such hot-spot features.

Furthermore, for each feature, we calculated the sum of the PRFIs of all feature indicator vectors containing that feature, and plotted them in decreasing order in Figures 7 and 8 for two of the systems. As shown in the figures, there *does* seem to be a “knee” in Figure 7 for x264, suggesting potential hot-spot features in the first six. However, sometimes the jump comes relatively late, showing broadly even degree of interactions for most features except only a few less active ones, as shown in Figure 8. Thus, the second heuristic may also need more careful examination.

As to a comparison between the two algorithms themselves, they are very similar in their line of approach. Given that the accuracies of the algorithms almost entirely depend on the learning oracles they use, a choice between them is very much up to the choice of their respective underlying oracles.

Algorithm 1 makes use of oracles that solve a more conventional machine learning problem, hence has a large number of choices, whereas Algorithm 2 takes advantage of particular sets of oracles that might be more efficient and potentially offer better error bounds in the particular Fourier domain.

4.8 Threats to Validity and Limitations

As mentioned in Section 1, our theoretical approach to feature interaction detection can smoothly generalize to any quantitative measure of software systems that fits the abstraction of Boolean functions. Therefore, except for the common threats to validity inherited from the used oracles, our algorithms do not suffer from obvious external threats to validity, in the sense that their correctness and effectiveness are independent of the systems that were used for our experiments and the particular conditions under which they were run.

However, our current approach still suffer from the following limitations. First of all, we have mostly considered Boolean features and ignored features of other types. Although it has been shown that discrete features can be easily transformed to Boolean features [18, 20], it is still very much an inconvenience. For example, an optional feature that can take values A, B or C can be split into three Boolean features, isA, isB and isC. This obviously increases the number of features in the system and introduces more complexity.

Furthermore, in many real-life software systems, not all possible feature combinations are valid. We still face the task of generating a random sample of valid configurations from the feature model of a software system, which may not at all be trivial. In addition, in Algorithm 2, we have implicitly taken the execution time of invalid configurations to be 0, which makes intuitive sense, but may lead to feature interaction values with large variances. This is however not a concern for Algorithm 1 if the function values are learned through CART, since CART learns a total function based on the measured sample. As a practical consequence, users may not be able to reconfigure the system as suggested by a feature interaction (e.g. given a strong negative feature interaction among three features, it may not be possible to disable one of the features because of constraints) and may need to try other interactions. For developers diagnosing undesired feature interactions, they will have to be investigating the interactions in some feasible configurations that exhibit them (rather than infeasible ones).

5. RELATED WORK

The related work of our study primarily comes from three angles: feature interaction detection, software performance prediction, and theoretical learning algorithms.

Most of the previous work on feature interaction have been focusing on their behavioral aspect [5]. On the software system and practical side, Apel et al. [2] and Liu et al. [13] both proposed white-box methods for modeling and detecting behavioral feature interactions through source code analysis. The latter also explored a different notion of derivatives that looks at software changes with respect to the change of features, which might have the potential to be mathematically formalized. Calder et al. [6] approaches behavioral feature interaction from the perspective of formal methods and logic. Apel et al. [3] studied feature interactions in the context of feature-based design and specification.

On a more abstract level, attempts to theoretically formalize feature interactions in terms of feature algebra have come a long way [9, 10, 2], and the feature composition notation in Batory et al. [4] has gained wide acceptance and recognition in recent years, whose model we compared against in Section 2.3.

Siegmund et al. [18] presented the first study to coin the notion of PRFIs following the definition of [4] and predict a configuration's performance by detecting PRFIs using sampling heuristics targeting different feature-coverage criteria. Further heuristic-based performance-influence models were also proposed in their follow-up work [17]. In contrast, we detect PFRIs using a general method without any heuristics, which can be applied to arbitrary samples. Guo et al. [8], Valov et al. [19] and Sarkar et al. [16], on the other hand, employed statistical machine learning methods to achieve performance prediction from random samples, the first of which we adopted as an oracle in our Algorithm 1.

As for the learning algorithms themselves, the learning and testing of Boolean functions have been well studied. Our mathematical model is based on the theory of Boolean functions analyzed in [15]. The Fourier learning method [20] used a trivial exponential time algorithm since the faster polynomial time algorithm from Kushilevitz and Mansour [11] unfortunately only applied to complete functions where all possible configurations are valid. Linial et al. [12] proposed a faster variation of it, which had even stricter requirements on the target function. Other similar algorithms and applications [7, 14] also exist.

6. CONCLUSION

With growing complexity in modern software systems, detecting significant PRFIs between multiple features and foreseeing potentially unexpected performance deviations of configurations is becoming increasingly important as well as challenging. In this work, we have proposed a mathematical model of viewing software systems as Boolean functions and their PRFIs as partial derivatives. This abstraction not only led to natural algorithms for detecting all PRFIs, potentially with small sample size, guaranteed accuracy and bounded confidence level, but also allowed general feature interactions with respect to any quantitative measure of software systems beyond their performance to be analyzed and detected in the same rigorous manner. By experiments, we demonstrated the effectiveness of our approach. Moreover, we provided new insights on the heuristics used for detecting PRFIs [18] and thus guide further use of them.

In future work, expanding our formulation of software feature interactions to wider forms of feature specifications, for example numerical features, would be a significant next step forward. Transforming software behavior to measurable quantities might be difficult, but we would also attempt to generalize our model to behavioral feature interactions in an appropriate way.

7. ACKNOWLEDGMENTS

This work was partially supported by the Natural Science Foundation of China (61173048, 61300041, 61375053), and a Specialized Research Fund for the Doctoral Program of Higher Education of China (20130074110015).

8. REFERENCES

[1] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebraic foundation for automatic feature-based program synthesis. *Science of Computer Programming*, 75(11):1022 – 1047, 2010.

[2] S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Detecting dependences and interactions in feature-oriented design. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 161–170, Nov 2010.

[3] S. Apel, A. von Rhein, T. Thüm, and C. Kästner. Feature-interaction detection based on feature-based specifications. *Computer Networks*, 57(12):2399 – 2409, 2013.

[4] D. Batory, P. Höfner, and J. Kim. Feature interactions, products, and composition. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, GPCE '11, pages 13–22. ACM, 2011.

[5] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115 – 141, 2003.

[6] M. Calder and A. Miller. Feature interaction detection by pairwise analysis of LTL properties – a case study. *Formal Methods in System Design*, 28(3):213–261, 2006.

[7] O. Goldreich and L. A. Levin. A hard-core predicate for all one-way functions. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 25–32, New York, NY, USA, 1989. ACM.

[8] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-aware performance prediction: A statistical learning approach. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 301–311, 2013.

[9] P. Höfner, R. Khedri, and B. Möller. Feature algebra. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 300–315. Springer Berlin Heidelberg, 2006.

[10] P. Höfner, R. Khedri, and B. Möller. An algebra of product families. *Software & Systems Modeling*, 10(2):161–182, 2011.

[11] E. Kushilevitz and Y. Mansour. Learning decision trees using the fourier spectrum. *SIAM Journal on Computing*, 22(6):1331–1348, 1993.

[12] N. Linial, Y. Mansour, and N. Nisan. Constant depth circuits, fourier transform, and learnability. *J. ACM*, 40(3):607–620, 1993.

[13] J. Liu, D. Batory, and S. Nedunuri. Modeling interactions in feature oriented software designs. In *FIW*, pages 178–197, 2005.

[14] Y. Mansour. Learning boolean functions via the Fourier transform. In *Theoretical advances in neural computation and learning*, pages 391–424. Springer, 1994.

[15] R. O'Donnell. *Analysis of Boolean Functions*. Cambridge University Press, 2014.

[16] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki. Cost-efficient sampling for performance prediction of configurable systems. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 342–352, Nov 2015.

[17] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner. Performance-influence models for highly configurable systems. In *In Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), 2015*, 2015.

[18] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmuller, and G. Saake. Predicting performance via automated feature-interaction detection. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 167–177, 2012.

[19] P. Valov, J. Guo, and K. Czarnecki. Empirical comparison of regression methods for variability-aware performance prediction. In *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, pages 186–190, 2015.

[20] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki. Performance prediction of configurable software systems by Fourier learning. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 365–373, Nov 2015.