Performance Prediction of Configurable Software Systems by Fourier Learning

Yi Zhang
University of Waterloo
y825zhan@uwaterloo.ca

Jianmei Guo
University of Waterloo
gjm@gsd.uwaterloo.ca

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

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

***Abstract*—Understanding how performance varies across a large number of variants of a configurable software system is important for helping stakeholders to choose a desirable variant. Given a software system with n optional features, measuring all its 2^n possible configurations to determine their performances is usually infeasible. Thus, various techniques have been proposed to predict software performances based on a small sample of measured configurations. We propose a novel algorithm based on Fourier transform that is able to make predictions of any configurable software system with *theoretical guarantees of accuracy and confidence level* specified by the user, while using minimum number of samples up to a constant factor. Empirical results on the case studies constructed from real-world configurable systems demonstrate the effectiveness of our algorithm.**

I. Introduction

Many software systems allow users to customize the software behavior with a finite set of configuration options, which we refer to as *features*, that users may decide to select or deselect. Each feature combination gives rise to a particular *variant* or *configuration* of the system, which in turn has a particular performance measure, such as execution time or throughput.

In order for users to decide on an optimal configuration for their particular purpose, they first need an understanding of the relationship between feature selection and performance, hence giving rise to the fundamental problem of performance prediction of configurable software systems. Take execution time as an example: since it is simply infeasible to run all 2^n configurations for a system with n features, the key challenge is to *accurately predict* the performance of the system on all configurations by measuring only a *small* number of *sample configurations*, as is being actively studied in many recent works [6], [13], [14], [15], [16], [17], [20], [4].

To address the challenge, we first formalize it as an equivalent mathematical problem. The mapping from any particular configuration of a system with n features to its performance value can be formalized as a Boolean function:

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

For example given $n = 4$, the statement $f(0101) = 3.22$s means that the execution time of the system with its second and fourth features selected is 3.22 seconds. Then for any given system with n features, the performance value of all its configurations can be solely and completely captured by a particular function f, as in (1), that maps any of its configurations $c \in \{0,1\}^n$ to a value in $\mathbb{R}$, which we will hereafter refer to as the *performance function*. Therefore the problem of predicting performance of a configurable system reduces to the problem of *learning* the performance function with a small number of its values given by the measurements, i.e., producing an estimate of $f(x)$ for every $x \in \{0,1\}^n$.

Although there are many established learning algorithms [1], [5], [8], [9], [10], [19], it is well known that arbitrary Boolean functions of this form simply cannot be learned. Since the domain of such functions is a finite set, with an arbitrary function f, knowing $f(a)$ provides practically no information about $f(b)$ unless $a = b$. That is to say, performance values cannot be predicted by merely taking samples of configurations.

Fortunately, previous empirical results [6], [13] have shown that performance functions of actual software systems are not arbitrary, but rather structured, hence can be potentially learned effectively. But what exactly the structures are, that actual performance functions exhibit, was not made clear.

In this work, we have explored one explicit notion of structure of actual performance functions of being *close* to *Fourier sparse* functions, namely functions that have many 0's in their Fourier decompositions [11]. Furthermore, we have proposed an algorithm that is able learn these Fourier sparse functions to user specified accuracy and confidence level, by taking only a small number of sample configurations, such that we in turn achieve accurate predictions of the actual underlying performance function of a configurable system.

In summary, the contributions of this work include:

- A theoretical model and formulation of configurable software performance prediction in terms of Fourier learning of Boolean functions.
- An algorithm for learning configurable software performance functions via their Fourier decomposition.
- An implementation, analysis and evaluation of the algorithm, with its strengths and relative weaknesses compared to existing performance prediction methods.

The rest of the paper is organized as follows. Section II introduces the mathematical background of Fourier transform and formalizes the problem of performance prediction.

Sections III and IV presents the algorithm itself and its implementation considerations. Sections V and VI discuss the experimental results and evaluations of the algorithm. The paper concludes after the related and future work in Section VII.

II. Problem Formalization

Having established software performance functions as Boolean functions of the form $f : \{0,1\}^n \to \mathbb{R}$, in this section we introduce the mathematical basis of Fourier transform of Boolean functions and formalize the problem of software performance prediction in terms of its mathematical equivalence.

A. Fourier Transform

Consider a function f of the form:

$$f : \{0,1\}^n \to \mathbb{R} \tag{2}$$

$$f(x) = f(x_1, x_2, \ldots, x_n) = y \in \mathbb{R} \tag{3}$$

A natural representation of f is the truth table view shown in Table I, where each x_i, a Boolean value indicating if a feature is selected, represents the ith coordinate of the input vector x, and $f(x)$ its function value.

TABLE I. Truth table view of a Boolean function f

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

We now explore an alternative view of the function f.

Given a function $f : \{0,1\}^n \to \mathbb{R}$, we can rewrite it in the following form:

$$f(x) := \sum_{z \in \{0,1\}^n} \hat{f}(z)\chi_z(x) \tag{4}$$

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

$$\chi_z(x) := \begin{cases} +1 & \text{if } \sum_{i=1}^n z_i x_i = 0 \mod 2 \\ -1 & \text{if } \sum_{i=1}^n z_i x_i = 1 \mod 2 \end{cases} \tag{5}$$

Intuitively, this Fourier transform [11] of f simply decomposes it into the sum of 2^n 'simpler' functions with coefficients indexed by all vectors $z \in \{0,1\}^n$. We say the function f is t-sparse if at most t out of 2^n of the Fourier coefficients are non-zero.

With a given dimension n, since all 2^n character functions are fixed, the 2^n Fourier coefficients then uniquely determine the function f. In other words, knowing all the Fourier coefficients is *equivalent* to knowing the function f itself, since given the Fourier coefficients, one can exactly calculate any function value $f(x)$, and perhaps less-trivially, vice versa.

With the inner product defined on functions of interest $f, g : \{0,1\}^n \to \mathbb{R}$:

$$\langle f, g \rangle := \frac{1}{2^n} \sum_{x \in \{0,1\}^n} f(x)g(x), \tag{6}$$

it can be easily checked that the set of functions

$$\{\chi_z : z \in \{0,1\}^n\}$$

forms an orthonormal basis of the entire set of functions from $\{0,1\}^n$ to $\mathbb{R}$. In other words, $\langle \chi_{z_1}, \chi_{z_2} \rangle$ evaluates to 1 if and only if $z_1 = z_2$ and 0 otherwise. Therefore we can derive the following properties:

1) For any two vectors $z_1, z_2 \in \{0,1\}^n$, we have $\chi_{z_1}(x)\chi_{z_2}(x) = \chi_{z_1 \oplus z_2}(x)$, where $\oplus$ is `xor` of the two bit vectors, or equivalently, addition modulo 2.
2) To calculate any particular Fourier coefficient $\hat{f}(z^*)$, we have:

$$\langle f, \chi_{z^*} \rangle = \left\langle \sum_{z \in \{0,1\}^n} \hat{f}(z)\chi_z, \chi_{z^*} \right\rangle \tag{7}$$

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

$$= \hat{f}(z^*) \langle \chi_{z^*}, \chi_{z^*} \rangle \tag{9}$$

$$= \hat{f}(z^*) \tag{10}$$

by orthonormality of the character functions.
3) Immediately we have Parseval's identity:

$$\langle f, f \rangle = \sum_{z \in \{0,1\}^n} \hat{f}^2(z) \tag{11}$$

4) With the inner product defined in (6) and combining with (11), we define the usual notion of L^2 norm of functions and distance between them.

$$L^2 \text{ norm:} \quad \|f\|_2 := \sqrt{\langle f, f \rangle}$$

$$\text{Distance:} \quad dist(f, g) := \|f - g\|_2^2 = \sum_{z \in \{0,1\}^n} \left(\hat{f}(z) - \hat{g}(z)\right)^2$$

From 2), it is now clear that Fourier coefficients are uniquely determined by the function f, hence we have the equivalence between a function f and its Fourier coefficients.

From 4), we define the distance between two functions f and g to be the square of the L^2 norm of their difference $f - g$, which by (11) is the sum of all the differences in their squared coefficients. Coincidentally, by (6) this distance is exactly the *mean square difference* between f and g on all inputs.

B. Problem Formalization

Recall our goal is to predict the performance of software systems based on their configurations. In this section we formalize exactly what it means.

For a software system with n features, there are overall 2^n possible configurations, each of which has a particular performance measure. Therefore it is natural to think of the

process of mapping a configuration to its performance as a Boolean function that can take any real number value:

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

where any input bit vector $x \in \{0,1\}^n$ represents a particular configuration, in which feature i is selected to be 'on' if and only if $x_i = 1$, and vice versa.

To make performance prediction is to estimate $f(x)$ given any configuration vector x. This is to say that we need to learn what the function f that corresponds to the software system produces at any given point. From the previous section, we realize that *learning a function f is equivalent to learning all of its Fourier coefficients*.

In particular, as we will discuss in Section III, functions constructed from real software systems are typically close to Fourier sparse, in which case, predicting its performances becomes equivalent to estimating only the *large* Fourier coefficients of its corresponding function f. This is exactly what our algorithm does.

The problem of learning a performance function can thus be formalized as follows.

Fix a target function f, the learning algorithm takes two inputs γ and δ, and outputs an estimate h of f such that:

$$dist(f,h) \leq \gamma$$

with probability $1 - \delta$.

III. The Fourier Learning Algorithm

A. Overview

Given a target function f, a high-level summary of the Fourier learning algorithm can be described as follows:

1) Take a random sample of configurations and their performance values of the system.
2) Use the sample to learn the Fourier coefficients of f, hence reconstructing an estimate h of f.
3) Estimate $dist(f,h)$, if it is larger than γ, increase the sample size and repeat.

With a clear definition of the problem at hand and the tool of Fourier transform introduced in the previous section, we are now ready to discuss each step of the learning algorithm in more detail.

B. Normalization

In order to aid error analysis of the algorithm, we normalize the original function:

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

to

$$f' : \{0,1\}^n \to [-1,1].$$

such that new function f' has a finite range.

In practice, it is often feasible to establish, perhaps with some domain knowledge, the upper and lower bounds on the performance values of a particular system, e.g. between 0 and max. Then f can be normalized by subtracting $max/2$ from each $f(x)$ and then dividing the result by $max/2$.

From this point on, we will then use f as the function after normalization.

C. Learning Fourier Coefficients

In this section, we describe how Fourier coefficients of a function $f : \{0,1\}^n \to [-1,1]$ can be learned.

From (10), the Fourier coefficient corresponding to each vector $z \in \{0,1\}^n$ can be calculated as:

$$\hat{f}(z) = \langle f, \chi_z \rangle \tag{12}$$

$$= \frac{1}{2^n} \sum_{x \in \{0,1\}^n} f(x)\chi_z(x) \tag{13}$$

This is essentially an *average* of the function values over the $\{0,1\}^n$ domain weighted by the particular character function corresponding to the Fourier coefficient that we are trying to estimated.

Now when a sample S is taken such that $f(x)$ is known for all $x \in S$, the same formula can be used to estimate all the Fourier coefficients. For each $z \in \{0,1\}^n$, $\hat{f}(z)$ can be estimated as:

$$\hat{f}(z) = \langle f, \chi_z \rangle \tag{14}$$

$$\approx \frac{1}{|S|} \sum_{x \in S} f(x)\chi_z(x) \tag{15}$$

Obviously, the more samples we take, i.e., the closer our sample set S is to the entire domain, the more accurate our estimations of $\hat{f}(z)$ will be. The Hoeffding's inequality formalizes this result.

Theorem III.1 (Hoeffding's inequality)**.** *[7] Let $X_1, X_2, \ldots, X_m$ be independent and identically distributed random variables with range $[a,b]$. Let the theoretical expected value of X_i be:*

$$E(X_i) = p \quad \forall i = 1 \ldots m$$

and let the sum random variable be:

$$Y := \sum_{i=1}^{m} X_i$$

Then we have:

$$Pr[|\frac{Y}{m} - p| \geq \epsilon] \leq 2\exp\left[-\frac{2m\epsilon^2}{(b-a)^2}\right] \tag{16}$$

Intuitively, the theorem states that the probability of the sample average, i.e. Y/m drifting distance ϵ apart from the theoretical average p is exponentially small in the number of samples m and the distance squared ϵ^2.

To apply the theorem to our estimations of Fourier coefficients, we realize the following:

1) For each Fourier coefficient $\hat{f}(z)$, the random variables are $X_i = f(x_i)\chi_z(x_i)$ and the expectation for each X_i is the actual value of the coefficient, therefore we have:
$$E(X_i) = \hat{f}(z)$$
according to (13).
2) We normalized the function values to be all between -1 and 1, therefore by Parseval's identity (11), we have all Fourier coefficients in the range $[-1, 1]$.
3) We demand *each* estimations of our Fourier coefficients to be accurate within ϵ with probability $1 - \delta/2^n$, such that the probability of *all* Fourier coefficients are accurate within ϵ will be at least:
$$1 - 2^n \cdot \frac{\delta}{2^n} = 1 - \delta$$

Then we can rewrite the inequality as:

$$Pr[|\hat{f}'(z) - \hat{f}(z)| > \epsilon] \leq 2\exp\left[-\frac{2m\epsilon^2}{4}\right] = \frac{\delta}{2^n} \tag{17}$$

where $\hat{f}'(z)$ is the approximation of $\hat{f}(z)$.

Rearrange and solve for m, we have:

$$m = \frac{2}{\epsilon^2}((\log 2)(n+1) + \log(1/\delta)) \tag{18}$$

To summarize, the relationship between the number of samples and the accuracy is as follows:

Theorem III.2. *Given any function $f : \{0,1\}^n \to [-1,1]$, with $1-\delta$ probability, all 2^n Fourier coefficients of f can be learned with at most ϵ additive error, using*

$$\frac{2}{\epsilon^2}((\log 2)(n+1) + \log(1/\delta)) \tag{19}$$

number of samples.

D. Error Analysis

In this section, we describe how the algorithm guarantees the accuracy of its estimate h, by constructing a series of estimates of f and bounding their distances from f.

Step 1: Given a function f, let g be the function obtained from f by only keeping its t largest Fourier coefficients and set the rest to 0, such that $dist(f, g) \leq d$. Then by construction, g is t-sparse.

Step 2: Let h_1 be the function obtained from g by replacing all its Fourier coefficients smaller than d/t by 0, namely:

$$h_1(x) = \sum_{z:\hat{g}(z) \geq d/t} \hat{g}(z)\chi_z(x)$$

Notice h_1 is also at most t-sparse.

Theorem III.3. *Let f, g and h_1 be defined as above, then [8]:*

$$dist(f, h_1) \leq d + d^2/t$$

Step 3: Let h_2 be an estimate of h_1 with each of its non-zero coefficients accurate to $d/4t$, i.e.

$$\forall \hat{h}_1(z) > 0 \quad \left|\hat{h}_1(z) - \hat{h}_2(z)\right| \leq \frac{d}{4t}$$

Then obviously:

$$dist(h_1, h_2) \leq \sum_{z \in \{0,1\}^n} \left(\hat{h}_1(z) - \hat{h}_2(z)\right)^2 \tag{20}$$

$$\leq t\left(\frac{d}{4t}\right)^2 = \frac{d^2}{16t} \tag{21}$$

Step 4: Let h be our estimation of f, obtained by learning each of f's Fourier coefficients to accuracy $d/4t$ with probability $1 - \delta$, as described in Section III-C. If the estimation of any coefficient is less than $3d/4t$, we set it to 0. Notice according to Theorem III.2, this can be done with:

$$\frac{32t^2}{d^2}((\log 2)(n+1) + \log(1/\delta)) \tag{22}$$

number of samples.

In this case, for any z such that the original Fourier coefficient $\hat{f}(z) \geq d/t$, we have an estimate of it $\hat{h}(z)$, accurate to $d/4t$. On the other hand, for all z such that $\hat{f}(z) < d/2t$, $\hat{h}(z) = 0$. Then we realize that $dist(h, f) \leq dist(h_2, f)$, since $\hat{h}_2(z)$ is set to 0 whenever $\hat{f}(z) < d/t$, which is a coarser cut.

Now we can establish:

$$\begin{aligned} dist(h, f) &\leq dist(h_2, f) = \|h_2 - f\|_2^2 \\ &\leq (\|h_2 - h_1\|_2 + \|h_1 - f\|_2)^2 \\ &= dist(h_2, h_1) + dist(h_1, f) \\ &+ 2\|h_2 - h_1\|_2 \cdot \|h_2 - h_1\|_2 \\ &\leq \frac{d^2}{16t} + d + \frac{d^2}{t} + 2 \cdot \frac{d}{4\sqrt{t}} \cdot \sqrt{d + \frac{d^2}{t}} \\ &= d + O(d^2/t) \end{aligned}$$

Since d, as distance between f and g, is bounded, as t gets large, $dist(h, f)$ is bounded above by $O(d)$. In this case, we have $dist(h, f) \leq 4d/3$ as soon as $t > 30$, which is a very small sparsity number.

To summarize this step, we realize that as long as there exists a t-sparse function g that is within d-close to f, we could take the number of samples specified in (22) to construct an estimate h such that $dist(h, f) \leq 4d/3$.

Step 5: Now we would like to estimate $dist(h, f)$ by drawing a sample S from f and calculate:

$$\frac{1}{|S|}\sum_{x \in S}(f(x) - h(x))^2 \tag{23}$$

Again by Theorem III.1, using

$$\frac{9}{2d^2}\log(2/\delta)$$

samples is sufficient to estimate $dist(h, f)$ within $d/3$ accuracy with probability $1 - \delta$.

If $dist(f,h) \geq 4d/3$, our estimate will be larger than d. Therefore we are confident that the original function f is *not* sufficiently close to any t-sparse function. In this case, we need to increase t and draw more samples to guarantee $dist(f,h) \leq \gamma$.

We have found that typical performance functions of software systems are sufficiently close to t-sparse functions with small t, hence having a relatively small sample size. However the algorithm works with functions with generic sparsity, since it is determined implicitly within the algorithm.

To conclude, the learning algorithm is written out in full in **Algorithm 1**. It applies to any generic function f, and with only a linear sample dependency on the dimension n, it outputs an estimation h such that $dist(h,f) \leq \gamma$ with probability $1-\delta$.

Algorithm 1 The Fourier Learning Algorithm

1: **Input**: n: dimension of f.
2: γ: target estimate error.
3: δ: confidence level parameter.
4: t_0: starting sparsity (Optional).
5: inc: multiplicative increment of t (Optional).
6: **Output**: h as an estimate of f.

7: **Initialization:**
8: $t := t_0$ or $1; d := \frac{3}{4} \cdot \gamma$
9: $\delta_1 := \delta_2 := 1 - \sqrt{1-\delta}$
10: $m_1 := \frac{32t^2}{d^2}((\log 2)(n+1) + \log(1/\delta_1))$
11: $m_2 := \frac{9}{2d^2}\log(2/\delta_2)$
12: **Learn:**
13: **if** $m_1 + m_2 > 2^n$ **then**
14: f is computed exactly.
15: **else**
16: Draw m_1 random samples to estimate all 2^n Fourier coefficients of f using (15). These are the coefficients of the estimation h.
17: For each $\hat{h}(z) < 3d/4t$, let $\hat{h}(z) = 0$.
18: Draw m_2 more samples and calculate $d' \approx dist(f,h)$ using (23).
19: **end if**
20:
21: **if** $d' \leq d$ **then**
22: **return** h
23: **else**
24: $t := inc \cdot t$ or $t := 2 \cdot t$
25: **go to line** 10.
26: **end if**

IV. Implementation

To evaluate the algorithm, we have implemented it in Java and run it against data constructed from real-world software systems across different domains. The experimental details will be presented in Section V, here we discuss a few implementation considerations of the algorithm.

A. Feature Selection

Before the algorithm is run, we do a preliminary feature selection. In a software system, if a particular set of features *must* be on or *must* be off, then they have effectively no discriminant power with respect to the software performances across configurations, therefore they are simply ignored, and the number of features can be thus reduced.

B. Partially Defined Functions

So far the algorithm we have described has only dealt with functions defined on the entire domain of $\{0,1\}^n$, i.e., software systems that every possible configuration is valid and has a performance value.

However this is often not the case in reality. Actual software systems often have features interacting with each other that may result in some combinations of them being invalid. A simple example could be that turning the `debug` mode on will force the `log to file` option being on as well. Therefore any configuration of the software with `debug` being 1 and `log to file` being 0 would *not* be valid.

Our learning algorithm deals with this neatly.

Let $D \subset \{0,1\}^n$ the set of bit vectors representing all *valid* configurations. Notice the size of D satisfies $|D| \leq 2^n$.

In this case, since the uniformly random samples are taken from the set D instead of the entire set of $\{0,1\}^n$, all estimated Fourier coefficients simply needs to be rescaled by a factor of $|D|/2^n$. Alternatively, scaling the estimated function values constructed from the Fourier coefficients by the same factor yields the same results.

In practice, valid configurations are often defined in terms of Boolean constraints. Therefore obtaining the number of valid configurations may require solving non-trivial #SAT instances expressed in the feature model, namely counting the number of satisfiable assignments of a Boolean satisfiability (SAT) problem, which is outside the scope of discussion of this work. Empirical results from both exact solver [18] and approximate solver [3] have suggested that this is usually feasible up to a relatively large number of features, so it is not a primary concern here.

V. Evaluation

A. Subject Systems

For evaluation of our algorithm, we used the public data sets of five software systems used in [6] and [13]. They are software systems in different domains and written in different languages. A brief summary of the subject software systems is shown in Table II:

As briefly mentioned in Section III, the number of samples the algorithm requires is $O(n)$ for a given set of accuracy parameters. Although this dependency on n is very good for large systems, for the small data sets we have, the required number of samples is more than the entire valid domain for any reasonable accuracy parameters due to the constant hidden in the $O(n)$ notation.

TABLE II. SUMMARY OF ORIGINAL SOFTWARE SYSTEMS

System	Domain	Lang.	$\|D\|$	n
Apache	Web Server	C	192	8
x264	Encoder	C	1,152	13
LLVM	Compiler	C++	1,024	10
Berkeley DB	Database	C	2,560	16
Berkeley DB	Database	Java	180	17

Lang. = Language of the software system.
$|D|$ = Number of valid configurations.
n = Number of features after trivial selection.

To still use the data sets for meaningful empirical evaluations of the algorithm, we constructed four hybrid-systems by combining some of the original systems together in a natural way.

Let $f : \{0,1\}^m \to \mathbb{R}$ and $g : \{0,1\}^n \to \mathbb{R}$ be the performance functions of two systems, we simply construct:

$$f \oplus g : \{0,1\}^{m+n} \to \mathbb{R} \tag{24}$$

$$f \oplus g(x) := f \oplus g(x_1, \ldots, x_{m+n}) \tag{25}$$

$$= f(x_1, x_2, \ldots, x_m) + g(x_{m+1}, \ldots, x_{m+n}) \tag{26}$$

to be the function of total performance of the combined configurations in f, and g.

And the hybrid systems, corresponding to the performance of running sequentially each of the two involved systems once, assuming no interactions between them are, summarized in Table III.

TABLE III. SUMMARY OF CONSTRUCTED HYBRID-SYSTEMS

System	Component	$\|D\|$	n
A	Apache + x264	221184	21
B	LLVM + x264	1179648	23
C	x264 + x264	1327104	26
D	LLVM + LLVM	1048576	20

B. Experimental Setup

Recall the algorithm essentially establishes a relationship between the number of samples it takes and its prediction accuracy and confidence level.

For our experiments, we will fix the confidence level at 80%, i.e. $\delta = 0.2$ and vary the desired accuracy parameters γ on each system to attempt to verify the theoretical error bounds of the algorithm. Our initial sparsity parameter was set to be $t_0 = 1$ and the multiplicative increase between iterations was set to be a conservative 1.2. Each experiment will be repeated 10 times and the results are presented in Section V-C.

The experiments are run on a single Ubuntu 14.04 machine with Intel Core i7 CPU 2.2 GHz and 8 GB of RAM.

C. Experimental Results

Here we present the results of our experiments on the data sets described in Section V-A.

A high-level summary of the experimental results in terms of the desired error, the actual error and the number of samples taken for each system is presented in Table IV.

TABLE IV. SUMMARY OF EXPERIMENTAL RESULTS

System	n	γ	Samples	mean	max
A	21	0.2	24236 (11%)	0.083	0.084
		0.15	43307 (20%)	0.081	0.081
		0.1	97440 (44%)	0.074	0.074
B	23	0.2	26332 (2.2%)	0.035	0.036
		0.15	46812 (4.0%)	0.026	0.026
		0.1	105326 (8.9%)	0.0068	0.0069
C	26	0.2	29289 (2.2%)	0.084	0.085
		0.15	52070 (3.9%)	0.084	0.084
		0.1	117156 (8.8%)	0.080	0.082
D	20	0.2	23375 (2.2%)	0.074	0.080
		0.15	41555 (4.0%)	0.034	0.037
		0.1	93497 (8.9%)	0.024	0.024

n = Number of features of system.
γ = User specified maximum error.
Samples = Number & proportion of samples used.
mean = Average actual error from the 10 runs.
max = Maximum actual error from the 10 runs.

As shown, the actual prediction errors of the algorithm has fallen within the specified bound on all instances. The small difference between the maximum error over the 10 runs and the mean error on all systems also suggests a very small variance that the algorithm produces on a given system, hence promoting its stability and reliability.

On the other hand, the prediction errors across different systems may vary widely, even with the same specified level of accuracy, principally due to the different intrinsic structures of systems. For example system B has an average error of 0.0068 when the user desires an error of 0.1, whereas that of system C is only 0.080.

Furthermore, although the maximum error for all these systems have fallen within 0.1 even when γ is set to be 0.2, the algorithm still decides to increase the number of samples when γ is decreased. This is because the number of samples is derived primarily from γ to guarantee the error and confidence interval for *all* systems of given size. Again, better-than-guaranteed errors on these systems show their further structure, but the algorithm is designed to work with full generality.

Notice the number of samples the algorithm used across different systems and accuracy parameters: it is evident that the sample complexity of the algorithm is $O(1/\gamma^2)$ and only linear in n. Therefore, in much larger systems, the sample complexity of the algorithm would have great benefits in making performance predictions within bounded errors.

Systems B, C and D are all consistent in achieving the desired accuracies with less than 9% of all samples. However for system A, the number of *valid* configurations is a smaller subset of the entire domain, hence giving rise to the number of required samples occupying a larger proportion of the space of all valid configurations.

VI. Discussion

A. Performance Measure

As briefly mentioned in Section II-A, we have used the notion of distance between two functions defined as:

$$dist(f,g) := \|f-g\|_2^2 \tag{27}$$

$$:= \frac{1}{2^n}\sum_{x\in\{0,1\}^n}\left(f(x)-g(x)\right)^2 \tag{28}$$

to be the error of our learning algorithm throughout.

Notice this definition coincides exactly with the standard *mean square error* as a common performance measure for machine learning algorithms.

Some previous studies [6], [13] have adopted the average relative error defined as:

$$\frac{1}{2^n}\sum_{x\in\{0,1\}^n}\frac{f(x)-g(x)}{f(x)} \tag{29}$$

as the performance measure, which is known to produce disproportionately large value of errors when $f(x)$ is close to 0. Since our algorithm normalized all function values to be around 0, using this performance metric here clearly does not make much sense.

B. Comparative Analysis

There has been two studies by Guo et al. [6] and Siegmund et al. [13] on software performance prediction that have used the same data sets for evaluating their algorithms. We give a brief comparative analysis between our algorithm and the previous ones in this section.

1) Algorithms: SPLConqueror [13] is a deterministic sampling method that focuses on predicting software performances by measuring specific samples according to some heuristics and understanding the interactions between different features, in particular pairwise feature interactions.

CART [6] on the other hand utilizes statistical methods and random sampling techniques in order to categorize unknown configurations into groups of known configurations according to their containment of certain features and hence predict their performance.

Both these methods employ a 'top-down' work flow where a number of samples are taken from the system, then the performance values of each configuration are predicted before they are compared to actual values for their accuracy.

The Fourier learning algorithm, on the other hand, attempts to determine the number of samples necessary for any desired accuracy in an adaptive, progressive sampling manner, which may be especially beneficial when the costs of sampling or measurements are high and prediction accuracy critical.

Given the different natures and work flows of the algorithms, it is not immediately obvious how they could be directly comparable to each other, hence the rest of the section will only attempt to give a qualitative account of several aspects of their relative strengths and weaknesses.

2) Sample Complexity & Requirements: SPLConqueror requires at least $\Omega(n^2)$ designated samples just to cover the feature-wise and pairwise measurements, where n again is the number of features.

CART does not have explicit sample complexity constraints, therefore [6] employs a progressive sampling scheme to double the number of random samples until the prediction accuarcy becomes satisfactory.

With an implicit progressive random sampling scheme similar to CART used in [6], but with the theoretical underpinnings of Fourier learning – in contrast to CART – we can explicitly bound our sample complexity to $O(n, 1/\gamma^2, \log(1/\delta))$ to be $1-\delta$ confident in achieving an error within γ.

3) Parameter Tuning: Although all three algorithms seem to be fully automated, CART seems to have the largest number of parameters to consider such that the algorithm does not over-fit the sample. The Fourier learning algorithm has two optional parameters t_0 and inc to control the initial and incremental number of samples respectively. Smaller t_0 and an inc closer to 1 represents more conservative approaches, which is more suitable if measurements are costly. SPLConqueror on the other hand relies more on heuristics rather than explicit parameters.

4) Execution Time: For performance prediction purposes, execution time of a learning algorithm is typically not an important concern. Since the process of gathering the data of software performances is typically much more costly in terms of time and effort, the minimum number of necessary measurements, namely the sample complexity therefore is of greater importance here.

SPLConqueror and CART are both fairly fast. The Fourier learning algorithm, having to approximate all 2^n Fourier coefficients, potentially more than once, can be relatively slow.

However some basic knowledge of the specific data sets may allow us to infer properties of their Fourier spectrum and hence dramatically speed up its execution time. For instance for the four hybrid systems we constructed, we know they consist of two independent components with no interactions between them. This saves the algorithm the effort of estimating many coefficients that we know are 0.

Furthermore, we used the most naive method for estimating 'large' coefficients in the algorithm for conceptual simplicity. More sophisticated and much faster methods, such as [8], [9] do exist and can be readily substituted as a subroutine into our algorithm with minimal alteration.

5) Summary: All three proposed algorithms have their distinct features and characteristics. For larger systems with many features and a relatively full set of valid configurations where measurement costs are potentially high and accuracy critical, the Fourier learning algorithm is a suitable choice. On the other hand, for smaller systems (e.g. $n < 20$), or systems with known low degree feature interactions, then CART and SPLConqueror respectively might be more appropriate in

terms their trade-off between accuracy and measurement efforts.

The main characteristics of the methods are summarized in Table V.

TABLE V. MAIN CHARACTERISTICS OF THE LEARNING ALGORITHMS

	SPLCONQUEROR	CART	Fourier
Accuracy	$\sim 95\%$	$\sim 94\%$	Any
Sample Size	$O(n^2)$	Any	$O(n, 1/\gamma^2)$
Sampling	Specific	Random	Random
Error Control	No	No	Yes

C. Threats to Validity

Potential threats to the validity of our results primarily comes from two angles: the model and the datasets.

Our model for configurable software performance prediction is based on a fundamental abstraction of a performance function $f : \{0,1\}^n \to \mathbb{R}$ for any given software system, which maps a (valid) configuration to a real number, representing the performance value of our interest. Although this abstraction is obviously valid for simple performance measures such as software execution time with fixed workload, it might require adjustments for more sophisticated software feature structures or performance objectives.

For systems with non-Boolean features, our approach can not be used directly. However Siegmund et al. [13] has shown that non-Boolean features can be easily expressed in terms of Boolean features in higher dimensions. For example a feature having options A, B and C could be expanded to 3 features, namely 'isA?', 'isB?' and 'isC?'. Then our approach can be readily employed.

Similarly, with multiple performance objectives, the performance function can be viewed as a multi-dimensional function $f : \{0,1\}^n \to \mathbb{R}^m$, where m is the number of performance measures. Then each performance measure can be dealt with simultaneously and separately.

Since our approach is a black box method that operates on a high level of abstraction, more software specific concerns such as varying workload and multi-user scenarios might pose more unexpected threats to our model. However one might be able to see how these variations can be feasibly incorporated into the modelling of features or performance objectives via some transformation such as the ones outlined above, and hence assume these threats are minimal.

As for the datasets, we used the openly available software system measurements originally generated by Siegmund et al. [13] for our experiments as off-the-shelf datasets. We are aware that using the combined hybrid systems as constructed in Section V-A may lose some representativeness of the systems, however the theoretical analysis of our algorithm should provide sufficient confidence that it is applicable to any general function that fits the abstraction regardless of its system structures, and the experiments should merely serve as empirical evidence confirming the correctness of the analysis.

VII. RELATED AND FUTURE WORK

One of the first established techniques for solving the problem, as we have described in Section VI-B1 was SPLCONQUEROR due to Siegmund et al. [13], which employs various heuristics and considers particular feature combinations and interactions to achieve performance prediction.

Guo et al. [6], Thereska et al. [17], Westermann et al. [20] and Sarkar et al. [12] have all utilized statistical sampling and machine learning approaches, in particular CART, for configurable software performance prediction.

There are two related Fourier learning algorithms that can be viewed as more advanced versions of our algorithm. The first one due to [8] uses better searching techniques, hence reducing the time complexity of estimating the large Fourier coefficients to only polynomial time in the number of features n. However it essentially requires the function to be defined on the entire $\{0,1\}^n$ domain. How it could be modified to be able to deal with partially defined functions is not immediately clear, hence can be explored in future work.

Another direction of adjustment of our algorithm is proposed by [9], where not only the magnitudes of estimated Fourier coefficients are restricted, but also their indices, giving even tighter probabilities of potential over-fitting. However it is not clear that some of the assumptions it makes fit well in the domain of software systems. For example it assumes that the indices of large Fourier coefficients are mostly of small weight, which, in software terms, means that most significant feature interactions only involve a small number of features. This method therefore would completely ignore potential feature interactions involving more than a certain number of features. Future work should investigate whether this assumption would hold in practice.

One further future work is related to Batory's [2] proposal of quantifying feature interactions in software systems. We believe that Batory's [2] notion of feature interaction can be formalized as derivatives of Boolean functions, therefore feature interaction detection can be reduced to estimating derivatives of Boolean functions, which is in turn closely related to estimating their Fourier coefficients [11]. Therefore an interesting future work would involve using the tools we proposed so far in estimating Fourier coefficients and translating them into a formal model of detecting feature interactions of configurable software systems with minimal extra cost.

VIII. CONCLUSION

Software performance prediction is a fundamental problem in software engineering that deserves much attention.

In this work, we formalized the model of configurable software performance prediction in terms of learning Boolean functions, explicitly explored the Fourier sparsity property of performance functions of real software systems, and proposed and implemented the Fourier learning algorithm that is able to make performance predictions with guaranteed accuracy and confidence level.

To conclude, we have introduced a new perspective on treating the problem of predicting software performance. With increasingly large software systems, desires of guarantees on prediction accuracy and more precise understanding of measurement efforts, more formal approaches such as this one may prove beneficial in the long term.

REFERENCES

[1] J. Anderson, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell. *Machine learning: An artificial intelligence approach*, volume 2. Morgan Kaufmann, 1986.

[2] 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.

[3] S. Chakraborty, K. S. Meel, and M. Y. Vardi. A scalable approximate model counter. *CoRR*, abs/1306.5726, 2013.

[4] S. Chen, Y. Liu, I. Gorton, and A. Liu. Performance prediction of component-based applications. *Journal of Systems and Software*, 74(1):35 – 43, 2005.

[5] 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.

[6] 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.

[7] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

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

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

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

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

[12] 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 IEEE/ACM 30th International Conference on, to appear*, 2015.

[13] N. Siegmund, S. S. Kolesnikov, C. Kastner, 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.

[14] N. Siegmund, M. Rosenmuller, C. Kastner, P.G. Giarrusso, S. Apel, and S.S. Kolesnikov. Scalable prediction of non-functional properties in software product lines. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 160–169, Aug 2011.

[15] J. Sincero, W. Schroder-Preikschat, and O. Spinczyk. Approaching non-functional properties of software product lines: Learning from products. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 147–155, Nov 2010.

[16] R. Tawhid and D.C. Petriu. Automatic derivation of a product performance model from a software product line model. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 80–89, Aug 2011.

[17] E. Thereska, B. Doebel, A. X. Zheng, and P. Nobel. Practical performance models for complex, popular applications. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '10, pages 1–12, New York, NY, USA, 2010. ACM.

[18] M. Thurley. sharpsat counting models with advanced component caching and implicit bcp. In Armin Biere and CarlaP. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, volume 4121 of *Lecture Notes in Computer Science*, pages 424–429. Springer Berlin Heidelberg, 2006.

[19] L. Wasserman. *All of statistics*. Springer Science & Business Media, 2011.

[20] D. Westermann, J. Happe, R. Krebs, and R. Farahbod. Automated inference of goal-oriented performance prediction functions. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 190–199, New York, NY, USA, 2012. ACM.