ABSTRACT

NAIR, VIVEK. Frugal Ways to Find Good Configurations. (Under the direction of Dr. Timothy J. Menzies).

Most software systems available today are configurable, which gives the users an option to customize the system to achieve different functional or non-functional (better performance) properties. As systems evolve, more configuration options are added to the software system, which leaves considerable optimization potential untapped and induces major economic cost. To solve this problem of finding the (near) optimal configurations, engineers have proposed various techniques. Most popular among them are model-based techniques, where accurate models of the configuration space are created using as few configuration measurements as possible. Previously, Guo et al. and Sarkar et al. asserted that one way to find good configurations is build very accurate model of the configuration space. However, we notice two major problems with the model-based techniques: 1) previous techniques are expensive to be practically viable, and 2) there are software systems whose configuration spaces cannot be accurately modeled. This dissertation focuses on proposing techniques which are easier to understand and are practically viable.

First, we present **WHAT** that exploits some lower dimensional knowledge to build performance models. WHAT's innovation is the use of the spectrum (eigenvalues) of the distance matrix between the configurations of a configurable software system, to perform dimensionality reduction. Within that reduced configuration space, many closely associated configurations can be studied by executing only a few sample configurations. For the subject systems studied here, a few dozen samples yield accurate and stable predictors less than 10 % prediction error, with a standard deviation of less than 2%. When compared to the state of the art, WHAT (a) requires 2 to 10 times fewer samples to achieve similar prediction accuracies, and (b) its predictions are more stable (i.e., have lower standard deviation).

While useful in the test domain, we found a significant drawback when exploring newer software systems. We found that the distance function used to generate the distance matrix changes with different software systems (since some configuration options have more influence on the performance than others). To overcome this,a rank-based method is proposed which shows how an accurate model is not required for performance optimization, but a rank-preserving model is sufficient. We evaluate rank-based method with 21 scenarios based on nine software systems and demonstrate that our approach is beneficial in 16 scenarios; for the remaining five scenarios, an accurate model can be built by using very few samples anyway, without the need for a rank-based approach. Additionally, in 8/21 of the scenarios, the number of measurements required by the rank-based method is an order of magnitude smaller than

methods used in prior work.

To further reduce the cost we present **FLASH**, a sequential model-based method, which sequentially explores the configuration space by reflecting on the configurations evaluated so far to determine the next best configuration to explore. FLASH scales up to software systems that defeat the prior state of the art model-based methods in this area and can solve both single-objective and multi-objective optimization problems. We evaluate FLASH using 30 scenarios based on 7 software systems to demonstrate that FLASH saves effort in 100% and 80% of cases in single-objective and multi-objective problems respectively by up to several orders of magnitude compared to the state of the art techniques.

Based on the above findings, we found that the previously held belief: " highly accurate model is required to optimize software system" is an overkill and expensive. To overcome the limitation of prior works, we have explored various alternate methods to shows how a near optimal solution can be found using much fewer resources and effort. Additionally, we observed that the prior work transformed the problem of finding an optimal solution to a modelling problem—which, in our opinion, was the main roadblock to build cheaper solutions. The central insight for all the frugal alternatives (as presented in this thesis) is to solve the optimization problem and not a modelling problem. Given the success of these frugal alternatives to find good configurations, we encourage practitioners to embrace and explore similar techniques prescribed in this thesis to (a) find better configurations and (b) not to be deterred with the cost of optimizing systems. Please note, this techniques presented in this thesis are very general in nature and could potentially be used in other domains (such as Cloud Computing, Software Engineering, etc.), where the cost of collecting samples is exorbitantly high. FLASH, for instance, has already been applied to domains like Cloud Architectural Tuning, Effort Estimation and software optimization problems. We recommend that future work explores options which consumes fewer resource, which is more suitable for real world scenario. In our opinion , some suggestions are: (a) explore cheaper models, and (b) use the models to guide explorations.

Frugal Ways to Find Good Configurations

by
Vivek Nair

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2019

APPROVED BY:

Dr. Kathyrn Stolee	Dr. Min Chi
Dr. Ranga Raju Vatsavai	Dr. Timothy J. Menzies Chair of Advisory Committee

DEDICATION

To Ammayi—for timely advice and support

BIOGRAPHY

Vivek Nair was born and raised in the city of Kolkata, India. He graduate from West Bengal University of Technology and National Institute of Technology, Durgapur with and Bachelors in Technology and Master in Technology respectively. His primary research interest is search based software engineering with a primary goal to solve search problems to maximize the quality of the search while minimizing the cost of search. His current research investigated problems in the software engineering domain and developed techniques not only to solve software configuration problem, but also software product lines as well as cloud architecture tuning problems. Before joining the PhD program, he was a software engineer at Samsung Software Engineering Labs. During his PhD program, his internship experiences include LexisNexis Risk Solutions (2015-2017), and Microsoft Research - Redmond, USA (Summer 2018).

ACKNOWLEDGEMENTS

**This acknowledgment contains plenty of subtexts and would be found as text within brackets. The subtext is meant to serve the purpose of comic relief and comic relief only.*

This thesis is no way an individual work. It is a product of a lot of love and sacrifice, tears and sweat of not just me but for various individuals who have taught lessons throughout my journey as a graduate student, here at NCSU. Although, I have endeavored to acknowledge everyone who has touched my life [for better or for worse], this list is in no way comprehensive.

One of the primary reasons, I can write this thesis is because of Dr. George N. Rouskas and my aunt, Dr. Latha Unni. My first year at NCSU was tumultuous and eventful. As new graduate student, dealing with the stresses of graduate school and the unfavorable environment of my previous lab was overwhelming. This resulted in me [prematurely] deciding to quit graduate school—which I have been so excited to attend. Dr. Rouskas, who provided me with financial support while transitioning from my old advisor to my current advisor. Dr. Rouskas and my aunt counseled and provided support when I needed it the most. This thesis would have never been possible without these two amazing people.

Secondly my thesis advisor [my academic father], Dr. Menzies, he is a legend [faking the Aussie accent]. He has been very [not so] patient with me and directed me towards areas which held promise. He helped me understand the value of collaboration and being a voracious reader of the literature. That said I have never had a more dramatic relationship with anyone, which in retrospect is an effective way of training graduate student. Our story has all the elements of a soap opera, there are love and respect, not so much love [hate sounds just too strong] and doubt. Either way, he played a massive part in making this thesis possible and made me a better researcher [hopefully]. Also, he is also a reason I went to a trip around Europe [with Dr. Fu and George].

Thirdly, I would then like to thank my dissertation committee members, Dr. Ranga Raju Vatsavai, Dr. Min Chi and Dr. Kathryn T. Stolee, for their valuable feedback and insight on my dissertation.

I am also grateful to my research collaborators: Sven Apel (University of Passau), Norbert Siegmund (Bauhaus-University Weimar), and Pooyan Jamshidi (University of South Carolina), Jianfeng Chen (North Carolina State University). I would like to gratefully acknowledge researchers who generously shared their research tools and results used in my dissertation. I am very much thankful to all my peers at the RAISE Lab, for their constant support, and useful feedback on my research. I thank Dr. Wei Fu, Dr. Chin-Jung Hsu, Rahul Krishna, George Mathew, and Zhe Yu for the insightful and exciting conversations [interestingly I was able to

write papers with all these amazing individuals—so all our discussions were not completely useless—as noted by Dr. Fu in his thesis acknowledgement]. I thank my internship mentors in the industry, who have immensely helped me expand my research to a broader scope: Dan Camper and Arjuna Chala (LexisNexis Risk Solutions), and Chris Duvarney, Kim Herzig, and Hitesh Sajnani (Microsoft Research, USA).

I extend heartfelt thanks to Kathy Luca, Carol Allen, and all the helpful staff at the Department of Computer Science. I thank Amruthkiran Hedge, Anand Gorthi, Sandesh Saokar, Siddartha Chauhan, Mayank Vaish, Akhilesh Tanneru [Mr. T], and George Mathew for sharing the apartment and making my stay at Raleigh eventful. Special thanks to Karen Warmbein, Blue and [Z]Simba whose love and support helped me survive the grind.

Finally, last but not least, I am very thankful to my family and friends. My family and friends were my constant source of support and encouragement throughout my journey as a doctoral student. My Ph.D. journey would not have gone so far, without y'all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Modern software systems provide configuration options to modify both functional behavior of the system (functionality of the system) and non-functional properties of the system (performance and memory consumption). Configuration options of a software system that are relevant to users are usually referred to as *features*. All the features of a system together (vector of configuration options) define a *configuration* of a software system. The features can take integer, decimal or string values. An important non-functional property is performance because it influences how a user interacts with the system. Environmental/external factors can affect performance. For example, the performance of the software system can be better for more number of cores and higher RAM. A software system is required to select and set configuration options to maximize the performance of that system. For example, say we have a software system with 10 (binary) configuration options—it results in a configuration space of size 2^{10} or 1024. The user of the software system now has to find the optimal configuration for the given task (or input) in hand. This problem can be tackled in two different ways: (1) exhaustively measuring the performance of all possible configurations—which means running 1024 benchmark runs, and (2) use domain knowledge (assuming the user has tuned similar software system before) to find the best configuration. However, as the number of configuration options increase, it becomes difficult for humans to keep track of the interactions between the configuration options. This means as the configuration space grows it is harder to either exhaustively measure performance for all possible configurations or find domain experts to do so confidently. Please note that the optimal configuration can change dramatically with different inputs (tasks) and the environment—which make domain knowledge-based decision less reliable.

Numerous researchers from different domains have reported this exact problem.

- Many software systems have poorly chosen defaults [1], [2]. Hence, it is useful to seek better configurations.

- Understanding, the configuration space of software systems with large configuration spaces, is challenging [3].
- Exploring more than just a handful of configurations is usually infeasible due to long benchmarking time [4].

The problem we are trying to tackle throughout this document is: "How can we find a set of configuration options which would maximize the performance of a system while minimizing the cost of search". Here we would limit our scope of the study to just the configurations options or features of a particular software system (and not its environment). While exploring the above mentioned, we produced three methods:

1. The paper titled "Faster Discovery of Faster System Configurations with Spectral Learning" introduced a method called **WHAT** , which improved upon the methods proposed by Guo et al. [36] and Sarkar et al. [78]. This paper was published in Automated Software Engineering Journal (ASEJ).
2. The paper titled "Using Bad Learners to find Good Configurations" introduced a rank-based method, which improved upon the method proposed by Guo et al. and Sarkar et al. This paper was published in Foundations of Software Engineering 2017 (FSE).
3. The paper titled "Finding faster configurations using Flash" introduce a method called Flash, which improved upon methods proposed by Guo et al. [36], Sarkar et al. [78], and Zuluaga et al. [111]. This paper was published in IEEE Transactions on Software Engineering (TSE).

While this has been used to configure a software system, in theory, is not different from general optimization strategies can be applied to many different domains. My colleagues in The RAISE Labs have been using the methods proposed in this thesis (notably FLASH) on various other domains. The preliminary results are positive.

1.1 Statement of Thesis

This dissertation advances knowledge through several contributions and defends the claims of the thesis. The thesis in full below:

> **Effective performance optimization of configurable software systems only requires approximate, cheaper and easy to build models.**

This thesis proposes three methods to find good configurations, and we endorse the Sequential Model-based Optimization. For a summary, see the rest of the chapter. For more details, please feel free to read the whole thesis.

1.2 Clustering

The prior work in this area [36, 78] used a combination of random sampling and regression trees. However, random sampling completely disregards the presence of clusters in the configuration space. It has been shown in the literature [72] that (1) most of the configuration options do not influence the performance, and (2) the performance curve is generally a step function.

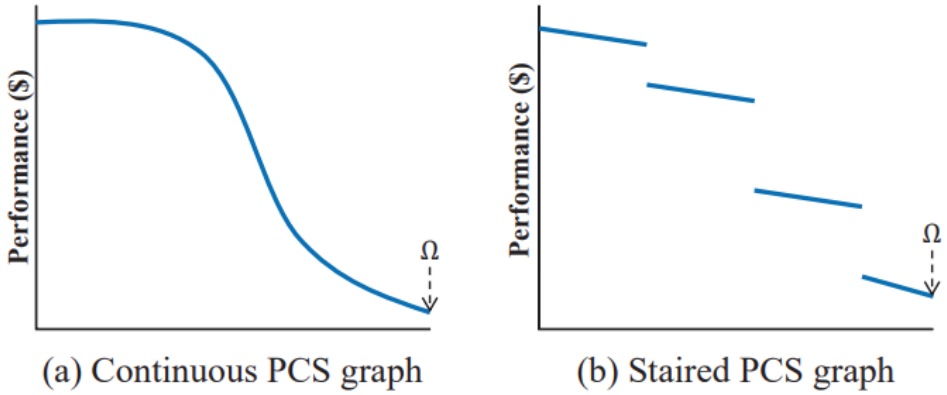

Figure 1.1 Performance curves. (From [72])

To elaborate, if we sort the configurations from worst-performance to best and plot configurations along the X-axis and performance along the Y-axis. We expected a continuous graph such as Figure 1.1(a), where high-valued ($) is bad (worst performance is at the far left), and low-valued ($) is good (best performance is at the far right). Interestingly, Marker et al. [60] discovered that performance curves often occurs (in the real world) as stairs, as in Figure 3b. Stairs arise from discrete feature decisions; some features are highly-influential in performance while others have little or no impact. Consequently, a few critical features influence the performance while less important feature decisions alter the performance of nearby configurations only slightly (giving a stair its width and slope).

Intuition: From the literature, it is evident that most of the configuration options in a (given) software system do not affect the performance of the configurations. Hence, the central insight of this work is that random sampling with no regards to this specific feature of this problem is ineffective and thus adds additional cost.

Proposal: This problem can be reformulated as a clustering problem, where we try to find an unsupervised method to cluster the configuration space into meaningful clusters. Once we have these clusters, we can use random samples (of configurations) from each of these clusters. This way, we can reduce redundant measurements.

1.3 Ranking

Prior work in this area (including [68]), tried to build accurate performance predictors, which can be used to predict the performance of a certain configuration. This work reflects on the prior work and asks: "*Our goal is to find a good configuration* [1] *but, why does the prior work transform this problem into building an accurate model?*" Another reason for this question is the very nature of the model building process previously described in Figure 2.2. We ask the question "How does a user define *Good*"? There is no way for the user to know whether a model can be built with MMRE less than 10%.

Concerning the questions above, we drastically modify our approach to this problem. We hypothesize that to find the best performing configuration; we do not want a model which can return a predicted performance score which is as close to the actual performance score. Instead, we build a model which preserves the relative ordering of the configurations.

Intuition: The central insight of this work is that exact performance values (e.g., the response time of a software system) are not required to rank configurations and to identify the optimal one. To elaborate more, let us assume that we have two humans (Adam—134cm, Billy—173cm) (analogous to configurations) and our objective is to identify the tallest person (Billy). To determine the tallest person, do we need a model which accurately predicts their height in a nano-meter scale? We can quickly identify Billy even if the bad model predicted Adams height as 700cm and 890cm.

Proposal: We show that, if we (slightly) relax the question we ask we can build useful predictors using minimal sample sets. Specifically, instead of asking "How long will this configuration run?", we ask instead "Will this configuration run faster than that configuration?" or "Which is the fastest configuration?".

1.4 Sequential-Model Based Sampling

Prior work in this area primarily used two strategies. Firstly, researchers used machine learning to model the configuration space. The model is built sequentially, where new configurations are sampled randomly, and the quality or accuracy of the model is measured using a holdout set. The size of the holdout set in some cases could be up to 20% of the configuration space [70] and needs to be evaluated (i.e., measured) before even the model is fully built. This strategy makes these methods not suitable in a practical setting since the generated holdout set can be (very) expensive. Secondly, the sequential model-based techniques used in prior work relied on Gaussian Process Models (GPM) to reflect on the configurations explored (or evaluated)

[1] Good is defined as the distance from the optimal configuration

so far [111]. However, GPM does not scale well for software systems with more than a dozen configuration options [101].

Intuition: To reduce the cost of sampling and eliminate the need for holdout set, we use sequential Model-based Optimization (SMBO). SMBO uses the Bayesian methodology to the iterative optimizer by incorporating a prior model (built using configurations which are already measured) on the space of possible target functions, f. By updating this model every time a configuration is evaluated, an SMBO routine keeps a posterior model of the target function f. This posterior model is the surrogate f^* for the function f (ground truth). Figure 1.2 encapsulates the process.

Algorithm 1 Sequential Model-Based Optimization

Input: $f, \mathcal{X}, S, \mathcal{M}$
$\mathcal{D} \leftarrow \text{INITSAMPLES}(f, \mathcal{X})$
for $i \leftarrow |\mathcal{D}|$ **to** T **do**
  $p(y \mid \mathbf{x}, \mathcal{D}) \leftarrow \text{FITMODEL}(\mathcal{M}, \mathcal{D})$
  $\mathbf{x}_i \leftarrow \arg\max_{\mathbf{x} \in \mathcal{X}} S(\mathbf{x},\ p(y \mid \mathbf{x}, \mathcal{D}))$
  $y_i \leftarrow f(\mathbf{x}_i)$ ▷ Expensive step
  $\mathcal{D} \leftarrow \mathcal{D} \cup (\mathbf{x}_i, y_i)$
end for

Figure 1.2 Sequential Model-based Optimization. (From http://tiny.cc/3hy80y)

Proposal: We use the intuition present above to develop a method called FLASH. The key idea of FLASH is to build a performance model that is just accurate enough for differentiating better configurations from the rest of the configuration space. Tolerating the inaccuracy of the model is useful to reduce the cost (measured in terms of the number of configurations evaluated) and the time required to find the better configuration. To increase the scalability of methods using GPM (Gaussian Process Models)—used widely in the machine learning domain, FLASH replaces the GPMs with a fast and scalable decision tree learner.

1.5 How to Read the Dissertation

This dissertation is organized as self-contained chapters that together support the thesis.

Chapter 1 explains, identifies and provides context for the research problem, articulates the objective, significance, and scope of the work.

Chapter 2 presents the background and related work in this area. The area of performance optimization has been explored not just in the domain of software engineering and has gained a

lot of interest in the systems community as well. In general, there has been considerable interest in the black-box optimization literature. In this chapter, an attempt has been made to describe the research work using the terminology used in the rest of the thesis.

Chapter 3 describes the design and implementation of *WHAT*. **WHAT** 's innovation is the use of the spectrum (eigenvalues) of the distance matrix between the configurations of a configurable software system, to perform dimensionality reduction. Within that reduced configuration space, many closely associated configurations can be studied by executing only a few sample configurations. For the subject systems studied here, a few dozen samples yield accurate and stable predictors—less than 10 % prediction error, with a standard deviation of less than 2 %. When compared to state of the art, our approach (a) requires 2 to 10 times fewer samples to achieve similar prediction accuracies, and (b) its predictions are more stable (i.e., have lower standard deviation). Furthermore, we demonstrate that predictive models generated by **WHAT** can be used by optimizers to discover system configurations that closely approach the optimal performance.

Chapter 4 proposes using ranking models instead of regression models to save cost while finding good configurations. The central insight of this chapter is that exact performance values (e.g., the response time of a software system) are not required to rank configurations and to identify the optimal one. As shown by our experiments, performance models that are cheap to learn but inaccurate (with respect to the difference between actual and predicted performance) can still be used rank configurations and hence find the optimal configuration. This novel *rank-based approach* allows us to significantly reduce the cost (in terms of the number of measurements of sample configuration) as well as the time required to build performance models. We evaluate our approach with 21 scenarios based on nine software systems and demonstrate that our approach is beneficial in 16 scenarios; for the remaining five scenarios, an accurate model can be built by using very few samples anyway, without the need for a rank-based approach.

In **Chapter 5**, we design and implement FLASH. The central insight of this paper is to use prior knowledge (gained from prior runs) to choose the next promising configuration. This strategy reduces the effort (measured in terms of the number of measurements) required to find the (near) optimal configuration. FLASH can be used to solve single-objective (e.g., run-time) and can also be adapted to multi-objective (e.g., energy and runtime) performance optimization problems. We evaluate FLASH using 30 scenarios based on seven software systems to demonstrate that FLASH saves effort in 100% and 80% of cases in single-objective and multi-objective problems respectively by up to several orders of magnitude.

In **Chapter 6**, we revisit the vital contributions of this thesis and provide directions for future work.

Chapter 2

Background and Related Work

This section offers background notes, evaluation criterion, and vocabulary which would be used throughout this thesis.

2.1 Motivating Example

To motivate our work, we use the same example from a previous work [36], a configurable command-line tool x264 for encoding video streams into the H.264/MPEG-4 AVC format. In this example, we consider 16 encoder features of x264, such as encoding with multiple reference frames and parallel encoding on multiple CPUs. The user can select different features to encode a video. The encoding time is used to indicate the performance of x264 in different configurations. A configuration represents a program variant with a certain selection of features. This example with only 16 features gives rise to 1,152 configurations. Intuitively, 16 binary features should provide 2^{16} different configurations, however, in this work we consider only valid configurations i.e. configurations that are allowed by the system under investigation. In practice, often only a limited set of configurations can be measured, either by simulation or by monitoring in the field. For example, Figure 2.1 lists a sample of 16 randomly selected configurations and their actual performance measurements. How can we determine the performance of other configurations based on a small random sample of measured configurations?

2.2 Problem Formulation

To formulate the above issue, we represent a feature as a binary decision variable x (please note that decision variable could be decimal as well). If a feature is selected in a configuration, then the corresponding variable is set to 1, and 0 otherwise. Assume that there are N features in total, all features of a program are represented as a set $X = x_1, x_2, ..., x_N$. A configuration

Conf.	Features																Perf. (s)
$\mathbf{c}_i$	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	y_i
$\mathbf{c}_1$	1	1	0	0	1	1	0	0	1	0	0	1	1	1	0	0	292
$\mathbf{c}_2$	1	0	0	0	1	1	0	1	1	1	0	0	1	0	1	0	571
$\mathbf{c}_3$	1	1	1	0	0	1	0	0	1	0	1	0	1	0	0	1	681
$\mathbf{c}_4$	1	1	0	1	1	0	0	0	1	0	1	0	1	1	0	0	263
$\mathbf{c}_5$	1	1	1	0	1	1	0	0	1	0	1	0	1	0	1	0	536
$\mathbf{c}_6$	1	0	0	1	0	1	1	1	1	0	1	0	1	1	0	0	305
$\mathbf{c}_7$	1	1	1	1	0	1	1	0	1	0	1	0	1	0	1	0	408
$\mathbf{c}_8$	1	0	0	1	1	0	1	1	1	0	1	0	1	1	0	0	278
$\mathbf{c}_9$	1	0	0	1	1	0	0	1	1	0	0	1	1	0	1	0	519
$\mathbf{c}_{10}$	1	1	0	0	1	1	0	1	1	0	1	0	1	0	0	1	781
$\mathbf{c}_{11}$	1	0	0	1	1	1	0	1	1	1	0	0	1	0	0	1	822
$\mathbf{c}_{12}$	1	1	0	0	1	0	0	0	1	0	1	0	1	0	0	1	713
$\mathbf{c}_{13}$	1	0	1	0	0	0	1	0	1	0	0	1	1	0	1	0	381
$\mathbf{c}_{14}$	1	0	1	1	1	1	0	1	1	1	0	0	1	0	1	0	564
$\mathbf{c}_{15}$	1	1	1	1	1	0	0	0	1	0	0	1	1	0	1	0	489
$\mathbf{c}_{16}$	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	0	275

Figure 2.1 A sample of 16 randomly-selected configurations of x264 and corresponding performance measurements (seconds)

is an N-tuple c, assigning 1 or 0 to each variable. For example, each configuration of x264 is represented by 16-tuple, e.g. $c_1 = (x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0, ..., x_{16} = 1)$. All valid configurations of a program are denoted by C. Each configuration c of a program has an actual measured performance value y. Performance values are taken from publicly available dataset deployed with SPLConqueror tool. All performance values of C form set Y . Suppose that we acquire a random sample of configurations $C_S \in C$ and their actual measured performance values $Y_S \in Y$, together forming sample S. The problem of variability-aware performance prediction is to predict the performance of other configurations in $C \in CS$ based on the measured sample S. We regard all variables in X as predictors and a configuration's actual performance value y as the response. In other words, we predict a quantitative response y based on a set of categorical predictors X, which is a typical regression problem. *Due to feature interactions, the above issue is reduced to a non-linear regression problem, where the response depends non-linearly on one or more predictors.*

2.3 Evaluation Metrics: Quality and Cost

The performance of the methods proposed in this thesis is evaluated the following two metrics: **Quality**: Quality of an approach refers to the distance between the solutions returned by the methods proposed and the ground truth (actual measurements). Quality can be calculated as the *rank difference* which is the distance between best solution (also referred to as configuration) found by (proposed) techniques to the actual best solution.

Cost: Cost of an approach refers to the effort (such as time, money, developer hours) required by an approach to find a good configurations. In this thesis, we use number of measurements (or benchmark runs) as an proxy to the search effort. This is something unique (and surprising that the other work in this area never considered this aspect) about the work presented in this thesis.

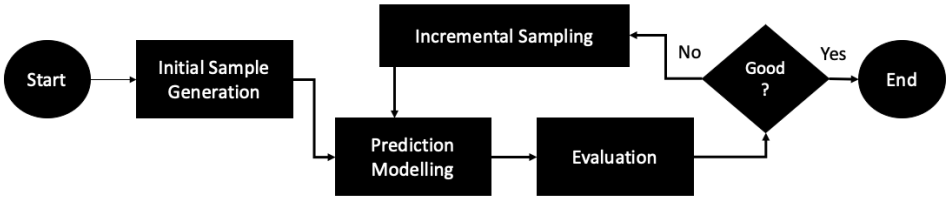

Figure 2.2 General process of performance prediction by sampling

2.4 Prior Work

Figure 2.2 illustrates the general process of performance prediction by sampling. It starts with an initial sample of measured configurations, which are used to build the prediction model. New configurations are added to iteratively improve the model such that it because reliable. Reliability of the model is measured using MMRE (Mean Magnitude of Relative Error), which is defined as:

$$MMRE = \frac{\mid predicted - actual \mid}{actual} * 100 \tag{2.1}$$

A good initial sample significantly reduces the iterations of the entire prediction process. State-of-the-art approaches fix the size of the initial sample to the number of features or potential feature interactions of a system [36, 49, 78, 86]. However, such a strategy might not be the optimal one, as the number of features (and their interactions) can be high and, at the same time, an acceptable prediction accuracy might be achieved using a substantially smaller set of measured configurations.

Observation: Although, the methods proposed by prior work exploited random sampling and CART decision trees (which are both scalable), these methods perform evaluations which are not necessarily useful in finding good configuration. While this approach worked reasonable well with respect to the quality of solutions, it failed to fare well in terms of cost (dual axis). Most of the method proposed in prior work use over 50% of the configurations (from the configuration space), to build a reliable model.

2.4.1 Related Work

Many researchers report that modern software systems come with *a daunting number of configuration options*. For example, the number of configuration options in Apache (a popular web server) increased from 150 to more than 550 configuration options within 16 years [103]. Van Aken et al. [94] also reports a similar trend. They indicate that, in over 15 years, the number of configuration options of POSTGRES and MYSQL increased by a factor of three and six, respectively. This is troubling since Xu et al. [103] report that developers tend to ignore over 80% of configuration options, which leaves considerable optimization potential untapped and induces major economic cost.[1]

Finally, another problem with configurable systems is the issue of *poorly chosen default configurations*. Often, it is assumed that software architects provide useful default configurations of their systems. This assumption can be very misleading. Van Aken et al. [94] report that the default MySQL configurations in 2016 assume that it will be installed on a machine that has 160MB of RAM (which, at that time, was incorrect by, at least, an order of magnitude). Herodotou et al. [42] show how standard settings for text mining applications in Hadoop result in worst-case execution times. In the same vein, Jamshidi et al. [49] reports for text mining applications on Apache storm, the throughput achieved using the worst configuration is *480 times slower* than the throughput achieved by the best configuration.

Yet another problem is that *exploring benchmark sets for different configurations is very slow*. Wang et al. [99] comments on the problems of evolving a test suite for software if every candidate solution requires a time-consuming execution of the entire system: such test suite generation can take weeks of execution time. Zuluaga et al. [110] report on the cost of analysis for software/hardware co-design: "synthesis of only one design can take hours or even days".

The challenges of having numerous configuration options are just *not limited to software systems*. The problem to find an optimal set of configuration options is pervasive and faced in numerous other sub-domains of computer science and beyond. In software engineering, software product lines–where the objective is to find a product which (say) reduces cost, defects [13, 41]—has been widely studied. Interaction testing—how to test configurable systems, when the same test case will behave differently while running under the same set of test cases [15, 51, 61, 75]. The problem of configuration optimization is present in domains, such as machine learning, cloud computing, and software security.

The area of *hyper-parameter optimization* (a.k.a. parameter tuning) is very similar to the performance configuration optimization problem studied in this paper. Instead of optimizing the performance of a software system, the hyper-parameter method tries to optimize the performance of a machine learner. Hyper-parameter optimization is an active area of research

[1] The size of the configuration space increases exponentially with the number of configuration options.

in various flavors of machine learning. For example, Bergstra and Bengiol [3] showed how random search could be used for hyper-parameter optimization of high dimensional spaces. Bergstra et a also showed how to tune computer vision software. Feurer el al. [26] showed how hyper-parameters could affect the performance of a Bayesian optimizer. Recently, there has been much interest in hyper-parameter optimization applied to the area of software analytics [1, 30–32, 92].

Another area of application for performance configuration optimization is *cloud computing*. With the advent of big data, long-running analytics jobs are commonplace. Since different analytic jobs have diverse behaviors and resource requirements, choosing the correct virtual machine type in a cloud environment has become critical. This problem has received considerable interest, and we argue, this is another useful application of performance configuration optimization — that is, optimize the performance of a system while minimizing cost [2, 16, 97, 104, 109].

As a sideeffect of the wide-spread adoption of cloud computing, the *security* of the instances or virtual machines (VMs) has become a daunting task. In particular, optimized security settings are not identical in every setup. They depend on characteristics of the setup, on the ways an application is used or on other applications running on the same system. The problem of finding security setting for a VM is similar to performance configuration optimization [6, 7, 21, 44, 76]. Among numerous other problems which are similar to performance configuration optimization, the problem of how to maximize conversions on landing pages or click-through rates on search-engine result pages [43, 100, 108] has gathered interest.

Chapter 3

Faster Discovery of Faster System Configurations with Spectral Learning

This chapter originally appeared as Nair, V., Menzies, T., Siegmund, N., & Apel, S. (2017). Faster discovery of faster system configurations with spectral learning. Automated Software Engineering, 1-31.

3.1 Abstract

Despite the huge spread and economical importance of configurable software systems, there is unsatisfactory support in utilizing the full potential of these systems with respect to finding performance-optimal configurations. Prior work on predicting the performance of software configurations suffered from either (a) requiring far too many sample configurations or (b) large variances in their predictions. Both these problems can be avoided using the **WHAT** spectral learner. **WHAT** 's innovation is the use of the spectrum (eigenvalues) of the distance matrix between the configurations of a configurable software system, to perform dimensionality reduction. Within that reduced configuration space, many closely associated configurations can be studied by executing only a few sample configurations. For the subject systems studied here, a few dozen samples yield accurate and stable predictors—less than 10 % prediction error, with a standard deviation of less than 2 %. When compared to the state of the art, our approach (a) requires 2 to 10 times fewer samples to achieve similar prediction accuracies, and (b) its predictions are more stable (i.e., have lower standard deviation). Furthermore, we demonstrate that predictive models generated by **WHAT** can be used by optimizers to discover system configurations that closely approach the optimal performance.

3.2 Introduction

Addressing the challenge of performance prediction and optimization in the face of large configuration spaces, researchers have developed a number of approaches that rely on sampling and machine learning [36, 78, 86]. While gaining some ground, state-of-the-art approaches face two problems: (a) they require far too many sample configurations for learning or (b) they are prone to large variances in their predictions. For example, prior work on predicting performance scores using regression-trees had to compile and execute hundreds to thousands of specific system configurations [36]. A more balanced approach by Siegmund et al. is able to learn predictors for configurable systems [86] with low mean errors, but with large variances of prediction accuracy (e.g. in half of the results, the performance predictions for the Apache Web server were up to 50 % wrong). Guo et al. [36] also proposed an incremental method to build a predictor model, which uses incremental random samples with steps equal to the number of configuration options (features) of the system. This approach also suffered from unstable predictions (e.g., predictions had a mean error of up to 22 %, with a standard deviation of up 46 %). Finally, Sarkar et al. [78] proposed a projective-learning approach (using fewer measurements than Guo at al. and Siegmund et al.) to quickly compute the number of sample configurations for learning a stable predictor. However, as we will discuss, after making that prediction, the total number of samples required for learning the predictor is comparatively high (up to hundreds of samples).

The problems of large sample sets and large variances in prediction can be avoided using the **WHAT** spectral learner, which is our main contribution. **WHAT** 's innovation is the use of the spectrum (eigenvalues) of the distance matrix between the configurations of a configurable system, to perform dimensionality reduction. Within that reduced configuration space, many closely associated configurations can be studied by measuring only a few samples. In a number of experiments, we compared **WHAT** against the state-of-the-art approaches of Siegmund et al. [86], Guo et al. [36], and Sarkar et al. [78] by means of six real-world configurable systems: Berkeley DB, the Apache Web server, SQLite, the LLVM compiler, and the x264 video encoder. We found that **WHAT** performs as well or better than prior approaches, while requiring far fewer samples (just a few dozen). This is significant and most surprising, since some of the systems explored here have up to millions of possible configurations.

Overall, we make the following contributions:

- We present a novel sampling and learning approach for predicting the performance of software configurations in the face of large configuration spaces. The approach is based on a *spectral learner* that uses an approximation to the first principal component of the configuration space to recursively cluster it, relying only on a few points as representatives of each cluster.

- We demonstrate the practicality and generality of our approach by conducting experiments on six real-world configurable software systems (see Figure 3.1). The results show that our approach is more accurate (lower mean error) and more stable (lower standard deviation) than state-of-the-art approaches.

- We report on a comparative analysis of our approach and three state-of-the-art approaches, demonstrating that our approach outperforms previous approaches in terms of sample size and prediction stability. A key finding is the utility of the principal component of a configuration space to find informative samples from a large configuration space.

3.3 Background & Related Work

A configurable software system has a set X of Boolean configuration options,[1] also referred to as features or independent variables in our setting. We denote the number of features of system S as n. The configuration space of S can be represented by a Boolean space Z_2^n, which is denoted by F. All valid configurations of S belong to a set V, which is represented by vectors $\vec{C}_i$ (with $1 \leq i \leq |V|$) in Z_2^n. Each element of a configuration represents a feature, which can either be *True* or *False*, based on whether the feature is selected or not. Each valid instance of a vector (i.e., a configuration) has a corresponding performance score associated to it.

The literature offers two approaches to performance prediction of software configurations: a *maximal sampling* and a *minimal sampling* approach: With *maximal sampling*, we compile all possible configurations and record the associated performance scores. Maximal sampling can be impractically slow. For example, the performance data used in this paper required 26 days of CPU time for measuring (and much longer, if we also count the time required for compiling the code prior to execution). Other researchers have commented that, in real world scenarios, the cost of acquiring the optimal configuration is overly expensive and time consuming [102].

If collecting performance scores of all configurations is impractical, *minimal sampling* can be used to intelligently select and execute just enough configurations (i.e., samples) to build a predictive model. For example, Zhang et al. [107] approximate the configuration space as a Fourier series, after which they can derive an expression showing how many configurations must be studied to build predictive models with a given error. While a theoretically satisfying result, that approach still needs thousands to hundreds of thousands of executions of sample configurations.

Another set of approaches are the four "additive" *minimal sampling* methods of Siegmund et al. [86]. Their first method, called feature-wise sampling (*FW*), is their basic method. To

[1] In this paper, we concentrate on Boolean options, as they make up the majority of all options; see Siegmund et al., for how to incorporate numeric options [87].

explain *FW*, we note that, from a configurable software system, it is theoretically possible to enumerate many or all of the valid configurations[2]. Since each configuration ($\vec{C}_i$) is a vector of n Booleans, it is possible to use this information to isolate examples of how much each feature individually contributes to the total run time:

1. Find a pair of configurations $\vec{C}_i$ and $\vec{C}_2$, where $\vec{C}_2$ uses exactly the same features as $\vec{C}_i$, plus one extra feature f_i.

2. Set the run time $\Pi(f_i)$ for feature f_i to be the difference in the performance scores between $\vec{C}_2$ and $\vec{C}_i$.

3. The run time for a new configuration $\vec{C}_i$ (with $1 \leq i \leq |V|$) that has not been sampled before is then the sum of the run time of its features, as determined before:

$$\Pi(C_i) = \sum_{f_j \in C_i} \Pi(f_j) \tag{3.1}$$

When many pairs, such as $\vec{C}_1, \vec{C}_2$, satisfy the criteria of point 1, Siegmund et al. used the pair that covers the *smallest* number of features. Their minimal sampling method, *FW*, compiles and executes only these smallest C_1 and C_2 configurations. Siegmund et al. also offers three extensions to the basic method, which are based on sampling not just the smallest $\vec{C}_i$,$\vec{C}_2$ pairs, but also any configurations with *interactions* between features. All the following minimal sampling policies compile and execute valid configurations selected via one of three heuristics:

PW (pair-wise): For each pair of features, try to find a configuration that contains the pair and has a minimal number of features selected.

HO (higher-order): Select extra configurations, in which three features, f_1, f_2, f_3, are selected if two of the following pair-wise interactions exist: (f_1, f_2) and (f_2, f_3) and (f_1, f_3).

HS (hot-spot): Select extra configurations that contain features that are frequently interacting with other features.

Guo et al. [36] proposed a progressive random sampling approach, which samples in steps of the number of features of the software system in question. They used the sampled configurations to train a regression tree, which is then used to predict the performance scores of other system configurations. The termination criterion of this approach is based on a heuristic, similar to the *PW* heuristics of Siegmund et al.

[2]Though, in practice, this can be very difficult. For example, in models like the Linux Kernel such an enumeration is practically impossible [79].

Sarkar et al. [78] proposed a cost model for predicting the effort (or cost) required to generate an accurate predictive model. The user can use this model to decide whether to go ahead and build the predictive model. This method randomly samples configurations and uses a heuristic based on feature frequencies as termination criterion. The samples are then used to train a regression tree; the accuracy of the model is measured by using a test set (where the size of the training set is equal to size of the test set). One of four projective functions (e.g., exponential) is selected based on how correlated they are to accuracy measures. The projective function is used to approximate the accuracy-measure curve, and the elbow point of the curve is then used as the optimal sample size. Once the optimal size is known, Sarkar et al. uses the approach of Guo et al. to build the actual prediction model.

The advantage of these previous approaches is that, unlike the results of Zhang et al., they require only dozens to hundreds of samples. Also, like our approach, they do not require to enumerate all configurations, which is important for highly configurable software systems. That said, as shown by our experiments (see Section 3.5), these approaches produce estimates with larger mean errors and partially larger variances than our approach. While sometimes the approach by Sarkar et al. results in models with (slightly) lower mean error rates, it still requires a considerably larger number of samples (up to hundreds, while **WHAT** requires only few dozen).

3.4 Approach

3.4.1 Spectral Learning

The minimal sampling method proposed in this paper is based on a spectral-learning algorithm that explores the spectrum (eigenvalues) of the distance matrix between configurations. In theory, such spectral learners are an appropriate method to handle noisy, redundant, and tightly inter-connected variables, for the following reasons: When data sets have many irrelevancies or closely associated data parameters d, then only a few eigenvectors e, $e \ll d$ are required to characterize the data. In that reduced space:

- Multiple inter-connected variables $i, j, k \subseteq d$ can be represented by a single eigenvector;
- Noisy variables from d are ignored, because they do not contribute to the signal in the data;
- Variables become (approximately) parallel lines in e space. For redundancies $i, j \in d$, we can ignore j since effects that change over j also change in the same way over i;

That is, in theory, samples of configurations drawn via an eigenspace sampling method would not get confused by noisy, redundant, or tightly inter-connected variables. Accordingly, we

expect predictions built from that sample to have lower mean errors and lower variances on that error.

Spectral methods have been used before for a variety of data mining applications [53]. Algorithms, such as PDDP [9], use spectral methods, such as principle component analysis (PCA), to recursively divide data into smaller regions. Software-analytics researchers use spectral methods (again, PCA) as a pre-processor prior to data mining to reduce noise in software-related data sets [93]. However, to the best of our knowledge, spectral methods have not been used before in software engineering as a basis of a minimal sampling method.

WHAT is somewhat different from other spectral learners explored in, for instance, image processing applications [84]. Work on image processing does not address defining a minimal sampling policy to predict performance scores. Also, a standard spectral method requires an $O(N^2)$ matrix multiplication to compute the components of PCA [48]. Worse, in the case of hierarchical division methods, such as PDDP, the polynomial-time inference must be repeated at every level of the hierarchy. Competitive results can be achieved using an $O(2N)$ analysis that we have developed previously [62], which is based on a heuristic proposed by Faloutsos and Lin [24] (which Platt has shown computes a Nyström approximation to the first component of PCA [73]).

Our approach inputs N (with $1 \leq |N| \leq |V|$) valid configurations ($\vec{C}$), $N_1, N_2, ...$, and then:

1. Picks any point N_i ($1 \leq i \leq |N|$) at random;
2. Finds the point $West \in N$ that is furthest away from N_i;
3. Finds the point $East \in N$ that is furthest from $West$.

The line joining *East* and *West* is our approximation for the first principal component. Using the distance calculation shown in Equation 3.2, we define δ to be the distance between *East* and *West*. **WHAT** uses this distance (δ) to divide all the configurations as follows: The value x_i is the projection of N_i on the line running from *East* to *West*[3]. We divide the examples based on the median value of the projection of x_i. Now, we have two clusters of data divided based on the projection values (of N_i) on the line joining *East* and *West*. This process is applied recursively on these clusters until a predefined stopping condition. In our study, the recursive splitting of

[3]The projection of N_i can be calculated in the following way:
$a = dist(East, N_i); b = dist(West, N_i); x_i = \sqrt{\frac{a^2 - b^2 + \delta^2}{2\delta}}$.

the N_i's stops when a sub-region contains less than $\sqrt{|N|}$ examples.

$$dist(x,y) = \begin{cases} \sqrt{\sum_i (x_i - y_i)^2} & \text{if } x_i \text{ and } y_i \text{ is numeric} \\ \begin{cases} 0, & \text{if } x_i = y_i \\ 1, & \text{otherwise} \end{cases} & \text{if } x_i \text{ and } y_i \text{ is Boolean} \end{cases} \tag{3.2}$$

We explore this approach for three reasons:

- *It is very fast*: This process requires only $2|n|$ distance comparisons per level of recursion, which is far less than the $O(N^2)$ required by PCA [22] or other algorithms such as K-Means [38].
- *It is not domain-specific*: Unlike traditional PCA, our approach is general in that it does not assume that all the variables are numeric. As shown in Equation 3.2,[4] we can approximate distances for both numeric and non-numeric data (e.g., Boolean).
- *It reduces the dimensionality problem*: This technique explores the underlying dimension (first principal component) without getting confused by noisy, related, and highly associated variables.

3.4.2 Spectral Sampling

When the above clustering method terminates, our sampling policy (which we will call S_1:Random) is then applied:

Random sampling (S_1): compile and execute one configuration, picked at random, from each leaf cluster;

We use this sampling policy, because (as we will show later) it performs better than:

East-West sampling (S_2): compile and execute the *East* and *West* poles of the leaf clusters;

Exemplar sampling (S_3): compile and execute all items in all leaves and return the one with lowest performance score.

Note that S_3 is *not* a *minimal* sampling policy (since it executes all configurations). We use it here as one baseline against which we can compare the other, more minimal, sampling policies. In the results that follow, we also compare our sampling methods against another baseline using information gathered after executing all configurations.

[4] In our study, *dist* accepts configurations ($\vec{C}$) and returns the distance between them. If x_i and $y_i \in R^n$, then the distance function would be same as the standard Euclidean distance.

3.4.3 Regression-Tree Learning

After collecting the data using one of the sampling policies (S_1, S_2, or S_3), as described in Section 3.4.2, we use a CART regression-tree learner [10] to build a performance predictor. Regression-tree learners seek the attribute-range split that most increases our ability to make accurate predictions. CART explores splits that divide N samples into two sets A and B, where each set has a standard deviation on the target variable of σ_1 and σ_2. CART finds the "best" split defined as the split that minimizes $\frac{A}{N}\sigma_1 + \frac{B}{N}\sigma_2$. Using this best split, CART divides the data recursively.

In summary, **WHAT** combines:

- The FASTMAP method of Faloutsos and Lin [24];
- A spectral-learning algorithm initially inspired by Boley's PDDP system [9], which we modify by replacing PCA with FASTMAP (called "WHERE" in prior work [62]);
- The sampling policy that explores the leaf clusters found by this recursive division;
- The CART regression-tree learner that converts the data from the samples collected by sampling policy into a run-time prediction model [10].

That is,

$$\begin{aligned} \text{WHERE} &= \text{PDDP} - \text{PCA} + \text{FASTMAP} \\ \textbf{WHAT} &= \text{WHERE} + \text{SamplingPolicy} + \text{CART} \end{aligned}$$

This unique combination of methods has not been previously explored in the software-engineering literature.

3.5 Experiments

All materials required for reproducing this work are available at https://goo.gl/689Dve.

3.5.1 Research Questions

We formulate our research questions in terms of the challenges of exploring large complex configuration spaces. Since our model explores the spectral space, our hypothesis is that only a small number of samples is required to explore the whole space. However, a prediction model built from a very small sample of the configuration space might be very inaccurate and unstable, that is, it may exhibit very large mean prediction errors and variances on the prediction error.

Also, if we learn models from small regions of the training data, it is possible that a learner will miss *trends* in the data between the sample points. Such trends are useful when building

optimizers (i.e., systems that input one configuration and propose an alternate configuration that has, for instance, a better performance). Such optimizers might need to evaluate hundreds to millions of alternate configurations. To speed up that process, optimizers can use a *surrogate model*[5] that mimics the outputs of a system of interest, while being computationally cheap(er) to evaluate [59]. For example, when optimizing performance scores, we might ask a CART for a performance prediction (rather than compile and execute the corresponding configuration). Note that such surrogate-based reasoning critically depends on how well the surrogate can guide optimization.

Therefore, to assess feasibility of our sampling policies, we must consider:

- Performance scores generated from our minimal sampling policy;
- The variance of the error rates when comparing predicted performance scores with actual ones;
- The optimization support offered by the performance predictor (i.e., can the model work in tandem with other off-the-shelf optimizers to generate useful solutions).

The above considerations lead to four research questions:

RQ1: *Can* **WHAT** *generate good predictions after executing only a small number of configurations?*

Here, by "good" we mean that the predictions made by models that were trained using sampling with **WHAT** are as accurate, or more accurate, as those generated from models supplied with more samples.

RQ2: *Does less data used in building the models cause larger variances in the predicted values?*

RQ3: *Can "good" surrogate models (to be used in optimizers) be built from minimal samples?*

Note that **RQ2** and **RQ3** are of particular concern with our approach, since our goal is to sample as little as possible from the configuration space.

RQ4: *How good is* **WHAT** *compared to the state of the art of learning performance predictors from configurable software systems?*

To answer RQ4, we will compare **WHAT** against approaches presented by Siegmund et al. [86], Guo et al. [36], and Sarkar et al. [78].

[5] Also known as response surface methods, meta models, or emulators.

Berkeley DB C Edition (BDBC) is an embedded database system written in C. It is one of the most deployed databases in the world, due to its low binary footprint and its configuration abilities. We used the benchmark provided by the vendor to measure response time.
Berkeley DB Java Edition (BDBJ) is a complete re-development in Java with full SQL support. Again, we used a benchmark provided by the vendor measuring response time.
Apache is a prominent open-source Web server that comes with various configuration options. To measure performance, we used the tools autobench and httperf to generate load on the Web server. We increased the load until the server could not handle any further requests and marked the maximum load as the performance value.
SQLite is an embedded database system deployed over several millions of devices. It supports a vast number of configuration options in terms of compiler flags. As benchmark, we used the benchmark provided by the vendor and measured the response time.
LLVM is a compiler infrastructure written in C++. It provides various configuration options to tailor the compilation process. As benchmark, we measured the time to compile LLVM's test suite.
x264 is a video encoder in C that provides configuration options to adjust output quality of encoded video files. As benchmark, we encoded the Sintel trailer (735 MB) from AVI to the xH.264 codec and measured encoding time.

System	LOC	Features	Configurations
BDBC	219,811	18	2,560
BDBJ	42,596	32	400
Apache	230,277	9	192
SQLite	312,625	39	3,932,160
LLVM	47,549	11	1,024
x264	45,743	16	1,152

Figure 3.1 Subject systems used in the experiments.

3.5.2 Subject Systems

The configurable systems we used in our experiments are described in Figure 3.1. Note, with "predicting performance", we mean predicting performance scores of the subject systems while executing test suites provided by the developers or the community, as described in Figure 3.1. To compare the predictions of our and prior approaches with actual performance measures, we use data sets that have been obtained by measuring *nearly all* configurations[6]. We say *nearly all* configurations, for the following reasoning: For all except one of our subject systems, the total number of valid configurations was tractable (192 to 2560). However, SQLite has 3,932,160 possible configurations, which is an impractically large number of configurations to test whether our predictions are accurate and stable. Hence, for SQLite, we use the 4500 samples for testing prediction accuracy and stability, which we could collect in one day of CPU time. Taking this into account, we will pay particular attention to the variance of the SQLite results.

[6] http://openscience.us/repo/performance-predict/cpm.html

3.5.3 Experimental Rig

RQ1 and **RQ2** require the construction and assessment of numerous runtime predictors from small samples of the data. The following rig implements that construction process.

For each configurable software system, we built a table of data, one row per valid configuration. We then ran all configurations of all software systems and recorded the performance scores (i.e., that are invoked by a benchmark). The exception is SQLite for which we measured only the configurations needed to detect interactions and additionally 100 random configurations to evaluate the accuracy of predictions. To this table, we added a column showing the performance score obtained from the actual measurements for each configuration.

Note that the following procedure ensures that we **never** test any prediction model on the data used to learn that model. Next, we repeated the following procedure 20 times (the figure of 20 repetitions was selected using the Central Limit Theorem): For each system in {BDBC, BDBJ, Apache, SQLite, LLVM, x264}

- Randomize the order of the rows in their table of data;
- For X in {10, 20, 30, ... , 90};
  - Let *Train* be the first X % of the data
  - Let *Test* be the rest of the data;
  - Pass *Train* to **WHAT** to select sample configurations;
  - Determine the performance scores associated with these configurations. This corresponds to a table lookup, but would entail compiling and executing a system configuration in a practical setting.
  - Using the *Train* data and their performance scores, build a performance predictor using CART.
  - Using the *Test* data, assess the accuracy of the predictor using the error measure of Equation 3.3 (see below).

The validity of the predictors built by the regression tree is verified on testing data. For each test item, we determine how long it *actually* takes to run the corresponding system configuration and compare the actual measured performance to the *prediction* from CART. The resulting prediction error is then computed using:

$$error = \frac{\mid predicted - actual \mid}{actual} * 100 \tag{3.3}$$

(Aside: It is reasonable to ask why this metrics and not some of the others proposed in the literature (e.g sum absolute residuals). In short, our results are stable across a range of different

metrics. For example, the results of this paper have been repeated using sum of absolute residuals and, in those other results, we seen the same ranking of methods; see http://tiny.cc/sumAR.)

RQ2 requires testing the standard deviation of the prediction error rate. To support that test, we:

- Determine the X-th point in the above experiments, where all predictions stop improving (elbow point);
- Measure the standard deviation of the error at this point, across our 20 repeats.

As shown in Figure 3.2, all our results plateaued after studying $X = 40\,\%$ of the valid configurations[7]. Hence to answer **RQ2**, we will compare all 20 predictions at $X = 40\,\%$.

RQ3 uses the learned regression tree as a *surrogate model* within an optimizer;

- Take $X = 40\,\%$ of the configurations;
- Apply **WHAT** to build a CART model using some minimal sample taken from that $40\,\%$;
- Use that CART model within some standard optimizer while searching for configurations with least runtime;
- Compare the faster configurations found in this manner with the fastest configuration known for that system.

This last item requires access to a ground truth of performance scores for a large number of configurations. For this experiment, we have access to that ground truth (since we have access to all system configurations, except for SQLite). Note that such a ground truth would not be needed when practitioners choose to use **WHAT** in their own work (it is only for our empirical investigation).

For the sake of completeness, we explored a range of optimizers seen in the literature in this second experiment: DE [90], NSGA-II [19], and our own GALE [56, 110] system. Normally, it would be reasonable to ask why we used those three, and not the hundreds of other optimizers described in the literature [28, 40]. However, as shown below, all these optimizers in this domain exhibited very similar behavior (all found configurations close to the best case performance). Hence, the specific choice of optimizer is not a critical variable in our analysis.

[7] Just to clarify one frequently asked question about this work, we note that our rig "studies" $40\,\%$ of the data. We do not mean that our predictive models require accessing the performance scores from the $40\,\%$ of the data. Rather, by "study" we mean reflect on a sample of configurations to determine what minimal subset of that sample deserves to be compiled and executed.

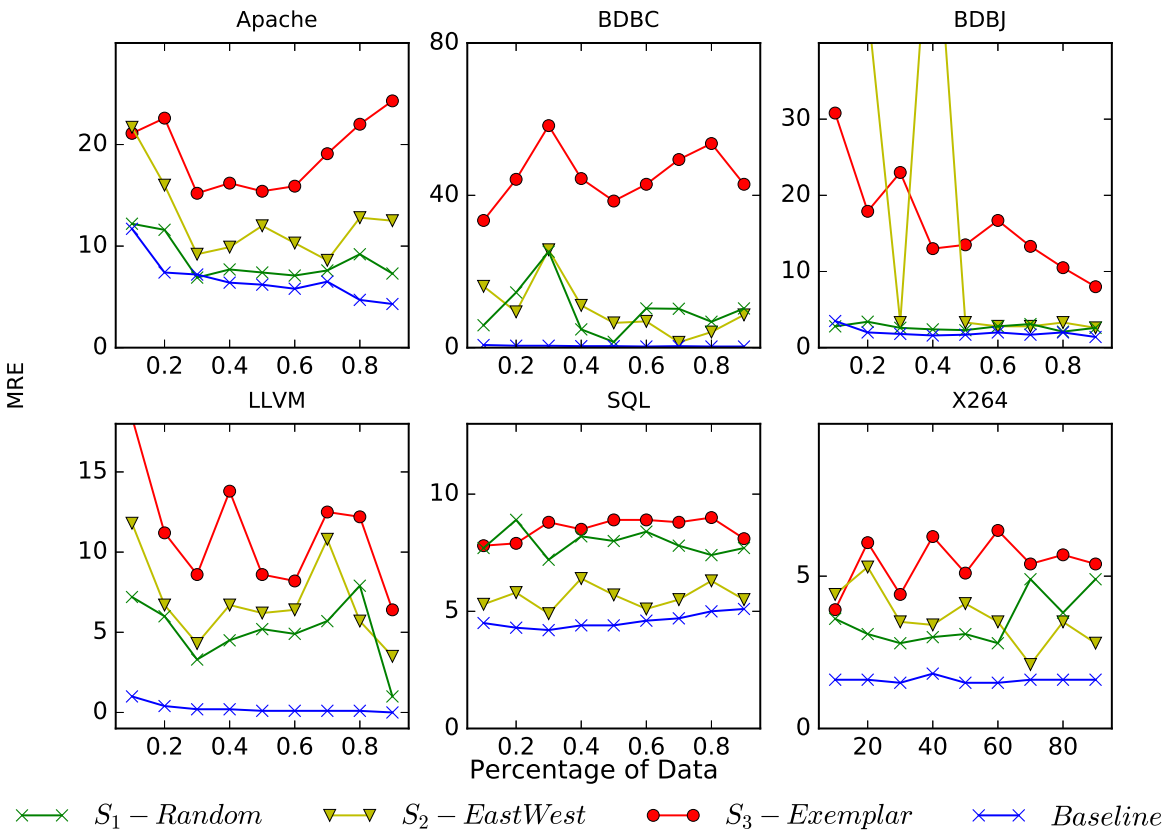

Figure 3.2 Errors of the predictions made by **WHAT** with four different sampling policies. Note that, on the y-axis, *lower* errors are *better*.

3.6 Results

3.6.1 RQ1

Can **WHAT** *generate good predictions after executing only a small number of configurations?*

Figure 3.2 shows the mean errors of the predictors learned after taking X % of the configurations, then asking **WHAT** and some sampling method (S_1, S_2, and S_3) to (a) find what configurations to measure; then (b) asking CART to build a predictor using these measurements. The horizontal axis of the plots shows what X % of the configurations are studied; the vertical axis shows the mean relative error (from Equation 3.3). In that figure:

- The $\times$—$\times$ lines in Figure 3.2 show a *baseline* result where data from the performance scores of 100 % of configurations were used by CART to build a runtime predictor.
- The other lines show the results using the sampling methods defined in Section 3.4.2.

Note that these sampling methods used runtime data only from a subset of 100 % of the performance scores seen in configurations from 0 to X %.

In Figure 3.2, *lower* y-axis values are *better* since this means lower prediction errors. Overall, we find that:

- Some software systems exhibit large variances in their error rate, below $X = 40\,\%$ (e.g., BDBC and BDBJ).
- Above $X = 40\,\%$, there is little effect on the overall change of the sampling methods.
- Mostly, S_3 shows the highest overall error, so that it cannot be recommended.
- Always, the $\times$—$\times$ baseline shows the lowest errors, which is to be expected since predictors built on the baseline have access to all data.
- We see a trend that the error of S_1 and S_2 are within 5 % of the *baseline* results. Hence, we can recommend these two minimal sampling methods.

Figure 3.3 provides information about which of S_1 or S_2 we should recommend. This figure displays data taken from the $X = 40\,\%$ point of Figure 3.2 and displays how many performance scores of configurations are needed by our sub-sampling methods (while reflecting on the configurations seen in the range $0 \leq X \leq 40$). Note that:

- S_3 needs up to thousands of performance-score points, so it cannot be recommended as minimal-sampling policy;
- S_2 needs twice as much performance-score information as S_1 (S_2 uses *two* samples per leaf cluster while S_1 uses only *one*).
- S_1 needs performance-score information on only a few dozen (or less) configurations to generate the predictions with the lower errors seen in Figure 3.2.

Combining the results of Figure 3.2 and Figure 3.3, we conclude that:

S_1 is our preferred spectral sampling method. Furthermore, the answer to **RQ1** is "yes", because applying **WHAT** , we can (a) generate runtime predictors using just a few dozens of sample performance scores; and (b) these predictions have error rates within 5 % of the error rates seen if predictors are built from information about all performance scores.

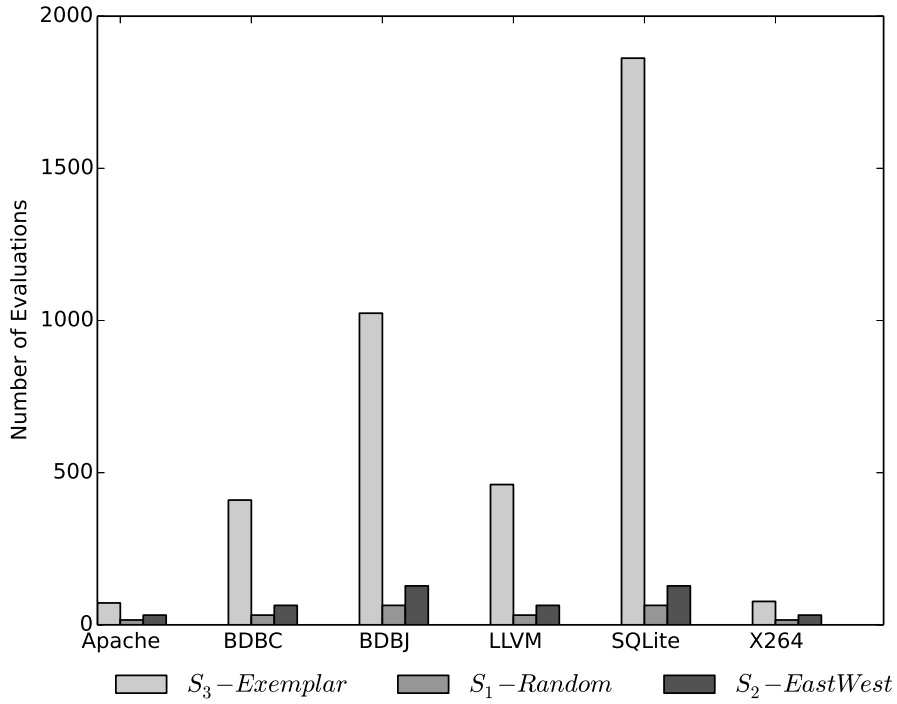

Figure 3.3 Comparing evaluations of different sampling policies. We see that the number of configurations evaluated for S_2 is twice as high as S_1, as it selects 2 points from each cluster, where as S_1 selects only 1 point.

3.6.2 RQ2

Do less data used in building models cause larger variances in the predicted values?

Two competing effects can cause increased or decreased variances in runtime predictions. The less we sample the configuration space, the less we constrain model generation in that space. Hence, one effect that can be expected is that models learned from too few samples exhibit large variances. But, a compensating effect can be introduced by sampling from the spectral space since that space contains fewer confusing or correlated variables than the raw configuration space. Figure 3.4 reports which one of these two competing effects are dominant. Figure 3.2 shows that after some initial fluctuations, after seeing $X = 40\,\%$ of the configurations, the variances in prediction errors reduces to nearly zero.

Hence, we answer **RQ2** with "no": Selecting a small number of samples does not necessarily increase variance (at least to say, not in this domain).

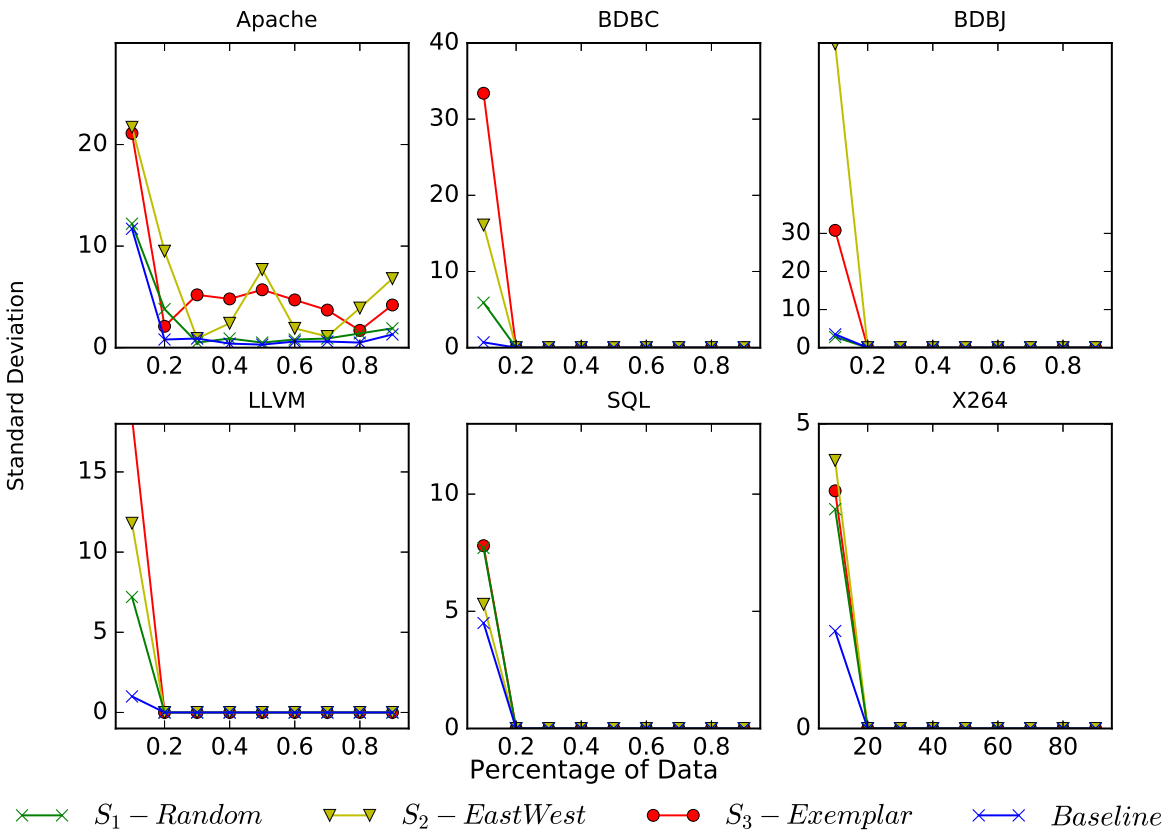

Figure 3.4 Standard deviations seen at various points of Figure 3.2.

3.6.3 RQ3

Can "good" surrogate models (to be used in optimizers) be built from minimal samples?

The results of answering **RQ1** and **RQ2** suggest to use **WHAT** (with S_1) to build runtime predictors from a small sample of data. **RQ3** asks if that predictor can be used by an optimizer to infer what *other* configurations correspond to system configurations with fast performance scores. To answer this question, we ran a random set of 100 configurations, 20 times, and related that baseline to three optimizers (GALE [56], DE [90] and NSGA-II [19]) using their default parameters.

When these three optimizers mutated existing configurations to suggest new ones, these mutations were checked for validity. Any mutants that violated the system's constraints (e.g., a feature excluding another feature) were rejected and the survivors were "evaluated" by asking the CART surrogate model. These evaluations either rejected the mutant or used it in generation $i + 1$, as the basis for a search for more, possibly better mutants.

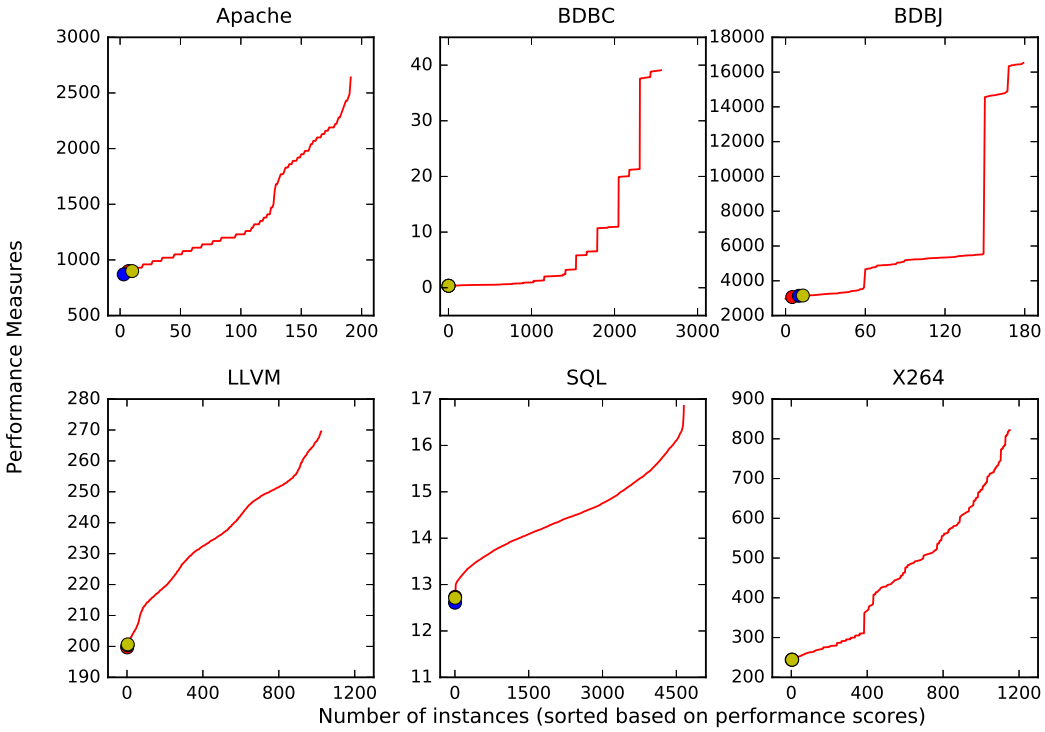

Figure 3.5 Solutions found by GALE, NSGA-II and DE (shown as points) laid against the ground truth (all known configuration performance scores). It can be observed that all the optimizers can find the configuration with lower performance scores.

Figure 3.5 shows the configurations found by three optimizers projected onto the ground truth of the performance scores of nearly all configurations (see Section 3.5.2). Again note that, while we use that ground truth for the validation of these results, our optimizers used only a small part of that ground-truth data in their search for the fastest configurations (see the **WHAT** + S_1 results of Figure 3.3).

The important feature of Figure 3.5 is that all the optimized configurations fall within 1 % of the fastest configuration according to the ground truth (see all the left-hand-side dots on each plot). Table 3.1 compares the performance of the optimizers used in this study. Note that the performances are nearly identical, which leads to the following conclusions:

The answer to **RQ3** is "yes": For optimizing performance scores, we can use surrogates built from few runtime samples. The choice of the optimizer does not critically effect this conclusion.

Table 3.1 The table shows how the minimum performance scores as found by the learners GALE, NSGA-II, and DE, vary over 20 repeated runs. Mean values are denoted μ and IQR denotes the 25th–75th percentile. A low IQR suggests that the surrogate model build by **WHAT** is stable and can be utilized by off the shelf optimizers to find performance-optimal configurations.

Dataset	Searcher					
	GALE		DE		NSGAII	
	Mean	IQR	Mean	IQR	Mean	IQR
Apache	870	0	840	0	840	0
BDBC	0.363	0.004	0.359	0.002	0.354	0.005
BDBJ	3139	70	3139	70	3139	70
LLVM	202	3.98	200	0	200	0
SQLite	13.1	0.241	13.1	0	13.1	0.406
X264	248	3.3	244	0.003	244	0.05

3.6.4 RQ4

How good is **WHAT** *compared to the state of the art of learning performance predictors from configurable software systems?*

We compare **WHAT** with the three state-of-the-art predictors proposed in the literature [86], [36], [78], as discussed in Section 3.3. Note that all approaches use regression-trees as predictors, except Siegmund's approach, which uses a regression function derived using linear programming. The results were studied using non-parametric tests, which was also used by Arcuri and Briand at ICSE '11 [64]). For testing statistical significance, we used non-parametric bootstrap test 95% confidence [23] followed by an A12 test to check that any observed differences were not trivially small effects; i.e. given two lists X and Y, count how often there are larger numbers in the former list (and there there are ties, add a half mark): $a = \forall x \in X, y \in Y \frac{\#(x>y)+0.5*\#(x=y)}{|X|*|Y|}$ (as per Vargha [96], we say that a "small" effect has $a < 0.6$). Lastly, to generate succinct reports, we use the Scott-Knott test to recursively divide our optimizers. This recursion used A12 and bootstrapping to group together subsets that are (a) not significantly different and are (b) not just a small effect different to each other. This use of Scott-Knott is endorsed by Mittas and Angelis [64] and by Hassan et al. [34].

As seen in the Figure 3.6, the FW approach of Siegmund et al. (i.e., the sampling approach using the fewest number of configurations) $(4/6)$ times has the higher errors rate and the highest standard deviation on that error rate. Hence, we cannot recommend this method or, if one wishes to use this method, we recommend using the other sampling heuristics (e.g., HO,

Rank	using	Mean MRE	STDev	Evaluations
Apache				
1	Sarkar	7.49	0.82	55
1	Guo(PW)	10.51	6.85	29
1	Siegmund	10.34	11.68	29
1	**WHAT**	10.95	2.74	16
1	Guo(2N)	13.03	15.28	18
BDBC				
1	Sarkar	1.24	1.46	191
2	Siegmund	6.14	4.41	139
2	**WHAT**	6.57	7.4	64
2	Guo(PW)	10.16	10.6	139
3	Guo(2N)	49.9	52.25	36
BDBJ				
1	Guo(2N)	2.29	3.26	52
1	Guo(PW)	2.86	2.72	48
1	**WHAT**	4.75	4.46	16
2	Sarkar	5.67	6.97	48
2	Siegmund	6.98	7.13	57
LLVM				
1	Guo(PW)	3.09	2.98	64
1	**WHAT**	3.32	1.05	32
1	Sarkar	3.72	0.45	62
1	Guo(2N)	4.99	5.05	22
2	Siegmund	8.5	8.28	43
SQLite				
1	Sarkar	3.44	0.1	925
2	**WHAT**	5.6	0.57	64
3	Guo(2N)	8.57	7.3	78
3	Guo(PW)	8.94	6.24	566
4	Siegmund	12.83	17.0	566
x264				
1	Sarkar	6.64	1.04	93
1	**WHAT**	6.93	1.67	32
1	Guo(2N)	7.18	7.07	32
1	Guo(PW)	7.72	2.33	81
2	Siegmund	31.87	21.24	81

Figure 3.6 Mean MRE seen in 20 repeats. Mean MRE is the prediction error as described in Equation 3.3 and STDev is the standard deviation of the MREs found during multiple repeats. Lines with a a dot in the middle (e.g. —•—) show the mean as a round dot withing the IQR (and if the IQR is very small, only a round dot will be visible). All the results are sorted by the mean values: lower mean value of MRE is better than large mean value. The left-hand side columns **Rank** the various techniques, smaller the value of **Rank** better the technique e.g. in Apache, all the techniques have the same rank since their mean values are not statistically different. **Rank** is computer using Scott-Knott, bootstrap 95% confidence, and the A12 test.

HS) to make more accurate predictions (but at the cost of much more measurements). Moreover, the size of the standard deviation of this method causes further difficulties in estimating which configurations are those exhibiting a large prediction error.

THere are two cases in Figure 3.6 where **WHAT** performs worse than at least one other methods:

- SQLite: here, the Sukar methods does better that **WHAT** (3.44 vs 5.6) but, as shown in the final column of Figure 3.6, does so at the cost of $\frac{925}{64} \approx 15$ times more evaluations that **WHAT** . In this case, a pragmatic engineering could well prefer our solution the that of Sukar (since Sukar uses more than an order of magnitude slowdown).

- BDBC: Here again, **WHAT** is not doing the best but, compared to the space of all other solutions, it is not doing particular bad.

As to the approach of Guo et al (with PW), this does not standout on any of our measurements. Its error results are within 1% of **WHAT** ; its standard deviation are usually larger and it requires much more data than **WHAT** (Evaluations column of the figure).

In terms of the number of measure samples required to build a model, the right-hand most column of Figure 3.6 shows that **WHAT** requires the fewest samples except for two cases: the approach of Guo et al. (with 2N) working on BDBC and LLVM. In both these cases, the mean error and standard deviation on the error estimate is larger than **WHAT** . Furthermore, in the case of BDBC, the error values are $\mu = 14\,\%$, $\sigma = 13\,\%$, which are much larger than **WHAT** 's error scores of $\mu = 6\,\%$, $\sigma = 5\,\%$.

Although the approach of Sarkar et al. produces an error rate that is sometimes less than the one of **WHAT** , it requires the most number of measurements. Moreover, **WHAT** ś accuracy is close to Sarkarś approach (1% to 2%) difference). Hence, we cannot recommend this approach, too.

Table 3.2 shows the number of evaluations used by each approaches. We see that most state-of-the-art approaches often require many more samples than **WHAT** . Using those fewest numbers of samples, **WHAT** has within 1 to 2 % of the lowest standard deviation rates and within 1 to 2 % of lowest error rates. The exception is Sarkar's approach, which has 5 % lower mean error rates. However, as shown in right-hand-side of Table 3.2, Sarkar's approach needs nearly three times more measurements than **WHAT** (191 vs 64 samples). Given the overall reduction of the error is small (5 % difference between Sarkar and **WHAT** in mean error), the overall cost of tripling the data-collection cost is often not feasible in a practical context and might not justify the small additional benefit in accuracy.

Table 3.2 Comparison of the number of the samples required with the state of the art. The grey colored cells indicate the approach which has the lowest number of samples. We notice that WHAT and Guo (2N) uses less data compared to other approaches. The high fault rate of Guo (2N) accompanied with high variability in the predictions makes WHAT our preferred method.

Dataset	Samples				
	Siegmund	Guo (2N)	Guo (PW)	Sarkar	WHAT
Apache	29	181	29	55	16
BDBC	139	36	139	191	64
BDBJ	48	52	48	57	16
LLVM	62	22	64	43	32
SQLite	566	78	566	925	64
X264	81	32	81	93	32

Hence, we answer **RQ4** with "yes", since **WHAT** yields predictions that are similar to or more accurate than prior work, while requiring fewer samples.

3.7 Why does it work?

In this section, we present an in-depth analysis to understand why our sampling technique (based on a spectral learner) achieves such low mean fault rates while being stable (low variance). We hypothesize that the configuration space of the system configuration lie on a low dimensional manifold.

3.7.1 History

Menzies et. al [62] suggested data heterogeneity by demonstrating the existence of local regions, which were very different from the global representation. Menzies et al. also makes an important observation about exploiting the underlying dimension to cluster the data. The authors used an algorithm called WHERE (see section 3.4.3), which recurses on two dimensions synthesized in linear time using a technique called FASTMAP [25]. The use of underlying dimension has been endorsed by various other researchers [4, 5, 20, 105]. There are numerous other methods in the literature, which are used to learn the underlying dimensionality of the data set like Principal Component Analysis (PCA) [52] [8], Spectral Learning [84], Random Projection [8]. These algorithms use different techniques to identify the underlying, independent/orthogonal

[8] WHERE is an approximation of the first principal component

dimensions to cluster the data points and differ with respect to the computational complexity and accuracy. We use WHERE since it computationally efficient $O(2N)$ while being accurate.

3.7.2 Testing Technique

Given our hypothesis – configuration space lies in a lower dimensional hyperplane– it is imperative to demonstrate that the intrinsic dimensionality of the configuration space is less than actual dimension. To formalize this notion we borrow the concept of correlation dimension from the domain of physics [35]. The correlation dimension of a dataset with k item is found by computing the number of items found at distance withing radius r (where r is the euclidean distance between two configurations) while varying r. This is then normalized by the number of connections between k items to find the expected number of neighbors at distance r. This can be written as:

$$C(r) = \frac{2}{k(k-1)} \sum_{i=1}^{n} \sum_{j=i+1}^{n} I(||x_i, x_j|| < r) \tag{3.4}$$

$$where : I(x < y) = \begin{cases} 1, & \text{if x <y} \\ 0, & \text{otherwise} \end{cases}$$

Given the dataset with k items and range of distances [r_0–r_{max}], we estimate the intrinsic dimensionality as the maximum slope between $\ln(C(r))$ vs $\ln(r)$.

3.7.3 Evaluation

We observe that **the intrinsic dimensionality of the software system is much lower than the actual dimension**. Figure 3.7 presents the intrinsic dimensionality along with the actual dimensions of the software systems. If we take a look at the intrinsic dimensionality and compare it with the actual dimensionality, then it becomes apparent that the configuration space lies on a lower dimensional hyperplane. For example, SQLite has 39 configuration options but the intrinsic dimensionality of the space is just 7.61 (this is a fractal dimension). At the heart of **WHAT** is WHERE (a spectral clusterer), which uses the approximation of the first principal component to divide the configuration space and hence can take advantage of the low intrinsic dimensionality.

As a summary, our observations indicates that the intrinsic dimension of the configuration space is much lower that its actual dimension. Hence, clustering based on the intrinsic dimensions rather than the actual dimension would be more effective. In other words, configurations with similar performance values lie closer in the intrinsic hyperplane when compared to the actual dimensions and may be the reason as to why **WHAT** achieves empirically good results.

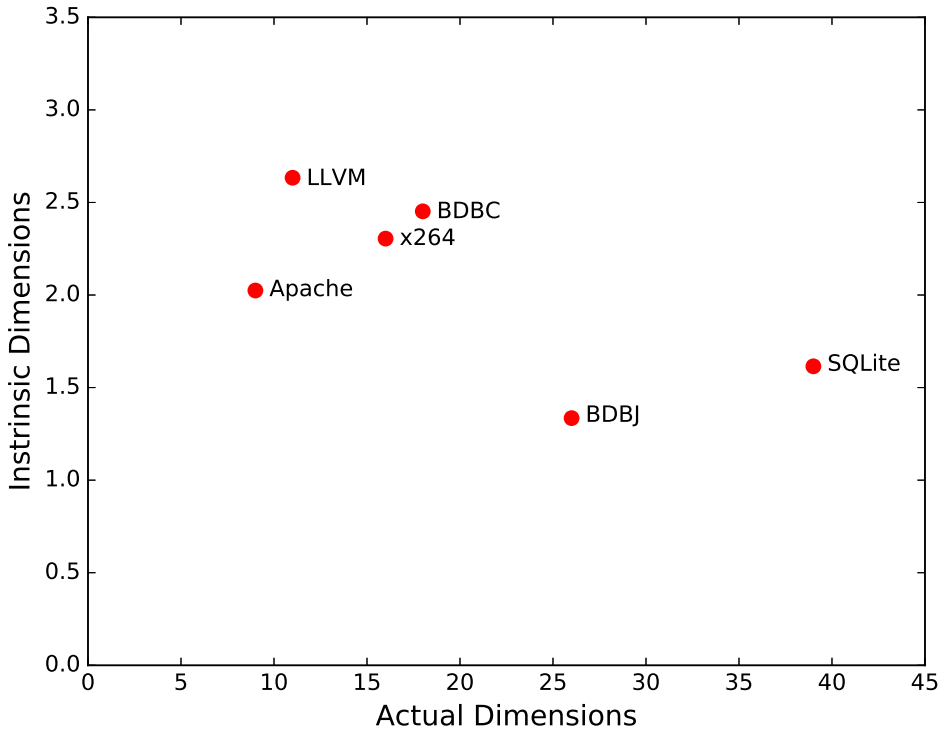

Figure 3.7 Intrinsic dimensionality of the subjects systems are shown on the y-axis. The number on the side is the actual dimension of the system. The intrinsic dimensionality of the systems are much lower than the actual dimensionality (number of columns in the dataset).

The intrinsic dimension of the configuration space is much lower than its actual dimension and hence clustering based on the intrinsic dimension is more effective.

3.8 Reliability and Validity

Reliability refers to the consistency of the results obtained from the research. For example, how well independent researchers could reproduce the study? To increase external reliability, this paper has taken care to either clearly define our algorithms or use implementations from the public domain (SciKitLearn) [81]. Also, all the data used in this work is available on-line in the PROMISE code repository and all our algorithms are on-line at github.com/ai-se/where.

Validity refers to the extent to which a piece of research actually investigates what the researcher purports to investigate [85]. *Internal validity* checks if the differences found in the

treatments can be ascribed to the treatments under study.

One internal validity issue with our experiments is the choice of *training and testing* data sets discussed in Figure 3.1. Recall that while all our learners used the same *testing* data set, our untuned learners were only given access to *training* data.

Another internal validity issues is *instrumentation*. The very low μ and σ error values reported in this study are so small that it is reasonable to ask whether they are due to some instrumentation quirk, rather than due to using a clever sample strategy:

- Our low μ values are consistent with prior work (e.g. [78]);
- As to our low σ values, we note that, when the error values are so close to 0 %, the standard deviation of the error is "squeezed" between zero and those errors. Hence, we would expect that experimental rigs that generate error values on the order of 5 % and Equation 3.3 should have σ values of $0 \leq \sigma \leq 5$ (e.g., like those seen in our introduction).

Regarding SQLite, we cannot measure all possible configurations in reasonable time. Hence, we sampled only 100 configurations to compare prediction and actual performance values. We are aware that this evaluation leaves room for outliers. Also, we are aware that measurement bias can cause false interpretations [62]. Since we aim at predicting performance for a special workload, we do not have to vary benchmarks.

We aimed at increasing the *external validity* by choosing software systems from different domains with different configuration mechanisms and implemented with different programming languages. Furthermore, the systems used are deployed and used in the real world. Nevertheless, assuming the evaluations to be automatically transferable to all configurable software systems is not fair. To further strengthen external validity, we run the model (generated by **WHAT** + S_1) against other optimizers, such as NSGA-II and differential evolution [90]. That is, we validated whether the learned models are not only applicable for GALE style of perturbation. In Table 3.1, we see that the models developed are valid for all optimizers, as all optimizers are able to find the near optimal solutions.

3.9 Related Work

In 2000, Shi and Maik [84] claimed the term "spectral clustering" as a reference to their normalized cuts image segmentation algorithm that partitions data through a spectral (eigenvalue) analysis of the Laplacian representation of the similarity graph between instances in the data.

In 2003, Kamvar et al. [53] generalized that definition saying that "spectral learners" were any data-mining algorithm that first replaced the raw dimensions with those inferred from the spectrum (eigenvalues) of the affinity (a.k.a. distance) matrix of the data, optionally adjusted via some normalization technique).

Our clustering based on first principal component splits the data on a approximation to an eigenvector, found at each recursive level of the data (as described in §3.4.1). Hence, this method is a "spectral clusterer" in the general Kamvar sense. Note that, for our data, we have not found that Kamvar's normalization matrices are needed.

Regarding sampling, there are a wide range of methods know as experimental designs or designs of experiments [74]. They usually rely on fractional factorial designs as in the combinatorial testing community [57].

Furthermore, there is a recent approach that learns *performance-influence models* for configurable software systems [87]. While this approach can handle even numeric features, it has similar sampling techniques for the Boolean features as reported in their earlier work [86]. Since we already compared to that earlier work and do not consider also numeric features, we did not compare our work to performance-influence models.

3.10 Conclusions

Configurable software systems today are widely used in practice, but expose challenges regarding finding performance-optimal configurations. State-of-the-art approaches require too many measurements or are prone to large variances in their performance predictions. To avoid these shortcomings, we have proposed a fast spectral learner, called **WHAT** , along with three new sampling techniques. The key idea of **WHAT** is to explore the configuration space with eigenvalues of the features used in a configuration to determine exactly those configurations for measurement that reveal key performance characteristics. This way, we can study many closely associated configurations with only a few measurements.

We evaluated our approach on six real-world configurable software systems borrowed from the literature. Our approach achieves similar to lower error rates, while being stable when compared to the state of the art. In particular, with the exception of Berkeley DB, our approach is more accurate than the state-of-the-art approaches by Siegmund et al. [86] and Guo et al. [36]. Furthermore, we achieve a similar prediction accuracy and stability as the approach by Sarkar et al [78], while requiring a far smaller number of configurations to be measured. We also demonstrated that our approach can be used to build cheap and stable surrogate prediction models, which can be used by off-the-shelf optimizers to find the performance-optimal configuration.

Chapter 4

Using Bad Learners to find Good Configurations

This chapter originally appeared as Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. Using bad learners to find good configurations. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017).

4.1 Abstract

The central insight of this paper is that exact performance values (e.g., the response time of a software system) are not required to rank configurations and to identify the optimal one. As shown by our experiments, performance models that are cheap to learn but inaccurate (with respect to the difference between actual and predicted performance) can still be used rank configurations and hence find the optimal configuration. This novel *rank-based approach* allows us to significantly reduce the cost (in terms of number of measurements of sample configuration) as well as the time required to build performance models. We evaluate our approach with 21 scenarios based on 9 software systems and demonstrate that our approach is beneficial in 16 scenarios; for the remaining 5 scenarios, an accurate model can be built by using very few samples anyway, without the need for a rank-based approach.

4.2 Introduction

This paper proposes an improvement of recent papers presented at ICSE'12, ASE'13, and ASE'15, which predict system performance based on learning influences of individual configuration options and combinations of thereof [36, 78, 86]. The idea is to measure a few configurations of a configurable software system and to make statements about the performance of its other

configurations. Thereby, the goal is to predict the performance of a given configuration as accurate as possible. We show that, if we (slightly) relax the question we ask, we can build useful predictors using very small sample sets. Specifically, instead of asking "How long will this configuration run?", we ask instead "Will this configuration run faster than that configuration?" or "Which is the fastest configuration?".

This is an important area of research since understanding system configurations has become a major problem in modern software systems. In their recent paper, Xu et al. documented the difficulties developers face with understanding the configuration spaces of their systems [103]. As a result, developers tend to ignore over 83% of configuration options, which leaves considerable optimization potential untapped and induces major economic cost [103].

With many configurations available for today's software systems, it is challenging to optimize for functional and non-functional properties. For functional properties, Chen et. al [13] and Sayyad et. al [80] developed fast techniques to find near-optimal configurations by solving a five-goal optimization problem. Henard et. al [41] used a SAT solver along with Multi-Objective Evolutionary Algorithms to repair invalid mutants found during the search process.

For non-functional properties, researchers have also developed a number of approaches. For example, it has been shown that the runtime of a configuration can be predicted with high accuracy by sampling and learning performance models [36, 78, 86]. State-of-the-art techniques rely on configuration data from which it is possible to build very accurate models. For example, prior work [68] has used sub-sampling to build predictors for configuration runtimes using predictors with error rates less than 5% (quantified in terms of *residual-based* measures such as Mean Magnitude of Relative Error, or MMRE, $(\sum_i^n(|a_i - p_i|/a_i))/n$ where a_i, p_i are the *actual* and *predicted values*). Figure 4.1 shows in green a number of real-world systems whose performance behavior can be modelled with high accuracy using state-of-the-art techniques.

Recently, we have come across software systems whose configuration spaces are far more complicated and hard to model. For example, when the state-of-the-art technique of Guo at al. [36] is applied to these software systems, the error rates of the generated predictor is up to 80%—see the yellow and **red** systems of Figure 4.1. The existence of these harder-to-model systems raises a serious validity question for all prior research in this area:

- Was prior research merely solving easy problems?
- Can we learn predictors for non-functional properties of more complex systems?

One pragmatic issue that complicates answering these two questions is the *minimal sampling problem*. It can be prohibitively expensive to run and test all configurations of modern software systems since their configuration spaces are very large. For example, to obtain the data used in our experiments, we required over a month of CPU time for measuring (and much longer, if

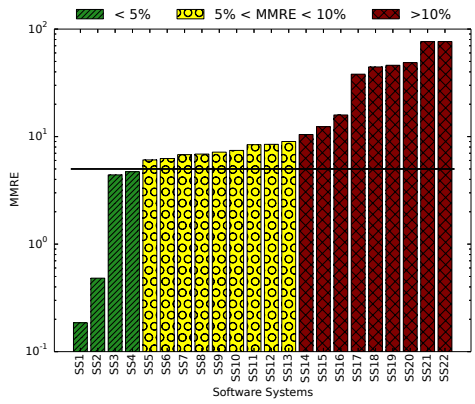

Figure 4.1 Errors of the predictions made by using CART, a machine learning technique (refer to Section 4.6), to model different software systems. Due to the results of Figure 4.2, we use 30%/70% of the valid configurations (chosen randomly) to train/test the model.

we also count the time required for compiling the code prior to execution). Other researchers have commented that, in real-world scenarios, the cost of acquiring the optimal configuration is overly expensive and time-consuming [102]. Hence, the goal of this paper must be:

1. Find predictors for non-functional properties for the hard-to-model systems of Figure 4.1, where learning accurate performance models is expensive.

2. Use as few sample configurations as possible.

The key result of this paper is that, even when *residual-based* performance models are inaccurate, *ranking* performance models can still be very useful for configuration optimization. Note that:

- Predictive models return a value for a configuration;

- Ranking models rank N configurations from "best" to "worst".

There are two very practical cases where such ranking models would suffice:

- Developers want to know the fastest configuration;

- Developers are debating alternate configurations and want to know which might run faster.

In this paper, we explore two research questions about constructing ranking models.
RQ1: *Can inaccurate predictive models still accurately rank configurations?*

We show below that, even if a model has, overall, a low predictive accuracy (i.e., a high MMRE), the predictions can still be used to effectively rank configurations. The rankings are

heuristic in nature and hence may be slightly inaccurate (w.r.t the actual performance value). That said, overall, our rankings are surprisingly accurate. For example, when exploring the configuration space of SQLite, our rankings are usually wrong only by less than 6 neighboring configurations – which is a very small number considering that SQLite has almost 4 million configurations.

RQ2: *How expensive is a rank-based approach (in terms of how many configurations must be executed)?*

To answer this question, we studied the configurations of 21 scenarios based on 9 open-source systems. We measure the benefit of our rank-based approach as the percentage of required measurements needed by state-of-the-art techniques in this field (see Sarkar et al. [78] presented at ASE'15). Those percentages were as follows – Note that *lower* values are *better* and values under 100% denote an improvement over the state-of-the-art (from Figure 4.9):

$$\{5,5,5,5,5,10,20,20,20,20,30,30,35,35,40,40,50,50,70,80,80,110\}\%$$

That is, the novel rank-based approach described in this paper is rarely worse than the state of the art and often far better. For example, as shown later in Figure 4.9, for one of the scenarios of Apache Storm, SS11, the rank-based approach uses only 5% of the measurements used by a residual-based approach.

The rest of this paper is structured as follows: we first formally describe the prediction problem. Then, we describe the state-of-the-art approach proposed by Sarkar et al. [78], henceforth referred to as residual-based approach, followed by the description of our rank-based approach. Then, the subject systems used in the paper are described followed by our evaluation. The paper ends with a discussion on why a rank-based approach works; finally, we conclude. To assist other researchers, a reproduction package with all our scripts and data are available on GitHub.[1]

4.3 Problem Formalization

A configurable software system has a set X of configurations $x \in X$. Let x_i indicate the *ith* configuration option of configuration x, which takes the values from a finite domain $Dom(x_i)$. In general, x_i indicates either an (i) integer variable or a (ii) Boolean variable. The configuration space is thus $Dom(x_1) \times Dom(x_2) \times ... \times Dom(x_n)$, which is the Cartesian product of the domains, where $n = |x|$ is the number of configuration options of the system. Each configuration (x) has a corresponding performance measure $y \in Y$ associated with it. The performance measure is also referred to as dependent variable. We denote the performance measure associated

[1] https://github.com/ai-se/Reimplement/tree/cleaned_version

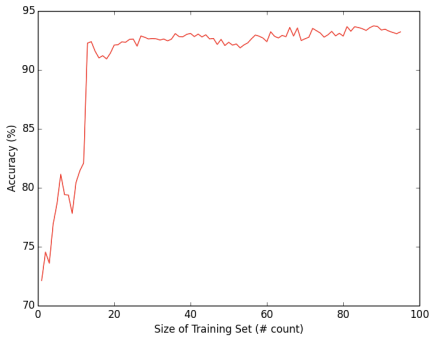

Figure 4.2 The relationship between the accuracy (in terms of MMRE) and the number of samples used to train the performance model of the Apache Web Server. Note that the accuracy does not improve substantially after 20 sample configurations.

with a given configuration by $y = f(x)$. We consider the problem of ranking configurations (x^*) that such that $f(x)$ is less than other configurations in the configuration space of X with few measurements.

$$f(x^*) \leq f(x), \quad \forall x \in X \setminus x^* \tag{4.1}$$

Our goal is to find the (near) optimal configuration of a system where it is not possible to build an accurate performance model as prescribed in earlier work.

4.4 Residual-based Approaches

In this section, we discuss the residual-based approaches for building performance models for configurable software systems. For further details, we refer to Sarkar et. al [78].

4.4.1 Progressive Sampling

When the cost of collecting data is higher than the cost of building a performance model, it is imperative to minimize the number of measurements required for model building. A learning curve shows the relationship between the size of the training set and the accuracy of the model. In Figure 4.2, the horizontal axis represents the number of samples used to create the performance model, whereas the vertical axis represents the accuracy (measured in terms of MMRE) of the model learned. Learning curves typically have a steep sloping portion early in the curve followed by a plateau late in the curve. The plateau occurs when adding data does not improve the accuracy of the model. As engineers, we would like to stop sampling as soon as the learning curve starts to flatten.

Figure 4.3 is a generic algorithm that defines the process of progressive sampling. *Progres-*

```
1   # Progressive Sampling
2   def progressive(training, testing, lives=3):
3     # For stopping criterion
4     last_score = −1
5     independent_vals = list()
6     dependent_vals = list()
7     for count in range(1, len(training)):
8       # Add one configuration to the training set
9       independent_vals += training[count]
10      # Measure the performance value for the newly
11      # added configuration
12      dependent_vals += measure(training_set[count])
13      # Build model
14      model = build_model(independent_vals, dependent_vals)
15      # Test Model
16      perf_score = test_model(model, testing, measure(testing))
17      # If current accuracy score is not better than
18      # the previous accuracy score, then loose life
19      if perf_score <= last_score:
20        lives −= 1
21      last_score = perf_score
22      # If all lives are lost, exit loop
23      if lives == 0: break
24    return model
```

Figure 4.3 Pseudocode of progressive sampling.

sive sampling starts by clearly defining the data used in the training set, called training pool, from which the samples would be selected (randomly, in this case) and then tested against the testing set. At each iteration, a (set of) data instance(s) of the training pool is added to the training set (Line 9). Once the data instances are selected from the training pool, they are evaluated, which in our setting means measuring the performance of the selected configuration (Line 12). The configurations and the associated performance scores are used to build the model (Line 14). The model is validated using the testing set[2], then, the accuracy is then computed. The accuracy can be quantified by any measure, such as MMRE, MBRE, Absolute Error, etc. In our setting, we assume that the measure is accuracy (higher is better). Once the accuracy score is calculated, it is compared with the accuracy score obtained before adding the new set of configurations to the training set. If the accuracy of the model (with more data) does not improve the accuracy when compared to the previous iteration (lesser data), then a life is lost. This termination criterion is widely used in the field of multi-objective optimization to determine degree of convergence [55].

[2]The testing data consist of the configurations as well as the corresponding performance scores.

```
# Projective Sampling                                                    1
def projective(training, testing, thres_freq=3):                         2
  collector = list()                                                     3
  independent_vals = list()                                              4
  dependent_vals = list()                                                5
  for count in range(1, len(training)):                                  6
    # Add one configuration to the training set                          7
    independent_vals += training[count]                                  8
    # Measure the performance value for the newly                        9
    # added configuration                                                10
    dependent_vals += measure(training_set[count])                       11
    # update feature frequency table                                     12
    T = update_frequency_table(training[count])                          13
    # Build model                                                        14
    model = build_model(independent_vals, dependent_vals)                15
    # Test Model                                                         16
    perf_score = test_model(model, testing, measure(testing))           17
    # Collect the the pair of |training set|                             18
    # and performance score                                              19
    collector += [count, perf_score]                                     20
    # minimum values of the feature frequency table                      21
    if min(T) >= thresh_freq: break                                      22
  return model                                                           23
```

Figure 4.4 Pseudocode of projective sampling.

4.4.2 Projective Sampling

One of the shortcomings of progressive sampling is that the resulting performance model achieves an acceptable accuracy only after a large number of iterations, which implies high modelling cost. There is no way to actually determine the cost of modelling until the performance model is already built, which defeats its purpose, as there is a risk of over-shooting the modelling budget and still not obtain an accurate model. *Projective* sampling addresses this problem by approximating the learning curve using a minimal set of initial sampling points (configurations), thus providing the stakeholders with an estimate of the modelling cost. Sarkar et. al [78] used projective sampling to predict the number of samples required to build a performance model. The initial data points are selected by randomly adding a constant number of samples (configurations) to the training set from the training pool. In each iteration, the model is built, and the accuracy of the model is calculated using the testing data. A feature-frequency heuristic is used as the termination criterion. The feature-frequency heuristic counts the number of times a feature has been selected and deselected. Sampling stops when the counts of features selected and deselected is, at least, at a predefined threshold (*thresh_freq*).

Figure 4.4 provides a generic algorithm for projective sampling. Similar to progressive sampling, projective sampling starts with selecting samples from the training pool and adding them to the training set (Line 8). Once the samples are selected, the corresponding configurations

are evaluated (Line 11). The feature-frequency table T is then updated by calculating the number of features that are selected and deselected in *independent_vals* (Line 13). The configurations and the associated performance values are then used to build a performance model, and the accuracy is calculated (Lines 15–17). The number of configurations and the accuracy score are stored in the collector, since our objective is to estimate the learning curve. $min(T)$ holds the minimum value of the feature selection and deselection frequencies in T. Once the value of $min(T)$ is greater than *thresh_freq*, the sampled points are used to estimate the learning curve. These points are used to search for a best-fit function that can be used to extrapolate the learning curve (there are several available, including Logarithmic, Weiss and Tian, Power Law and Exponential [78]). Once the best-fit function is found, it is used to determine the point of convergence.

4.5 Rank-based approach

Typically, performance models are evaluated based on the accuracy or error. The error can be computed using[3]:

$$MMRE = \frac{\mid predicted - actual \mid}{actual} \cdot 100 \tag{4.2}$$

The key idea in this paper is to use ranking as an approach for building regression models. There are a number of advantages of using a rank-based approach:

- For the use cases listed in the introduction, *ranking is the ultimate goal*. A user may just want to identify the top-ranked configurations rather than to rank the whole space of configurations. For example, a practitioner trying to optimize an Apache Web server is searching for a set of configurations that can handle maximum load, and is not interested in the whole configuration space.

- *Ranking is extremely robust* since it is only mildly affected by errors or outliers [54, 77]. Even though measures such as Mean Absolute Error are robust, in the configuration setting, a practitioner is often more interested in knowing the rank rather than the predicted performance scores.

- *Ranking reduces the number of training samples required to train a model*. We will demonstrate that the number of training samples required to find the optimal configuration using a rank-based approach is reduced considerably, compared to residual-based approaches which use MMRE.

[3] Aside: There has been a lot of criticism regarding MMRE, which shows that MMRE along with other accuracy statistics such as MMRE, MBRE has been shown to cause conclusion instability [29, 65, 66].

```
# rank−based approach
def rank_based(training, testing, lives=3):
  last_score = −1
  independent_vals = list()
  dependent_vals = list()
  for count in range(1, len(training)):
    # Add one configuration to the training set
    independent_vals += training[count]
    # Measure the performance value for the newly
    # added configuration
    dependent_vals += measure(training_set[count])
    # Build model
    model = build_model(independent_vals, dependent_vals)
    # Predicted performance values
    predicted_performance = model(testing)
    # Compare the ranks of the actual performance
    # scores to ranks of predicted performance scores
    actual_ranks = ranks(measure(testing))
    predicted_ranks = ranks(predicted_performance)
    mean_RD = RD(actual_ranks, predicted_ranks)
    # If current rank difference is not better than
    # the previous rank difference, then loose life
    if mean_rank_difference <= last_rank_difference:
      lives −= 1
    last_rank_difference = mean_RD
    # If all lives are lost, exit loop
    if lives == 0: break
  return model
```

Figure 4.5 Psuedocode of rank-based approach.

It is important to note that we aim at building a performance model similar to the accurate performance model building process used by prior work as described in Section 4.4. But instead of using residual measures of errors, as described in Equation 4.2, which depend on residuals ($r = y - f(x)$), [4] we use a rank-based measure. While training the performance model ($f(x)$), the configuration space is iteratively sampled (from the training pool) to train the performance model. Once the model is trained, the accuracy of the model is measured by sorting the values of $y = f(x)$ from 'small' to 'large', that is:

$$f(x_1) \leq f(x_2) \leq f(x_3) \leq ... \leq f(x_n). \tag{4.3}$$

The predicted rank order is then compared to the actual rank order. The accuracy is calculated using the mean rank difference:

$$accuracy = \frac{1}{n} \cdot \sum_{i=1}^{n} \left| rank(y_i) - rank(f(x_i)) \right| \tag{4.4}$$

[4] Refer to Section 4.3 for definitions.

This measure simply counts how many of the pairs in the test data were ordered incorrectly by the performance model $f(x)$ and measures the average of magnitude of the ranking difference.

In Figure 4.5, we list a generic algorithm for our rank-based approach. Sampling starts by selecting samples randomly from the training pool and by adding them to the training set (Line 8). The collected sample configurations are then evaluated (Line 11). The configurations and the associated performance measure are used to build a performance model (Line 13). The generated model (CART, in our case) is used to predict the performance measure of the configurations in the testing pool (Line 16). Since the performance value of the testing pool is already measured, hence known, the ranks of the actual performance measures, and predicted performance measure are calculated. (Lines 18–19). The actual and predicted performance measure is then used to calculate the rank difference using Equation 4.4. If the rank difference of the model (with more data) does not decrease when compared to the previous generation (lesser data), then a life is lost (Lines 23–24). When all lives are expired, sampling terminates (Line 27).

The motivation behind using the parameter *lives* is: to detect convergence of the model building process. If adding more data does not improve the accuracy of the model (for example, in Figure 4.2 the accuracy of the model generated does not improve after 20 samples configuration), the sampling process should terminate to avoid resource wastage; see also Section 4.9.4.

4.6 Subject Systems

To compare residual-based approaches with our rank-based approach, we evaluate it using 21 test cases collected in 9 open-source software systems[5].

1. SS1 X264 is a video-encoding library that encodes video streams to H.264/MPEG-4 AVC format. We consider 16 features, which results in 1152 valid configurations.

2. SS2 BERKELEY DB (C) is an embedded key-value-based database library that provides scalable high performance database management services to applications. We consider 18 features resulting in 2560 valid configurations.

3. SS3 SQLITE is the most popular lightweight relational database management system. It is used by several browsers and operating systems as an embedded database. In our experiments, we consider 39 features that give rise to more than 3 million valid configurations.

[5]For more details on the subject systems and configurations options refer to http://tiny.cc/3wpwly

4. SS4 WGET is a software package for retrieving files using HTTP, HTTPS, and FTP. It is a non-interactive command line tool. In our experiments, we consider 16 features, which result in 188 valid configurations.

5. SS5 LRZIP or Long Range ZIP is a compression program optimized for large files, consisting mainly of an extended rzip step for long distance redundant reduction and a normal compressor step. We consider 19 features, which results in 432 valid configurations.

6. SS6 DUNE or the Distributed and Unified Numerics Environment, is a modular C++ library for solving partial differential equations using grid-based methods. We consider 11 feature resulting in 2305 valid configurations.

7. SS7 HSMGP or Highly Scalable MG Prototype is a prototype code for benchmarking Hierarchical Hybrid Grids data structures, algorithms, and concepts. It was designed to run on super computers. We consider 14 features resulting in 3456 valid configurations.

8. SS8 APACHE HTTP Server is a Web Server; we consider 9 features resulting in 192 valid configurations.

In addition to these 8 subject systems, we also consider Apache Storm, a distributed system, in several scenarios. The datasets were obtained from the paper by Jamshidi et al. [49]. The experiment considers three benchmarks namely:

- WordCount (WC) counts the number of occurences of the words in a text file.
- RollingSort (RS) implements a common pattern in real-time analysis that performs rolling counts of messages.
- SOL (SOL) is a network intensive topology, where the message is routed through an inter-worker network.

The experiments were conducted with all the above mentioned benchmarks on 5 cloud clusters. The experiments also contain measurement variabilities, the WC experiments were also carried out on multi-tenant cluster, which were shared with other jobs. For example, WC+RS means WC was deployed in a multi-tenant cluster with RS running on the same cluster. As a result, not only latency increased but also variability became greater. The environments considered in our experiments are:

9. SS9 WC-6D-THROUGHPUT is an environment configuration where WC is executed by varying 6 features resulting in 2879 configurations; throughput is calculated.

10. SS10 RS-6D-THROUGHPUT is an environment configuration where RS is run by varying 6 features which results in 3839 configurations; the throughput is measured.

11. SS11 WC-6D-LATENCY is an environment configuration where WC is executed by varying 6 features resulting in 2879 configurations; latency is calculated.

12. SS12 RS-6D-LATENCY is an environment configuration where RS is executed by varying 6 features, which results in 3839 configurations; latency is measured.

13. SS13 WC+RS-3D-THROUGHPUT is an environment configuration where WC is run in a multi-tenant cluster along with RS. WC is executed by varying 3 features resulting in 195 configurations; throughput is measured.

14. SS14 WC+SOL-3D-THROUGHPUT is an environment configuration where WC is run in a multi-tenant cluster along with SOL. WC is executed by varying 3 features resulting in 195 configurations; throughput is measured.

15. **SS15** WC+WC-3D-THROUGHPUT is an environment configuration where WC is run in a multi-tenant cluster along with WC. WC is executed by varying 3 features resulting in 195 configurations; throughput is measured.

16. **SS16** SOL-6D-THROUGHPUT is an environment configuration where SOL is executed by varying 6 features resulting in 2865 configurations; throughput is measured.

17. **SS17** WC-WC-3D-THROUGHPUT is an environment configuration where WC is executed by varying 3 features resulting in 755 configurations; throughput is calculated.

18. **SS18** WC+SOL-3D-LATENCY is an environment configuration where WC is run in a multi-tenant cluster along with SOL. The WC is executed by varying 3 features resulting in 195 configurations; latency is measured.

19. **SS19** WC+WC-3D-LATENCY is an environment configuration where WC is run in a multi-tenant cluster along with WC. The WC is executed by varying 3 features resulting in 195 configurations; latency is measured.

20. **SS20** SOL-6D-LATENCY is an environment configuration where SOL is executed by varying 6 features resulting in 2861 configuration setting; latency is measured.

21. **SS21** WC+RS-3D-LATENCY is an environment configuration where WC is run in a multi-tenant cluster along with RS. WC is executed by varying 3 features resulting in 195 configurations; latency is measured.

4.7 Evaluation

4.7.1 Research Questions

In the past, configuration ranking required an accurate model of the configuration space, since an inaccurate model implicitly indicates that the model has missed the trends of the configuration space. Such accurate models require the evaluation/measurement of hundreds of configuration options for training [36, 78, 86]. There are also cases where building an accurate model is not possible, as shown in Figure 4.1 (right side).

Our research questions are geared towards finding optimal configurations when building an accurate model of a given software system is not possible. As our approach relies on ranking, our hypothesis is that we would be able to find the (near) optimal configuration using our rank-based approach while using fewer measurements, as compared to an accurate model learnt using residual-based approaches[6].

Our proposal is to embrace rank preservation but with inaccurate models and to use these models to guide configuration rankings. Therefore, to assess the feasibility and usefulness of the inaccurate model in configuration rankings, we consider the following:

- Accurate rankings found by inaccurate models using a rank-based approach, and
- the effort (number of measurements) required to build an inaccurate model.

The above considerations lead to two research questions:

RQ1*: Can inaccurate models accurately rank configurations?* Here, the optimal configurations found using an inaccurate model are compared to the more accurate models generated using residual-based approaches. The accuracy of the models is calculated using MMRE (from Equation 4.2).

RQ2*: How expensive is a rank-based approach (in terms of how many configurations must be executed)?* It is expensive to build accurate models, and our goal is to minimize the number of measurements. It is important to demonstrate that we can find optimal configurations of a system using inaccurate models as well as reducing the number of measurements.

4.7.2 Experimental Rig

For each subject system, we build a table of data, one row per valid configuration. We then run all configurations of all systems and record the performance scores (i.e., that are invoked by a

[6]Aside: It is worth keeping in mind that the approximation error in a model does not always harm. A model capable to smoothing the complex landscape of a problem can be beneficial for the search process. This sentiment has been echoed in the evolutionary algorithm literature as well [58].

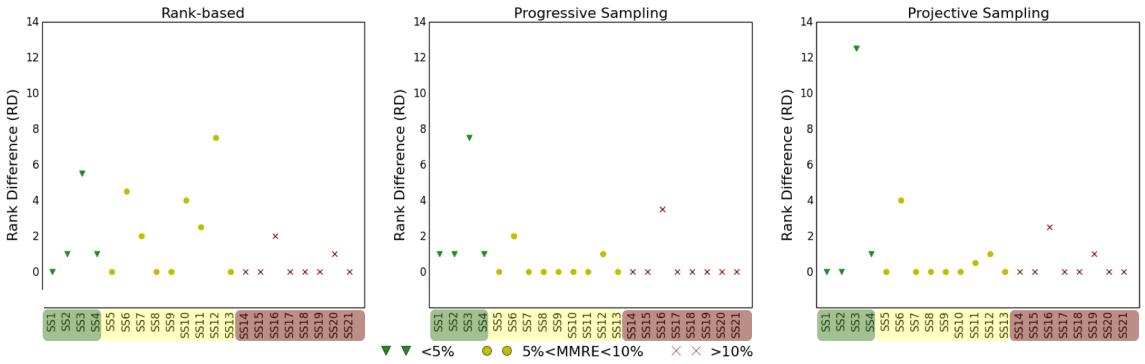

Figure 4.6 The rank difference of the prediction made by the model built using residual-based and rank-based approaches. Note that the y-axis of this chart rises to some very large values; e.g., SS3 has over three million possible configurations. Hence, the above charts could be summarised as follows: "the rank-based approach is surprisingly accurate since the rank difference is usually close to 0% of the total number of possible configurations". In this figure, ▼, ●, and × represent the subject systems (using the technique mentioned at the top of the figure), in which we could build a prediction model, where accuracy is $< 5\%$, $5\% < MMRE < 10\%$, and $> 10\%$ respectively. This is based on Figure 4.1.

benchmark). The exception is SQLite, for which we measure only 4400 configurations corresponding to feature-wise and pair-wise sampling and additionally 100 random configurations. (This is because SQLite has 3,932,160 possible configurations, which is an impractically large number of configurations to test.) To this table, we added a column showing the performance score obtained from the actual measurements for each configuration. Note that, while answering the research questions, we ensure that we never test any prediction model on the data that we used to learn the model. Next, we repeat the following procedure 20 times.

To answer the research questions, we split the datasets into training pool (40%), testing pool (20%), and validation pool (40%). The experiment is conducted in the following way:

- Randomize the order of rows in the training data
- **Do**
  - Select one configuration (by sampling with replacement) and add it to the training set
  - Determine the performance scores associated with the configuration. This corresponds to a table look-up, but would entail compiling or configuring and executing a system configuration in a practical setting.
  - Using the training set and the accuracy, build a performance model using CART.
  - Using the data from the testing pool, assess the accuracy either using MMRE (as described in Equation 4.2) or the rank difference (as described in Equation 4.4).

- **While** the accuracy is greater or equal to the threshold determined by the practitioner (rank difference in the case of our rank-based approach and MMRE in the case of the residual-based approaches).

Once the model has been iteratively trained, it is used on the data in the validation pool. Please note, the learner has not been trained on the validation pool. RQ1 relates the results found by the inaccurate performance models (rank-based) to more accurate models (residual-based). We use the absolute difference between the ranks of the configurations predicted to be the optimal configuration and the actual optimal configuration. We call this measure rank difference (*RD*).

$$RD = \left| rank(actual_{optimal}) - rank(predicted_{optimal}) \right| \tag{4.5}$$

Ranks are calculated by sorting the configurations based on their performance scores. The configuration with the least performance score, $rank(actual_{optimal})$, is ranked 1 and the one with highest score is ranked as N, where N is the number of configurations.

4.8 Results

4.8.1 RQ1: *Can inaccurate models accurately rank configurations?*

Figure 4.6 shows the *RD* of the predictions built using the rank-based approach and residual-based approaches [7] by learning from 40% of the training set and iteratively adding data to the training set (from the training pool), while testing against the testing set (20%). The model is then used to find the optimal configuration among the configurations in the validation dataset (40%). The horizontal axis shows subject systems. The vertical axis shows the rank difference (*RD*) from Equation 4.5. In this figure:

- The perfect performance model would be able to find the optimal configuration. Hence, the ideal result of this figure would be if all the points lie on the $y = 0$ or the horizontal axis. That is, the model was able to find the optimal configuration for all the subject systems (*RD* = 0).
- The markers ▼, ●, and × represent the software systems where a model with a certain accuracy can be built, measured in MMRE is $<$ 5%, 5% $<$MMRE$<$ 10%, and $>$ 10% respectively.

Overall, in Figure 4.6, we find that:

[7] The MMRE scores for the models http://tiny.cc/bu14iy

Rank	Treatment	Median	IQR	Median and IQR chart
SS1				
1	Projective	0.0	0.0	
1	Progressive	0.0	1.0	
2	Rank-based	2.0	8.0	
SS2				
1	Projective	0.0	1.0	
2	Rank-based	1.0	6.0	
2	Progressive	2.0	18.0	
SS3				
1	Progressive	10.0	17.0	
2	Projective	15.0	139.0	
2	Rank-based	21.0	40.0	
SS11				
1	Progressive	0.0	1.0	
2	Rank-based	1.0	3.0	
2	Projective	1.0	2.0	
SS20				
1	Projective	0.0	1.0	
1	Progressive	0.0	1.0	
2	Rank-based	5.0	19.0	

Figure 4.7 Median rank difference of 20 repeats. Median ranks is the rank difference as described in Equation 4.5, and IQR the difference between 75th percentile and 25th percentile found during multiple repeats. Lines with a dot in the middle (⊸), show the median as a round dot within the IQR. All the results are sorted by the median rank difference: a lower median value is better. The left-hand column (*Rank*) ranks the various techniques for example, when comparing various techniques for SS1, a rank-based approach has a different rank since their median rank difference is statistically different.

- The × represents software systems where the performance models are inaccurate ($>$ 10% MMRE) and still can be used for ranking configurations, since the rank difference of these systems is always less than 4. Hence, even an inaccurate performance model can rank configurations.
- All three models built using both rank-based and residual-based approaches are able to find near optimal configurations. For example, progressive sampling for SQLite predicted the configuration whose performance score is ranked 9th in the testing set. This is good enough since progressive sampling is able to find the 9th most performant configuration among 1861 configurations[8].
- The mean rank difference of the $predicted_{optimal}$ is 1.4, 0.77, and 0.93[9] for the rank-based approach, progressive sampling, and projective sampling respectively. Thus, a performance model can be used to rank configurations.

We claim that the rank of the optimal configuration found by the residual and rank-based approaches is the same. To verify that the similarity is statistically significant, we further

[8]Since we test only on 40% of the possible configuration space (40% of 4653).
[9]The median rank difference is 0 for all the approaches.

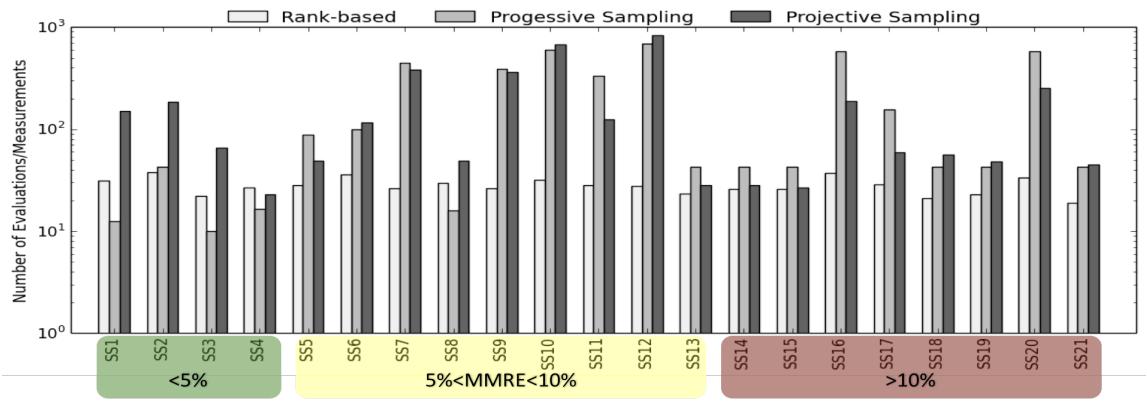

Figure 4.8 [Number of measurements required to train models by rank-based methods and other approaches.] Number of measurements required to train models by different approaches. The software systems are ordered based on the accuracy scores of Figure 4.1.

studied the results using non-parametric tests, which were used by Arcuri and Briand at ICSE'11 [64]. For testing statistical significance, we used a non-parametric bootstrap test with 95% confidence [23], followed by an A12 test to check that any observed differences were not trivially small effects; that is, given two lists X and Y, count how often there are larger numbers in the former list (and if there are ties, add a half mark): $a = \forall x \in X, y \in Y \frac{\#(x>y)+0.5\cdot\#(x=y)}{|X|\cdot|Y|}$ (as per Vargha [96], we say that a "small" effect has $a < 0.6$). Lastly, to generate succinct reports, we use the Scott-Knott test to recursively divide our approaches. This recursion used A12 and bootstrapping to group together subsets that are (a) not significantly different and are (b) not just a small effect different to each other. This use of the Scott-Knott test is endorsed by Mittas and Angelis [64] and by Hassan et al. [34].

In Figure 4.7, the table shows the Scott-Knott ranks for the three approaches. The quartile charts are the Scott-Knott results for our subject systems, where the rank-based approach did not do as well as the residual-based approaches [10]. For example, the statistic test for SS1 shows that the ranks of the optimal configuration by the rank-based approach was statistically different from the ones found by the residual-based approaches. We think this is reasonably close since the median rank found by the rank-based approach is 2 out of 460 configurations, whereas residual-based approaches find the optimal configurations with a median rank of 0. As our motivation was to find optimal configurations for software systems for which performance models were difficult or infeasible to build, we look at SS20. If we look at the Skott-Knott chart for SS20, the median rank found by the rank-based approach is 5, whereas the residual-based approaches could find the optimal configurations very consistently (IQR=1). But as engineers, we feel that this is close because we are able to find the 5th best configuration using

[10] For complete Skott-Knott charts, refer to http://geekpic.net/pm-1GUTPZ.html.

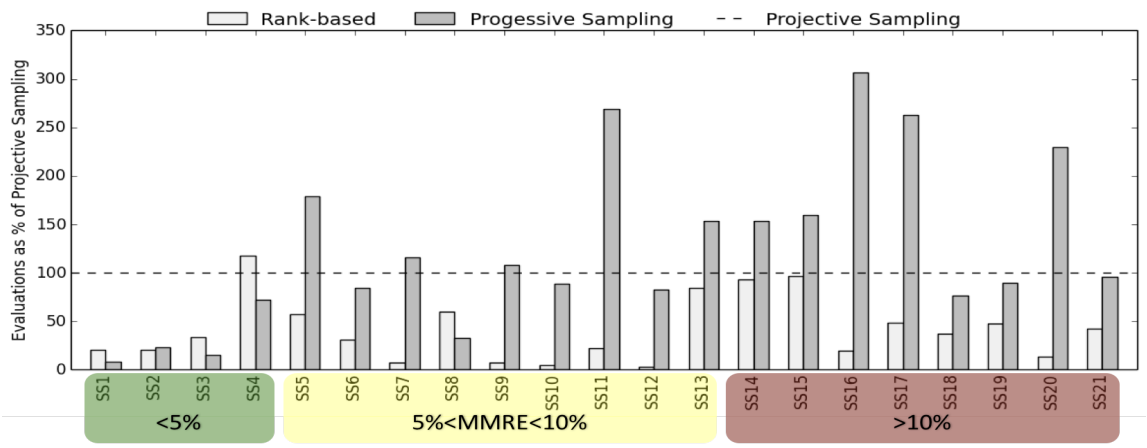

Figure 4.9 The percentage of measurement used for training models with respect to the number of measurements used by projective sampling (dashed line). The rank-based approach uses almost 10 times less measurements than the residual-based approaches. The subject systems are ordered based on the accuracy scores of Figure 4.1.

33 measurements compared to 251 and 576 measurements used for progressive and projective sampling, respectively. Overall, our results indicate that:

> A rank preserving (probably inaccurate) model can be useful in finding (near) optimal configurations of a software system using a rank-based approach.

4.8.2 RQ2: *How expensive is a rank-based approach?*

To answer the question of whether we can recommend the rank-based approach as a cheap method for finding the optimal configuration, it is important to demonstrate that rank-based models are indeed cheap to build. In our setting, the cost of a model is determined by the number of measurements required to train the model. Figure 4.8 demonstrates this relationship. The vertical axis denotes the number of measurements in log scale and horizontal axis represents the subject systems.

In the systems SS1–SS4 (green band), the number of measurements required by the rank-based approach is less than for projective sampling and more than for progressive sampling. This is because the subject systems are easy to model. For the systems SS4–SS13 (yellow band), the number of measurements required to build models using the rank-based approach is less than residual-based approaches, with the exception of SS8. Note that, as building accurate models becomes difficult, the difference between the number of measurements required by the rank-based approach and residual-based approaches increases. For the systems SS14–SS21 (red

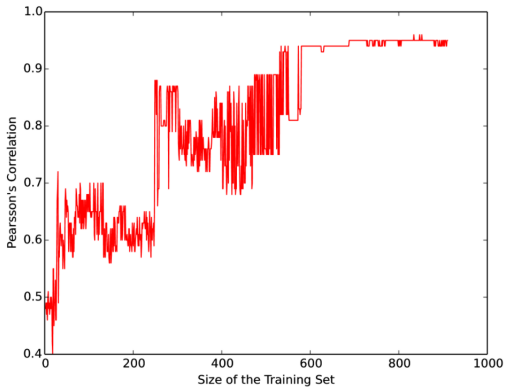

Figure 4.10 The correlation between actual and predicted performance values (not ranks) increases as the training set increases. This is evidence that the model does learn as training progresses.

band), the number of measurements required by the rank-based approach to build a model is always less than for residual-based approaches, with significant gains for SS19–SS21.

In Figure 4.9, the ratio of the measurements of different approaches are represented as the percentage of number of measurements required by projective sampling – since it uses the most measurements in 50% of the subject systems. For example, in SS5, the number of measurements used by progressive sampling is twice as much as used by projective sampling, whereas the rank-based approach uses half of the total number of measurements used by projective sampling. We observe that the number of measurements required by the rank-based approach is much lower than for the residual-based approaches, with the only exceptions of SS4 and SS8. We argue that such outliers are not of a big concern since the motivation of rank-based approach is to find optimal configurations for software systems, where an accurate model is infeasible.

To summarize, the number of samples required by the rank-based approach is much smaller than for residual-based approaches. There are $\frac{4}{21}$ cases where residual-based approaches (progressive sampling) use fewer measurements. The subject systems where residual-based approaches use fewer measurements are systems where accurate models are easier to build (green and yellow band):

> Models built using the rank-based approach require fewer measurements than residual-based approaches. In $\frac{8}{21}$ of the cases, the number of measurements is an order of magnitude smaller than residual-based approaches.

4.9 Discussion

4.9.1 How is rank difference useful in configuration optimization?

The objective of the modelling task considered here is to assist practitioners to find optimal configuration/s. We use a performance model, like traditional approaches, to solve this problem, but rather than using a residual-based accuracy measure, we use rank difference. Rank difference can also be thought as a correlation-based metric, since rank difference essentially tries to preserve the ordering of performance scores.

In our rank-based approach, we do not train the model to predict the dependent values of the testing set. But rather, we attempt to train the model to predict the dependent value that is correlated to the actual values of the testing set. So, during iterative sampling based on the rank-based approach, we should see an increase in the correlation coefficient as the training progresses. Figure 4.10 shows how the correlation between actual and the predicted values increases as the size of the training set grows. From the combination of Figure 4.1 and Figure 4.6, we see that even an inaccurate model can be used to find an optimal configuration.[11]

4.9.2 Can inaccurate models be built using residual-based approaches?

We have already shown that a rank preserving (probably inaccurate) model is sufficient to find the optimal configuration of a given system. MMRE can be used as a stopping criterion, but as we have seen with residual-based approaches, they require a larger training set and hence are not cost effective. This is because, with residual-based approaches, unlike our rank-based approach, it is not possible to know when to terminate sampling. It may be noted that rank difference can be easily replaced with a correlation-based approach such as Pearson's or Spearman's correlation.

4.9.3 Can we predict the complexity of a system to determine which approach to use?

From our results, we observe that a rank-based approach is not as effective as the residual-based approaches for software systems that can be modelled accurately (green band). Hence, it is important to distinguish between software systems where the rank-based approach is suitable and software systems where residual-based approaches are suitable. This is relatively straightforward since both rank-based and residual-based approaches use random sampling to select the samples. The primary difference between the approaches is the termination criterion. The rank-based approach uses rank difference as the termination criterion whereas residual-based

[11] This also shows how a correlation-based measure can be used as a stopping criterion.

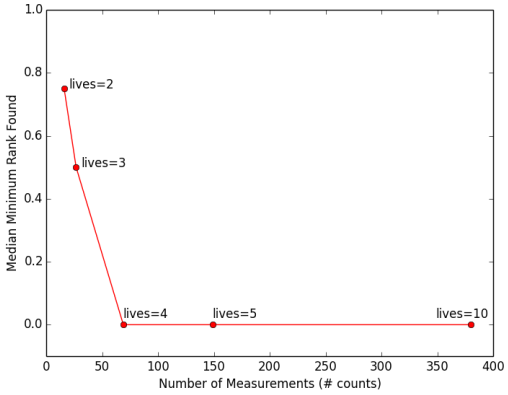

Figure 4.11 The trade-off between the number of measurements or size of the training set and the number of *lives*.

approaches use criterion based on MMRE, etc. Hence, it is possible to use both techniques simultaneously. If a practitioner observes that the accuracy of the model during the building process is high (as in case of SS2), residual-based approaches would be preferred. Conversely, if the accuracy of the model is low (as in the case of , SS21), the rank-based approach would be preferred.

4.9.4 What is the trade-off between the number of lives and the number of measurements?

Our rank-based approach requires that the practitioner defines a termination criterion (*lives* in our setting) before the sampling process commences, which is a similar to progressive sampling. The termination criterion preempts the process of model building based on an accuracy measure. The rank-based approach uses rank difference as the termination criterion, whereas residual-based approaches use residual-based measures. In our experiments, the number of measurements or the size of the training set depends on the termination criterion (*lives*). An early termination of the sampling process would lead to a sub-optimal configuration, while late termination would result in resource wastage. Hence, it is important to discuss the trade-off between the number of *lives* and the number of measurements. In Figure 4.11, we show the trade-off between the median minimum ranks found and the number of measurements (size of training set). The markers of the figure are annotated with the values of *lives*. The trade-off characterizes the relationship between two conflicting objectives, for example, point (*lives*=2) requires very few measurements but the minimum rank found is the highest, whereas point (*lives*=10) requires a large number of measurements but is able to find the best performing configuration. Note, this curve is an aggregate of the trade-off curves of all the software systems

discussed in this paper[12]. Since our objective is to minimize the number of measurements while reducing rank difference, we assign the value of 3 to *lives* for the purposes of our experiments.

4.10 Reliability and Validity

Reliability refers to the consistency of the results obtained from the research. For example, how well can independent researchers reproduce the study? To increase external reliability, we took care to either clearly define our algorithms or use implementations from the public domain (scikit-learn) [81]. Also, all data and code used in this work are available on-line.[13]

Validity refers to the extent to which a piece of research actually investigates what the researcher purports to investigate [85]. *Internal validity* is concerned with whether the differences found in the treatments can be ascribed to the treatments under study.

For SQLite, we cannot measure all possible configurations in reasonable time. Hence, we sampled only 100 configurations to compare prediction and actual values. We are aware that this evaluation leaves room for outliers and that measurement bias can cause false interpretations [62]. Since we limit our attention to predicting performance for a given workload, we did not vary benchmarks.

We aimed at increasing *external validity* by choosing subject systems from different domains with different configuration mechanisms. Furthermore, our subject systems are deployed and used in the real world.

4.11 Conclusion

Configurable systems are widely used in practice, but it requires careful tuning to adapt them to a particular setting. State-of-the-art approaches use a residual-based technique to guide the search for optimal configurations. The model-building process involves iterative sampling used along with a residual-based accuracy measure to determine the termination point. These approaches require too many measurements and hence are expensive. To overcome the requirement of a highly accurate model, we propose a rank-based approach, which requires a lesser number of measurements and finds the optimal configuration just by using the ranks of configurations as an evaluation criterion.

Our key findings are the following. First, a highly accurate model is not required for configuration optimization of a software system. We demonstrated how a rank-preserving (possibly even inaccurate) model can still be useful for ranking configurations, whereas a model with accuracy as low as 26% can be useful for configuration ranking. Second, we introduce

[12]Complete trade-off curves can be found at http://tiny.cc/kgs2iy or http://tiny.cc/rank-param.
[13] https://github.com/ai-se/Reimplement/tree/cleaned_version

a new rank-based approach that can be used to decide when to stop iterative sampling. We show how a rank-based approach is not trained to predict the raw performance score but rather learns the model, so that the predicted values are correlated to actual performance scores.

To compare the rank-based approach to the state-of-the-art residual-based approaches (projective and progressive sampling), we conducted a number of experiments on 9 real-world configurable systems to demonstrate the effectiveness of our approach. We observed that the rank-based approach is effective to find the optimal configurations for most subject systems while using fewer measurements than residual-based approaches. The only exceptions are subject systems for which building an accurate model is easy, anyway.

Chapter 5

Finding faster configurations using FLASH

This chapter originally appeared as Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. Finding faster configurations using FLASH. In IEEE Transactions on Software Engineering 2018.

5.1 Abstract

Finding good configurations for a software system is often challenging since the number of configuration options can be very large. Software engineers often make poor choices about configuration or, even worse, they usually use a sub-optimal configuration in production, which leads to inadequate (or sub-optimal) performance. To assist engineers to find the (near) optimal configuration, this paper introduces FLASH, a sequential model-based method, which sequentially explores the configuration space by reflecting on the configurations evaluated till date to determine the next best configuration to explore. FLASH scales up to software systems that defeat the prior state of the art model-based methods in this area. FLASH runs much faster than existing methods and can solve both single-objective and multi-objective optimization problems. The central insight of this paper is to use the prior knowledge (gained from prior runs) to choose the next promising configuration. This strategy reduces the effort (measured in terms of the number of measurements) required to find the (near) optimal configuration. We evaluate FLASH using 30 scenarios based on 7 software systems to demonstrate that FLASH saves effort in 100% and 80% of cases in single-objective and multi-objective problems respectively by up to several orders of magnitude.

5.2 Introduction

Most software systems available today are *configurable*; that is, they can be easily adjusted to achieve a wide range of functional or non-functional (e.g., energy or performance) properties. Once that configuration space becomes large, it becomes tough for developers, maintainers, and users to keep track of the interactions between all various configuration options. Section 2 of this paper offers more details on many of the problems seen with software configuration. In summary:

- Many software systems have poorly chosen defaults [42, 94]. Hence, it is useful to seek better configurations.
- Understanding, the configuration space of software systems with large configuration space, is challenging [103].
- Exploring more than just a handful of configurations is usually infeasible due to long benchmarking time [108].

This paper describes FLASH, a novel way to find (near) optimal configurations for a software system (for a given workload). FLASH is a sequential model-based method [2, 12, 88] that reflects on the evidence (configurations) to date to select the best configuration to measure next. FLASH uses fewer evaluations to finds (near) optimal configurations similar to more expensive prior work [36, 37, 69, 70, 78].

Prior work in this area primarily used two strategies. Firstly, they used a machine learning method to model the configuration space. The model is built sequentially, where new configurations are sampled randomly, and the quality or accuracy of the model is measured using a holdout set. The size of the holdout set in some cases could be a 20% [70] and need to be evaluated before even the machine learning model is built. This strategy makes these methods not suitable in a practical setting since the generating holdout set can be (very) expensive. Secondly, the sequential model-based technique used in prior work builds the model using Gaussian Process Models (GPM) to reflect on the configurations explored (or evaluated) till date [111]. However, GPMs do not scale well for software systems with more than a dozen configuration options.

The key idea of FLASH is to build a performance model that is just accurate enough to allow differentiating between (near) optimal configuration from the rest of the configuration space. Tolerating the inaccuracy of the model is useful to reduce the cost (measured in terms of the number of configurations evaluated) and the time required to find the (near) optimal configuration. To increase the scalability of methods using GPM, FLASH replaces the GPMs with a fast and scalable decision tree learner.

Table 5.1 Configuration problems explored in this paper. The abbreviations of the systems (Abbr) are sorted in the order of the number of configuration options of the system. The column #Config Options represent the number of configuration options of the software system and #Configurations represents the total number of configurations of the system. See http://tiny.cc/flash_systems/ for more details.

Family	Software Systems	Objectives	#Config Options	Configuration Options	Description	Abbr	# Configurations	Prev Used
Stream Processing Systems	wc-c1-3d	Throughput / Latency	3	max spout, spliters, counters	Word Count is executed by varying 3 config -urations of Apache Storm on cluster C1	SS-A1 / SS-A2	1343	[49]
	wc-c3-3d	Throughput / Latency	3	max spout, spliters, counters	Word Count is executed by varying 3 configu -rations of Apache Storm on cluster C3	SS-C1 / SS-C2	1512	
	wc+wc-c4-3d	Throughput / Latency	3	max spout, spliters, counters	Word Count is executed, collocated with Word Counttask, by varying 3 configurations of Apache Storm on cluster C3	SS-D1 / SS-D2	195	
	wc-c4-3d	Throughput / Latency	3	max spout, spliters, counters	Word Count is executed by varying 3 config -urations of Apache Storm on cluster C4	SS-E1 / SS-E2	756	
	wc+rs-c4-3d	Throughput / Latency	3	max spout, spliters, counters	Word Count is executed, collocated with Rolling Sort task, by varying 3 configurations of Apache Storm on cluster C4	SS-F1 / SS-F2	196	
	wc+sol-c4-3d	Throughput / Latency	3	max spout, spliters, counters	Word Count is executed, collocated with SOL task, by varying 3 configurations of Apache Storm on cluster C3	SS-G1 / SS-G2	195	
	wc-5d-c5	Throughput / Latency	5	spouts, splitters, counters,buffer-size, heap	Word Count is executed by varying 5 config -urations of Apache Storm on cluster C3	SS-I1 / SS-I2	1080	
	rs-6d-c3	Throughput / Latency	6	spouts, max spout, sorters, emitfreq, chunksize, message size	Rolling Sort is executed by varying 6 config -urations of Apache Storm on cluster C3	SS-J1 / SS-J2	3839	
	wc-6d-c1	Throughput / Latency	6	spouts, max spout, sorters,emitfreq, chunksize,,message size	Word Count is executed by varying 6 configurations of Apache Storm on cluster C1	SS-K1 / SS-K2	2880	
FPGA	sort-256	Area / Throughput	3	Not specified	The design space consists of 206 different hardware implementations of a sorting network for 256 inputs	SS-B1 / SS-B2	206	[111]
	noc-CM-log	Energy / Runtime	4	Not specified	The design space consists of 259 different implementations of a tree-based network-on-chip, targeting application specific circuits (ASICs)	SS-H1 / SS-H2	259	
Compiler	llvm	Performance / Memory Footprint	11	time passes, gvn, instcombine,inline, ... , ipsccp, iv users, licm	The design space consists of 1023 different compiler settingsfor the LLVM compiler framework. Each setting is specified byd = 11 binary flags.	SS-L1 / SS-L2	1023	
Mesh Solver	Trimesh	# Iteration / Time to Solutions	13	F, smoother, colorGS,relaxParameter, V, Jacobi, line, zebraLine, cycle, alpha, beta,preSmoothing, postSmoothing	Configuration space of Trimesh, a library to manipulate triangle meshes	SS-M1 / SS-M2	239,260	[86]
Video Encoder	X264-DB	PSNR / Energy	17	no mbtree, no asm, no cabac, no scenecut,..., keyint, crf, scenecut, seek, ipratio	Configuration space of X-264 a video encoder	SS-N1 / SS-N2	53,662	
Seismic Analysis Code	SaC	Compile-Exit / Compile-Read	59	extrema, enabledOptimizations, disabledOptimizations,ls, dcr, cf, lir, inl, lur, wlur, ...maxae, initmheap, initwheap	Configuration space of SaC	SS-O1 / SS-O2	65,424	

The novel contributions of the paper are:

- We show that FLASH can solve single-objective performance configuration optimization problems using an order of magnitude fewer measurements than state of the art (Section 5.8.1). This is a critical feature because, as discussed in the next section, it can be very slow to sample multiple properties of modern software systems, when each such sample requires (say) to compile and benchmark the corresponding software system.
- We adapt FLASH to multi-objective performance configuration optimization problems.
- We show that FLASH overcomes the shortcomings of prior work and achieves similar performance and scalability, with greatly reduced runtimes (Section 5.8.2).
- Background material, a replication package, all measurement data, and the open source version of FLASH are available at http://tiny.cc/flashrepo/.

The rest of the paper is structured as follows: Section 2 motivates this work. Section 3 describes the problem formulation and the theory behind Bayesian optimization. Section 4 describes the prior work in software performance configuration optimization followed by the core algorithm of FLASH in Section 5. In Section 6, we present our research questions along with experimental settings used to answer them. In Section 7, we apply FLASH on single objective performance configuration optimization and multi-objective performance configuration optimization. The paper ends with a discussion on various aspects of FLASH, and finally, we conclude along with future work.

5.3 Performance configuration optimization for Software

This section motivates our research by reviewing the numerous problems associated with software configuration.

Many researchers report that modern software systems come with *a daunting number of configuration options*. For example, the number of configuration options in Apache (a popular web server) increased from 150 to more than 550 configuration options within 16 years [103]. Van Aken et al. [94] also reports a similar trend. They indicate that, in over 15 years, the number of configuration options of POSTGRES and MYSQL increased by a factor of three and six, respectively. This is troubling since Xu et al. [103] report that developers tend to ignore over 80% of configuration options, which leaves considerable optimization potential untapped and induces major economic cost [103].[1] For illustration, Figure 5.1 offer examples of the kinds of configuration options seen in software systems.

[1]The size of the configuration space increases exponentially with the number of configuration options.

Another problem with configurable systems is the issue of *poorly chosen default configurations*. Often, it is assumed that software architects provide useful default configurations of their systems. This assumption can be very misleading. Van Aken et al. [94] report that the default MySQL configurations in 2016 assume that it will be installed on a machine that has 160MB of RAM (which, at that time, was incorrect by, at least, an order of magnitude). Herodotou et al. [42] show how standard settings for text mining applications in Hadoop result in worst-case execution times. In the same vein, Jamshidi et al. [49] reports for text mining applications on Apache storm, the throughput achieved using the worst configuration is *480 times slower* than the throughput achieved by the best configuration.

Yet another problem is that *exploring benchmark sets for different configurations is very slow*. Wang et al. [99] comments on the problems of evolving a test suite for software if every candidate solution requires a time-consuming execution of the entire system: such test suite generation can take weeks of execution time. Zuluaga et al. [110] report on the cost of analysis for software/hardware co-design: "synthesis of only one design can take hours or even days".

The challenges of having numerous configuration options are just *not limited to software systems*. The problem to find an optimal set of configuration options is pervasive and faced in numerous other sub-domains of computer science and beyond. In software engineering, software product lines–where the objective is to find a product which (say) reduces cost, defects [13, 41]—has been widely studied. The problem of configuration optimization is present in domains, such as machine learning, cloud computing, and software security.

The area of *hyper-parameter optimization* (a.k.a. parameter tuning) is very similar to the performance configuration optimization problem studied in this paper. Instead of optimizing the performance of a software system, the hyper-parameter method tries to optimize the performance of a machine learner. Hyper-parameter optimization is an active area of research in various flavors of machine learning. For example, Bergstra and Bengiol [3] showed how random search could be used for hyper-parameter optimization of high dimensional spaces. Recently, there has been much interest in hyper-parameter optimization applied to the area of software analytics [1, 30–32, 92].

Another area of application for performance configuration optimization is *cloud computing*. With the advent of big data, long-running analytics jobs are commonplace. Since different analytic jobs have diverse behaviors and resource requirements, choosing the correct virtual machine type in a cloud environment has become critical. This problem has received considerable interest, and we argue, this is another useful application of performance configuration optimization — that is, optimize the performance of a system while minimizing cost [2, 16, 97, 104, 109].

As a sideeffect of the wide-spread adoption of cloud computing, the *security* of the instances or virtual machines (VMs) has become a daunting task. In particular, optimized security settings

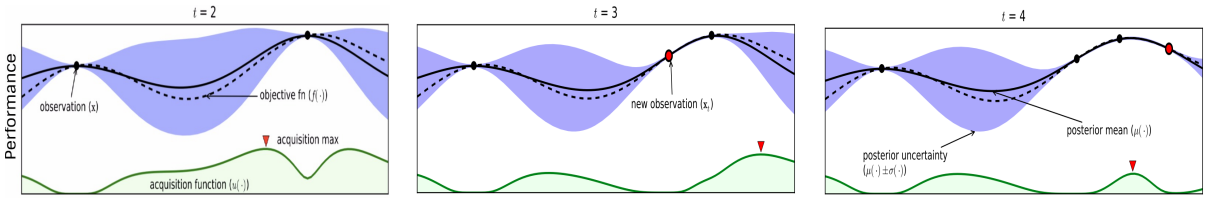

Figure 5.1 An example of Sequential Model-based method's working process from [12]. The figures show a Gaussian process model (GPM) of an objective function over four iterations of sampled values. Green shaded plots represent acquisition function. The value of the acquisition function is high where the GPM predicts larger objective and where the prediction uncertainty (confidence) is high such points (configurations in our case) is sampled first. Note that the area on the far left is never sampled even when it has high uncertainty (low confidence) associated.

are not identical in every setup. They depend on characteristics of the setup, on the ways an application is used or on other applications running on the same system. The problem of finding security setting for a VM is similar to performance configuration optimization [6, 7, 21, 44, 76]. Among numerous other problems which are similar to performance configuration optimization, the problem of how to maximize conversions on landing pages or click-through rates on search-engine result pages [43, 100, 108] has gathered interest.

The rest of this paper discusses how FLASH addresses configuration problems (using the case studies of Figure 5.1).

5.4 Theory

The following theoretical notes define the framework used in the remaining paper.

5.4.1 What are Configurable Software Systems?

A configurable software system has a set X of configurations $x \in X$. Let x_i represent the *ith* configuration of a software system. $x_{i,j}$ represent the *jth* configuration option of the configuration x_i. In general, $x_{i,j}$ indicates either an (i) integer variable or a (ii) Boolean variable. The configuration space (X) represents all the valid configurations of a software system. The configurations are also referred to as *independent variables*. Each configuration (x_i), where $1 \leq i \leq |X|$, has one (single objective) or many (multi-objective) corresponding performance measures $y_{i,k} \in Y$, where $y_{i,k}$ indicates the $1 \leq kth \leq m$ objective associated with a configuration x_i. The performance measure is also referred to as *dependent variable*. For multi-objective problems, there are multiple dependent variables. We denote the performance measures ($y \in Y$) associated with a given configuration by $(y_{i,1}, ..y_{i,m}) = f(x_i)$, in multi-objective setting y_i is a vector, where: $f : X \mapsto Y$

is a function which maps $X \in R^n$ to $Y \in R^m$. In a practical setting, whenever we try to obtain the performance measure corresponding to a certain configuration, it requires actual running a benchmark run with that configuration. In our setting, evaluation of a configuration (or using f) is expensive and is referred to as a measurement. The cost or measurement is defined as the number of times f is used to map a configuration $x_i \in X$ to Y. In our setting, the cost of an optimization technique is the total number of measurements required to find the (near) optimal solution.

In the following, we will explore two different kinds of configuration optimization: *single objective* and *multiple objective*. In single-objective performance configuration optimization, we consider the problem of finding the optimal configuration (x^*) such that $f(x^*)$ is less than other configurations in X. Our objective is to find x^* while minimizing the number of measurements.

$$f(x^*) \leq f(x), \quad \forall x \in X \setminus x^* \tag{5.1}$$

That is, our goal is to find the (near) optimal configuration of a system with least cost or measurements as possible when compared to prior work.

In multi-objective performance configuration optimization, we consider the problem of finding a configuration (x^*) that is better than other configurations in the configuration space of X while minimizing the number of measurements. Unlike, the single-objective configuration optimization problem, where one solution can be the best (optimal) solution (except multiple configurations have the same performance measure), in multi-objective configuration optimization there may be no best solution (best in all objectives). Rather there may be a set of solutions that are equivalent to each other. Hence, to declare that one solution is *b*etter than another, all objectives must be polled separately. Given two vectors of configurations x_1, x_2 with associated objectives y_1, y_2, then x_1 is binary dominant ($\succ$) over x_2 when:

$$\begin{gathered} y_{1,p} \leq y_{2,p} \forall p \in \{1, 2, ..., m\} \text{ and} \\ y_{1,q} < y_{2,q} \text{ for at least one index } q \in \{1, 2, ..., m\} \end{gathered} \tag{5.2}$$

where $y \in Y$ is the performance measures. We refer to binary dominant configurations as better configurations. For the multi-objective configuration optimization problem, our goal is to find a set of better configurations of a given software system using fewer measurements compared to prior work.

5.4.2 Sequential Model-based Optimization

Sequential Model-based Optimization (SMBO) is a useful strategy to find extremes of an unknown objective (or performance) function which is expensive (both in terms of cost and

time) to evaluate. In literature, a certain variant of SMBO is also called Bayesian optimization. SMBO is efficient because of its ability to incorporate prior belief as already measured solutions (or configurations), to help direct further sampling. Here, the prior represents the already known areas of the search (or performance optimization) problem. The prior can be used to estimate the rest of the points (or unevaluated configurations). Once we have evaluated one (or many) points based on the prior, we can define the posterior. The posterior captures our updated belief in the objective function. This step is performed by using a machine learning model or a surrogate model.
The concept of SMBO is simple stated:

- Given what we know about the problem...
- ... what should we do next?

The "given what we know about the problem" part is achieved by using a *machine learning model* whereas "what should we do next" is performed by an *acquisition function*. Such acquisition function automatically adjusts the exploration ("should we sample in uncertain parts of the search space) and exploitation ("should we stick to what is already known") behavior of the method.

This can also be explained in the following way: firstly, few points (or configurations) are (say) randomly selected and measured. These points along with their performance measurements are used to build a model (prior). Secondly, this model is then used to estimate or predict the performance measurements of other unevaluated points (or configurations).[2] This can be used by an acquisition function to select the configurations to measure next. This process continues till a predefined stopping criterion (budget) is reached.

Much of the prior research in configuration optimization of software systems can be characterized as an exploration of different acquisition functions. These acquisition function (or sampling heuristics) were used to satisfy two requirements: (i) use 'reasonable' number of configurations (along with corresponding measurements) and (ii) the selected configurations should incorporate the relevant interactions—how different configuration options influence the performance measure [87]. In a nutshell, the intuition behind such functions is that it is not necessary to try all configuration options—for pragmatic reasons. Rather, it is only necessary to try a small representative configurations—which incorporates the influences of various configuration options. Randomized functions select random items [36, 78]. Other, more intricate, acquisition functions first cluster the data then sample only a subset of examples within each cluster [69]. But the more intricate the acquisition function, the longer it takes to execute—particularly for software with very many configurations. For example, recent studies with a

[2]The probabilistic machine learning model which, unlike fixed point regression model, provides the estimated value (mean) along with uncertainty associated with the mean (variance).

new state of the art acquisition function show that such approaches are limited to models with less than a dozen decisions (i.e. configuration options) [111].

As an example of an acquisition function, consider Figure 5.1. It illustrates a time series of a typical run of SMBO. The bold black line represents the actual performance function (f—which is unknown in our setting) and the dotted black line represents the estimated objective function (in the language of SMBO, this is the *prior*). The purple regions represent the configuration or uncertainty of estimation in a region—the thicker that region, the higher the uncertainty.

The green line in that figure represents the acquisition function. The optimization starts with two points (t=2). At each iteration, the acquisition function is maximized to determine where to sample next. The acquisition function is a user-defined strategy, which takes into account the estimated performance measures (mean and variance) associated with each configuration.. The chosen sample (or configuration) maximizes the acquisition function (*argmax*). This process terminates when a predefined stopping condition is reached which is related to the budget associated with the optimization process.

Gaussian Process Models (GPM) is often the surrogate model of choice in the machine learning literature. GPM is a probabilistic regression model which instead of returning a scalar ($f(x)$) returns the mean and variance associated with x. There are various acquisition functions to choose from: (1) Maximum Mean, (2) Maximum Upper Interval, (3) Maximum Probability of Improvement, (4) Maximum Variance, and (5) Maximum Expected of Improvement.
Building GPMs can be very challenging since:

- GPMs can be very fragile, that is, very sensitive to the parameters of GPMs;
- GPMs do not scale to high dimensional data as well as a large dataset (software system with large configuration space) [83]. For example, in SE, the state of the art in this area using GPMs for optimization was limited to models with around ten decisions [101].

5.5 Performance optimization of Configurable Software Systems

In this section, we discuss the model-based methods used in the prior work to find the (near) optimal configurations of software systems. In this work, we do not discuss the work by Oh et al. [72] which uses true random sampling to find the (near) optimal configurations. We do not compare FLASH with Oh et al.'s method mainly for two reasons. Firstly, Oh et al. builds a BDD (Binary Decision Diagram) which requires access to the feature model, which describes, properties of configuration options and their interactions, and user-imposed configuration constraints, of the software system. However, we argue that this is not always available in

real-world setting [82]. [3] Secondly, Oh et al.'s work only support boolean configuration options which limit its practical applicability. FLASH or the prior work considered in this paper do not have these limitations, and hence, we do not include method proposed by Oh et al. in this work.

5.5.1 Residual-based: "Build an Accurate Model"

In this section, we discuss the residual-based method for building performance models for software systems, which, in SMBO terminology, is an optimizer with a *flat acquisition function*, that is, all the points are equally likely to be selected (random sampling).

When the cost of collecting data (benchmarking time) is higher than the cost of building a performance model (surrogate model), it is imperative to minimize the number of measurements required for model building. A learning curve shows the relationship between the size of the training set and the accuracy of the model. In Figure 5.2, the horizontal axis represents the number of samples used to create the performance model, whereas the vertical axis represents the accuracy (measured in terms of MMRE—Mean Magnitude of Relative Error) of the model learned. Learning curves typically have a steep sloping portion early in the curve followed by a plateau late in the curve. The plateau occurs when adding data does not improve the accuracy of the model. As engineers, we would like to stop sampling as soon as the learning curve starts to flatten. Two types of residual-based methods have been introduced in Sarkar et al. namely *progressive* and *projective sampling*. We use progressive sampling as a representative.[4]

The residual-based method discussed here considers only performance configuration optimization scenarios with a single objective. In the residual-based method, the correctness of the performance model built is measured using error measures such as MMRE:

$$MMRE = \frac{|\ predicted - actual\ |}{actual} \cdot 100 \tag{5.3}$$

For further details, please refer to Sarkar et. al [78].

Figure 5.3 is a generic algorithm that defines the process of progressive sampling. *Progressive* sampling starts by clearly defining the data used in the training set (called training pool) and used for testing the quality of the surrogate model (in terms of residual-based measures) called *holdout set*. The training pool is the set from which the configurations would be selected (randomly, in this case) and then tested against the holdout set. At each iteration, a (set of) data instance(s) of the training pool is added to the training set (Line 9). Once the data instances are selected from the training pool, they are evaluated, which in our setting means measuring

[3] Please note that BDD is used to draw configuration randomly from a large configuration space when enumerating all the configurations is infeasible. However, the method can still be used without BDD at what point would be very close to random sampling.

[4] We choose progressive sampling as a representative because the projective sampling adds only a sample estimation technique to the progressive sampling and does not add anything to the sampling itself.

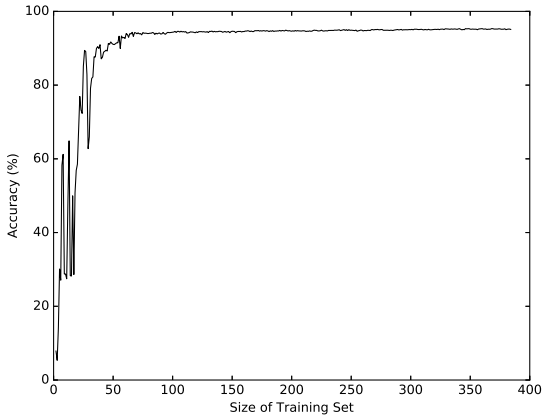

Figure 5.2 The relationship between the accuracy and the number of samples used to train the performance model of the running Word Count application on Apache Storm. Note that the accuracy does not improve substantially after 20 sample configurations.

the performance of the selected configuration (Line 12). The configurations and the associated performance scores are used to build the performance model (Line 14). The model is validated using the testing set[5], then the accuracy is computed. In our setting, we assume that the measure is accuracy (higher is better). Once the accuracy score is calculated, it is compared with the accuracy score obtained before adding the new set of configurations to the training set. If the accuracy of the model (with more data) does not improve the accuracy when compared to the previous iteration (lesser data), then life is lost. This termination criterion is widely used in the field of Evolutionary Algorithms to determine the degree of convergence [55].

5.5.2 Rank-based: "Build a Rank-preserving Model"

As an alternative to the residual-based method, a rank-based method has recently been proposed [70]. The rank-based method is similar to residual-based method in that it has a *flat acquisition function*, which resembles random sampling. Like the residual-based method, the rank-based method discussed here also considers only performance configuration optimizations with a single objective. For further details, please refer to Nair et. al [70].

In a nutshell, instead of using residual measures of errors, as described in Equation 5.3, which depend on residuals ($r = y - f(x)$)[6], it uses a rank-based measure. While training the performance model ($f(x)$), the configuration space is iteratively sampled (from the training pool) to train the performance model. Once the model is trained, the accuracy of the model is

[5]The testing data consist of the configurations as well as the corresponding performance scores.
[6]Refer to Section 5.4.1 for definitions.

```
# Progressive Sampling                                                    1
def progressive(training, holdout, lives=3):                              2
  # For stopping criterion                                                3
  last_score = -1                                                         4
  independent_vals = list()                                               5
  dependent_vals = list()                                                 6
  for count in range(1, len(training)):                                   7
    # Add one configuration to the training set                           8
    independent_vals += training[count]                                   9
    # Measure the performance value for the newly                         10
    # added configuration                                                 11
    dependent_vals += measure(training_set[count])                        12
    # Build model                                                         13
    model = build_model(independent_vals, dependent_vals)                 14
    # Test Model                                                          15
    perf_score = test_model(model, holdout, measure(holdout))            16
    # If current accuracy score is not better than                        17
    # the previous accuracy score, then loose life                        18
    if perf_score <= last_score:                                          19
      lives -= 1                                                          20
    last_score = perf_score                                               21
    # If all lives are lost, exit loop                                    22
    if lives == 0: break                                                  23
  return model                                                            24
```

Figure 5.3 Python code of progressive sampling, a residual-based method.

measured by sorting the values of $y = f(x)$ from 'small' to 'large', that is:

$$f(x_1) \leq f(x_2) \leq f(x_3) \leq ... \leq f(x_n). \tag{5.4}$$

The predicted rank order is then compared to the actual rank order. The accuracy is calculated using the mean rank difference (μRD):

$$\bar{}RD = \frac{1}{n} \cdot \sum_{i=1}^{n} \left| rank(y_i) - rank(f(x_i)) \right| \tag{5.5}$$

This measure simply counts how many of the pairs in the test data have been ordered incorrectly by the performance model $f(x)$ and measures the average of magnitude of the ranking difference.

In Figure 5.4, we list a generic algorithm for the rank-based method. Sampling starts by selecting samples randomly from the training pool and by adding them to the training set (Line 8). Then, the collected sample configurations are evaluated (Line 11). The configurations and the associated performance measure are used to build a performance model (Line 13). The generated model (CART, in our case) is used to predict the performance measure of the configurations in the testing pool (Line 16). Since the performance value of the holdout set is

```
# rank−based method                                                    1
def rank_based(training, holdout, lives=3):                            2
  last_score = −1                                                      3
  independent_vals = list()                                            4
  dependent_vals = list()                                              5
  for count in range(1, len(training)):                                6
      # Add one configuration to the training set                      7
      independent_vals += training[count]                              8
      # Measure the performance value for the newly                    9
      # added configuration                                            10
      dependent_vals += measure(training_set[count])                   11
      # Build model                                                    12
      model = build_model(independent_vals, dependent_vals)            13
      # Predicted performance values                                   14
      predicted_performance = model(holdout)                           15
      # Compare the ranks of the actual performance                    16
      # scores to ranks of predicted performance scores                17
      actual_ranks = ranks(measure(holdout))                           18
      predicted_ranks = ranks(predicted_performance)                   19
      mean_RD = RD(actual_ranks, predicted_ranks)                      20
      # If current rank difference is not better than                  21
      # the previous rank difference, then loose life                  22
      if mean_rank_difference <= last_rank_difference:                 23
        lives −= 1                                                     24
      last_rank_difference = mean_RD                                   25
      # If all lives are lost, exit loop                               26
      if lives == 0: break                                             27
    return model                                                       28
```

Figure 5.4 Python code of rank-based method.

already measured, hence known, the ranks of the actual performance measures, and predicted performance measure are calculated. (Lines 18–19). The actual and predicted performance measure is then used to calculate the rank difference using Equation 5.5. If the rank difference (μRD) of the model (with more data) does not decrease when compared to the previous generation (lesser data), then a life is lost (Lines 23–24). When all lives are expired, sampling terminates (Line 27).

5.5.3 ePAL: "Traditional Bayesian Optimization"

Unlike the residual-based and rank-based methods, epsilon Pareto Active Learning (ePAL) reflects on the evaluated configurations (and corresponding performance measures) to decide the next best configuration to measure using *Maximum Variance* (predictive uncertainty) as an acquisition function. ePAL incrementally updates a model (GPM) representing a generalization of all samples (or configurations) seen so far. The model can be used to decide the next most promising configuration to evaluate. This ability to avoid unnecessary measurement (by just exploring a model) is very useful in the cases where each measurement can take days to weeks.

In Figure 5.5, we list a generic algorithm for ePAL. ePAL starts by selecting samples ran-

```
# ePAL Multi−objective Bayesian Optimizer                                    1
def ePAL(all_configs, ε, size = 20):                                         2
  # Add |\textit{size}| number of randomly selected                          3
  # configurations to training data                                          4
  train = random.sample(all_configs, size)                                   5
  # Measure the performance value for sampled training data                  6
  train = [ measure(x) for x in train ]                                      7
  # Remove the measured configurations from configuration space              8
  all_configs = all_configs \ train                                          9
  # Till all the configurations in all_configs has been either sampled or discarded  10
  while len(all_configs) > 0:                                                11
    # Build GPM                                                              12
    model = GPM(train)                                                       13
    # Get prediction and corresponding confidence intervals                  14
    # associated with each configuration in all_configs                      15
    μ, σ = model.predict(all_configs)                                        16
    # Only keep configurations discard based on uncertainty                  17
    all_configs = all_configs − discard(all_configs,μ, σ, ε)                 18
    # Find and measure another point based on acquisition function.          19
    new_point = measure(acquisition_function(all_configs, μ, σ))            20
    # Add new_point to train                                                 21
    train += new_point                                                       22
  return train                                                               23
```

Figure 5.5 Python code of ePAL, a multi-objective Bayesian optimizer.

domly from the configuration space (*all_configs*) and by adding them to the training set (Line 5). The collected sample configurations are then evaluated (Line 7). The configurations and the associated performance values are used to build a performance model (Line 13). The generated model (GPM, in this case) is used to predict the performance values of the configurations in the testing pool (Line 16). Note that the model returns both the value (μ) as well as the associated confidence interval (σ). These predicted values are used to discard configurations, which have a high probability of being dominated by another point (Line 18). Domination is defined in Equation 5.2 [7]. After configurations (which have a high probability of being dominated) have been discarded, a new configuration (*new_point*) is selected and measured (Line 20). The selected configuration *new_point* is the most uncertain in *all_configs*. Then, *new_point* is added to *train*, which is then used to build the model in the subsequent iteration (Line 22). When all the configuration in *all_configs* have been discarded (or evaluated and moved to train) the process terminates.

Note again, since ePAL is a traditional SMBO, it shares its shortcomings (refer to Section 5.4.2).

[7] ePAL then removes all ε-dominated points: a is discarded due to b if $\mu_b + \sigma_b$ ε-dominates $\mu_a - \sigma_a$, where x ε-dominates y if $x + \varepsilon \succeq y$ and "$\succeq$" is binary domination- see Equation 5.2

5.6 FLASH: A Fast Sequential Model-based Method

To overcome the shortcomings of the traditional SMBO, FLASH makes certain design choices:

- FLASH's acquisition function uses *Maximum Mean*. Maximum Mean returns the sample (configuration) with highest expected (performance) measure;
- GPM is replaced with CART [11], a fixed-point regression model. This is possible because the acquisition function requires only a single point value rather than the mean and the associated variance.

When used in a multi-objective optimization setting, FLASH models each objective as a separate performance (CART) model. This is because the CART model can only be trained for one performance measure or dependent value.

FLASH replaces the *actual* evaluation of all configurations (which can be a very slow process) with a *surrogate evaluation*, where the CART decision trees are used to guess the objective scores (which is a very fast process). Once guesses are made, then some *select* operator must be applied to remove less-than-satisfactory configurations. Inspired by the decomposition approach of MOEA/D [106], FLASH uses the following stochastic Maximum Mean method, which we call *BarrySort*[8].

For problems with o objectives that we seek to maximize, *BarrySort* returns the configuration that has outstandingly maximum objective values across N random projections. Using the *predictions* for the o objectives from the learned CART models, then *BarrySort* executes as follows:

- Generate N vectors V of length m. Fill with random numbers in the range 0..1.
- Set $max = 0$ and $best = nil$.
- For each configuration x_i
  - Guess its objective scores $y_{i,j}$ using the *predictions* from the CART models.
  - Compute its mean weight as follows:

$$mean_i = \frac{1}{N} \sum_{n}^{N} \sum_{j}^{m} \left(V_{n,o} \cdot x_{i,j} \right) \tag{5.6}$$

  - If $mean > max$, then $max := mean$ and $best := x_i$.
- Return *best*.

```
def FLASH(uneval_configs, fitness, size, budget):                                  1
  # Add |size| number of randomly selected configurations to training data.       2
  # All the randomly selected configurations are measured                        3
  eval_configs = [measure(x) for x in sample(uneval_configs, size)]              4
  # Remove the evaluations configuration from data                               5
  uneval_configs.remove(eval_configs)                                            6
  # Till all the lives has been lost                                             7
  while budget > 0:                                                              8
    # build one CART model per objective                                         9
    for o in objectives: model[o] = CART(eval_configs)                           10
    # Find and measure another point based on acquisition function               11
    acquired_point = measure(acquisition_fn(uneval_configs, model))              12
    eval_configs += acquired_point # Add acquired point                          13
    uneval_config −= acquired_point # Remove acquired point                      14
    # Stopping Criteria                                                          15
    budget −= 1                                                                  16
  return best                                                                    17
                                                                                 18
def acquisition_fn(uneval_configs, model, no_directions=10):                     19
  # Predict the value of all the unevaluated configurations using model          20
  predicted = model.predict(uneval_configs)                                      21
  # If number of objectives is greater than 1 (In our setting len(objectives) = 2) 22
  if len(objectives) > 1: # For multi−objective problems                         23
    return BarrySort( predicted )                                                24
  else: # For single objective problems                                          25
    return max(predicted)                                                        26
```

Figure 5.6 Python code of FLASH

The resulting algorithm is shown in Figure 5.6. Before initializing FLASH, a user needs to define three parameters N, *size* and *budget* (refer to Section 5.10.2 for a sensitivity analysis). FLASH starts by randomly sampling a predefined number (*size*) of configurations from the configuration space and evaluate them (Line 4). The evaluated configurations are removed from the unevaluated pool of configurations (Line 6). The evaluated configurations and the corresponding performance measure/s are then used to build CART model/s (Line 10). This model (or models, in case of multi-objective problems) is then used by the acquisition function to determine the next point to measure (Line 13). The acquisition function accepts the model (or models) generated in Line 10 and the pool of unevaluated configurations (*uneval_configs*) to choose the next configuration to measure. The model is used to generate a prediction for the unevaluated configurations. For single-objective optimization problems, the next configuration to measure is the configuration with the highest predicted performance measure (Line 26). For multi-objective optimization problems, *BarrySort* is applied. The configuration chosen by the acquisition function is evaluated and added to the evaluated pool of configurations (Line 12-13) and removed from the unevaluated pool (Line 14). FLASH terminates once it runs out of budget

[8]Named after (1) Barry Allen of the Flash T.V. series; and (2) the childhood nickname of the 44th United States President Barack Obama.

(Line 8).

Note the advantages of this approach: Firstly, FLASH only executes a full evaluation, once per iteration; so, in terms of number of evaluations, FLASH runs very fast.

Secondly, for single-objective optimization problems, prior approaches built accurate (either residual-based or rank-based) models to find (near) optimal configurations. The quality of the (sequential) model built is computed using Equations 5.3 and 5.5 on a holdout set. The size of this holdout set can be significantly high based on the size of the configuration space of the software system. FLASH *removes the need for such holdout set and finds the (near) optimal configurations using fewer samples*.

Thirdly, in multi-objective optimization problems, prior approaches uses the traditional Bayesian optimization technique which is not scalable.

Fourthly, *BarrySort* has a time complexity of $O(C)$ to return the best configuration from C candidates. Other methods are much slower. NSGA-II non-dominated sort procedure[18] which is $O(C^2)$. In practice, that took hours to compute, especially for software systems with very large configuration spaces. That is, *BarrySort removes the short-comings of prior work, which makes* FLASH *more scalable.*

5.7 Evaluation

5.7.1 Research Questions

5.7.1.1 Single-objective Optimization

In prior work, performance configuration optimization was conducted by sequentially sampling configurations to build models, both accurate (residual-based method) and inaccurate (rank-based method). Both these techniques use a holdout set which is used to evaluate the quality of the model (built sequentially). The size of the holdout set is based on an engineering judgment and expected to be a representative of the whole configuration space. Prior work [70] used 20% of the configuration space as a holdout set, but did not consider the cost of using the holdout set.

Our research questions are geared towards assessing the performance of FLASH based on two aspects: (i) **Effectiveness** of the solution or the rank-difference between the best configuration found by FLASH to the actual best configuration, and (ii) **Effort** (number of measurements) required to find the (near) optimal configuration.

The above considerations lead to two research questions: **RQ1***: Can* FLASH *find (near) optimal configuration?*

Here, the (near) optimal configurations found using FLASH are compared to the ones identified

in prior work, using the residual-based and rank-based method. The effectiveness of the methods is compared using rank-difference (Equation 5.7).
RQ2*: How expensive is* FLASH *(in terms of how many configurations must be measured)?*
It is expensive to build (residual-based or rank-based) models since they require using a holdout set. Our goal is to demonstrate that FLASH can find (near) optimal configurations of a software system using fewer measurements.

5.7.1.2 Multi-objective Optimization

To the best of knowledge, SMBO has never been used for multi-objective performance configuration optimization in software engineering. However, similar work has been done by Zuluaga et al. [111] in the machine learning community, where they introduced ePAL. We use ePAL as a state of the art method to compare FLASH.

Since ePAL suffers from the shortcomings of traditional SMBO, our research questions are geared towards finding the estimated Pareto-optimal solutions (predicted Pareto Frontier [9]), which is *closest* to the true Pareto Frontier (which requires measuring all configurations) with *least effort*. We assess the performance of FLASH by considering three aspects: (i) **Effectiveness** of the configurations between the Pareto Frontier and the ones approximated by a optimizer, and **Effort** evaluated in terms of (ii) number of measurements, and (iii) time to approximate the Pareto Frontier.
The above considerations lead to three research questions: **RQ3:** *How effective is* FLASH *for multi-objective performance configuration optimization?*
The effectiveness of the solution or the difference between the predicted Pareto Frontier found by optimizers to the true Pareto Frontier,
RQ4: *Can* FLASH *be used to reduce the effort of multi-objective performance configuration optimization compared to ePAL?*
Effort (number of measurements) required to estimate the Pareto Frontier which is closest to the true Pareto Frontier, and
RQ5 *Does* FLASH *save time for multi-objective performance configuration optimization compared to ePAL?*
Since ePAL is may take substantial time to find the approximate the Pareto Frontier, it is imperative to show that FLASH can approximate the Pareto Frontier and converge faster.

Our goal is to minimize the effort (time and number of measurements) required to find an approximate Pareto Frontier as close to the actual Pareto Frontier as possible.

[9] Pareto Frontier is a set of solutions which are non-dominated by any other solution.

5.7.2 Case Studies

We evaluated FLASH in two different types of problems namely: (1) single-objective optimization problems and (2) multi-objective optimization problems using 30 scenarios (15 scenarios in multi-objective settings) from 7 software systems. These systems are summarized in Figure 5.1. More details about the software systems are available at http://tiny.cc/flash_systems/.

We selected these software systems since they are widely used in the configuration and search-based SE literature [36, 49, 69, 70, 72, 78, 86, 111] as benchmark problems for this kind of optimization work. Furthermore, extensive documentation is available at the supplementary web site for all these models.

5.7.3 Experimental Rig

5.7.3.1 Single-objective Optimization

For each subject system, we build a table of data, one row per valid configuration. We then run all configurations of all systems (that is, that are invoked by a benchmark) and record the performance scores. To this table, we added a column showing the performance score obtained from the actual measurements for each configuration. Note that, while answering the research questions, we ensure that we never test any prediction model on the data that we used to learn the model.

To answer our research questions, we split the datasets into training pool (40%), holdout set (20%), and validation pool (40%). The size of the holdout set is taken from prior work [70]. It is worthy to note that this is a magic parameter and is set based on an engineering judgment. To perform a fair comparison while comparing FLASH with prior work, the training pool and validation pool are merged for FLASH experiments.

The experiment to find (near) optimal configuration using the residual-based and rank-based methods is conducted in the following way:

- Randomize the order of rows in the training data
- **Do**
  - Select one configuration (by sampling with replacement) and add it to the training set
  - Determine the performance scores associated with the configuration. This corresponds to a table look up but would entail compiling or configuring and executing a system configuration in a practical setting.
  - Using the training set and the accuracy, build a performance model using CART.
  - Using the data from the testing pool, assess the accuracy either using MMRE (as described in Equation 5.3) or rank difference (as described in Equation 5.5).

- **While** the accuracy is greater or equal to the threshold determined by the practitioner (rank difference in the case of rank-based method and MMRE in the case of residual-based method).

Once the model is trained, it is tested on the data in the validation pool. Please note, the learner has not been trained on the validation pool. The experiment to find (near) optimal configuration by FLASH is conducted in the following way:

- Choose 80% of the data (at random)
- Randomize the order of rows in the training data
- **Do**
  - Select 30 configurations (by sampling with replacement) and add them to the training set
  - Determine the performance scores associated with the configurations. This corresponds to a table look up, but would entail compiling or configuring and executing a system configuration in a practical setting.
  - Using the training set, build a performance model using CART.
  - Using the CART model, find the configuration with best predicted performance.
  - Add the configuration with best predicted performance to the training set.
- **While** the stopping criterion (*budget*) is not met, continue adding configurations to the training set.

Once FLASH has terminated, the configuration with the best performance is selected as the (near) optimal configuration. Please note that unlike the methods proposed in prior work, there is no training and validation pool in FLASH. It uses the whole space and returns the configuration with the best performance.

RQ1 relates the results found by FLASH to ones of residual-based and rank-based methods. We use the absolute difference between the ranks of the configurations predicted to be the optimal configuration and the actual optimal configuration. We call this measure rank difference.

$$RD = \left| rank(actual_{optimal}) - rank(predicted_{optimal}) \right| \tag{5.7}$$

Ranks are calculated by sorting the configurations based on their performance scores. The configuration with the least performance score, $rank(actual_{optimal})$, is ranked 1 and the one with the highest score is ranked as N, where N is the number of configurations.

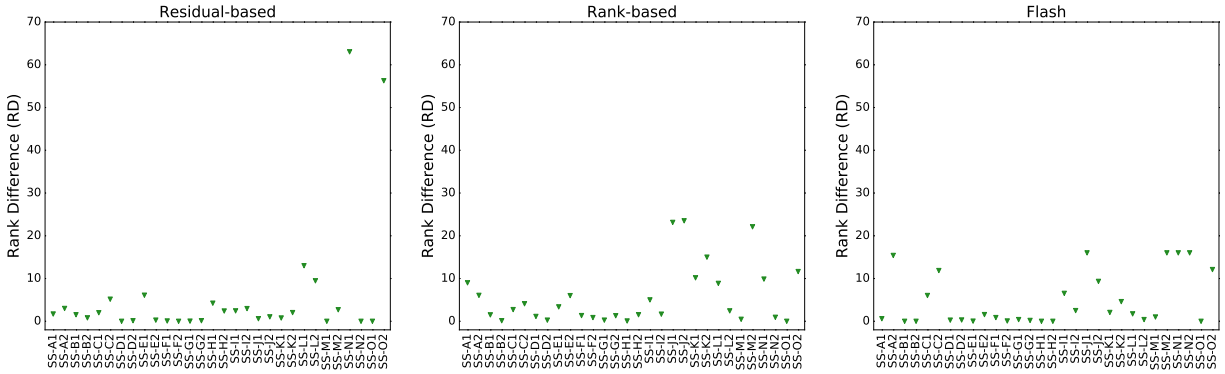

Figure 5.7 The rank difference of the prediction made by model built using the residual-based method, the rank-based methods, and FLASH. Note that the y-axis of this chart rises to large values; e.g., SS-M has 239,260 possible configurations. Hence, the above charts could be summarized as follows: "the FLASH is surprisingly accurate since the rank difference is usually close to 0% of the total number of possible configurations."

5.7.3.2 Multi-objective Optimization

Similar to **RQ1** and **RQ2**, for each subject system, we build a table of data, one row per valid configuration. We then run all configurations of all systems and record the performance scores. To this table, we add two columns of measurements (one for each objective) obtained from measurements.

To measure effectiveness, we use quality indicators as advised by Wang et al. [98]. The quality indicators are:

- The *Generational Distance* (GD) [95] measures the closeness of the solutions from by the optimizers to the *Pareto frontier* that is, the actual set of non-dominated solutions.
- The *Inverted Generational Distance* (IGD) [14] is the mean distance from points on the *true* Pareto-optimal solutions to its nearest point in the predicted Pareto-optimal solutions returned by the optimizer.

Note that, for both measures, *smaller* values are *better*. Also, according to Coello et al. [14], IGD is a better measure of how well solutions of a method are *spread* across the space of all known solutions. A lower value of GD indicates that the predicted Pareto-optimal solutions have converged (or are near) to the actual Pareto-optimal solutions. However, it does not comment on the diversity (or spread) of the solutions. GD is useful while interpreting the results of **RQ3** and **RQ6**, where we would notice that FLASH has low GD values but relatively high IGD values.

To answer our research questions, we initialize ePAL and FLASH with randomly selected configurations along with their corresponding performance scores. Since, ePAL does not have

an explicit stopping criterion, we allow ePAL to run until completion. For FLASH, we allowed a budget of 110 configurations. The value 110 was assigned by parameter tuning (from Section 5.10.2). The configurations evaluated during the execution of the three methods are then used to measure the quality measures (to compare methods). Note that we use two versions of ePAL: ePAL with $\epsilon = 0.01$ (ePAL_0.01), and ePAL with $\epsilon = 0.3$ (ePAL_0.3) [10]. These ePAL versions represents two extremes of ePAL from the most cautious ($\epsilon = 0.01$)—maximizing quality to most careless ($\epsilon = 0.3$)—minimizing measurements. [11]

Other aspects of our experimental setting were designed in response to the specific features of the experiments. For example, all the residual-based, rank-based and FLASH methods are implemented in Python. We use Zuluaga et al.'s implementation of ePAL, which was implemented in Matlab. Since we are comparing methods implemented in different languages, we measure "speed" in terms of the number of measurements (a language-independent feature) along with runtimes.

5.8 Results

5.8.1 Single-objective Problems

RQ1: Can FLASH find (near) optimal configuration?
Figure 5.7 shows the *Rank Difference* of the predictions using FLASH, the rank-based method and the residual based method. The horizontal axis shows subject systems. The vertical axis shows the rank difference (Equation 5.7).

- The ideal result would be when all the points lie on the line *y=0* or the horizontal axis, which means the method was able to find the optimal configurations for all subject systems or the rank difference between the predicted optimal solution and the actual optimal solution is 0.
- The sub-figures (left to right) represent the residual-based method, rank-based method, and FLASH.

Overall, in Figure 5.7, we find that:

- All methods can find (near) optimal configurations. For example, FLASH for SS-J1 predicted the configuration whose performance score is ranked 20th in configuration space. That is *good enough* since FLASH can find the 20th most performant configuration among 3072 configurations[12].

[10] Refer to Section 5.5.3 for definition of ϵ

[11] We have measured other values of epsilon between 0.01 and 0.3, but due to space constraints we show results from two variants of ePAL

[12] Since we only test on 80% of the total configuration space.

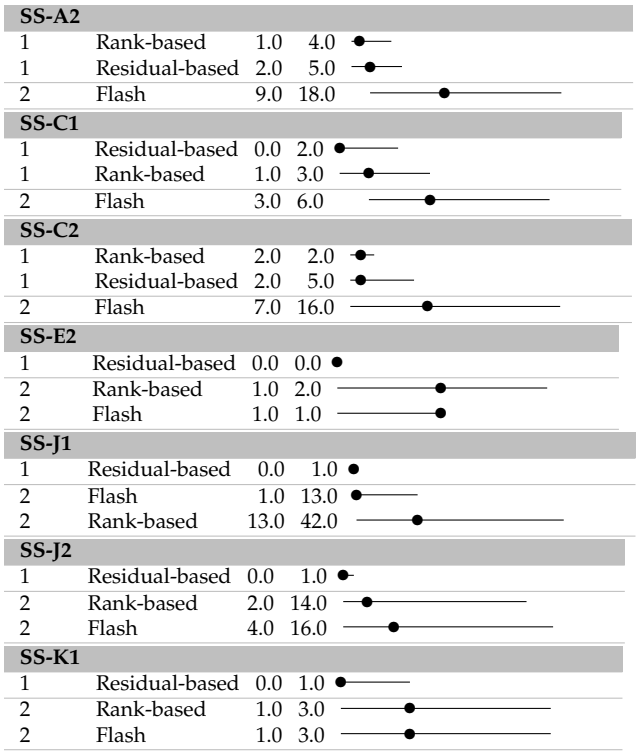

Figure 5.8 The median rank difference of 20 repeats. Median ranks are the rank difference as described in Equation 5.7, and IQR the difference between 75th percentile and 25th percentile found during multiple repeats. Lines with a dot in the middle (—•—), show the median as a round dot within the IQR. All the results are sorted by the median rank difference: a lower median value is better. The left-hand column (*Rank*) ranks the various methods for example when comparing three techniques. For SS-A2, a rank-based method has a different rank since their median rank difference is statistically different. Please note, this chart only shows software systems where FLASH is significantly worse than methods from prior work.

- The mean rank difference of the predicted optimal configuration is 6.082, 5.81, and 5.58 [13] for residual-based, rank-based, and FLASH.

So, the rank of the (near) optimal configuration found by all the three methods is practically the same. To verify the similarity is statistically significant, we further studied the results using non-parametric tests (as proposed by Arcuri & Briand [64]). For testing statistical significance, we used a non-parametric bootstrap test with 95% confidence interval [23], followed by an A12 test to check that any observed difference is not trivially small,[14] Lastly, to generate succinct

[13]The median rank difference is 1.61, 2.583, and 1.28 for residual-based, rank-based, and FLASH.

[14] Given two lists X and Y, count how often there are larger numbers in the former list (and if there are ties, add a half mark): $a = \forall x \in X, y \in Y \frac{\#(x>y)+0.5\cdot\#(x=y)}{|X|\cdot|Y|}$ (as per Vargha [96], we say that a "small" effect has $a < 0.6$).

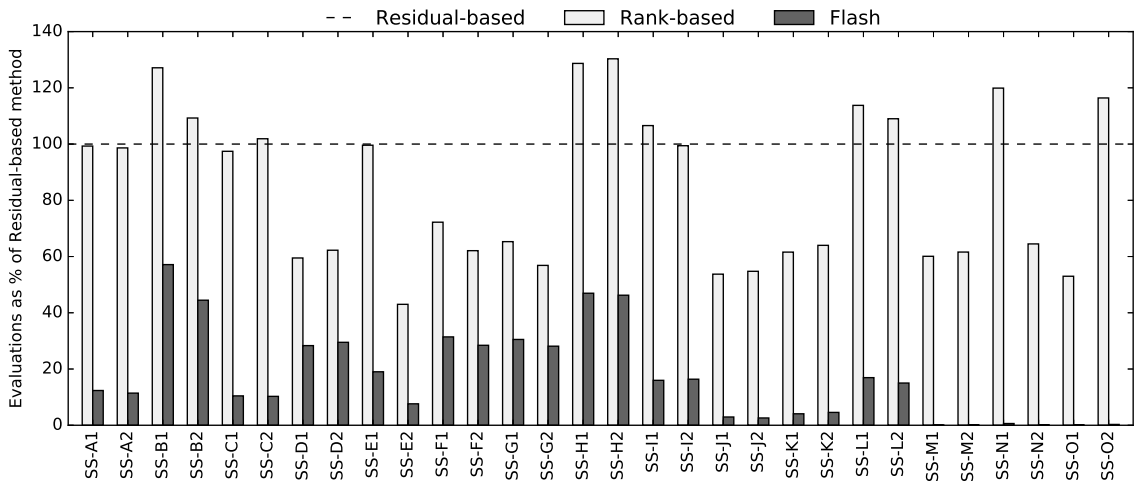

Figure 5.9 Number of measurements required to find (near) optimal configurations with the residual-based method as the baseline.

reports, we use the Scott-Knott test to recursively divide our methods. This recursion used A12 and bootstrapping to group subsets that are not significantly different and whose difference has non-neglectable effect size. This use of the Scott-Knott test is endorsed by Mittas & Angelis [64] and by Hassan et al. [34].

In Figure 5.8, we show the Scott-Knott ranks for the three methods. The quartile charts show the Scott-Knott results for the subject systems, where FLASH did not do as well as the other two methods. For example, the statistic test for SS-C2 shows that the rank difference of configurations found by FLASH is statistically larger from the other methods. This is reasonably close, since the median rank of the configurations found by FLASH is 7 of 1512 configurations, where for the other methods found configurations have a median rank of 2. As engineers, we feel that this is close because we can find the 7th best configuration using 34 measurements compared to 339 and 346 measurements used by the residual-based and rank-based methods. Overall, our results indicate that:

> FLASH can find (near) optimal configurations, similar to the residual-based and the rank-based method, of a software system without using a holdout set.

RQ2: How expensive is FLASH (in terms of how many configurations must be executed)?
To recommend FLASH as a cheap method for performance optimization, it is important to demonstrate that it requires fewer measurements to find the (near) optimal configurations. In our setting, the cost of finding the (near) optimal configuration is quantified in terms of

number of measurements required. Figure 5.9 demonstrates this. The vertical axis represents the ratio of the measurements of different methods are represented as the percentage of number of measurements required by residual-based method âĂŞ since it uses the most measurements in 66% of the scenarios.

Overall, we see that FLASH requires the least number of measurements to find (near) optimal configurations. For example, in SS-E1, FLASH requires 9% of the measurements when compared with the residual-based method and the rank-based method. There are few cases (SS-M1 to SS-O2) where FLASH requires less than 1% of the residual-based method, which is because these systems have a large configuration space and the holdout set required by the residual-based method and the rank-based method (except FLASH) uses 20% of the measurements.

For performance configuration optimization, FLASH is cheaper than the state of the art method. In 57% of the software systems, FLASH requires an order of magnitude fewer measurement compared to the residual-based method and rank-based method.

5.8.2 Multi-objective Optimization

RQ3: How effective is FLASH for multi-objective performance configuration optimization? Table 5.2 shows the results of a statistical analysis that compares the quality measures of the approximated Pareto-optimal solutions generated by FLASH to those generated by ePAL.

- The rows of the table shows median numbers of 20 repeated runs over 15 different scenarios.
- The columns report the quality measures, generational distance (GD) and inverted generation distance (IGD). Recall *smaller* values of GD and IGD are *better*.
- '**X**' denotes cases where a method did not terminate within a reasonable amount of time (10 hours).
- **Bold**-typed results are statistically better than the rest.
- The last row of the table (Win (%)) report the percentage of times a method is significantly better than other methods overall software systems.

One way to get a quick summary of this table is to read the last row (Win(%)). This row is the percentage number of times a method was marked statistically better than the other methods. From Table 5.2, we can observe that FLASH outperforms variants of ePAL since FLASH has the highest Win(%) in both quality measures. This is particularly true for scenarios with more than 10 configuration options, where ePAL failed to terminate while FLASH always did.

Table 5.2 Statistical comparisons of FLASH and ePAL regarding the Performance measures are GD (Generational Distance), IGD (Inverted Generational Distance) and a number of measurements. For all measures, less is better; âĂIJXâĂİ denotes cases where methods did not terminate within a reasonable amount of time (10hrs). The numbers in bold represent statistically better runs than the rest. For example, for SS-G, GD of FLASH is statistically better than of ePAL.

Software	GD			IGD			Evals		
	epal_0.01	epal_0.3	Flash	epal_0.01	epal_0.3	Flash	epal_0.01	epal_0.3	Flash
SS-A	**0.002**	**0.002**	**0**	**0.002**	0.002	**0**	109.5	73.5	**50**
SS-B	**0**	**0**	**0.005**	**0**	0.003	0.001	84.5	**20**	**50**
SS-C	**0.001**	**0.001**	**0.003**	**0.004**	**0.004**	**0**	247	101	**50**
SS-D	**0**	**0.004**	**0.014**	**0.002**	0.007	0.009	119.5	67	**50**
SS-E	**0.001**	**0.001**	**0.012**	**0.004**	0.008	0.002	208	54.5	**50**
SS-F	**0**	0.016	0.008	**0**	0.006	0.016	138	71	**50**
SS-G	**0**	**0**	**0.023**	**0.003**	0.006	0.004	131	69	**50**
SS-H	**0**	**0**	**0**	**0**	**0**	**0**	52	**28**	**50**
SS-I	0.008	0.018	**0**	0.008	0.018	**0**	48	**30**	**50**
SS-J	**0**	**0**	**0.002**	**0.002**	**0.002**	**0**	186	**30**	**50**
SS-K	**0.001**	**0.001**	**0.003**	**0.001**	**0.002**	**0.001**	209	140	**50**
SS-L	**0.01**	**0.028**	**0.006**	**0.007**	**0.008**	**0.009**	68.5	**35**	**50**
SS-M	X	X	**0**	X	X	**0**	X	X	**50**
SS-N	X	X	**0.065**	X	X	**0.015**	X	X	**50**
SS-O	X	X	**3.01E-07**	X	X	**3.20E-06**	X	X	**50**
Win (%)	73	67	**93**	**67**	33	**67**	0	33	**80**

We further notice that ePAL-0.01 has a higher win percentage than ePAL-0.3. This is not surprising since (as discussed in Section 5.7.3.2) ePAL-0.01 (optimized for quality) is more cautious than ePAL-0.3 (which is optimized for speed measured in terms of number of measurements). This can be regarded as a sanity check. It is interesting to note that FLASH has impressive convergence score (lower GD scores)—it converges better for 93% of the systems, but not so remarkable in terms of the spread (lower IGD scores). However, the performance of FLASH is similar to ePAL. It is also interesting that, for software systems where FLASH was not statistically better, these are cases where the statistically better method always converged to the actual Pareto Frontier (with few exceptions).

FLASH is effective for multi-objective performance configuration optimization. It also works in software systems with more than 10 configuration options whereas ePAL does not terminate in reasonable time.

RQ4: Can FLASH be used to reduce the effort of multi-objective performance configuration optimization compared to ePAL?
In the RQ4 section (right-hand side) of Table 5.2, the number of measurements required by methods, ePAL, and FLASH are shown. Rows show different software systems and columns shows the number of measurements associated with each method. The numbers highlighted in bold mark methods that are statistically better than the other. For example in SS-K, FLASH uses statistically few samples than (variants of) ePAL.

From the table we observe:

- FLASH uses fewer samples than ePAL_0.01. In 9 of 15 cases, ePAL_0.01 is, at least, two times better than FLASH.
- (Variants of) ePAL does not terminate for SS-M, SS-N, and SS-O even after ten hours of execution—a pragmatic choice. The reason for this can be seen in Figure 5.1: these software systems have more than 10 configuration options and the GPMs used by ePAL does not scale beyond that number.
- The obvious feature of Table 5.2 is that FLASH used fewer measurements in 12 of 15 software systems.

FLASH requires fewer measurements than ePAL to approximate Pareto-optimal solutions. The number of evaluations used by FLASH is less than (more careful) ePAL-0.01 for all the software systems and 12 of 15 software systems for (more careless) ePAL-0.3.

RQ5:Does FLASH save time for multi-objective performance configuration optimization compared to ePAL?
Figure 5.10 compares the run times of ePAL with FLASH. Please note that we use authors version of ePAL in our experiments, which is implemented in Matlab. However, FLASH was implemented in Python. Even though this may not be a fair comparison, for the sake of completeness, we report the run-times of the test. The sub-figure to the left shows how the run times vary with the number of configurations of the system. The x-axis represents the number of configurations (in log scale), while the y-axis represents the time taken to perform 20 repeats in seconds (in log scale), which means lower the better. The dotted lines in the figure, shows the cases where a method (in this case, ePAL) did not terminate. The sub-figure in the middle represents how the run-time varies with the number of configuration options. The x-axis represents the number of configuration options, and the y-axis represents the time taken for 20 repeats in seconds (in log scale), which means lower the better. The sub-figure to the right represent the performance gain achieved by FLASH over (variants of) ePAL. The x-axis shows the software systems, and

the Y-axis represents the gain ratio. Any bar higher than the line (y=1) represent cases where FLASH is better than ePAL.

From the figure, we observe:

- From sub-figures left and middle, FLASH is much faster than (variants of) ePAL except in 2 of 15 cases.
- The run times of ePAL increase exponentially with the number of configurations and configuration options, similar to the trend reported in the literature.
- (Variants of) ePAL does not terminate for cases with large numbers of configurations and configuration options, whereas FLASH always terminates an order of magnitude faster than ePAL. This effect is magnified in case of a scenarios with large configuration space.

Overall our results indicate:

> FLASH saves time and is faster than (variants of) ePAL in 13 of 15 cases. Furthermore, FLASH is an order of magnitudes faster than ePAL in 5 of 15 software systems. In other 2 out of 15 cases, the FLASH 's runtimes are similar to (variants of) ePAL.

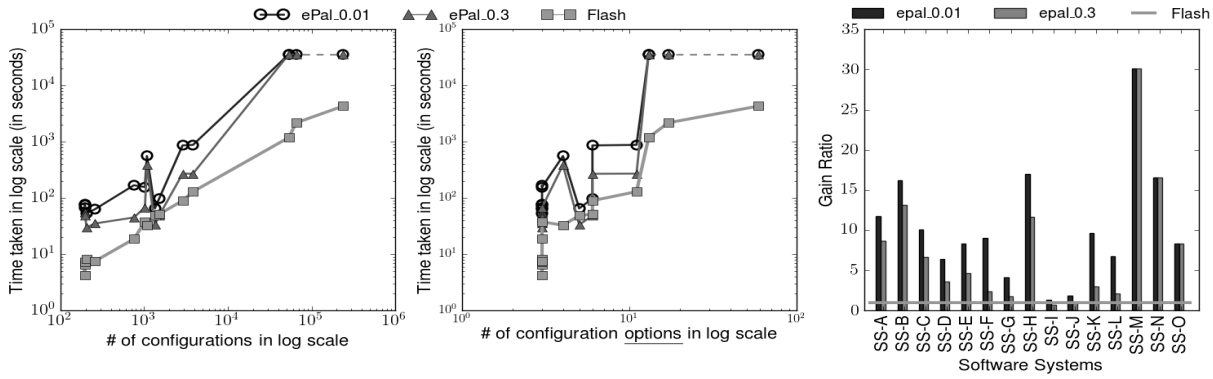

Figure 5.10 The time required to find (near) optimal solutions using ePAL and Flash (sum of 20 repeats). Note that the axis's of the first two figures (left, and center) are in log scale. The time required for FLASH compared to (variants of) ePAL is much lower (with an exception on 2 of 15 software systems). The dashed line in the figure (left and middle) represents cases where ePAL did not terminate within a reasonable time (10 hours). In the right-hand figure, we show the performance gain (wrt. to time) achieved by using FLASH. All the bars above the dashed line (y=1) performs worse than FLASH.

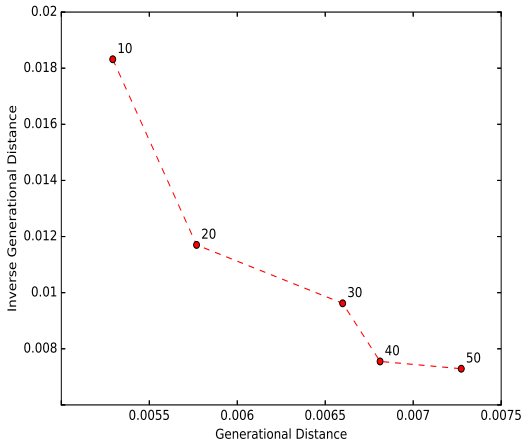

Figure 5.11 The trade-off between the number of starting samples (exploration) and number of steps to converge (exploitation). The ideal point in these trade-off curves is (0,0), which mean the algorithm has perfect convergence ($GD = 0$) and perfect diversity ($IGD = 0$). The trade-off curve for multi-objective performance configuration optimization is shown with budget of 50 evaluations.

5.9 Threats to Validity

Reliability refers to the consistency of the results obtained from the research. For example, how well can independent researchers reproduce the study? To increase external reliability, we took care to either define our algorithms or use implementations from the public domain (SciKitLearn) [81]. All code used in this work are available online [15].

Validity refers to the extent to which a piece of research investigates what the researcher purports to investigate [85]. *Internal validity* concerns with whether the differences found in the treatments can be ascribed to the treatments under study.

For the case-studies relating to configuration control, we cannot measure all possible configurations in a reasonable time. Hence, we sampled only few hundred configurations to compare the prediction to actual values. We are aware that this evaluation leaves room for outliers and that *measurement bias* can cause false interpretations [62]. We also limit our attention to predicting PF for a given workload; we did not vary benchmarks.

Internal bias originates from the stochastic nature of multi-objective optimization algorithms. The evolutionary process required many random operations, same as the FLASH was introduced in this paper. To mitigate these threats, we repeated our experiments for 20 runs and reported the median of the indicators. We also employed statistical tests to check the significance of the achieved results.

[15] http://tiny.cc/flashrepo/

It is challenging to find the representatives sample test cases to covers all kinds of domains. We just selected four most common types of decision space to discuss the FLASH basing on them. In the future, we also need to explore more types of SBSE problems, especially the problem with other types of decisions. We aimed to increase *external validity* by choosing case-studies from different domains.

5.10 Discussion

5.10.1 Can ideas from different communities be combined?

A wide range of SE tasks is a optimization problem where an optimization algorithm must navigate through a complex space of constraints to find solutions while balancing between multiple competing objectives. Software engineering problems used in this paper are examples for such problems. This is curious since similar problems have been tacked in various other communities using different methods:

- Researchers in *software analytics* have shown that their standard algorithms can be readily adapted to optimizing SE tasks – sometimes significantly out-performing EA methods [55, 67, 69].
- For such optimization tasks, machine learning researchers prefer *Bayesian optimization* [33, 89, 110].
- Researchers in *Evolutionary Algorithms* (EAs) are always improving their algorithms [17, 18, 106].

Given this situation a valid question to ask is: *is there any advantage in combining ideas from different research communities?*

In this paper, we experimented with a new method called FLASH which combines ideas from these research communities. When tested on performance configuration optimization, FLASH is often a "better" method.

5.10.2 What is the trade-off between the starting size and budget of FLASH?

There are two main parameters of FLASH which require being set very carefully. In our setting the parameters are *size* and *budget*. The parameter *size* controls the exploration capabilities of FLASH whereas parameter *budget* controls the exploitation capabilities. In figure 5.11, we show the trade-off between generational distance and inverted generational distance by varying parameters *size* and *budget*. The markers in Figure 5.11 are annotated with the starting size of FLASH. The trade-off characterizes the relationship between two conflicting objectives, for

example in Figure 5.11, point (10) achieves high convergence (low GD value) but low diversity (high IGD value). Note, that the curves are an aggregate of the trade-off curves for all the software systems. From the figure 5.11 we observe that:The number of initial samples (*size* in figure 5.6) determines the diversity of the solution. With ten initial samples the algorithm converges (lowest GD values) but lowest diversity (high IGD values). However, with 50 initial samples (random sampling) FLASH achieves highest diversity (low IGD values) but lowest convergence (high GD values). We choose the starting size of 30 because it achieves a good trade-off between convergence and diversity. These values were used in experiments in section 5.8.2.

5.10.3 Can rules learned by CART guide the search?

Currently FLASH does not explicitly reflect on the decision tree to select the next sample (or configuration). But, rules learned by Decision Tree can be used to guide the process of search. Though we have not tested this approach, a decision tree can be used to learn about importance of various configuration options which can be then used to recursively prune the configuration space. We hypothesize that this would make FLASH more scalable and be used to much larger models. We leave this for future work.

5.10.4 Why CART is used as the surrogate model?

Decision Trees is a very simple way to learn rules from a set of examples and can be viewed as a tool for the analysis of a large dataset. The reason why we chose CART is two-fold. Firstly, it is shown to be *scalable* and there is a growing interest to find new ways to speed up the decision tree learning process [91]. Secondly, a decision tree can *describe* with the tree structure the dependence between the decisions and the objectives, which is useful for induction and comprehensibility. These are the primary reason for choosing decision-trees to replace Gaussian Process as the surrogate model for FLASH.

Chapter 6

Conclusions and Future Work

6.1 Thesis Revisited

Modern software systems come with a lot of configuration options, which can be tweaked to modify the functional or non-functional (e.g., throughput or runtime) requirements. Finding the good or optimal configuration to run a particular workload is essential since there is a significant difference between the best and the worst configurations. Many researchers report that modern software systems come with a daunting number of configuration options [103]. The size of the configuration space increases exponentially with the number of configuration options. The long runtimes or cost required to run benchmarks make this problem more challenging.

Prior work in this area used a machine learning method to model the configuration space accurately. The model is built sequentially, where new configurations are sampled randomly, and the quality or accuracy of the model is measured using a holdout set. This strategy makes these methods unsuitable in a practical setting since they can be very expensive. On the other hand, there are software systems for which an accurate model cannot be built.

In this thesis, we introduced techniques and method, which can find good configurations while minimizing the search cost. The central insight of this thesis is: to build a machine learning model (not accurate) which can differentiate between the good and not so good solutions.

The important lessons, we learned during the preparation of the thesis are:

- **Clustering:** Early attempts managed to use random sampling along with machine learning models to build accurate models, which can then be used to find good configurations. However, random sampling and accurate machine learning models can lead to an expensive model building process. We attempt to **reduce the cost of the model building by adding the idea of stratified sampling instead of random sampling** i.e.; we try to cluster the configuration space using a top-down clustering and sampling from each sub-population. We have found stratified sampling can significantly reduce the cost of

model building and also reduce the variance in the model built.
Lessons Learned: Even though we found the stratified sampling approach of **WHAT** works for the software systems explored in our paper [68]. However, while performing external validation (with a more diverse set of software systems), we found that **WHAT** is not efficient (as shown by our preliminary study). Upon further investigation, we found that we make an important assumption: "our clustering algorithm (WHERE in our case) returns meaningful cluster". This assumption meant that we know the distance function applicable to the configuration space, i.e., configurations with shorter distance is similar to the configurations with a larger distance. The ineffectiveness of **WHAT** is primarily because our assumption did not hold for the newer configuration spaces.

- **Ranking:** A retrospective look on the prior work including our work, made us realize that researchers have been tackling the problem of finding good configuration by transforming the problem to a building accurate model problem. However, the fundamental idea in this work is to use ranking as an approach for building models. The ranking is a useful paradigm to solve the software configuration optimization because (1) ranking is the ultimate goal, (2) Ranking is extremely robust since it is only mildly affected by errors or outliers, and (3) Ranking reduces the number of training samples required to train a model.
  Lessons Learned: The model is built sequentially, where new configurations are sampled randomly, and the quality or accuracy of the model is measured using a holdout set. The size of the holdout set in some cases could be up to 20% of the configuration space and need to be evaluated (i.e., measured) before even the machine learning model is entirely built. This strategy makes these methods unsuitable in a practical setting since the generated holdout set can be (very) expensive.

- **Sequential Model-based Optimization:** The problem of finding the (near) optimal configuration is expensive and often infeasible using the current techniques. A useful strategy could be to build a machine learning model which can differentiate between the good and not so good solutions. FLASH, a Sequential Model-based Optimization (SMBO), is a useful strategy to find extremes of an unknown objective. FLASH is efficient because of its ability to incorporate prior belief as already measured solutions (or configurations), to help direct further sampling. Here, the prior represents the already known areas of the search (or performance optimization) problem. The prior can be used to estimate the rest of the points (or unevaluated configurations). Once one (or many) points are evaluated based on the prior, the posterior can be defined. The posterior captures the updated belief in the objective function. This step is performed by using a machine learning model, also called *surrogate model*. The concept of FLASH can be simply stated as:

- Given what one knows about the problem,
- what can be done next?

The "given what one knows about the problem" part is achieved by using a machine learning model whereas "what can be done next" is performed by an acquisition function. Such acquisition function automatically adjusts the exploration ("should we sample in uncertain parts of the search space") and exploitation ("should we stick to what is already known") behavior of the method.

Lessons Learned: Much current research (including all the approaches discussed in this thesis) has explored this problem, usually by creating performance models that predict performance characteristics. While this approach is cheaper and more effective than manual configuration, it still incurs the expensive of extensive data collection about the software. This is undesirable since this data collection has to be repeated if ever the software is updated on the workload of the system changes abruptly. In such a scenario, all prior research suffers from the same drawback: these approaches do not learn from previous optimization experiments and must be rerun whenever the environment of the experiments change. Note our use of the term environment. This refers to the external factors influencing the performance of the system such as workload, hardware, version of the software.

6.2 Future Work

There are many fruitful avenues of research that are worthwhile to pursue but beyond the scope of this dissertation. Here are just a few directions:

Can expert knowledge be used to increase the rate of convergence?

During our many conversations with the people in the industry, we get pushback regarding the automated approaches. The common theme of this conversation is generally, "we have been configuring our software systems for decades, and we have people who can configure these systems in their sleep". While this line of reasoning is generally flawed and there are several pieces of evidence (as discussed in our related work session). Pooyan Jamshidi has shown several such examples. Figure 6.1 is one such example with Cassandra. However, the rate of convergence is FLASH is dependent on the initial configurations (currently selected using random sampling). So, if expert knowledge can be successfully extracted and embedded into the initial sample, it can further reduce the cost of optimization. There has been some effort made in this direction [27, 45].

Can we learn from our experience to increase the rate of convergence or decrease the cost?

Transfer learning can only be useful in cases where the source environment is similar to

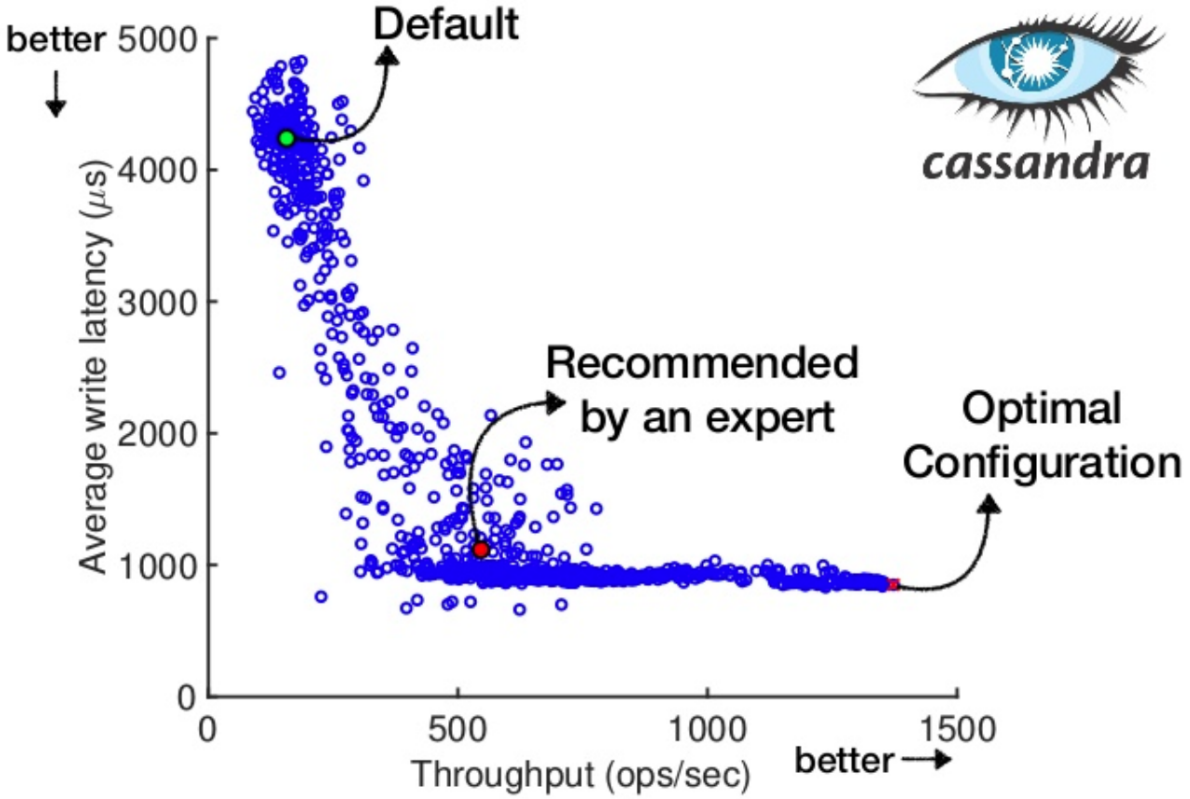

Figure 6.1 Comparison between default configurations, configurations selected by experts and optimal configuration. From http://tiny.cc/existingTechniques.

the target environment. If the source and the target are not similar, knowledge should not be transferred. In such an extreme situation, transfer learning can be unsuccessful and can lead to a *negative transfer*. Prior work on transfer learning focused on "What to transfer" and "How to transfer", by implicitly assuming that the source and target are related to each other. Hence, that work failed to address âĂIJWhen to transfer.âĂİ Jamshidi et al. [50] alluded to this and explained when transfer learning works but, did not provide a method which can help in selecting a suitable source. There is a need for some effort in solving the problem of performance optimization by choosing an appropriate source to transfer knowledge. We have made some progress in this area [71].

Does learning about landscape [1] provide us with information about which search strategy to use?

The techniques used in FLASH (model and the acquisition function) is a subset of a larger space of options. During the development phase of FLASH, we tried various options and narrowed

[1] Landscape refers to features of the problem such as modality (local optimal and plateau), basins of attractions [63]

down on the existing options using trial and error. However, there may be configuration spaces where FLASH is not effective. Since the configuration space is currently a black box to us, there is a need for a fast technique to sample and understand that **landspace** quickly. This information can then be used to select the right combination of model and acquisition function. This problem is currently explored as a different problem namely algorithm configuration, and there is some progress in this area [63]. There is also a great tool available to get a head start in this domain [39].

Can these ideas be applied to other domains?
In an abstract sense, the problems discussed in this thesis falls under a broad realm of black-box optimization. The only challenging component (or rather a constraint) to this problem is each evaluation is costly. Hence, we need to be aware of the cost (both time and computing resources). There are various other domains where the methods, discussed in this thesis, can be applied such as hyperparameter tuning, cloud configuration, etc. We made some progress in applying these techniques in the cloud configuration [45–47].

How to optimize for a large macro-system which involves many micro-systems? The software systems, explored in this thesis, are standalone system and not a collection of heterogeneous systems. We hypothesize that in such collective systems the configuration of one of the participating software system can affect the performance of another participating software system. In such cases, how can we alter the techniques discussed in this thesis to be valuable in such situations.

6.3 Epilogue

The prior works in the area of performance optimization have transformed the problem of finding good configurations into a modeling problem. Researchers have tried to solve this problem by building accurate models, which made this process of optimization (using models) expensive. In this thesis, we have shown that the problem of performance optimization can be tacked as an optimization problem and not a modeling problem. This can be done by building inaccurate models and use the models to explore new regions of the configuration space.

The idea of this thesis can be best encapsulated as:

> **"There's no sense in being precise when you**
> **don't even know what you're talking about."**
>
> **— John von Neumann**

REFERENCES

[1] Agrawal, A. et al. "What is wrong with topic modeling?(and how to fix it using search-based se)". *arXiv preprint* (2016).

[2] Alipourfard, O. et al. "CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics." *Symposium on Networked Systems Design and Implementation*. 2017.

[3] Bergstra, J. et al. "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures". *International Conference on Machine Learning*. 2013.

[4] Bettenburg, N. et al. "Think locally, act globally: Improving defect and effort prediction models". *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. IEEE Press. 2012, pp. 60–69.

[5] Bettenburg, N. et al. "Towards improving statistical modeling of software engineering data: think locally, act globally!" *Empirical Software Engineering* **20**.2 (2015), pp. 294–335.

[6] Biedermann, S. et al. "Hot-hardening: getting more out of your security settings". *Computer Security Applications Conference*. ACM. 2014.

[7] Biedermann, S. et al. "Leveraging Virtual Machine Introspection for Hot-Hardening of Arbitrary Cloud-User Applications." *HotCloud*. 2014.

[8] Bingham, E. & Mannila, H. "Random projection in dimensionality reduction: applications to image and text data". *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2001, pp. 245–250.

[9] Boley, D. "Principal direction divisive partitioning". *Data mining and knowledge discovery* **2**.4 (1998), pp. 325–344.

[10] Breiman, L. et al. *Classification and regression trees*. CRC press, 1984.

[11] Breiman, L. et al. *Classification and regression trees*. Wadsworth & Brooks/Cole Advanced Books & Software, 1984.

[12] Brochu, E. et al. "A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning". *arXiv preprint* (2010).

[13] Chen, J. et al. "Is Sampling better than Evolution for Search-based Software Engineering?" *arXiv preprint* (2016).

[14] Coello, C. A. C. & Sierra, M. R. "A study of the parallelization of a coevolutionary multi-objective evolutionary algorithm". *Mexican International Conference on Artificial Intelligence*. 2004.

[15] Cohen, M. B. et al. "Testing across configurations: implications for combinatorial testing". *SIGSOFT Software Engineering Notes* (2006).

[16] Dalibard, V. et al. "BOAT: Building auto-tuners with structured Bayesian optimization". *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2017.

[17] Deb, K. & Jain, H. "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: Solving problems with box constraints." *IEEE Transactions of Evolutionary Computation* (2014).

[18] Deb, K. et al. "A fast and elitist multiobjective genetic algorithm: NSGA-II". *IEEE transactions on evolutionary computation* (2002).

[19] Deb, K. et al. "A fast and elitist multiobjective genetic algorithm: NSGA-II". *IEEE Transactions on Evolutionary Computation* **6**.2 (2002), pp. 182–197.

[20] Deiters, C. et al. "Using spectral clustering to automate identification and optimization of component structures". *Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2013 2nd International Workshop on*. IEEE. 2013, pp. 14–20.

[21] Drabik, J. *Method and apparatus for automatic configuration and management of a virtual private network*. US Patent App. 10/460,518. 2003.

[22] Du, Q. & Fowler, J. E. "Low-complexity principal component analysis for hyperspectral image compression". *International Journal of High Performance Computing Applications* **22**.4 (2008), pp. 438–448.

[23] Efron, B. & Tibshirani, R. J. *An introduction to the bootstrap*. CRC, 1993.

[24] Faloutsos, C. & Lin, K.-I. "FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets". Vol. 24. 2. ACM, 1995.

[25] Faloutsos, C. & Lin, K.-I. *FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets*. Vol. 24. 2. ACM, 1995.

[26] Feurer, M. et al. "Initializing Bayesian Hyperparameter Optimization via Meta-Learning." *AAAI Conference on Artificial Intelligence*. 2015.

[27] Feurer, M. et al. "Scalable Meta-Learning for Bayesian Optimization". *arXiv preprint arXiv:1802.02219* (2018).

[28] Fletcher, R. *Practical methods of optimization*. John Wiley & Sons, 2013.

[29] Foss, T. et al. "A Simulation Study of the Model Evaluation Criterion MMRE". *IEEE Transactions on Software Engineering (TSE)* **29** (2003), pp. 985–995.

[30] Fu, W. & Menzies, T. "Easy over Hard: A Case Study on Deep Learning". *Foundations of Software Engineering*. ACM, 2017.

[31] Fu, W. et al. "Tuning for software analytics: Is it really necessary?" *Information and Software Technology* (2016).

[32] Fu, W. et al. "Why is Differential Evolution Better than Grid Search for Tuning Defect Predictors?" *arXiv preprint arXiv:1609.02613* (2016).

[33] Gelbart, M. A. et al. "Bayesian optimization with unknown constraints". *arXiv preprint* (2014).

[34] Ghotra, B. et al. "Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models". *37th IEEE International Conference on Software Engineering*. 2015.

[35] Grassberger, P. & Procaccia, I. "Measuring the strangeness of strange attractors". *The Theory of Chaotic Attractors*. Springer, 2004, pp. 170–189.

[36] Guo, J. et al. "Variability-aware performance prediction: A statistical learning approach". *IEEE/ACM 28th International Conference on Automated Software Engineering*. IEEE. 2013, pp. 301–311.

[37] Guo, J. et al. "Data-efficient performance learning for configurable systems". *Empirical Software Engineering* (2017), pp. 1–42.

[38] Hamerly, G. "Making k-means Even Faster." Society for Industrial and Applied Mathematics.

[39] Hanster, C. & Kerschke, P. "flaccogui: exploratory landscape analysis for everyone". *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM. 2017, pp. 1215–1222.

[40] Harman, M. et al. "Search-based software engineering: Trends, techniques and applications". *ACM Computing Surveys* **45**.1 (2012), p. 11.

[41] Henard, C. et al. "Combining multi-objective search and constraint solving for configuring large software product lines". *International Conference on Software Engineering*. 2015.

[42] Herodotou, H. et al. "Starfish: A Self-tuning System for Big Data Analytics". *The 5th Biennial Conference on Innovative Data Systems Research (CIDR 2011)*. Vol. 11. 2011. 2011, pp. 261–272.

[43] Hill, D. N. et al. "An Efficient Bandit Algorithm for Realtime Multivariate Optimization". *SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM. 2017.

[44] *HPE Security Research*. http://files.asset.microfocus.com/4aa5-0858/en/4aa5-0858.pdf. [Online; accessed 10-Nov-2017]. 2015.

[45] Hsu, C.-J. et al. "Scout: An Experienced Guide to Find the Best Cloud Configuration". *ArXiv e-prints* (2018). arXiv: 1803.01296 [cs.DC].

[46] Hsu, C.-J. et al. "Arrow: Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM". *The 38th IEEE International Conference on Distributed Computing Systems (ICDCS 2018)*. 2018.

[47] Hsu, C.-J. et al. "Micky: A Cheaper Alternative for Selecting Cloud Instances". *The 11th IEEE International Conference on Cloud Computing (IEEE CLOUD 2018)*. 2018.

[48] Ilin, A. & Raiko, T. "Practical approaches to principal component analysis in the presence of missing values". *The Journal of Machine Learning Research* **11** (2010), pp. 1957–2000.

[49] Jamshidi, P. & Casale, G. "An Uncertainty-Aware Approach to Optimal Configuration of Stream Processing Systems". *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2016, pp. 39–48.

[50] Jamshidi, P. et al. "Transfer learning for performance modeling of configurable systems: An exploratory analysis". *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press. 2017, pp. 497–508.

[51] Jin, D. et al. "Configurations everywhere: Implications for testing and debugging in practice". *International Conference on Software Engineering*. ACM. 2014.

[52] Jolliffe, I. *Principal component analysis*. Wiley Online Library, 2002.

[53] Kamvar, K. et al. "Spectral learning". *International Joint Conference of Artificial Intelligence*. Stanford InfoLab. 2003.

[54] Kloke, J. & McKean, J. W. "Rfit: Rank-based Estimation for Linear Models". *The R Journal* **4** (2012), pp. 57–64.

[55] Krall, J. et al. "Gale: Geometric active learning for search-based software engineering". *IEEE Transactions on Software Engineering* (2015).

[56] Krall, J. et al. "GALE: Geometric Active Learning for Search-Based Software Engineering". *IEEE Transactions on Software Engineering* **41**.10 (2015), pp. 1001–1018.

[57] Kuhn, D. R. et al. *Introduction to combinatorial testing*. CRC press, 2013.

[58] Lim, D. et al. "Generalizing Surrogate-Assisted Evolutionary Computation". *IEEE Transactions on Evolutionary Computation* **14** (2010), pp. 329–355.

[59] Loshchilov, I. G. "Surrogate-assisted evolutionary algorithms". PhD thesis. 2013.

[60] Marker, B. et al. "Understanding performance stairs: Elucidating heuristics". *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM. 2014, pp. 301–312.

[61] Medeiros, F. et al. "A comparison of 10 sampling algorithms for configurable systems". *International Conference on Software Engineering*. ACM. 2016.

[62] Menzies, T. et al. "Local versus global lessons for defect prediction and effort estimation". *IEEE Transactions on Software Engineering* **39**.6 (2013), pp. 822–834.

[63] Mersmann, O. et al. "Exploratory landscape analysis". *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. ACM. 2011, pp. 829–836.

[64] Mittas, N. & Angelis, L. "Ranking and Clustering Software Cost Estimation Models through a Multiple Comparisons Algorithm". *IEEE Transactions of Software Engineering* (2013).

[65] Myrtveit, I. & Stensrud, E. "Validity and Reliability of Evaluation Procedures in Comparative Studies of Effort Prediction Models". *Empirical Software Engineering (ESE)* **17** (2012), pp. 23–33.

[66] Myrtveit, I. et al. "Reliability and Validity in Comparative Studies of Software Prediction Models". *IEEE Transactions on Software Engineering (TSE)* **31** (2005), pp. 380–391.

[67] Nair, V. et al. "An (Accidental) Exploration of Alternatives to Evolutionary Algorithms for SBSE". *Symposium Search Based Software Engineering*. 2016.

[68] Nair, V. et al. "Faster discovery of faster system configurations with spectral learning". *Automated Software Engineering* (2017), pp. 1–31.

[69] Nair, V. et al. "Faster discovery of faster system configurations with spectral learning". *Automated Software Engineering* (2017).

[70] Nair, V. et al. "Using bad learners to find good configurations". *arXiv preprint arXiv:1702.05701* (2017).

[71] Nair, V. et al. "Transfer Learning with Bellwethers to find Good Configurations". *arXiv preprint arXiv:1803.03900* (2018).

[72] Oh, J. et al. "Finding Near-optimal Configurations in Product Lines by Random Sampling". *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: ACM, 2017, pp. 61–71.

[73] Platt, J. "FastMap, MetricMap, and Landmark MDS are all Nystrom Algorithms". *aistats05*. Ed. by Cowell, R. G. & Ghahramani, Z. Society for Artificial Intelligence and Statistics, 2005, pp. 261–268.

[74] Pukelsheim, F. *Optimal Design of Experiments*. Vol. 50. Society for Industrial and Applied Mathematics, 1993.

[75] Qu, X. et al. "Combinatorial interaction regression testing: A study of test case generation and prioritization". *International Conference on Software Maintenance*. IEEE. 2007.

[76] *Real-World Access Control*. https://www.schneier.com/blog/archives/2009/09/real-world_acce.html. [Online; accessed 10-Nov-2017]. 2009.

[77] Rosset, S. et al. "Ranking-based Evaluation of Regression Models". *Proc. of International Conference on Data Mining (ICDM)*. IEEE. 2005.

[78] Sarkar, A. et al. "Cost-Efficient Sampling for Performance Prediction of Configurable Systems". *30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2015, pp. 342–352.

[79] Sayyad, A. S. et al. "Scalable product line configuration: A straw to break the camel's back". *IEEE/ACM 28th International Conference on Automated Software Engineering*. IEEE. 2013, pp. 465–474.

[80] Sayyad, A. S. et al. "Scalable product line configuration: A straw to break the camel's back". *Automated Software Engineering*. 2013.

[81] *scikit-learn*. http://scikit-learn.org.

[82] She, S. et al. "Reverse engineering feature models". *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE. 2011, pp. 461–470.

[83] Shen, Y. et al. "Fast gaussian process regression using kd-trees". *Advances in neural information processing systems*. 2006.

[84] Shi, J. & Malik, J. "Normalized cuts and image segmentation". *IEEE Transactions on Pattern Analysis and Machine Intelligence* **22**.8 (2000), pp. 888–905.

[85] Siegmund, J. et al. "Views on internal and external validity in empirical software engineering". *Proceedings of the 37th International Conference on Software Engineering*. IEEE. 2015, pp. 9–19.

[86] Siegmund, N. et al. "Predicting performance via automated feature-interaction detection". *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press. 2012, pp. 167–177.

[87] Siegmund, N. et al. “Performance-influence models for highly configurable systems”. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 284–294.

[88] Snoek, J. et al. “Practical bayesian optimization of machine learning algorithms”. *Advances in neural information processing systems*. 2012.

[89] Srinivas, N. et al. “Gaussian process optimization in the bandit setting: No regret and experimental design”. *arXiv preprint* (2009).

[90] Storn, R. & Price, K. “Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces”. *Journal of global optimization* **11**.4 (1997), pp. 341–359.

[91] Su, J. & Zhang, H. “A fast decision tree learning algorithm”. *AAAI Conference on Artificial Intelligence*. 2006.

[92] Tantithamthavorn, C. et al. “Automated parameter optimization of classification techniques for defect prediction models”. *International Conference on Software Engineering*. IEEE. 2016.

[93] Theisen, C. et al. “Approximating attack surfaces with stack traces”. *Proceedings of the 37th International Conference on Software Engineering*. IEEE Press. 2015, pp. 199–208.

[94] Van Aken, D. et al. “Automatic Database Management System Tuning Through Large-scale Machine Learning”. *International Conference on Management of Data*. ACM. 2017.

[95] Van Veldhuizen, D. A. *Multiobjective Evolutionary Algorithms: Classifications, Analyses, and New Innovations*. Tech. rep. 1999.

[96] Vargha, A. & Delaney, H. D. “A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong”. *Journal of Educational and Behavioral Statistics* (2000).

[97] Venkataraman, S. et al. “Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics.” *Symposium on Networked Systems Design and Implementation*. 2016.

[98] Wang, S. et al. “A practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering”. *International Conference on Software Engineering*. 2016.

[99] Wang, T. et al. “Searching for better configurations: a rigorous approach to clone evaluation”. *Foundations of Software Engineering*. 2013.

[100] Wang, Y. et al. "Beyond ranking: Optimizing whole-page presentation". *International Conference on Web Search and Data Mining*. ACM. 2016.

[101] Wang, Z. et al. "Bayesian optimization in a billion dimensions via random embeddings". *Journal of Artificial Intelligence Research* **55** (2016), pp. 361–387.

[102] Weiss, G. M. & Tian, Y. "Maximizing classifier utility when there are data acquisition and modeling costs". *Data Mining and Knowledge Discovery* **17**.2 (2008), pp. 253–282.

[103] Xu, T. et al. "Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software". *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 307–319.

[104] Yadwadkar, N. J. et al. "Selecting the Best VM Across Multiple Public Clouds: A Data-driven Performance Modeling Approach". *Symposium on Cloud Computing*. ACM, 2017.

[105] Zhang, F. et al. "Cross-project Defect Prediction Using A Connectivity-based Unsupervised Classifier". *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. 2016.

[106] Zhang, Q. & Li, H. "MOEA/D: A multiobjective evolutionary algorithm based on decomposition". *IEEE Transactions on Evolutionary Computation* (2007).

[107] Zhang, Y. et al. "Performance Prediction of Configurable Software Systems by Fourier Learning". *30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2015, pp. 365–373.

[108] Zhu, H. et al. "Optimized Cost per Click in Taobao Display Advertising". *arXiv preprint* (2017).

[109] Zhu, Y. et al. "BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning". *Symposium on Cloud Computing*. ACM, 2017.

[110] Zuluaga, M. et al. "Active learning for multi-objective optimization". *Proceedings of the 30th International Conference on Machine Learning*. 2013, pp. 462–470.

[111] Zuluaga, M. et al. "ε-PAL: An Active Learning Approach to the Multi-Objective Optimization Problem". *The Journal of Machine Learning Research* **17**.1 (2016), pp. 3619–3650.