Finding Near-Optimal Configurations in Product Lines by Random Sampling

Jeho Oh, Don Batory, Margaret Myers
University of Texas at Austin
USA

Norbert Siegmund
Bauhaus-University Weimar
Germany

ABSTRACT

Software Product Lines (SPLs) are highly configurable systems. This raises the challenge to find optimal performing configurations for an anticipated workload. As SPL configuration spaces are huge, it is infeasible to benchmark all configurations to find an optimal one. Prior work focused on building performance models to predict and optimize SPL configurations. Instead, we randomly sample and recursively search a configuration space directly to find near-optimal configurations without constructing a prediction model. Our algorithms are simpler and have higher accuracy and efficiency.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; **Search-based software engineering**;

KEYWORDS

software product lines, searching configuration spaces, finding optimal configurations

ACM Reference format:
Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding Near-Optimal Configurations in Product Lines by Random Sampling. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 11 pages.
https://doi.org/10.1145/3106237.3106273

1 INTRODUCTION

Software Product Lines (SPLs) are highly configurable systems. This raises the challenge to find a configuration that has near-optimal performance. An SPL configuration space is often astronomical in size (exponential in terms of *features* – increments in program functionality), and searching it efficiently is hard [35]. There are many reasons: (1) A feature's influence on performance is not easy to determine, because (2) feature interactions introduce performance dependencies with other features [5, 31]. (3) Techniques for true random sampling of configuration spaces are not known; approximations to true random sampling are used instead. And (4) how few samples can be taken for performance models to have acceptable accuracy?

This paper focuses on a fundamental problem in SPLs to find acceptable configurations whose performance is near-optimal. We do ***not*** create a performance prediction model, which then requires an optimizer (*e.g.* using a genetic algorithm [27]) to find good configurations. Instead, we use BDDs to count the number of valid configurations in a configuration space, thereby enabling true random sampling of the space. Doing so allows us to prove theoretically tight bounds on sampling results. Further, we identify features that are statistically certain to improve or degrade program performance [11]. We use these features to recursively constrict the configuration space towards near-optimal configurations. The advantages in doing so are (a) we use simpler algorithms to accomplish what more complicated algorithms do now, (b) our accuracy is better than existing algorithms, and (c) we use fewer samples.

The novel contributions of our paper are:

- True random sampling of valid configurations in an SPL;
- Theoretical bounds on search accuracy from uniform random sampling of configurations;
- A way to progressively shrink a configuration space by exploiting its shape and statistical reasoning;
- Analyses of real systems that shows our approach outperforms prior work in accuracy and the number of samples needed; and
- A demonstration of the scalability of our work to huge configuration spaces.

2 BIG PICTURE OF PRIOR WORK

To predict performance of SPL products (programs), a mathematical performance model is created. Historically, such models are developed manually using domain-specific knowledge [1, 12]. More recently, emphasis has been on general approaches from which performance prediction models are learned or deduced from performance measurements of sampled configurations. Such a performance model is then given to an optimizer, which not only can find near-optimal configurations, but also near-optimal configurations that observe *user-imposed feature constraints* (*e.g.* configuration predicates that exclude feature F and include feature G).

Prediction models estimate the performance of valid configurations [14, 25, 29, 31, 37]. They are deduced from performance measurements of sampled configurations. The goal is to use as few samples as possible to yield a model that is 'accurate'. Finding a good set of samples to use is one challenge; another is minimizing the variance in predictions.

Given an SPL feature model [3], properties of features and their interactions, and user-imposed feature constraints, an optimizer can

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106273

derive valid configurations that satisfy one or more performance objectives using a general search strategy [15, 16, 27, 28, 35].

Let $\mathbb{C}$ be the set of all legal SPL configurations. 1^{st}-order performance models have the following form: $\$P_c$ is the estimated performance of an SPL product P_c with configuration $c \in \mathbb{C}$, where c is a set of selected features and $\$F_i$ is the performance contribution of feature F_i:[1]

$$\$P_c \;=\; \sum_{i\in c} \$F_i \tag{1}$$

Linear models are inaccurate as they do not consider feature interactions. Let $\$F_{ij}$ denote the performance contribution of the interaction of features F_i and F_j, which requires both F_i and F_j to be present in a configuration; $\$F_{ij}$=0 if $F_i \notin c \vee F_j \notin c$. 2^{nd}-order models take into account 2-way interactions:

$$\$P_c \;=\; \left(\sum_{i\in c} \$F_i\right) + \left(\sum_{i\in c}\sum_{j\in c} \$F_{ij}\right) \tag{2}$$

and more generally, n-way interactions add more nested-summation terms to Eq. (2) [31].

When compared to manually-developed performance models [1, 6, 12], an important difference becomes apparent. A manually-developed model:

- Identifies operations $\{O_1 \ldots\}$ invoked by system clients,
- Defines a function $\$O_i$ to estimate the performance of each operation O_i,
- Encodes system workloads in terms of operation execution frequencies, where v_i is the frequency of O_i, and
- Expresses performance $\$P$ of a program P as a weighted sum of frequency times operation cost:

$$\$P \;=\; \sum_{i} v_i \cdot \$O_i \tag{3}$$

Features complicate the cost function of each operation, where configuration $c \in \mathbb{C}$ becomes an explicit parameter:

$$\$P_c \;=\; \sum_{i} v_i \cdot \$O_i(c) \tag{4}$$

The key observation is that manual performance models ***include*** *workload variances in their predictions*, whereas current SPL performance models use a *fixed* workload. Workload variations play a significant role in the performance of SPL products and should not be omitted.

Random sampling. Optimizers and prediction models [14–16, 25, 27, 28, 37] rely on 'random sampling', but the samples used are not provably random. True random sampling would, in effect, enumerate all n legal configurations, randomly choose a number $k \in \{1..n\}$, and use the k^{th} configuration – but this is not done because n could be astronomically large.

One popular alternative is to randomly select features to create a configuration, followed by a filter to eliminate invalid configurations [14, 15, 25, 28, 37]. The drawback of this approach is that it creates too many invalid configurations [16]. Another approach uses SAT solvers to generate valid configurations [16, 27], but this produces configurations with similar features due to the way solvers enumerate solutions. Further, SAT solvers count the number of solutions by enumeration, which is inefficient [7, 8]. Although Henard et al. [16] mitigated these issues by randomly permuting the parameter settings in SAT solvers, true random sampling was not demonstrated.

The top path of Figure 1 summarizes prior work: the configuration space is pseudo-randomly sampled to derive a performance model; samplings are interleaved with performance model learning until a model is 'sufficiently' accurate. That model is then used by an optimizer, along with user-imposed feature constraints, to find a near-optimal performing configuration.

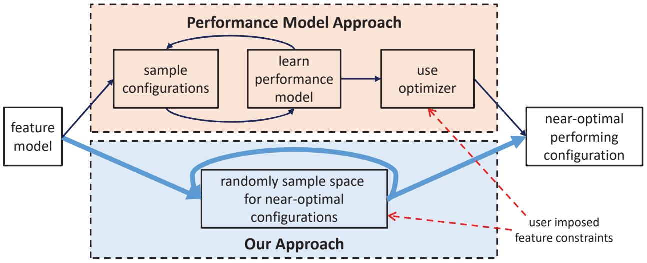

Performance Model Approach
sample configurations
learn performance model
use optimizer
feature model
near-optimal performing configuration
randomly sample space for near-optimal configurations
user imposed feature constraints
Our Approach

Figure 1: Different ways to find good configurations.

Our approach is different. First, we do not use performance models or optimizers. We find good configurations by randomly probing the configuration space directly, measuring the performance of these samples under the required workload. User-imposed feature constraints simply reduce the space that we probe.

Second, we use true random sampling. We encode feature models as *Binary Decision Diagrams* (BDDs) [2], for which counting the number of legal configurations is straightforward. Given the number of legal configurations n, we can randomly select a number $k \in \{1..n\}$, and traverse a BDD to find the k^{th} configuration. This allows us to create accurate mathematical models based on true random selection.

Third, we progressively constrict the configuration space by determining statistically significant features (or their absence) that contribute to good performance. Selecting these features focuses on progressively smaller regions of the configuration space that have near-optimal configurations.

The bottom path of Figure 1 summarizes our approach: we use true random sampling of a constrained configuration space and measure the performance of selected configurations *for a given workload*. We continue sampling until we reach a configuration that exhibits a satisfactory 'accuracy'. We demonstrate later that our technique is more efficient than prior work in the number of samples used, and more accurate than prediction models. *Only when prediction models with fixed workloads are reused will they be less costly – but not necessarily more accurate – than our approach.*

3 SEARCH BY RANDOM SAMPLING

3.1 Counting Binary Decision Diagrams

Two tools are commonly used to analyze propositional formulas: *SAT(isfiability) solvers* [13] and *BDDs*. SAT relies on a *Conjunctive Normal Form* (CNF) representation of a formula to find a solution efficiently. In contrast, a BDD is a data structure that encodes a disjunction of formula solutions, *i.e. Disjunctive Normal Form* (DNF). BDD tools convert non-DNF formulas into BDDs.

Figure 2 shows how a given feature model (feature diagram + cross-tree constraints) can be transformed into a propositional

[1] $\$F_i$ need not be a constant; it could be a sophisticated expression [14].

formula [4], then into a BDD.[2] For now, ignore the integer labels on edges. The name of each node is a variable v; its dashed-line child denotes a *false* or 0 assignment to v and its bold-line child is a *true* or 1 assignment. A terminal node of a BDD is a 0 or 1 box. A path from the root to a box assigns values to variables. A path terminating at the 1 box means that the variable assignments are satisfiable. Path $(1, 1, 1, 0, -)$ means that all configurations with $root = 1, A = 1, C = 1$, and $D = 0$ (the remaining variable B is don't care) are valid solutions of this model.[3]

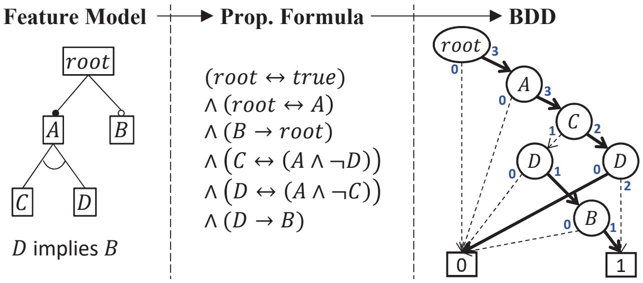

Feature Model
Prop. Formula
BDD
root
A
B
C
D
D implies B
$(root \leftrightarrow true)$
$\wedge (root \leftrightarrow A)$
$\wedge (B \rightarrow root)$
$\wedge (C \leftrightarrow (A \wedge \neg D))$
$\wedge (D \leftrightarrow (A \wedge \neg C))$
$\wedge (D \rightarrow B)$
root
3
0
A
3
0
C
1
2
D
0
1
D
0
2
B
0
1
0
1

Figure 2: Transforming a Feature Model into BDD.

BDDs make it fast and easy to count the valid configurations for a given feature model and user-imposed feature constraints. The integer on each edge in Figure 2 indicates the number of solutions with those variable assignments. We call this a *Counting BDD* (CBDD). The path $(1, 0, -, -, -)$ has zero solutions; path $(1, 1, 0, -, -)$ has one solution. The root or path $(-, -, -, -, -)$ has three solutions, the sum of edges from the root.

Here is why CBDDs are important: CBDDs solve an open problem of how to randomly and uniformly select configurations from a valid configuration space. We can quickly count the size n of a configuration space, generate a random number $k \in \{1..n\}$ (where all numbers in $\{1..n\}$ are equally likely), and convert k into an SPL configuration by a CBDD traversal. In contrast, SAT solvers count solutions by enumeration; for a large configuration spaces, enumeration is impractical. The downside of BDDs is that when formulas are large, BDD creation time may exceed user patience or storage requirements of available memory [2].

An algorithm to create and traverse a CBDD that maps a number to a configuration is straightforward and is presented in [22]. A simple extension includes user-imposed feature constraints. In short, our algorithm creates a CBDD and counts solutions to feature-constrained configuration spaces to sample configurations.

3.2 Performance Stairs in Configuration Spaces

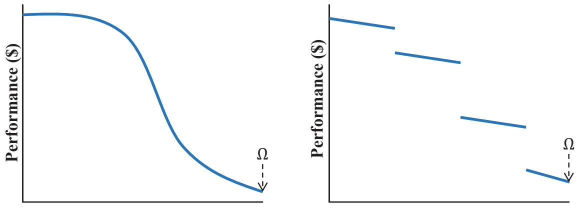
Exploiting the 'shape' of a configuration space is key to searching it efficiently. We may not find the *optimal configuration* Ω – the configuration with the optimal performance – but if we can come provably close to Ω, that will do nicely.

Let $\mathbb{C}$ be set of all legal SPL configurations. Let $c \in \mathbb{C}$ and $\$(c)$ denote the measured or predicted performance of configuration c. A *performance configuration space* (*PCS*) is the set of all (config, performance) pairs:

$$PCS = \{ (c, \$(c)) \mid c \in \mathbb{C} \} \tag{5}$$

where configuration $\Omega \in \mathbb{C}$ has the best performance $\$(\Omega)$.

Now, sort the pairs of *PCS* from worst-performance to best and plot configurations along the X-axis and performance along the Y-axis. We call this a *PCS graph*. We expected a continuous graph such as Figure 3a, where high-valued \$ is bad (worst performance is at the far left) and low-valued \$ is good (best performance is at the far right). Ω anchors the far-right point on X-axis of PCS graphs.



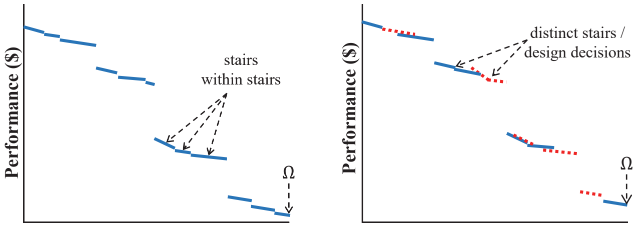
(a) Continuous PCS graph (b) Staired PCS graph

Figure 3: PCS Graphs.

Interestingly, Marker et al. [18] discovered that PCS graphs are staired, 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 feature decisions define the performance characteristics of a segment of a PCS graph (the configuration membership of a stair) while less important feature decisions alter the performance of nearby configurations only slightly (giving a stair its width and slope). In short, *the configurations of a stair share major design decisions* [18].

Figure 4 illustrates two common situations. First, like a fractal, stairs have substairs, recursively. Substairs within different stairs repeat because the same less significant decisions are applied within each stair (see Figure 4a). Second, distinct stairs can overlap because they have similar performance, making it difficult to distinguish common decisions. We use the term *pollution* when the superposition of distinct stairs (forming a downward trending *shelf*) arises.

Figure 4b is the basic shape of a PCS graph that we believe is common in SPLs and will exploit in this paper.



(a) PCS graph with substairs (b) PCS graph with overlaps

Figure 4: Stairs within Stairs.

3.3 Random Selection in PCS Graphs

Let $\mathbb{N}$ be the interval of integers $[1, |\mathbb{C}|]$, one per configuration in $\mathbb{C}$. Randomly sampling n integers from this interval can be regarded as a combinatorial problem. As we are interested in finding Ω, the probability that the largest selected integer, c_{best}, is i units away from $|\mathbb{C}|$ is:

$$p_{|\mathbb{C}|,n}(i) = \binom{|\mathbb{C}| - i - 1}{n - 1} / \binom{|\mathbb{C}|}{n} \tag{6}$$

[2] This is an *ordered BDD*, where Boolean variables are encountered from root-to-terminals in the same order.

[3] This is a *reduced* BDD, meaning unnecessary nodes/variables whose values are immaterial to a solution are eliminated. Otherwise a BDD would contain $2^{number\ of\ variables}$ nodes.

The expected value of i, or the mean distance c_{best} is to $|\mathbb{C}|$, is:

$$E_{|\mathbb{C}|,n} \;=\; \sum_{i=0}^{|\mathbb{C}|-n} i \cdot p_{|\mathbb{C}|,n}(i) \tag{7}$$

As we are interested in huge configuration spaces, we can generalize this analysis by replacing $\mathbb{N}$ with the real unit interval $\mathbb{I} = [0, 1]$, with:

- Dividing each number in $\mathbb{N}$=$[1, |\mathbb{C}|]$ by $|\mathbb{C}|$ to yield $\mathbb{NN} = [\frac{1}{|\mathbb{C}|}, 1]$, and
- Taking the limit $lim_{|\mathbb{C}|\to\infty}\mathbb{NN}$ to produce $\mathbb{I}$.

Every PCS graph is monotonically decreasing. If we randomly select n points in $\mathbb{I}$, c_{best} will be closest to 1. The cumulative probability distribution function for c_{best} is:[4]

$$p_n(X \le x) \;=\; \int_0^x n \cdot x^{n-1} \cdot dx = x^n \tag{8}$$

The average error E_n, or the mean distance c_{best} is to 1, is:

$$E_n \;=\; \int_0^1 (1-x) \cdot n \cdot x^{n-1} \cdot dx \;=\; \frac{1}{n+1} \tag{9}$$

That is, *n randomly selected points partition $\mathbb{I}$ on average into n+1 uniform intervals of length $\frac{1}{n+1}$*. Eq. (9) tells us a simple way to search for a good configuration in a PCS graph: randomly select n configurations and evaluate the performance of each. The best performing selection, c_{best}, is *on average* a distance $\frac{1}{n+1}$ from the best performance at x=1.

Other useful statistics of p_n are $\overline{E}_n$, the second-moment of E_n, and σ_n, its standard deviation:

$$\overline{E}_n \;=\; \int_0^1 (1-x)^2 \cdot n \cdot x^{n-1} \cdot dx \;=\; \frac{2}{(n+1)\cdot(n+2)} \tag{10}$$

$$\sigma_n \;=\; \sqrt{\overline{E}_n - E_n^2} \;=\; \sqrt{\frac{2}{(n+1)\cdot(n+2)} - \left(\frac{1}{n+1}\right)^2} \tag{11}$$

Figure 5 compares the result of Eq. (7) and Eq. (9), sampling 20, 60, and 100 numbers on spaces with different size $|\mathbb{C}|$, shown on the X-axis. To compare two equations, the results of Eq. (7) were normalized by the size of $|\mathbb{C}|$, so that both equations indicate normalized distances to 1, shown on the Y-axis.

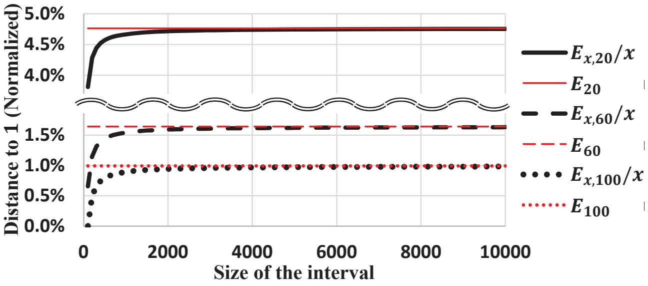


Figure 5: Comparing $E_{|\mathbb{C}|,n}$ and E_n.

For small spaces ($|\mathbb{C}| \le 2000$), Eq. (7) predictions are slightly lower than Eq. (9). When $|\mathbb{C}| \ge 2000$, there is no difference between Eq. (7) and Eq. (9). As we are more interested in huge configuration spaces, we resort to Eq. (9) for the rest of this paper.

Figure 6 plots E_n and σ_n as percentages *in an infinite-size configuration space*; E_n and σ_n values are virtually identical as they lie on top of each other.

Here is what Figure 6 means: If we randomly select n=100 points, c_{best} will be 1% away from 1 on the $\mathbb{I}$-axis with a standard deviation of 1%. If we select n=50 points, c_{best} will be 2% away from 1 with a standard deviation of 2%. As n increases, the interval $[E_n{-}\sigma_n, E_n{+}\sigma_n]$ shrinks. *We will see later that these numbers are good; they say that we do not need many random selections to find a good performing point.*

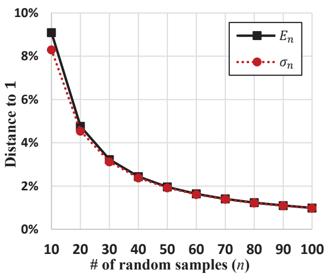


Figure 6: E_n and σ_n.

3.4 Axes of Projections and Main Conjecture

Consider the PCS graph of $\mathbb{I}$×\$ plane of Figure 7. In the last section, we analyzed the performance of selecting n points along the $\mathbb{I}$ axis and choosing the point closest to 1.

We are more interested in the PCS graph of plane $\mathbb{C}$×\$. Each point in $\mathbb{I}$ has an equal probability of being selected. As the CBDD mapping is 1-to-1, each point along $\mathbb{C}$ axis also has an equally probability of being selected. We still have to measure \$(c) for each selected configuration c, but the theoretical results of Eq. (8)–(11) about error distances from c_{best} to 1 in $\mathbb{I}$ are transferred to error distances from c_{best} to Ω in $\mathbb{C}$.

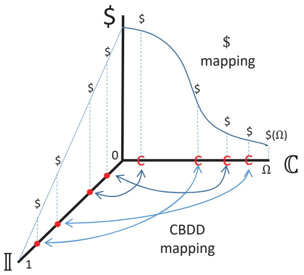
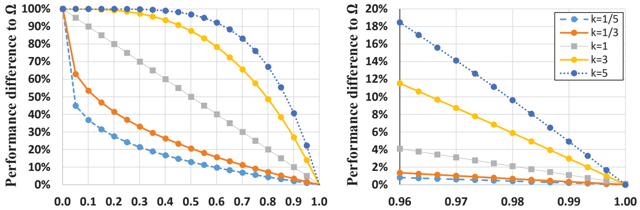


Figure 7: Axes of Projections.

Here is our main conjecture: *there is a correspondence between being q% from Ω along X-axis and q% from \$(Ω) along Y-axis in a PCS graph for small q*. Suppose a PCS graph is defined by $\$_{\Bbbk}(x)$:

$$\$_{\Bbbk}(x) \;=\; 1 - x^{\Bbbk} \tag{12}$$

Figure 8a plots graphs for $\Bbbk \in \{1/5, 1/3, 1, 3, 5\}$. We say $\Bbbk$ is the *curvature* of a PCS graph.



(a) $\$_{\Bbbk}(x)$ PCS Graphs (b) Curvatures in Critical Zone

Figure 8: PCS Graphs and Their Critical Zones.

Let's focus on the interval [.96,1], which contains all configurations whose X-axis value is within 4% of Ω. We call this the *critical zone*; 4% within optimal is a ball-park setting for prior work. Figure 8b shows the critical zone of Figure 8a. The Y-axis plots the %-distance from the best-performance at y=0, namely \$(Ω).

Although the PCS graphs in Figure 8a are non-linear, the curvature $\Bbbk$ reduces to the graph's slope at x=1 in the critical zone.

[4] $x^{n-1} \cdot dx$ is the probability that the first n−1 selections are in the interval [0, x] and dx is the probability that the last selection is at x; n is the normalization constant. Eq. (8) is an instance of the Beta function [9].

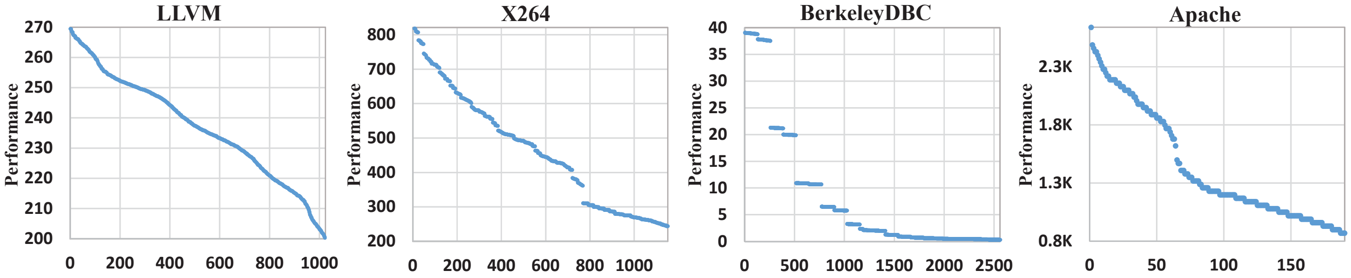


Figure 9: PCS Graphs of Different SPLs.

This slope is the first derivative, $\frac{d}{dx}(1 - x^{\Bbbk})$=$-\Bbbk \cdot x^{\Bbbk-1}$, and in the limit, $lim_{x\to 1} - \Bbbk \cdot x^{\Bbbk-1}$=$-\Bbbk$. That is, the slope of a PCS graph in the critical region is negative $\Bbbk$. Observe:

- When $\Bbbk$=1, the PCS graph $\$_{\Bbbk}(x)$ is a line. If we are q% away from Ω on the X-axis we are also precisely q% away from $\$(\Omega)$ on the Y-axis.
- When $\Bbbk$<1 the graph is *convex*. If we are q% from Ω, we know that performance is $\Bbbk \cdot q$% away from $\$(\Omega)$. A convex PCS graph means that Ω lies on a flat shelf where any configuration on that shelf is near-optimal.
- When $\Bbbk$>1 the graph is *concave*. If we are q% away from Ω, we are $\Bbbk \cdot q$% away from $\$(\Omega)$. A concave PCS means that Ω does *not* lie on a flat shelf and further searching may be warranted.

At this point, we need to look at actual PCS graphs to examine their shape and curvature.

3.5 PCS Graphs of Actual SPLs

Four SPLs were analyzed by Siegmund et al. [30], which were extensively used as the test set for prediction models.[5] Figure 9 shows their PCS graphs. Each is described briefly:

- **LLVM** is a compiler infrastructure, written in C++. It has 11 features and 1024 configurations, where test suite compilation times were measured.
- **X264** is a video encoder library for H.264/MPEG-4 AVC format, written in C. It has 16 features and 1152 configurations; Sintel trailer encoding times were measured.
- **BerkeleyDBC** is an embedded database system, written in C. It has 18 features and 2560 configurations where benchmark response times were measured.
- **Apache** is an open-source Web server. It has 9 features with 192 configurations, where the maximum server load size was measured through `autobench` and `httperf`.

Figure 10 shows their PCS graphs in the critical zone. Most have $\Bbbk$<1; this means that as c_{best} approaches Ω on the X-axis, we know its performance is very close to $\$(\Omega)$. The reason is that all configurations lie on a flat shelf whose performance differences are minimal. Choosing any configuration on this shelf will do.

SPLs whose PCS graphs where $\Bbbk$>1 pose more of a challenge. Their configurations do not lie on a flat shelf; performance noticeably improves as one gets closer to Ω. If we know the curvature $\Bbbk$ of a PCS graph, we can estimate how far we are from $\$(\Omega)$. Examples: **LLVM** has a curvature of $\Bbbk$=2. If we believe our best sample is q% from Ω, we can infer that we are $2 \cdot q$% from $\$(\Omega)$.

From the above, a key metric that determines when to stop sampling or if more sampling is needed is to estimate a PCS graph's curvature $\Bbbk$. More on this in Section 4.2.

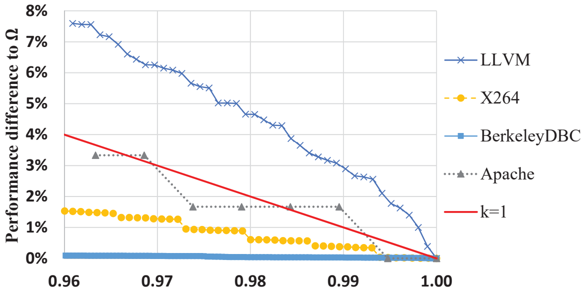


Figure 10: PCS Graphs of SPLs in the Critical Zone.

4 RECURSIVE SEARCHING

The best configuration c_{best} out of 10 random samples will have an average error/distance of 9% = $\frac{1}{11}$ along the X-axis from Ω. 100 random samples (or 10× the previous number) are needed to find c_{better} that reduces the error to 1%=$\frac{1}{101}$. Note that approximately 90% of the additional 90 samples will not perform better than c_{best}. This is wasteful. We call this *Non-Recursive Searching (NRS)* to distinguish it from our upcoming approach.

Random sampling with recursion offers improvement. Ideally, 10 samples of the original configuration space can identify the best 10% of this space, and another 10 samples can constrict this smaller space by another 10% to the best 1% for a total cost of 20 samples. This is better; this is *recursive searching*.

The key driver for recursion is performance stairs. As stairs have different average performances due to different feature decisions, finding the best performing stair that contains Ω improves the result of sampling.

Consider the PCS graph of LLVM in Figure 9. This graph looks almost linear with no stairs. However, stairs become visible as configurations are analyzed based on features they have in common. In each graph of Figure 11, configurations are partitioned by whether they include a particular feature or not, which is done recursively. The most influential feature (or its negation) is selected to partition the configurations. Then, from the remaining features, the most influential feature (or its negation) regarding the partition that better performs in overall is selected as the next feature to partition.

Each graph in Figure 11 clearly shows the effect of performance stairs, where one partition is constricted by the next via the selection

[5] Among 6 available datasets, we used only systems that had all legal configurations measured. Our analyses of systems with incomplete datasets we felt were misleading, although they exhibited similar performance to systems with complete dataset. Note that gathering the performance data of all systems took 2 months of CPU time [30].

of a 'good-performing' or 'noteworthy' feature. Each such feature defines a 'stair'. Thus, devising an algorithm that finds a good stair on which to recurse is the crucial next step. We use the *Statistical Recursive Searching* (SRS) algorithm defined next.

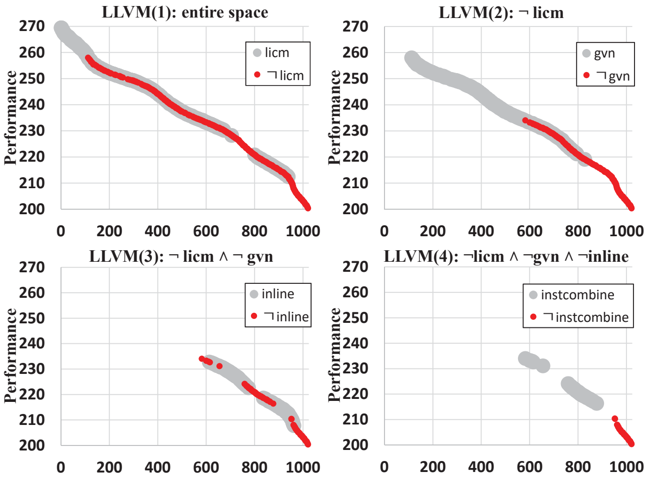


Figure 11: Recursive Stairs of LLVM.

4.1 Statistical Recursive Searching (SRS)

There are at least two basic approaches to find a good stair. One directly focuses on the feature decisions that are expected to form the best stair by using common feature decisions in the k-best sampled configurations. Another exploits how stairs are recursively formed, observing feature influence on performance from samples.

We discovered the k-best approach has drawbacks: finding a good k value is hard. Small k often yields highly variant and inaccurate results. Larger k requires more samples to collect as fewer commonalities are found among them.

Similarly, we discovered the second approach also has drawbacks: feature interactions and constraints often led to misinterpreting a feature's influence by making decisions inconsistent with Ω.

SRS combines the advantages of both approaches while minimizing their disadvantages. SRS utilizes the k-best approach by setting k=2. Then SRS identifies features that are common to the k=2 best – and here's the difference – identifying *noteworthy* features among them – those features (or their negation) that statistically are certain to improve performance [11]. SRS then constricts the search space to configurations that comply with noteworthy features decisions, and the SRS algorithm recurses; see Algorithm 1.

4.1.1 Recursion Logic.

At each recursive step, n random samples are taken. The performance influence of feature decision d is determined as follows:

- $\$(d)$ measures the average performance over the n samples that have feature d. $\$(\neg d)$ measures the average performance of the n samples that do not have d.
- $\Delta(d)$=$\$(d)$ − $\$(\neg d)$ is the performance influence of feature d. The sign of $\Delta(d)$ indicates whether d improves (negative value) or degrades (positive value) average performance.
- $t\text{-}Test(d)$ is the result of Welch's t-test [34] on whether $\$(d)$ is better than $\$(\neg d)$ with 95% confidence.

Welch's t-test evaluates the hypothesis that the mean of one sample group is higher than the other [34]. That is, it determines whether the $\Delta(d)$ from samples is reliable to distinguish whether d or $\neg d$ is a noteworthy feature. Noteworthy features constrict the configuration space for the next recursion by becoming additional constraints that samples must satisfy at the next recursive step.

4.1.2 Termination Logic.

Recursion terminates when no new noteworthy features are discovered. SRS assumes that the configuration space cannot be reduced further, so that random sampling on this region yields a good configuration. If the size of the constricted configuration space is smaller than n, all configurations in the space are measured.[6]

Algorithm 1: SRS algorithm

```
Procedure SRS($n, FM, dSet$):
    Input  : $n$ (number of samples per recursion)
             $FM$ (feature model propositional formula)
             $dSet$ (set of feature decisions (initially empty))
    Output: $c_{best}$ (best configuration found (set of features))
    $samples$ ← sample $n$ configs. from $FM \wedge dSet$;
    sort $samples$ so that $samples[0]$ has best performance;
    $commons$ ← common feature decisions in $samples[0]$ and
      $samples[1]$;
    for each $decision$ in $commons$ do
        if $(\Delta(decision) < 0) \wedge tTest(decision)$ then
            add $decision$ to $dSet$;
    if $dSet$ unchanged from previous recursion then
        return $samples[0]$;
    else
        return SRS ($n, FM, dSet$);
```

4.2 Estimating PCS Graph Curvature 𝕜

Let $\Delta_x(r)$ denote the size of a stair in terms of the number of configurations at the r^{th} recursion, where $\Delta_x(1)$ = the number of configurations in the original space. $\Delta_x(r)$ decreases with increasing r. Using CBBDs, we can compute $\Delta_x(r)$ with pin-point accuracy.

There are two ideal values – values that cannot be computed unless performance data for the entire configuration space is available. Let $\delta_x(r)$ be the error (distance) from the best sampled configuration c_{best} to Ω along the X-axis at r-th recursion:

$$\delta_x(r) = \frac{\$(c_{best}) \le \#configs \le \$(\Omega)}{total\,\#of\,configs}+ \tag{13}$$

And let $\delta_y(r)$ be the relative performance difference between the best configuration c_{best} and Ω at r-th recursion as:

$$\delta_y(r) = \frac{\$(c_{best}) - \$(\Omega)}{\$(\Omega)} \tag{14}$$

We estimate $\delta_x(r)$ and $\delta_y(r)$ from random samples by making the following best-case assumptions:

- Samples are $\frac{\Delta_x(r)}{n+1}$ away from each other on X-axis.
- Recursion always finds the best stair that contains Ω.
- Pollution is negligible between c_{best} and Ω.

[6] SRS mostly avoids local minima by searching a PCS graph, which is monotonically decreasing [36]. This is elaborated in [22].

Figure 12 depicts how $\delta_x(r)$ and $\delta_y(r)$ can be estimated with these assumptions.

- $E(\delta_x(r))$, our estimate of $\delta_x(r)$, is based on the size of the current stair and number of samples:

$$E(\delta_x(r)) = \frac{\Delta_x(r)}{\Delta_x(1)} \cdot \frac{1}{(n+1)} \tag{15}$$

  where c_{best} is $\frac{\Delta_x(r)}{n+1}$ configurations from Ω along X-axis.
- We compute the slope or curvature $\mathbb{k}$ of a stair using the rightmost $\frac{1}{3}$ of its samples. We found $\frac{1}{3}$ works well, computed by a standard least squares method [9].
- $E(\delta_y(r))$, our estimate of $\delta_y(r)$, is a linear extrapolation of $\$(c_{best})$, using slope $\mathbb{k}$ to estimate $\$(\Omega)$:

$$E(\$(\Omega)) = \$(c_{best}) \; + \; \mathbb{k} \cdot \frac{\Delta_x(r)}{n+1} \tag{16}$$

Then $E(\delta_y(r))$ is:

$$E(\delta_y(r)) = \frac{\$(c_{best}) - E(\$(\Omega))}{E(\$(\Omega))} \tag{17}$$

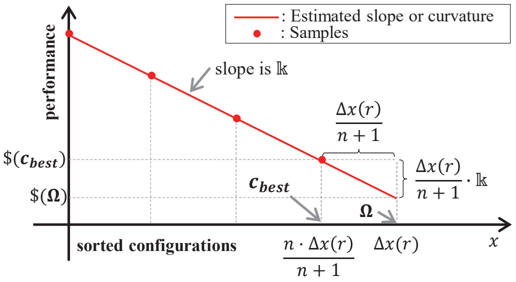

: Estimated slope or curvature
: Samples
performance
slope is $\mathbb{k}$
$\frac{\Delta x(r)}{n+1}$
$\$(c_{best})$
$\frac{\Delta x(r)}{n+1} \cdot \mathbb{k}$
$\$(\Omega)$
c_{best}
Ω
sorted configurations
$\frac{n \cdot \Delta x(r)}{n+1}$
$\Delta x(r)$
x

Figure 12: Estimating $\delta_x(r)$ and $\delta_y(r)$.

At each recursion, we report $[E(\delta_x(r)), E(\delta_y(r)), \mathbb{k}]$ triples to the user to decide whether the best solution found so far is accurate enough, thereby stopping the recursion before Algorithm 1 stops itself and eliminating the need for further costly sampling. The results of the next section are based on Algorithm 1 stopping itself.

5 EVALUATION

Five research questions evaluate our work:

RQ1 : Does our sampling theory for NRS match observations?
RQ2 : Is SRS more efficient than NRS?
RQ3 : Why does SRS work?
RQ4 : Is SRS better than existing approaches?
RQ5 : Do NRS and SRS scale to large configuration spaces?

To answer these questions, we used the data by Siegmund et al. [30] as ground-truth, presented in Section 3.5.

5.1 RQ1: Does Our Sampling Theory for NRS Match Observations?

We compared the theoretical predictions of NRS using E_n, Eq. (9), with the average of measured values for $\delta_x(1)$, Eq. (13). For Apache, Eq. (7) was used instead as the configuration space is tiny. We performed 100 experiments for each value of n. For each system, the experiments started with n at 10 to 100 incremented by 10, plotted for comparison with E_n; see Figure 13. These graphs confirm a close agreement between NRS theory and observations: their differences are imperceptible.

For **RQ1**, our sampling theory matches empirical observations.

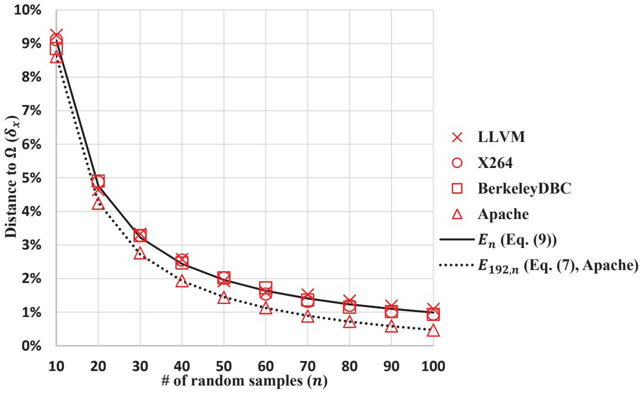


Figure 13: NRS Theory vs. Empirical Observations.

5.2 RQ2: Is SRS More Efficient than NRS?

We compared the accuracy of SRS and NRS using an equal number of samples and collected the following data:

- δ_x is the true X-axis accuracy of SRS when it terminates;
- n is the number of samples per recursion;
- N is the total number of samples taken by SRS; and
- E_N is the theoretical accuracy of NRS assuming N configurations are randomly sampled.

Note: We do not report δ_y values here. A decrease in δ_x is never matched by an increase in δ_y in a PCS graph. δ_y values are important but only when comparing SRS with existing approaches, which we do in **RQ4**.

Figure 14 plots averages of 100 experiments with different n values. While both δ_x and E_N decrease sharply with increasing n, δ_x is on average better than E_N.

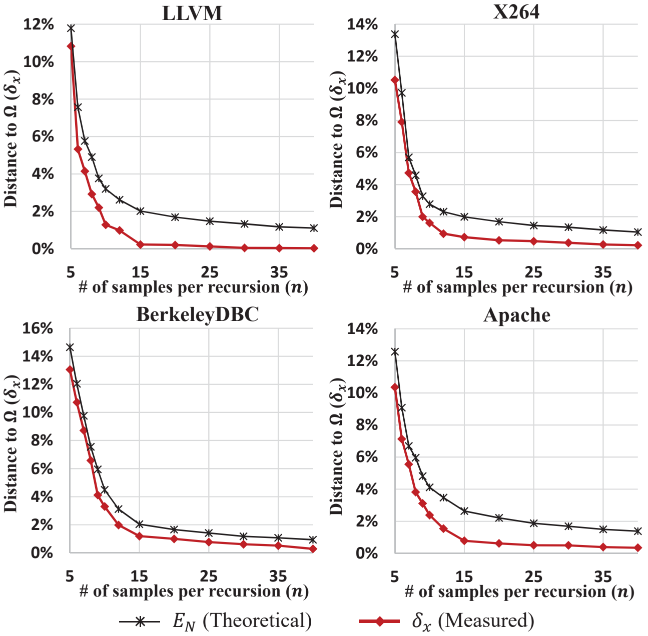


Figure 14: Comparison between SRS and NRS.

For **RQ2**, SRS is more efficient than NRS when the number of samples per recursion n exceeds 15.

5.3 RQ3: Why Does SRS Work?

We collected the following measurements to understand how SRS performs, all taken at SRS termination:

- N the total # of samples taken,
- d is the total # of noteworthy features selected,
- ρ is the % of noteworthy features that belong to Ω, and
- r is depth of recursion.

Figure 15 plots these measures *w.r.t.* n for all 4 systems. Reinforcing the results of **RQ2**, the d, ρ, and r saturate at n=15; indicating that recursion works as desired. As n increases, accuracy increases at the cost of a linearly increasing N.

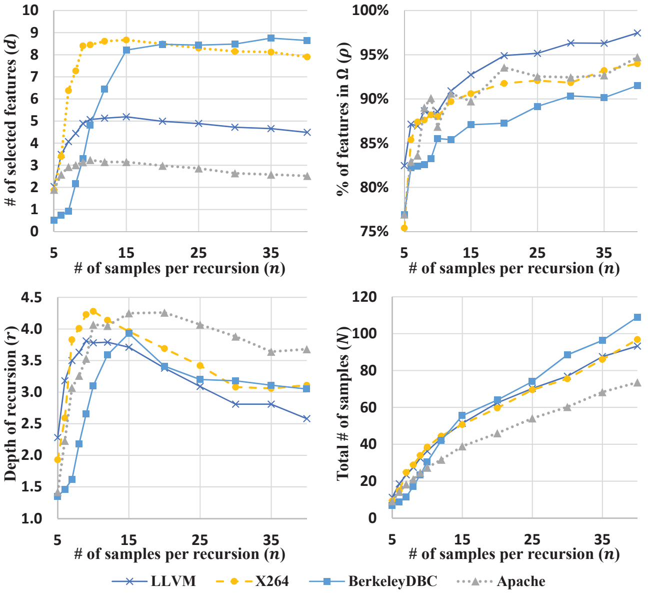


Figure 15: Results of SRS Recursion.

For **RQ3**, SRS works because it requires relatively few samples per recursion, it accurately predicts features that belong to Ω, and relatively few recursions are needed.

5.4 RQ4: Is SRS Better than Existing Approaches?

We determined the best configuration that can be returned by existing prediction models, and derived their δ_y value with respect to the total number of samples N used to construct the prediction model. We compared our results with the best results to date, Sarkar2015 [25] and Siegmund2012 [31],[7] which had their tool and generated prediction models available at [26, 30].

5.4.1 *Comparison with Sarkar2015.*

Sarkar2015's prediction model uses a *Classification And Regression Tree* (CART) of features, based on how randomly-sampled configurations can be partitioned by features. Each leaf node of the CART is a group of sampled configurations that share the same decisions (feature selections). The tree does not cover all features, but only the ones that are significant to performance. When a configuration is queried, CART is traversed to find a leaf that matches its decisions. The average performance of the sampled configurations within the leaf is returned as the predicted performance.

To compare with SRS, the leaf with the smallest average performance was regarded as the predicted performance of the best configuration. δ_y was derived as the relative error between (Ω) and this value. Using their tool, 20 prediction models were created to derive δ_y and averaged, for different sample sizes.

Figure 16 plots δ_y of SRS over N, as well as the values derived from Sarkar2015, plotted as squares. The graphs show that *SRS obtained the same δ_y value with fewer samples* (N) *and found better δ_y values with same* N, except when N<10, where statistical reasoning is not meaningful. For example, in BerkeleyDBC, Sarkar2015 used 110 samples to obtain an accuracy of δ_y=20% (see point ▣ in Figure 16). SRS needs 20 samples to produce this accuracy. And when SRS uses 110 samples, it obtains an accuracy of $\delta_y \leq$0.5%.

Further, their results did not show a clear trend over the number of samples, as larger N did not necessarily lead to a smaller δ_y. In contrast, SRS clearly shows a decrease of δ_y as N increases.

Here is why: CART takes the average performance of configurations as predicted performance, which cannot be better than the best sampled configuration among them. Instead, SRS searches the space directly to find the best-performing configuration it can.

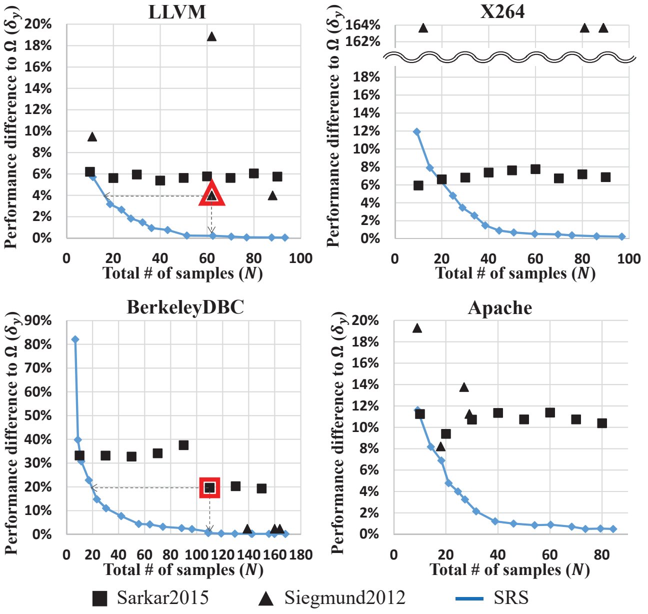


Figure 16: Comparison with Sarkar2015 and Siegmund2012.

5.4.2 *Comparison with Siegmund2012.*

Siegmund2012's prediction model assigns performance values to key features and their interactions using configuration measurements and linear programming. The resulting model can predict the performance of any legal configuration. We defined δ_y for this prediction model as follows:

$$\delta_y = \frac{\$(c_{predicted\ best}) - \$(\Omega)}{\$(\Omega)} \tag{18}$$

where $\$(c_{predicted\ best})$ is the actual, not predicted, performance of $c_{predicted\ best}$, the best performing configuration according to their prediction model. To build their model, Siegmund2012 used different strategies to select configurations. As different strategies

[7] It is unclear to us on how to compare our results to a newer version of Sigmund2012 [29], as they extend their work with numerical features (features whose values are within a range of real numbers); simply ignoring numerical features was not possible.

used different numbers of samples, we measured δ_y for different strategies. Figure 16 plots the prediction model results of Siegmund2012 as triangles.

As with Sarkar2015, *SRS obtained the same δ_y value with fewer samples (N) and found better δ_y values with the same N*. For example, Siegmund2012 used 62 samples to obtain an accuracy of δ_y=4% for LLVM (see see point ▲ in Figure 16). SRS needed only 17 samples to produce this accuracy. And when SRS uses 62 samples, it obtained an accuracy of δ_y=0.2%.

Like Sarkar2015, more samples did not guarantee a better δ_y value, nor was there consistency across systems, as greatly different δ_y and N values were observed. SRS clearly shows a decrease of δ_y as N increases.

For **RQ4**, SRS outperforms existing prediction models, even assuming an optimizer always finds the best configuration based on the prediction model.

5.5 RQ5: Do NRS and SRS Scale to Large Configuration Spaces?

Zhang et al. [37] created large configuration spaces by ×-composing multiple SPLs (see Table 1). Configurations from each SPL are combined by taking the union of their features and summing their performance values.

Table 1: Combined SPLs for Scalability Evaluation

Combined Systems	# of Features	# of Configs.
Apache × LLVM × BerkeleyDBC	38	503,316,480
Apache × X264 × BerkeleyDBC	51	566,231,040
LLVM × X264 × BerkeleyDBC	53	3,019,898,880
Apache × X264 × LLVM × BerkeleyDBC	62	579,820,584,960

Demonstrating the scalability of NRS is simple: Eq. (9) defines NRS performance for configuration spaces of size 2000 to infinity. Figure 13 showed how SPLs of size up to 2560 match the predictions of Eq. (9). Averaging 100 experiments for each value of n, Figure 17 shows how the two smaller composite SPLs of Table 1, which are 200K times larger than the biggest (2560 of BerkeleyDBC) in Figure 13, match Eq. (9). (Extending Figure 17 for n = 10, 20 was infeasible, as ≥3.5% of these huge spaces exceeded the memory capacity of our machines).

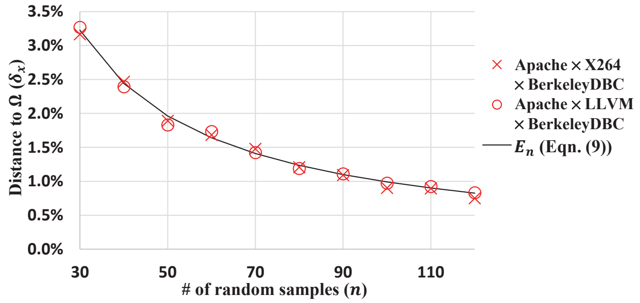


Figure 17: NRS Theory vs Observations on Large Spaces.

Demonstrating the scalability of SRS is similar. Performance graphs for composite SPLs should all have the same shape and should match those of SPLs with small configuration spaces, as in Figure 14. Figure 18 shows this isomorphism in all four composites. Note that SRS performs *better* than NRS in large spaces than in small; we believe that the initial noteworthy features SRS selects are the most effective candidates for each individual SPL in a composite, hence the percentage improvement appears better. So this might be an artifact of using composite SPLs.

For **RQ5**, NRS and SRS scale to large configuration spaces. As before with smaller spaces, SRS outperforms NRS.

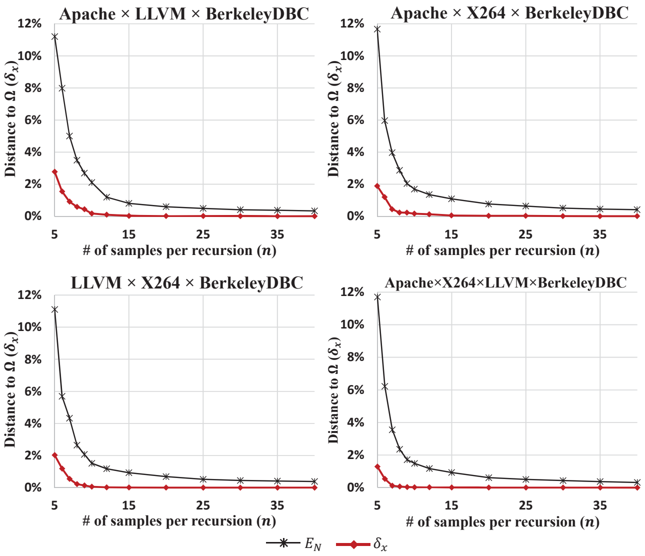


Figure 18: δ_x of Large Configuration Spaces.

5.6 Threats to Validity

Internal Validity. We used ground-truth data of [30] which are measurements of real systems. While there may be errors in measurements, this dataset was utilized by other researchers [14, 25, 29, 31, 37]. We believe that the threat of comparing different approaches was sufficiently controlled.

To control the randomness of sampling, we performed 100 experiments and averaged the results. While there are outliers that threaten our results, δ_x for both NRS and SRS followed a Beta-distribution, indicating that they are marginal and can be controlled.

External Validity. We evaluated our approach based on 6 real-world systems with different domains and numbers of features. We provided a mathematical argument on the system-independence of NRS, statistical reasoning of SRS may depend on the number of features and their influence on performance. We are aware that SRS performance may not generalize to all systems due to this, identical trends from our evaluations across systems and their combinations gives confidence that our conclusions should hold for other SPLs.

6 RELATED WORK

Section 3 placed our research in perspective with prior work. We elaborate key topics in more detail below.

6.1 Performance Prediction Models

A performance prediction model is a function Φ(c) that returns an estimate of the expected performance of an SPL configuration c, for all legal configurations. Aside from the two approaches described in

Section 5.4, Sarkar et al. [25] used projective sampling to minimize the cost of constructing a CART model for performance prediction. Projective sampling attempts to find the optimal sample size by approximating the learning curve of the prediction model's accuracy. Siegmund et al. [29] extended their previous work [31] with numeric features and an iterative process to build performance-influence models, which we do not cover yet. Zhang et al. [37] used Fourier transformation to create a prediction model that not only predicts performance, but also estimates its accuracy; it is unclear if this approach scales beyond 30 features. All of these works are not directly comparable with SRS, as their evaluation measured the average prediction accuracy over multiple test configurations and do not provide means for finding the near-optimal configurations.

6.2 Optimizers

An optimizer finds configurations that satisfy multiple performance constraints from a given feature model and properties of each feature. White et al. [35] proposed an approach based on linear programming, which transforms the given feature model with budget constraints into a knapsack problem. Guo et al. [15] applied a genetic algorithm to search for the optimal configuration. From randomly selected configurations, they crossover good performing configurations for mutation and modify invalid configurations. Sayyad et al. [27, 28] elaborated on *Indicator-Based Evolutionary Algorithm* (IBEA) for selecting optimal features regarding multiple objectives, which outperformed other evolutionary algorithms. They also proposed a heuristic that uses precomputed valid configurations as a seed for the evolutionary algorithm, to improve the scalability of the approach. Henard et al. [16] extended IBEA with SAT solver to generate random configurations and filter out the invalid configurations from mutations, to improve scalability.

These evolutionary approaches perform randomized mutation of configurations, which often leads to invalid configurations. They require significant effort to find suitable parameter settings, which are system-specific and require more than 100 initial samples [23].

6.3 Counting Configurations

Counting configurations is known as the *model counting problem*, which is regarded as a more complicated problem than checking the satisfiability [8]. SAT solvers were extended to exactly [32] or approximately [10] count the number of solutions from a given propositional formula. BDDs can count the number of solutions via their construction. This is advantageous when multiple queries are made to a single formula, as the BDD can be reused [8].

Benavides et al. and Pohl et al. [7, 24] compared CSP, SAT, and BDD solvers on counting configurations, where BDD was much faster than the others, given enough memory. Mendonca et al. [20] provided a reasoning and configuration engine SPLOT, which uses BDD to count the number of valid configurations. Mendonca et al. [21] proposed heuristics to reduce the size of BDD through variable ordering inferred from a feature model, which improves the scalability up to 2000 features.

6.4 Sampling Configurable Systems

Efficient testing strategies for configurable systems rely on sampling. Liebig et al. [17] compared different sampling algorithms with regards to scalability. Random sampling was considered infeasible as most samples were invalid when features are randomly selected, due to feature constraints. Medeiros et al. [19] also compared different sampling algorithms for fault detection capability. Their work randomly selected features, eliminating invalid configurations. But again, random sampling features does not guarantee random sampling of configurations [14, 15, 25, 28, 37].

In contrast, we randomly sample from a set of valid configurations. #SAT solvers [10, 32] are SAT solvers that also can count the number of solutions. The key to using #SAT is to determine how to uniquely map a given number to a specific configuration. CBDDs provide this capability directly [8, 33].

7 CONCLUSIONS

Creating performance models that can predict the performance of any SPL configuration is a worthy goal; it must be used with an optimizer that knows how to search a large configuration space efficiently. But it is also an expensive approach, as the performance model must be reused in different situations to amortize the cost of its development. A key assumption in this line of work is measuring performance for a fixed workload; should that workload change, a new performance model may need to be created.

We eliminated the middle-men of performance models and optimizers by random sampling the configuration space directly and using sampled configurations to progressively constrict the space. Our paper makes five contributions:

(1) We showed how true random sampling of a SPL configuration space can be achieved by Counting BDDs (CBDDs). Prior work relied on pseudo-random sampling;
(2) We explained how configuration spaces can be searched by using n random samples and returning the best-performance-in-n. We called this approach Non-Recursive Sampling (NRS), which has theoretically good performance;
(3) We demonstrated that information gleaned from sampled configurations yields noticeably better performance than NRS using Statistical Recursive Searching (SRS) at a minimal increase in algorithm complexity;
(4) We compared SRS to prior work and showed that SRS consistently found better-performing configurations using fewer samples; and
(5) We demonstrated how our approach scales to huge spaces.

We believe that our work advances and simplifies the state-of-the-art in finding near-optimal configurations in large SPL configuration spaces.

ACKNOWLEDGMENTS

Work by Oh, Batory, and Myers is supported by NSF grants CCF-1212683 and ACI-1550493. Siegmund's work is supported by the DFG under the contract SI 2171/2.

REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R Narasayya. 2000. Automated selection of materialized views and indexes in SQL Databases. In *26th International Conference on Very Large Databases (VLDB)*. 496–505.
[2] Sheldon B. Akers. 1978. Binary decision diagrams. *IEEE Trans. Comput.* 27, 6 (1978), 509–516.
[3] Sven Apel, Don Batory, Christian Kaestner, and Gunter Saake. 2013. *Feature-oriented software product lines: concepts and implementation.* Springer.

[4] Don Batory. 2005. Feature models, grammars, and propositional formulas. In *9th International Software Product Line Conference (SPLC)*. Springer, 7–20.

[5] Don Batory, Peter Höfner, and Jongwook Kim. 2011. Feature interactions, products, and composition. In *10th International Conference on Generative Programming and Component Engineering (GPCE)*, Vol. 47. ACM, 13–22.

[6] D. S. Batory and C. C. Gotlieb. 1982. A unifying model of physical databases. *ACM Transactions on Database Systems (TODS)* 4 (1982), 509–539.

[7] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2006. A first step towards a framework for the automated analysis of feature models. In *10th International Software Product Line Conference (SPLC)*. 39–47.

[8] Armin Biere, Marijn Heule, and Hans van Maaren. 2009. *Handbook of satisfiability*. Vol. 185. IOS press.

[9] Allan G Bluman. 1995. *Elementary statistics*. Brown Melbourne.

[10] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. 2013. A scalable approximate model counter. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 200–216.

[11] Girish Chandrashekar and Ferat Sahin. 2014. A survey on feature selection methods. *Computers & Electrical Engineering* 40, 1 (2014), 16–28.

[12] Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, 34–43.

[13] Carla P Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. 2008. Satisfiability solvers. *Foundations of Artificial Intelligence* 3 (2008), 89–134.

[14] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *28th International Conference on Automated Software Engineering (ASE)*. IEEE, 301–311.

[15] Jianmei Guo, Jules White, Guangxin Wang, Jian Li, and Yinglin Wang. 2011. A genetic algorithm for optimized feature selection with resource constraints in software product lines. *Journal of Systems and Software* 84, 12 (2011), 2208–2221.

[16] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. 2015. Combining multi-objective search and constraint solving for configuring large software product lines. In *37th International Conference on Software Engineering (ICSE)*, Vol. 1. IEEE, 517–528.

[17] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable analysis of variable software. In *9th Joint Meeting on Foundations of Software Engineering (FSE)*. ACM, 81–91.

[18] Bryan Marker, Don Batory, and Robert Van De Geijn. 2014. Understanding performance stairs: Elucidating heuristics. In *29th International Conference on Automated Software Engineering (ASE)*. ACM, 301–312.

[19] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *38th International Conference on Software Engineering (ICSE)*. ACM, 643–654.

[20] Marcilio Mendonca, Moises Branco, and Donald Cowan. 2009. SPLOT: Software product lines online tools. In *24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. ACM, 761–762.

[21] Marcilio Mendonca, Andrzej Wasowski, Krzysztof Czarnecki, and Donald Cowan. 2008. Efficient compilation techniques for large scale feature models. In *7th International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 13–22.

[22] Jeho Oh, Don Batory, and Margaret Myers. 2016. *Finding Product Line Configurations with High Performance by Random Sampling*. Technical Report TR-16-22. University of Texas at Austin, Department of Computer Science.

[23] Rafael Olaechea, Derek Rayside, Jianmei Guo, and Krzysztof Czarnecki. 2014. Comparison of exact and approximate multi-objective optimization for software product lines. In *18th International Software Product Line Conference (SPLC)*, Vol. 1. ACM, 92–101.

[24] Richard Pohl, Kim Lauenroth, and Klaus Pohl. 2011. A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models. In *26th International Conference on Automated Software Engineering (ASE)*. IEEE, 313–322.

[25] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Cost-efficient sampling for performance prediction of configurable systems. In *30th International Conference on Automated Software Engineering (ASE)*. IEEE, 342–352.

[26] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Dataset for Sarkar2015. https://github.com/atrisarkar/ces. (2015).

[27] Abdel Salam Sayyad, Joseph Ingram, Tim Menzies, and Hany Ammar. 2013. Scalable product line configuration: A straw to break the camel's back. In *28th International Conference on Automated Software Engineering (ASE)*. IEEE, 465–474.

[28] Abdel Salam Sayyad, Tim Menzies, and Hany Ammar. 2013. On the value of user preferences in search-based software engineering: a case study in software product lines. In *35th International Conference on Software Engineering (ICSE)*. IEEE, 492–501.

[29] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence models for highly configurable systems. In *10th Joint Meeting on Foundations of Software Engineering (FSE)*. ACM, 284–294.

[30] Norbert Siegmund, Sergiy S Kolesnikov, K Christian, Sven Apel, and Gunter Saake. 2012. Dataset for Siegmund2012. http://fosd.de/SPLConqueror. (2012).

[31] Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. 2012. Predicting performance via automated feature-interaction detection. In *34th International Conference on Software Engineering (ICSE)*. IEEE, 167–177.

[32] Marc Thurley. 2006. sharpSAT–counting models with advanced component caching and implicit BCP. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 424–429.

[33] Takahisa Toda and Takehide Soh. 2016. Implementing efficient all solutions sat solvers. *Journal of Experimental Algorithmics (JEA)* 21, 1 (2016), 1–12.

[34] Bernard L Welch. 1947. The generalization ofstudent's' problem when several different population variances are involved. *Biometrika* 34, 1/2 (1947), 28–35.

[35] Jules White, Brian Doughtery, and Douglas C Schmidt. 2009. Filtered Cartesian flattening: An approximation technique for optimally selecting features while adhering to resource constraints. *Journal of Systems and Software* 82(8) (2009), 1268–1284.

[36] Wikipedia. 2016. Monotonic Function. https://en.wikipedia.org/wiki/Monotonic_function. (2016).

[37] Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. 2015. Performance prediction of configurable software systems by fourier learning. In *30th International Conference on Automated Software Engineering (ASE)*. IEEE, 365–373.