Finding Product Line Configurations with High Performance by Random Sampling

Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund

ABSTRACT
Software Product Lines (SPLs) have many features for customization. This raises the challenge to find good 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; we randomly sample and recursively search a configuration space to find good configurations without constructing a prediction model. Our algorithms are simpler and have noticeably higher accuracy and efficiency.

ACM Reference format:
DOI:

1 INTRODUCTION
Software Product Lines (SPLs) are highly configurable systems. This raises the challenge to find a configuration that has good or 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 [36]. 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, 32]. (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 closely approximates that of optimal. Our approach does not create a performance prediction model, which then requires an optimizer (e.g. using a genetic algorithm [28]) 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 good 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 performance by randomly sampling 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 good performing configurations, but also good-performing configurations that observe configuration constraints (e.g. exclude feature F and include feature G in an answer).

Prediction models estimate the performance of any valid configuration [16, 26, 30, 32, 39]. 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 [4], properties of features and their interactions, and user-requirement constraints, an optimizer can derive valid configurations that satisfy one or more multiple performance objectives using a general search strategy [17, 18, 28, 29, 36]. Let \( C \) 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 variant \( P_{c} \) with configuration \( c \in C \), where \( c \) is a set of features and \( \$F_{i} \) is the performance contribution of feature \( F_{i} \):

\[
\$P_{c} = \sum_{i \in c} \$F_{i}
\]

Linear models are inaccurate as they do not consider feature interactions. Let \( \$Fi_{ij} \) 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 \( F_{i} \not\subset c \lor F_{j} \not\subset c \). 2\(^{nd}\)-order

\[
\$P_{c} = \sum_{\forall i \in c} \sum_{\forall j \in c} \$Fi_{ij}
\]
models take into account 2-way interactions:
\[ S_P = \left( \sum_{i \in c} F_i \right) + \left( \sum_{i \in c} \sum_{j \in i} F_{ij} \right) \] (2)
and more generally, \( n \)-way interactions, that add more nested-
summation terms to Eq. (2) [32].

When compared to manually-developed performance models
[1, 6, 12], an important difference becomes apparent. A manually-
developed model:
- Identifies operations \{\( O_i \ldots \)\} 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:
\[ P = \sum_i v_i \cdot S O_i \] (3)

Features complicate the cost function of each operation, where
configuration \( c \in C \) becomes an explicit parameter:
\[ S_P = \sum_i v_i \cdot S O_i (c) \] (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 [16–
18, 26, 28, 29, 39] 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 configura-
tions [16, 17, 26, 29, 39]. The drawback of this approach is that it
creates too many invalid configurations [16]. Another approach
uses SAT solvers to generate valid configurations [18, 28], 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. [18] 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 configura-
tion 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 configuration constraints, to find a good
or optimal performing configuration.

Our approach is different. First, we do not use performance
models or optimizers. We find good configurations by randomly
probing the configuration space, measuring the performance of
these samples under the required workload.

Second, we use true random sampling. We encode feature models
as BDDs, 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 good-performing 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(is/fiability)
solvers [14] and Binary Decision Diagrams (BDDs) [2]. SAT relies on a
Conjunctive Normal Form (CNF) represen-
tation of a formula to find a so-
lution efficiently. In contrast, a BDD
is a data structure that encodes a
Dis-
junctive Normal Form (DNF) formula – essentially a disjunction of formula
solutions. BDD tools convert non-
DNF formulas into BDDs.

Consider the formula:
\[ \psi = (x_1 \leftrightarrow y_1) \land (x_2 \leftrightarrow y_2) \]

Figure 2 shows a BDD\footnote{This is an ordered BDD, where Boolean variables are encountered from root-to-
terminals in the same order.} that encodes \( \psi \) [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

![Figure 2: CBDD.](image-url)
assignments are a solution to \( \psi \). Path \((1, 0, -, -)\) means that all configurations with \( x_1=1, y_1=0 \) (the remaining variables are don’t cares) are not solutions to \( \psi \) as the path ends at the 0 box.\(^3\)

BDDs make it easy and fast to count the number of solutions to a formula. 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 \((1, 1, -, -)\) has two solutions. The root or path \((-,-, -, -)\) has four solutions (the sum of the edges leaving the root). The 3rd solution is found by a guided descent from the root, yielding path \((1, 1, 0, 0)\).

Here is why CBDDs are important: It solves an open problem of how to randomly select configurations in a configuration space. CBDDs provide a solution: 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 CBDDs 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 [3] (for lack of space). Further, it is also possible to constrain a formula by setting a truth assignment to one or more variables and counting the number of solutions of the reduced formula. The utility of this algorithm is that a CBDD can be created and used for counting solutions to constrained configuration spaces [3].

### 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** \( \Omega \) – the configuration with the optimal performance – but if we can come provably close to \( \Omega \), that will do nicely.

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

\[
\text{PCS} = \{ (c, \$ (c)) \mid c \in \mathbb{C} \}
\]

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

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). \( \Omega \) anchors the far-right point on \( X \)-axis of PCS graphs.

Interestingly, Marker et al. [20] discovered that PCS graphs are staircased, 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,\

\(\text{Figure 3: PCS Graphs.}\)

**3.3 Random Selection in PCS Graphs**

Let \( \mathbb{N} \) be the interval of integers \( \{1, |\mathbb{C}|\} \), one per configuration in \( \mathbb{C} \). To simplify mathematics, we replace \( \mathbb{N} \) with the real unit interval \( I = [0, 1] \) by:

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

Every PCS graph is monotonically decreasing. If we randomly select \( n \) points in \( I \), the point/configuration with the best performance, henceforth \( c_{\text{best}} \), will be the closest to 1. The cumulative probability distribution function for \( c_{\text{best}} \) is:

\[
\frac{p_n (X \leq x)}{x^n} = \int_0^x n \cdot x^{n-1} \cdot dx = x^n
\]

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

\[
E_n = \int_0^1 (1 - x) \cdot n \cdot x^{n-1} \cdot dx = \frac{1}{n + 1}
\]

That is, \( n \) randomly selected points partition \( I \) on average into \( n+1 \) uniform intervals of length \( \frac{1}{n+1} \). Eq. (7) tells us a simple way to

\(\frac{x^{n-1} \cdot dx}{x^n} \) 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. (6) is an instance of the Beta function [9].
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 $\sigma_n$, its standard deviation:

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

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

Figure 5 plots $E_n$ and $\sigma_n$ as percentages in an infinite-size configuration space. $E_n$ and $\sigma_n$ values are virtually identical as their graphs lie on top of each other.

Here is what Figure 5 means: If we randomly select $n=100$ points, $c_{best}$ will be 1% away from 1 on the $x$-axis with a standard deviation of 1%. If we select $n=50$ points, the best point 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 we do not need many random selections to find a good performing point.

### 3.4 Axes of Projections and Main Conjecture

Consider the PCS graph of $\Omega \times \mathbb{S}$ plane of Figure 6. In the last section, we analyzed the performance of selecting $n$ points along the $x$-axis and choosing the point closest to 1.

We are more interested in the PCS graph of plane $\mathbb{C} \times \mathbb{S}$. Each point in $\mathbb{C}$ 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 Eq. (6)–(9) about error distances from $c_{best}$ to 1 in $\mathbb{C}$ are transferred to error distances from $c_{best}$ to $\Omega$ in $\mathbb{C}$.

Here is our main conjecture: there is a correspondence between being $q\%$ from $\Omega$ along $x$-axis and $q\%$ from $\$ (\Omega)$ along $y$-axis in a PCS graph for small $q$. Suppose a PCS graph is defined by $\$ (x)$:

$$\$ (x) = 1 - x^k$$

Figure 7a plots graphs for $k \in \{1/5, 1/3, 1, 3, 5\}$. We say $k$ is the curvature of a PCS graph.

Let’s focus on the interval $[0.6, 1]$, which contains all configurations whose $x$-axis value is within 4% of $\Omega$. We call this the critical zone. Figure 7b shows a close-up of the critical zone of Figure 7a.

The $y$-axis plots the %-distance from the best-performance at $y=0$, namely $\$ (\Omega)$.

Although the PCS graphs in Figure 7a are non-linear, the curvature $k$ reduces to the graph’s slope at $x=1$ in the critical zone. This slope is the first derivative, $\frac{\partial}{\partial x} (1 - x^k) = -k \cdot x^{k-1}$, and in the limit, $\lim_{x\to 1} -k \cdot x^{k-1} = -k$. That is, the slope of a PCS graph in the critical region is negative $k$. Observe:

- When $k=1$, the PCS graph $\$ (x)$ is a line. If we are $q\%$ away from $\Omega$ on $x$-axis we are also precisely $q\%$ away from $\$ (\Omega)$ on $y$-axis.
- When $k<1$ the graph is convex. If we are $q\%$ away from $\Omega$, we know that performance is $k\cdot q\%$ away from $\$ (\Omega)$. A convex PCS graph means that $\Omega$ lies on a flat shelf where any configuration on that shelf is good.
- When $k>1$ the graph is concave. If we are $q\%$ away from $\Omega$, we are $k\cdot q\%$ away from $\$ (\Omega)$. A concave PCS means that $\Omega$ 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

Six SPLs were analyzed by Siegmund et al. [31]. Figure 8 shows their PCS graphs. Each is described briefly:

- **AJStats** is a tool for collecting code statistics from AspectJ programs. It has 20 features and 131072 configurations, where code analysis times on @@@@, a customizable CORBA implementation, 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.
- **BerkeleyDBC** is an embedded database system written in C. It has 18 features and 2560 configurations where benchmark response times were measured.
- **BerkeleyJ** is a Java re-implementation of BerkeleyDBC. It has 26 features and 400 configurations, where benchmark response times were measured.
- **LLVM** is a compiler infrastructure for languages written in C++. It has 11 features and 1024 configurations, where test suite compilation times were measured.
- **X264** is a video codec library for H.264/MPEG-4 AVC format written in C. It has 16 features and 1152 configurations, Sintel trailer encoding times were measured.

Figure 9 shows their critical zones. Most SPLs in our sample have $k<1$; this means that as $c_{best}$ approaches $\Omega$ 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 \( k > 1 \) pose more of a challenge. Their configurations do not lie on a flat shelf; performance noticeably improves as one gets closer to \( \Omega \). If we know the curvature \( k \) of a PCS graph, we can estimate how far we are from \( \Omega \). Examples: LLVM has a curvature of \( k = 2 \). If we believe our best sample is \( q \% \) from \( \Omega \), we can infer that we are \( 2 \cdot q \% \) from \( \Omega \). \textbf{AjStats} has a curvature of \( k = 6 \). If we believe our best sample is \( q \% \) from \( \Omega \), we can infer that we are \( 6 \cdot q \% \) 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 \( k \). More on this in Section 4.2.

4 RECURSIVE SEARCHING

The best configuration \( c_{\text{best}} \) out of 10 random samples will have an average error/distance of \( 9\% = \frac{1}{11} \) along the X-axis from \( \Omega \). 100 random samples (or \( 10 \times \) the previous number) are needed to find \( c_{\text{better}} \) that reduces the error to \( 1\% = \frac{1}{100} \). Note that approximately 90% of the additional 90 samples will not perform better than \( c_{\text{best}} \). This is wasteful. We call this \textbf{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 \textbf{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 \( \Omega \) improves the result of sampling.

Consider the PCS graph of LLVM in Figure 8. This graph looks almost linear with no stairs. However, stairs become visible as configurations are analyzed based on features they have in common.

Figure 10 shows this analysis. In each graph, configurations are partitioned by whether they include a particular feature or not. This partitioning is done recursively, where the most influential feature is used to partition first, and the partition that performs better overall is selected for the next partition.
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.

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.

Algorithm 1: SRS algorithm

1 Procedure SRS(n, FM, dSet):
   2   Input: n (number of samples per recursion)
   3       FM (feature model propositional formula)
   4       dSet (set of feature decisions (initially null))
   5   Output: best (searched best config. (set of features))
   6   samples ← sample n configs. from FM ∧ dSet;
   7   sort samples so that samples[0] has best performance;
   8   commons ← common feature decisions in samples[0] and samples[1];
   9   for each decision in commons do
   10       if (Δ(decision) < 0) ∧ TTest(decision) then
   11           add decision to dSet;
   12       if dSet unchanged from previous recursion then
   13           return configs[0];
   14   else
   15       return SRS(n, FM, dSet);

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:

• $\Delta(d)$ measures the average performance over the n samples that have feature d. $\Delta(-d)$ measures the average performance of the n samples that do not have d.
• $\delta(d)=\Delta(d)-\Delta(-d)$ 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-Test(d) is the result of Welch’s t-test [35] on whether $\delta(d)$ is better than $\delta(-d)$ with 95% confidence.

Welch’s t-test evaluates the hypothesis that the mean of one sample group is higher than the other [35]. 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.

4.2 Estimating PCS Graph Curvature

Let $\Delta_k(r)$ denote the size of a stair in terms of the number of configurations at the rth recursion, where $\Delta_k(1) = \# configs$ in the original space. $\Delta_k(r)$ decreases with increasing r. Using CBBDs, we can compute $\Delta_k(r)$ 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 $\delta_k(r)$ be the error (distance) from the best sampled configuration $c_{best}$ to Ω along the x-axis at r-th recursion:

$$\delta_k(r) = \frac{\# (c_{best}) \leq \text{# of configs} \leq \# (\Omega)}{\text{# of configs}}$$

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

$$\delta_k(r) = \frac{\# (c_{best}) - \# (\Omega)}{\# (\Omega)}$$

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

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

Figure 11 depicts how $\delta_k(r)$ and $\delta_k(r)$ can be estimated with these assumptions.

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

$$E(\delta_k(r)) = \frac{\Delta_k(r)}{\Delta_k(1)} \frac{1}{(n + 1)}$$

where $c_{best}$ is $\Delta_k(r)$ configurations from Ω along X-axis.

- We compute the slope or curvature k of a stair using the rightmost $\frac{1}{2}$ of its samples. We found $\frac{1}{2}$ works well, computed by a standard least squares method [9].

- $E(\delta_k(r))$, our estimate of $\delta_k(r)$, is a linear extrapolation of $\delta_k$, using slope k and to estimate $\delta_k$:

$$E(\delta_k(r)) = \frac{\# (c_{best}) - \# (\Omega)}{E(\delta_k)}$$

$SRS$ mostly avoids local minima by searching a PCS graph, which is monotonically decreasing [37]. This is elaborated at [3].
Finding SPL Configurations with High Performance by Random Sampling

At each recursion, we report \([E(\delta_x(r)), E(\delta_y(r)), k]\) 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.

Figure 11: Estimating \(\delta_x(r)\) and \(\delta_y(r)\).

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?

We used the data by Siegmund et al. [31] as ground-truth. We present results based on 3 systems, LLVM, BerkeleyDBC, and X264, as they have configuration spaces with complete data. The other 3 systems had incomplete data but showed identical evaluation results as the first three. The results are presented in [3].

5.1 RQ1: Does our sampling theory for NRS match observations?

We compared the theoretical predictions of NRS using \(E_N\), Eq. (7), with the average of measured values for \(\delta_x(1)\), Eq. (11). We performed 100 experiments for each value of \(N\). For each system, the experiments started with \(N = 10\) to 100 incremented by 10, plotted for comparison with \(E_N\); see Figure 12. These graphs confirm a close agreement between NRS theory and observations: their differences are imperceptible.

For RQ1, our sampling theory matches 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:

- \(\delta_x\) 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 \(\delta_y\) values here. A decrease in \(\delta_x\) is never matched by an increase in \(\delta_y\) in a PCS graph. \(\delta_y\) values are important but only when comparing SRS with existing approaches, which we do in RQ4.

Looking closer, Figure 14 plots the following measures for BerkeleyDBC, which had the smallest \(\delta_x\) and \(E_N\) gap: \(N\) is the total # of samples taken at SRS termination and \(E_N^{-1}(\delta_x)\) is the number of samples required for NRS to achieve the same accuracy as \(\delta_x\). The greater the number of samples \(n\) per recursion, the more accurate SRS is, requiring significantly more samples from NRS to match its performance. We observed this for other systems as well.

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,
For RQ2, the $d$, $\rho$, and $r$ saturate at $n=15$; indicating that recursion works as desired. As $n$ increases, accuracy increases at the cost of a linearly increasing $N$.

5.4.2 Comparison with Sarkar2015 and 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 for any legal configuration.

We defined $\delta_y$ for this prediction model as follows:

$$\delta_y = \frac{\$\left(\text{predicted best}\right) - \$\left(\Omega\right)}{\$\left(\Omega\right)}$$

where $\$\left(\text{predicted best}\right)$ is the actual, not predicted, performance of the sampled configuration within the leaf, and $\$\left(\Omega\right)$ denotes the total required samples.

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

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. $\delta_y$ was derived as the relative error between $\$\left(\Omega\right)$ and this value. Using their tool, 20 prediction models were created to derive $\delta_y$ and averaged, for different sample sizes.

Figure 16 plots $\delta_y$ of SRS over $N$, as well as the values derived from Sarkar2015, plotted as squares. The graphs show that SRS obtained the same $\delta_y$ value with fewer samples ($N$) and found better $\delta_y$ 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 $\delta_y=20\%$ (see point 5 in Figure 16). SRS needs less than 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 $\delta_y$. In contrast, SRS clearly shows a decrease of $\delta_y$ 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.
used different strategies to select the configurations. As different strategies used different numbers of samples, we measured $\delta_y$ for different strategies. Figure 16 plots the prediction model results of Siegmund2012 as triangles.

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

Like Sarkar2015, more samples did not guarantee a better $\delta_y$ value, nor was there consistency across systems, as greatly different $\delta_y$ and $N$ values were observed. SRS clearly shows a decrease of $\delta_y$ 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. [39] 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

<table>
<thead>
<tr>
<th>Combined Systems</th>
<th># of Features</th>
<th># of Confs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache × LLVM × BerkeleyDBC</td>
<td>38</td>
<td>503,316,480</td>
</tr>
<tr>
<td>Apache × X264 × BerkeleyDBC</td>
<td>51</td>
<td>566,231,040</td>
</tr>
<tr>
<td>LLVM × X264 × BerkeleyDBC</td>
<td>53</td>
<td>3,019,898,880</td>
</tr>
<tr>
<td>Apache × X264 × LLVM × BerkeleyDBC</td>
<td>62</td>
<td>579,820,584,960</td>
</tr>
</tbody>
</table>

Demonstrating the scalability of NRS is simple: the equations that define NRS performance, namely Eq. (7) and Eq. (9), do not depend on the size of the configuration space; NRS performance is defined over the unit interval that represents a space with an infinite number of configurations. Compare Figure 5 with Figure 12: the $E_N$ graphs are isomorphic. Figure 17 plots $E_N$ predicted vs. $E_N$ observed for Apache×X264×BerkeleyDBC for ranges of $N = [40, 120]$ that we could compute Eq. (11). The wavy line of observations is no different than that in Figure 12, except minute differences averaging 100 experiments are magnified. Computing observed $E_N$ requires us to compute configuration performance for most of this huge space, so for smaller fractions of this space, computation time was reasonable. Again, theoretical predictions match observations.

Figure 17: NRS Theory vs Observations.

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

5.6 Threats to Validity

Internal Validity. We used ground-truth data from [31], which are measurements of real systems. While there may be errors in measurements, this dataset was utilized by other researchers [16, 26, 30, 32, 39]. 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, $\delta_2$ 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. While 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 $\Phi(c)$ 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. [26] 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 using accuracy. Siegmund et al. [30] extended their previous work [32] with numeric features and an iterative process to build performance-influence models, which we do not cover yet. Zhang et al. [39] 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 optimal configuration.

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. [36] proposed an approach based on linear programming, which transforms the given feature model with budget constraints into a knapsack problem. Guo et al. [17] applied a genetic algorithm to search for the optimal configuration. From randomly selected configurations, they crossover good performing configurations for mutation and modify invalid generated configurations. Sayyad et al. [28, 29] 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. [18] 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 [24], and require more than 100 initial samples.

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 [33] 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, 25] compared CSP, SAT, and BDD solvers on counting configurations, where BDD was much faster than the others, given enough memory. Mendonca et al. [22] provided a reasoning and configuration engine SPLOT, which uses BDD to count the number of valid configurations. Mendonca et al. [23] 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. [19] 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. [21] 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 [16, 17, 26, 29, 39].

In contrast, we randomly sample from a set of valid configurations. AllSAT solvers [13, 15, 38] are SAT solvers that also can count the number of solutions. The key to using AllSAT is to determine how, given the number of a configuration, to translate it efficiently into that configuration. BDDs provide this capability directly [8, 34].

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-d. 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 good performing configurations in large SPL configuration spaces.

REFERENCES