Finding Product Line Configurations with High Performance by Random Sampling

Jeho Oh, Margaret Myers, Don Batory

University of Texas at Austin

{jeho, myers, batory}@cs.utexas.edu

Abstract

Highly configurable systems, such as software product lines, have many options for customization. This raises the challenge to find good performing configurations for an anticipated workload. As 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 configurations; we randomly sample and recursively search a configuration space to find good configurations without constructing a prediction model. Consequently, our algorithms are simpler and have noticeably higher accuracy and efficiency.

1. Introduction

Highly configurable systems, like software product lines (SPLs), provide many options for customization. This raises the challenge to find a configuration that has good or optimal performance. A configuration space is often astronomical in size (exponential in terms of features – increments in program functionality), and searching it efficiently is hard [33]. 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 [4, 30]. (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 must be taken for performance models to have acceptable accuracy?

This paper focuses on a fundamental problem in configurable systems to find acceptable configurations whose performance closely approximates that of optimal. Our approach does not create a performance prediction model, which requires an optimizer (eg using a genetic algorithm) to find good configurations. Instead, we randomly sample the configuration space to directly find acceptable configurations, eliminating the overhead of performance model learning and optimizers.

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 can identify features that are statistically certain to improve or degrade program performance [10]. We use these features to recursively shrink 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 contributions of our paper are:

• A novel way to truly random sample valid configurations in an SPL;
• Theoretical bounds on performance by randomly sampling configurations;
• A way to progressively shrink a configuration space using performance stairs and statistical reasoning;
• Analyses of real systems that shows our approach to outperform 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 and Review of Prior Work

To predict performance of a highly configurable system, a mathematical performance model is created. Historically, such models are developed manually using domain-specific knowledge [1, 11]. 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 (eg exclude feature F and include feature G in an answer).
**Prediction models** estimate the performance of any valid configuration [14, 25, 28, 30, 34]. 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-requirement constraints, an optimizer can derive valid configurations that satisfy one or more multiple performance objectives using a general search strategy [15, 16, 26, 27, 33].

Let \( C \) be the set of all legal SPL configurations. 1\textsuperscript{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 \( F_i \):

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

Linear models are inaccurate as they do not take into account 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 \( F_i \notin c \) or \( F_j \notin c \). \(^2\) 2\textsuperscript{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, that add more nested-summation terms to Eqn (2) [30].

When compared to manually-developed performance models [1, 5, 11], an important difference becomes apparent. A manually-developed model:

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

\[
\$P = \sum_{i} \nu_i \cdot \$O_i \tag{3}
\]

Features complicate the cost function of each operation, where configuration \( c \in C \) becomes an explicit parameter:

\[
\$P_c = \sum_{i} \nu_i \cdot \$O_i(c) \tag{4}
\]

The key observation is that manual performance models **include** workload variances in their predictions, whereas current SPL performance models use a **fixed** workload. There is every reason to believe that workload variations play a significant role in SPL product performance and should not be omitted.

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

---

**Random sampling.** Optimizers and prediction models [14–16, 25–27, 34] rely on ‘random sampling’, but the samples used are not provably random. True random sampling would, in effect, enumerate all \( n \) legal configurations in an array, randomly choose a number \( k \in \{1..n\} \), and use the \( k^{\text{th}} \) 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, 27, 34]. The drawback of this approach is that it creates too many invalid configurations [16]. Another approach uses SAT solvers to generate valid configurations [16, 26], 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 [6, 7]. Although Henard et al. [16] tried to mitigate these issues by randomly permuting the parameter settings in SAT solvers, true random sampling was not demonstrated.

---

**Figure 1. Different Ways to Find Good Configurations.**

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 configuration constraints, to find a good or optimal performing configuration.

Our work is different. First, we do not use performance models or optimizers. We find good configurations by randomly probing the configuration space, using performance measurements of 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 \( k^{\text{th}} \) configuration. This allows us to create accurate mathematical models based on true random selection. We also use samples to recognize statistically significant features (or their absence) that contribute to good performance, thereby allowing us to recursively focus 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 configura-
tions for a given workload. We continue sampling until we reach a configuration that exhibits a satisfactory ‘accuracy’. We later demonstrate that our technique is more efficient than prior work in terms of sampling, 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. Searching by Random Sampling

3.1 Counting Binary Decision Diagrams

Two tools are commonly used to analyze propositional formulas: SAT(isfiability) solvers [12] and Binary Decision Diagrams (BDDs) [2]. 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 Disjunctive 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\(^2\) that encodes \(\psi\) [2]. For now, ignore the integer labels on edges. The name of each node is a variable; 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 the variable assignments are a solution to \(\psi\). Path 

\((1, 0, -, -)\)

means all configurations with \(x_1=1, y_1=0\) and the remaining are don’t cares, are not solutions to \(\psi\) as this path terminates at the 0 box.\(^3\)

BDDs make it easy and fast to count the number of solutions of 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 

\((1, 1, 0, -)\)

has two solutions. The root or path 

\((-,-,-,-)\)

has four solutions (the sum of the edges leaving the root). The 3\(^{rd}\) solution is found by a guided descent from the root, yielding path 

\((1, 1, 0, 0)\).

Appendix A lists an algorithm to create and traverse a CBDD that maps a number to a configuration.

Here is why CBDDs are important: an open problem is 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 BDDs is that when formulas are large, BDD creation time may exceed user patience or storage requirements of available memory [2].

appTraversal goes further. It lists an algorithm 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 these algorithms is that a CBDD can be created once and reused for counting solutions in a configuration space with differing constraints.

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 \(C\) 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 \((\text{config}, \text{performance})\) pairs:

$$\text{PCS} = \{ (c, \$ (c)) | c \in \mathbb{C} \}$$

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 [19]. 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 the \(x\)-axis of PCS graphs.

Interestingly, real PCS graphs are stairied, as in Figure 3b [19]. Stairs arise from discrete feature decisions, which includes or excludes certain features; 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). In short, the configurations of a stair share the same major design decisions [19].

\(^2\)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 \(2^{\text{number of variables}}\) nodes.
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.

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 \( \mathbb{I} = [0, 1] \) by:

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

Every PCS graph is monotonically decreasing. So if we randomly select \( n \) points in \( \mathbb{I} \), the point with the best performance will be closest to 1. Its cumulative probability distribution function is:

\[
p_n(x \leq x) = \int_0^x n \cdot x^x \cdot dx = x^n \tag{6}
\]

The average error \( E_n \), or the mean distance the closest selected point is to 1, is:

\[
E_n = \int_0^1 (1 - x) \cdot n \cdot x^n \cdot dx = \frac{1}{n+1} \tag{7}
\]

That is, \( n \) randomly selected points on average partition \( \mathbb{I} \) into \( n+1 \) uniform intervals of length \( \frac{1}{n+1} \). Eqn (7) 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 is on average a distance \( \frac{1}{n+1} \) from the best performance at \( x=1 \).

Other useful statistics of \( p_n \) are \( E_n \), the second-moment of \( E_n \), and \( \sigma_n \), its standard deviation:

\[
E_n^2 = \int_0^1 (1 - x)^2 \cdot n \cdot x^n \cdot dx = \frac{2}{(n+1) \cdot (n+2)} \tag{8}
\]

\[
\sigma_n = \sqrt{E_n^2 - E_n^2} = \sqrt{\frac{2}{(n+1) \cdot (n+2)} - \left(\frac{1}{n+1}\right)^2} \tag{9}
\]

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 lines overlay each other. If we randomly select \( n=100 \) points, the best point will be 1% away from 1 on the \( \mathbb{I} \)-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 quickly. 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 the \( \mathbb{I} \times \mathbb{S} \) plane of Figure 6. 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 the plane \( \mathbb{C} \times \mathbb{S} \). Each point in \( \mathbb{I} \) has an equal probability of being selected. As the CBDD mapping is 1-to-1, each point along the \( \mathbb{C} \) axis also has an equally probability of being selected.

So the theoretical results Eqns (6)–(9) about error distances from the best-of-\( n \) selections to 1 in \( \mathbb{I} \) are transferred to error distances from best-in-\( n \)-configuration selections to \( \Omega \) in \( \mathbb{C} \).

Here is our main conjecture: there is a correspondence between being \( q\% \) from \( \Omega \) along the \( \mathbb{I} \)-axis and \( q\% \) from \( \mathbb{S}(\Omega) \) along the \( \mathbb{Y} \)-axis in a PCS graph for small \( q \). Suppose a PCS graph is defined by \( \mathbb{S}_k(x) \):

\[
\mathbb{S}_k(x) = 1 - x^k \tag{10}
\]

Figure 7 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 \( \mathbb{X} \)-axis value is within 4% of \( \Omega \). We call this the critical zone. Figure 7 shows a close-up of the critical zone of Figure 7a. The \( \mathbb{Y} \)-axis plots the \( q\% \)-distance from the best-performance at \( y=0 \), namely \( \mathbb{S}(\Omega) \).
Although the PCS graphs in Figure 7a are non-linear, the curvature $\kappa$ 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, $\lim_{x \to 1} \frac{\partial}{\partial x} (1 - x^k) = -k$. That is, the slope of the PCS graph in the critical region is negative $k$. Observe:

- When $k=1$, the PCS graph $\frac{\partial}{\partial x} (1 - x^k)$ is a line. If we are $q\%$ away from $\Omega$ on the $x$-axis we are also precisely $q\%$ away from $\frac{\partial}{\partial x} (1 - x^k)$ on the $y$-axis.
- When $k<1$ the graph is convex. If we are $q\%$ from $\Omega$, we know that performance is $k \cdot q\%$ away from $\frac{\partial}{\partial x} (1 - x^k)$. 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 $\frac{\partial}{\partial x} (1 - x^k)$. 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 Highly Configurable Systems

Six highly configurable systems were analyzed by Siegmund et al. [29]. 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 the code analysis time on ORBACUS, a customizable CORBA implementation, was 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 encoder library for H.264/MPEG-4 AVC format written in C. It has 16 features and 1152 configurations; Sintel trailer encoding time was measured.

Figure 9 shows their PCS graphs in the critical zone. Most SPLs in our sample have $k<1$; this means that as our best-n configurations approaches $\Omega$ on the $x$-axis, we know its performance is close to $\frac{\partial}{\partial x} (1 - x^k)$. The reason is 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 $\frac{\partial}{\partial x} (1 - x^k)$. Examples: **LLVM** has a curvature of $k=2$. If we believe our best sample is $q\%$ from $\Omega$, we can infer that we are $2 \cdot q\%$ away from $\frac{\partial}{\partial x} (1 - x^k)$. **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 \( S(\Omega) \).

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 \( 9\% = \frac{1}{10} \) 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 \( 1\% = \frac{1}{90} \). Note that approximately 90\% of the additional 90 samples will not perform better than \( c_{\text{best}} \). This is wasteful.

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 reduce this smaller space 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 \( \Omega \) improves the result of sampling. Thus, devising an algorithm that finds a good stair on which to recurse is the crucial next step. We use the 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 is to exploit how stairs are recursively formed, by observing each feature’s 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 that are inconsistent with \( \Omega \).

Algorithm 1: SRS algorithm

\begin{center}
\begin{tabular}{|c|}
\hline
\textbf{Procedure SRS}(n, FM, dSet) \protect\footnote{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 statically are certain to improve performance [10]. SRS then shrinks the search space to configurations that comply with noteworthy features decisions, and the SRS algorithm recurses. See Algorithm 1.} \\
\textbf{Input} : n (number of samples per recursion) \\
& FM (feature model propositional formula) \\
& dSet (set of feature decisions (initially null)) \\
\textbf{Output}: best (searched best config. (set of features)) \\
\textbf{samples} ← sample n configs. from FM ∧ dSet; \\
\textbf{sort samples} so that \textbf{samples}[0] has best performance; \\
\textbf{commons} ← common feature decisions in \textbf{samples}[0] and \textbf{samples}[1]; \\
\textbf{for} each decision in \textbf{commons} do \\
| if \( \Delta(\text{decision}) < 0 \) \& tTest(\text{decision}) then \\
| add decision to \textbf{dSet}; \\
| if dSet unchanged from previous recursion then \\
| return \textbf{config}[0]; \\
\textbf{else} \\
| return SRS(n, FM, dSet); \\
\hline
\end{tabular}
\end{center}
configuration space is smaller than $n$, all configurations in the space are measured.\footnote{SRS mostly avoids local minima by searching a PCS graph. More on this in Appendix C.}

4.2 Estimating PCS Graph Curvature

Let $\Delta_x(r)$ denote the size of a stair in terms of the number of configurations at the $r$\textsuperscript{th} recursion, where $\Delta_x(1)$ is 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 complete performance data about a configuration space is available. Let $\delta_x(r)$ be the error (distance) from the best sampled configuration $c_{best}$ to $\Omega$ along the $X$-axis at $r$-th recursion:

$$\delta_x(r) = \frac{$c_{best} - \# configs \leq \$\Omega}{\text{total \# of configs}} (11)$$

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

$$\delta_y(r) = \frac{$c_{best} - \$\Omega}{\$\Omega} (12)$$

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 $\Omega$.
- Pollution is negligible between $c_{best}$ and $\Omega$.

Figure 10 depicts how $\delta_x(r)$ and $\delta_y(r)$ can be estimated with these assumptions.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure10}
\caption{Estimated $\delta_x(r)$ and $\delta_y(r)$.}
\end{figure}

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

$$E(\delta_x(r)) = \frac{\Delta_x(r)}{\Delta_x(1) \cdot (n + 1)} (13)$$

where $c_{best}$ is $\frac{\Delta_x(1)}{n+1}$ configurations from $\Omega$ along $X$-axis.

- We compute the slope or curvature $k$ of a stair using the right-most $\frac{1}{2}$ of its samples. We found $\frac{1}{2}$ works well, computed by a standard least squares method [8].

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

$$E(\$\Omega) = \$c_{best} - k \cdot \frac{\Delta_x(r)}{n + 1} (14)$$

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

$$E(\delta_y(r)) = \frac{$c_{best} - E(\$\Omega)}{E(\$\Omega)} (15)$$

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.

5. Evaluation

Five research questions evaluate our work:

- **RQ1**: Does SRS sample theory match observations?
- **RQ2**: Is SRS more efficient than non-recursive searching?
- **RQ3**: Why does SRS work?
- **RQ4**: Is SRS better than existing approaches?
- **RQ5**: Does SRS scale to large configuration spaces?

We used the data by Siegmund et al. [29] 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 with identical evaluation results as the first three. See Appendix B.

5.1 RQ1: Does our sampling theory match observations?

Theoretical predictions of Section 3 were compared to measurements. Specifically, we examined the theoretical values of Eqn (7), with the average of measured values for $\delta_x(1)$, Eqn (11). In RQ1 we did not use recursion in our search – just straight random sampling.

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 Eqn. See Figure 11. These graphs confirm a close agreement between theory and observations: their differences are imperceptible.

For RQ1, we conclude the theory matches observations.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure11}
\caption{Theory vs. Observations on Random Sampling.}
\end{figure}
5.2 RQ2: Is SRS More Efficient Than Non-Recursive Searching (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.

![Figure 12. Comparison between SRS and NRS.](image)

Figure 12 plots averages of 100 experiments with different $n$ values. While both $\delta_x$ and $E_N$ decrease sharply with increasing $n$, $\delta_x$ is always better than $E_N$.

Looking closer, Figure 13 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.

![Figure 13. Efficiency Comparison between NRS and SRS.](image)

Figure 13 shows that for RQ2, SRS is more efficient than NRS when the number of samples $n$ per recursion 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,
- $\rho$ is the % of noteworthy features that belong to $\Omega$, and
- $r$ is depth of recursion.

![Figure 14. Results of SRS Recursion.](image)

Figure 14 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 RQ4: Is SRS better than existing approaches?

We determined the best configuration that can be returned by existing prediction models, and derived their $\delta_y$ value with regards to number of samples $N$ used to construct the prediction model. We compared our results with Siegmund2012 [30] and Guo2013 [2], as their prediction models and evaluation data were available [13] and [29].

5.4.1 Comparison with Guo2013

Guo2013’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 relative error between $\$\left(\Omega\right)$ and this value. In their data, multiple prediction models were created per system, with different sample sizes.

Figure 15 plots $\delta_y$ of SRS over $N$, as well as the values derived from Guo2013, 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 $N$. For example, in BerkeleyDBC, Guo2013 used 139 samples to obtain an accuracy of $\delta_y=40\%$ (see point 2 in Figure 15). SRS needs only 9 samples to produce this accuracy. And when SRS uses 139 samples, it obtains an accuracy of $\delta_y=0.3\%$.

Further, their results did not show a clear trend when increasing the number of samples, as larger $N$ did not necessarily lead to a smaller $\delta_y$. In contrast, SRS clearly showed a decrease of $\delta_y$ as $N$ increases.

![Figure 15. Comparison with Guo2013 and Siegmund2012.](image)

**5.4.2 Comparison with Siegmund2012**

Siegmund2012’s prediction model assigns performance values to key features and their interactions using configuration measurements and machine learning. The resulting model can predict the performance for any legal configuration.

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

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

where $\$\left(c_{\text{predicted}\ best}\right)$ is the actual, not predicted, performance of $c_{\text{predicted}\ best}$, the best performing configuration according to their prediction model. To build the prediction model, Siegmund2012 used different strategies to select the configurations. As different strategies used different numbers of samples, we measured $\delta_y$ for different strategies. Figure 15 plots the prediction model results of Siegmund2012 as triangles. As with Guo2013, 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 ▲ in Figure 15). 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 Guo2013, 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 showed 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: Does SRS scale to large configuration spaces?**

To create ground-truth data for a large configuration space, we combined multiple SPLs where their combinations of configurations create a larger configuration space (see Table 1) which was utilized in [34]. Configurations from each SPL are combined by taking the union of their features and summing their performance values.

![Combined systems for scalability evaluation](image)

**Table 1. Combined systems for scalability evaluation**

<table>
<thead>
<tr>
<th>Combined Systems</th>
<th># of Features</th>
<th># of Configs.</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 and SRS is simple: The equations that define NRS performance (Eqn (7) and Eqn (9)) do not depend on the size of the configuration space; their performance is defined over the unit interval $[0,1]$ that represents a space with an infinite number of configurations. Consequently performance graphs for composite SPLs should all have same shape and should match those of SPLs with small configuration spaces, as in Figure 12. Figure 16 shows this isomorphism in all four composites. The minimal differences that one can spot are due to the fact that each of these SPLs has finite and different configuration space sizes.

By comparing Figure 12 and Figure 16, you will see they are all isomorphic, despite significant variations in configuration space size. Further 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 RQ6, NRS performance $E_n$ is independent of the size of the configuration space. And consistent with results on small spaces, SRS outperforms NRS.

**5.6 Threats to Validity**

**Internal Validity.** We used ground-truth data from [29], which are measurements of real systems. While there may be errors on measurements, it mitigates the threat of searching over a system with features having trivial influence on performance. As this dataset was utilized in other researchers...
[14, 25, 28, 30, 34], we believe that the threat of comparing different approaches is sufficiently controlled.

To control the randomness of sampling, we performed 100 experiments and averaged the results. While there are outliers that threaten our results, \( \delta_x \) 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 domain and number 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. While 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 indicates that our conclusions should hold for most systems.

### 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. [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 models accuracy. Siegmund et al. [28] extended their previous work [30] with numeric features, which we do not cover yet. Zhang et al. [34] used Fourier transformation to create a prediction model that not only predicts performance, but also estimates its accuracy. 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. [33] 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 generated configurations. Sayyad et al. [26, 27] 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 [23], and require more than 100 initial samples.

#### 6.3 Sampling Configurable Systems

Efficient testing strategies for configurable systems rely on random sampling. Liebig et al. [18] 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. [20] also compared 10 different sampling algorithms for fault detection capability. Their work also randomly selected features, eliminating invalid configurations. But again, random sampling features does not guarantee true random sampling [14, 15, 25, 27, 34].

#### 6.4 Counting Configurations

Counting configurations is known as the model counting problem, which is regarded as a more complicated problem than checking the satisfiability [7]. SAT solvers were extended to exactly [31] or approximately [9] 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 constructed BDD can be reused [7].

Benavides et al. and Pohl et al. [6, 24] compared CSP, SAT, and BDD solvers on counting configurations, where BDD was much faster than the others, given enough memory. Kästner et al. [17] provided the tool FeatureIDE, where
the number of valid configurations is counted using a SAT solver. However, FeatureIDE was not able to count the number of configurations when the size is large. Mendonca et al. [21] provided a reasoning and configuration engine SPLOT, which uses BDD to count the number of valid configurations. Mendonca et al. [22] proposed heuristics to reduce the size of BDD through variable ordering inferred from feature model, which improves the scalability up to 2000 features. We believe we are the first to use counting BDDs to sample large configuration spaces.

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 eliminate the middle-men of performance models and optimizers by directly searching a configuration space by random sampling. 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 showed 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 provably good performance;
3. We showed that by using information gleaned from sampled configurations we could get 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 astronomical-sized configuration spaces.

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

Acknowledgments. We gratefully acknowledge support for this work by NSF grants CCF-1212683 and ACI-1550493. We also thank Norbert Siegmund for his helpful comments on an earlier draft.

References

Nodes. The number of possible configurations of a node, node.\#, is counted bottom-up from 1 box, computed as:
\[\text{node.}\# = \text{node.low.}\# \times 2^{\text{node.low.depth} - \text{node.depth} - 1} + \text{node.high.}\# \times 2^{\text{node.high.depth} - \text{node.depth} - 1}\]
where node.low and node.left are child nodes and node.depth is the order of the node, to deal with reduced nodes. node.\# of the root node shows the total number of configurations.

### Algorithm 2: Random sampling algorithm

<table>
<thead>
<tr>
<th>Procedure</th>
<th>SampleConfig(r, node, config);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>r (random number for sampling)</td>
</tr>
<tr>
<td></td>
<td>node (node of BDD under traversal)</td>
</tr>
<tr>
<td>Output</td>
<td>config (sampled set of features)</td>
</tr>
<tr>
<td>if node = 1 box then return ;</td>
<td></td>
</tr>
<tr>
<td>if ( r \leq \text{node.low.}# ) then</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SelectReduced(node, node.low, r, config);</td>
</tr>
<tr>
<td></td>
<td>SampleConfig(node.low, r, config);</td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>config.add(node.feature);</td>
</tr>
<tr>
<td></td>
<td>( r \leftarrow r - \text{node.low.}#; )</td>
</tr>
<tr>
<td></td>
<td>SelectReduced(node, node.high, r, config);</td>
</tr>
<tr>
<td></td>
<td>SampleConfig(node.high, r, config);</td>
</tr>
<tr>
<td>Procedure</td>
<td>SelectReduced(r, node, child, config);</td>
</tr>
<tr>
<td>Input</td>
<td>r (random number for sampling)</td>
</tr>
<tr>
<td></td>
<td>node (node of BDD under traversal)</td>
</tr>
<tr>
<td></td>
<td>child (successor of node to be traversed)</td>
</tr>
<tr>
<td>Output</td>
<td>config (sampled set of features)</td>
</tr>
<tr>
<td>for ( i ) from node.depth + 1 to child.depth – 1 do</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if ( r % 2 = 1 ) then</td>
</tr>
<tr>
<td></td>
<td>config.add(node.feature at i-th depth);</td>
</tr>
<tr>
<td></td>
<td>( r \leftarrow r/2; )</td>
</tr>
</tbody>
</table>

Sampling a configuration is shown on Algorithm 2. A random value \( r \) is selected from 1 to total number of configurations. Then, BDD is traversed top-down, where node.high is traversed if \( r > \text{node.low.}\# \), and node.low otherwise. Traversing node.high means that the sample includes this feature. In this case, \( r \) is subtracted by node.low.\# to propagate the selection. To deal with the reduced nodes throughout the traversal, \( r \% 2 \) is computed and its feature is included if the result is 1. \( r \) is divided by 2 so that it can propagate the selection.

Counting and sampling configurations with constraints can be done by setting the corresponding node.\# value to 0. If the feature should be included, node.low.\# of that feature is set to 0. node.high.\# is set to 0 if it should be excluded. If a feature corresponds to a reduced node, it is ignored for the counting as the value is predefined. Then, counting is performed again to apply Algorithm 2.

Counting configurations takes \( \mathcal{O}(n) \), where \( n \) is number of nodes in BDD. Sampling a configuration takes \( \mathcal{O}(d) \), where \( d \) is number of features. Counting and sampling under constraints require same complexity, but reusing BDD saves computation from BDD reconstruction. In SRS, constraints monotonically increase, where this reuse is beneficial.

### Appendices

#### A. CBDD Traversal Algorithm

Counting the number of solutions using a BDD is well known [2]. The number of possible configurations of a node, node.\#, is counted bottom-up from 1 box, computed as:
\[\text{node.}\# = \text{node.low.}\# \times 2^{\text{node.low.depth} - \text{node.depth} - 1} + \text{node.high.}\# \times 2^{\text{node.high.depth} - \text{node.depth} - 1}\]
where node.low and node.left are child nodes and node.depth is the order of the node, to deal with reduced nodes. node.\# of the root node shows the total number of configurations.


B. Results of Other Systems

Figure 17 shows $\delta_x$ values over $n$ for Apache, BerkeleyDBJ, and AjStats, which corresponds to the results of Section 5.2. These SPLs did not have performance measurements for all legal configurations. To use existing results, we enumerated all configurations for which measurements existed, and used this as the system’s configuration space. Stored in an array $A[1..n]$, we randomly selected $k \in \{1..n\}$ and used $A[k]$ as the corresponding configuration. The trend we observed for these systems is identical to those discussed in Section 5.2: SRS showed better accuracy than NRS.

![Figure 17. Comparison between SRS and NRS.](image)

Figure 17 shows $\delta_y$ values over $N$ for Apache and BerkeleyDBJ which corresponds to the comparison with Guo2013 and Siegmund2012 of Section 5.4.

![Figure 18. Comparison with Guo2013 and Siegmund2012.](image)

C. Local Minima and PCS Graphs

A standard pitfall when searching configuration spaces is getting stuck in a local minima. Figure 19a shows four points ($\times$) that are local minima, where one of which is the global minimum ($\Omega$). Search procedures might get stuck at a local minimum if they exploit the local contours of a search space.

![Figure 19. Local Minima in PCS Graphs.](image)

A PCS graph is different: it is a monotonically descending performance graph with $\Omega$ at the far right of the $X$-axis. What were local minima in Figure 19a just become undistinguished points in a PCS graph, except for the global minimum, which is anchored at $\Omega$.

Other than one exception, there are no ‘local minima’ at which to get stuck. The exception is stair pollutions, mentioned in Section 3.2. Two stairs may have the approximate same performance, but may represent very different sets of configurations. Our random sampling algorithm can not distinguish multiple stairs with the same performance or distinguish the stair with $\Omega$ from one that does not. Our algorithms select good performing configurations but sometimes will home in on a good-performing stair that does not contain $\Omega$. This is the extent to which ‘local minima’ affects our work.