Finding Near-Optimal Configurations in Colossal Spaces with Statistical Guarantees

JEHO OH and DON BATORY, The University of Texas at Austin
RUBÉN HERADIO, Universidad Nacional de Educación a Distancia

A Software Product Line (SPL) is a family of similar programs. Each program is defined by a unique set of features, called a configuration, that satisfies all feature constraints. “What configuration achieves the best performance for a given workload?” is the SPL Optimization (SPLO) challenge. SPLO is daunting: just 80 unconstrained features yields $10^{24}$ unique configurations, which equals the estimated number of stars in the universe. We explain (a) how uniform random sampling and random search algorithms solve SPLO more efficiently and accurately than current machine-learned performance models, and (b) how to compute statistical guarantees on the quality of a returned configuration, i.e., it is within x% of optimal with y% confidence.

Additional Key Words and Phrases: software product lines, configuration optimization, product spaces, machine learning, uniform random sampling, random search, order statistics

ACM Reference format:

1 INTRODUCTION

1 A Software Product Line (SPL) is a family of programs with similar functionalities. Each SPL program is defined in terms of features, i.e., standardized increments of program functionality. Features are notorious for having constraints: using a feature may require and/or preclude other features. A feature model defines all features and their constraints. A configuration is a unique set of features that satisfies the feature model. The configuration space or product space of an SPL, denoted $C$, is the set of all SPL configurations, exactly one program per configuration. A configuration space can be colossal ($\gg 10^{10}$); a set of $f$ unconstrained features yields a space of size $2^f$. A space of size 250K, which is near the upper-limit to product space enumeration [116], has $f \approx 18$ features, which is tiny for an SPL. Most SPLs are larger. Table 1 lists the sizes of contemporary SPLs taken from [48, 50, 69, 79, 88].

Typical clients want an SPL program to satisfy constraints. A functionality constraint declares required or forbidden features. There are environmental (hardware and platform) constraints. There are specification challenges: many mutually exclusive features can implement the same

| SPL   | #Features | $|C|$ |
|-------|-----------|-----|
| JHipster | 3.6.1     | 45  | $2.6 \cdot 10^4$ |
| axTLS  | 1.5.3     | 64  | $3.9 \cdot 10^{12}$ |
| uClib-ng | 1.0.29   | 269 | $8.0 \cdot 10^6$ |
| ToyBox | 0.7.5     | 316 | $1.4 \cdot 10^4$ |
| BusyBox | 1.23.2   | 613 | $7.4 \cdot 10^{16}$ |
| EmbToolKit | 1.7.0 | 2,331 | $4.0 \cdot 10^{34}$ |
| LargeAutomotive | 17,365 | 5.3 $\cdot 10^{41}$ |

Table 1. SPL space sizes.

This paper extends two prior publications: [91] from 2017 and [15] from 2021.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2022 Association for Computing Machinery.
1049-331X/2022/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Transactions on Software Engineering and Methodology, Vol. 1, No. 1, Article 1. Publication date: January 2022.
functionality in different ways, each offering a unique performance surface. There are performance constraints on program usage workloads. There are (sometimes unknown) combinations of features that are advantageous or detrimental to performance. Given these hurdles, what product of an SPL achieves the best performance? This is the challenge of SPL Optimization (SPLO).

SPLO is daunting. The complexity of feature constraints and the performance influence of features and their interactions is beyond human reasoning. Simply using default configurations is notoriously bad [6]. The solution is to find a configuration with near-optimal performance automatically, which is known to be challenging [41, 43, 49, 55, 60, 83–85, 87, 91, 94, 100, 103, 115, 127].

The Contestants. There are two known ways to find a near-optimal configuration (cno) in an SPL space: (a) create a Performance Model (PM) and use an optimizer or (b) randomly search using Uniform Random Sampling (URS), meaning every configuration in \( C \) has an equal probability of being selected (e.g., \( \frac{1}{|C|} \) where \( |C| \) is the cardinality of \( C \)).

The upper path in Fig. 1 abstracts the process of Machine Learning (ML) PMs: a configuration space is randomly (and not necessarily uniformly) sampled; samples are interleaved with model learning until a model is sufficiently accurate. An optimizer then uses this PM, along with a workload and functionality constraints, to find a cno.

The bottom path abstracts random searching: a functionality-and-workload-constrained subspace is uniformly sampled until a cno is found.

Why is SPLO hard? Three reasons:

- URS is a gold standard for statistical analysis. Uniformly sampling an enumerated SPL space is easy: randomly select an integer from \([1..|C|]\) and index to that configuration. Enumeration of colossal spaces is infeasible, so non-URS sampling methods were used instead [2, 24, 27, 34, 42, 49, 62, 65]. The challenge is each SPL configuration is a solution to a propositional formula and how to index to a solution is unknown.

- Building and benchmarking a configuration is extremely expensive. Minimizing the sample size to while achieving accuracy is critical to all approaches. Today, only heuristics are known, like: use sample size (f, 2f, 3f, . . . ) where f is the SPL’s number of features [42, 46].

- Statistical guarantees on the quality of returned cnos should be required: a cno is within x% of optimal with y% confidence. Such statistical guarantees are unknown today.

1.1 The Central Questions of SPLO

Let a sample be a set of configurations, whose cardinality it its size. Let cbest be a product in \( C \) that has the best performance for a given workload and functionality constraints. Then:

1. How does one find a cno in an SPL configuration space?
2. How accurate (e.g., how near cbest) is the returned cno?
3. What sample size should be used?

1.2 Contributions of This Paper

- Order statistics and URS [9, 126] provide an SPLO statistical guarantee: i.e., a returned cno is within x% of cbest with y% confidence;

- Given any two of (a) accuracy (x%), (b) confidence (y%), and (c) sample size, the third can be determined mathematically, which leads to standardized answer tables;
2 RESULTS ON PERFORMANCE MODELING

2.1 Basic Facts

Performance Modeling. ML approaches to PM creation are enormously diverse [73, 95]; we do not try to be exhaustive or complete. Instead, we review results on Linear Regression (LR), a popular ML approach used in SPL0, that may not be widely known. Let \( \hat{S}(c) \) be the estimated performance of configuration \( c \in \mathcal{C} \). A common form of \( \hat{S}(c) \) is [33, 43, 67, 107, 108]:

\[
\hat{S}(c) = \beta_0 + \beta_1 \cdot x_1(c) + \beta_2 \cdot x_2(c) + \ldots + \beta_h \cdot x_h(c)
\]

(1)

Consider any \( x_1(c) \) in Eqn (1). Either \( x_1(c) \) represents a unique feature, say \( F_j \) in Fig. 2a, meaning \( x_1(c) = 1 \) if \( F_j \) is present in \( c \); 0 otherwise. Or \( x_1(c) \) represents a \( t \)-way interaction of \( t > 1 \) distinct features. Suppose \( x_1(c) \) is the 3-way interaction of features \( \{F_j, F_k, F_q\} \) in Fig. 2b, meaning \( x_1(c) = 1 \) if \( F_j, F_k, \) and \( F_q \) are all present in \( c \); 0 otherwise. If an SPL has \( f \) features, the number of \( x_1(c) \) terms is \( 2^f - 1 \). For any reasonable \( f, 2^f \) is far too big. So an analysis finds 2-way to 5-way interactions that are important to performance [43, 106]. Eqn (1) assumes only \( h \ll 2^f \) terms remain.

Let \( S(c_r) \) be the benchmarked value of configuration \( c_r \). Given a set of \( \{(c_r, S(c_r))\}_{r=1..t} \) pairs, LR finds values for the \( \beta_0 \ldots \beta_h \) that minimizes the sum of the squares of differences between measured and predicted values, i.e., \( \sum_{r=1}^{t} (S(c_r) - \hat{S}(c_r))^2 \) [22].

Optimizer Complexity. All \( x_1(c) \) assume the value 0 or 1. Applying the constraints of a feature model so that only legal configurations are examined, Eqn (1) becomes an instance of 0-1 Linear Programming, which is NP-hard [15, 122]. Although this complexity is specific to LR PMs, any comparable formulation will not alter this result. **Conclusion: An optimizer must solve an NP-hard problem to find \( c_{\text{best}} \).**

Workload and Environment Fragility. A **workload** is a set of tasks that are to be executed by a program. A benchmark measures one or more performance metrics (build size, completion-time, maximum memory footprint, etc.) of a program when executing a workload. All ML PM models known to us are created with a fixed workload. It is well-known that changing the workload alters \( c_{\text{best}} \); the same for changes in execution environment [5, 6, 14, 29, 128, 130]. **Conclusion: a PM may need to be relearned if its workload or environment changes.** More on this in Section 8.2.

2.2 PM Answers to Central Questions

Answers to Section 1.1 questions for contemporary PM research are:

1. A PM “fits” a line or curve through a set of observations \( \{(c_r, S(c_r))\}_{r=1..t} \). Prediction errors are unavoidable, although errors are minimized.

2. Unless an SPL configuration space is enumerated and benchmarked, it is unknown how close a \( c_{\text{no}} \) is to \( c_{\text{best}} \). Of course, enumeration is impractical or impossible in most circumstances.
The expected rank depends on the learning algorithm (see SPLConqueror in Section 5.1), although there are rules-of-thumb. Namely, choose size \((f, 2f, 3f, \ldots)\) where \(f\) is the SPL’s number of features [42, 46].

### 3 RESULTS ON SIMPLE RANDOM SEARCHING

#### 3.1 Performance Configuration Space (PCS) Graphs

Imagine it is possible to benchmark every \(c \in \mathbb{C}\), where \(\$\(c)\) is \(c\)’s measured performance. Small \(\$\) is good (efficient) and large \(\$\) is bad (inefficient). Sort all \((c_i, \$\(c_i))\) pairs in increasing \(\$\(c)\) order and plot them equally-spaced along the \(X\)-axis. The result is a **Performance Configuration Space (PCS)** graph. A **normalized PCS** graph normalizes the \(X\)-axis to the unit interval \([0..1]\), where \(c_{\text{best}}=0\) and \(c_{\text{worst}}=1\). The \(Y\)-axis is similarly normalized, where \(\$\(c_{\text{best}}\)=0\) and \(\$\(c_{\text{worst}}\)=1\). See Fig. 3 [91].

All SPLs have finite (albeit colossal) configuration spaces. Consequently, their PCS graphs are discrete, discontinuous and stair-stepped like Fig. 3, because consecutive configurations along the \(X\)-axis encode discrete decisions/features that make discontinuous jumps in performance [78]. Further, every PCS graph is monotonically non-decreasing, meaning consecutive configurations along the \(X\) axis, like \(c_1\) and \(c_{i+1}\), satisfy \(\$\(c_1\) \leq \$\(c_{i+1}\)\), as some features have no impact on performance.

Random search algorithms are well-suited for non-differentiable and discontinuous functions, like PCS graph plots.

#### 3.2 Simple Random Search (SRS)

URS requires every configuration to have equal probability \(\frac{1}{|\mathbb{C}|}\) to be selected. Given that \(|\mathbb{C}|\) is colossal, we can approximate a discrete distribution with the continuous distribution \(\text{Uniform}(0, 1)\):

\[
\lim_{|\mathbb{C}| \to \infty} \frac{1}{|\mathbb{C}|} \left[ 1 \ldots |\mathbb{C}| \right] = \lim_{|\mathbb{C}| \to \infty} \left[ \frac{1}{|\mathbb{C}|} \ldots \frac{|\mathbb{C}|}{|\mathbb{C}|} \right] = [0..1]
\]

The **Simple Random Search (SRS)** algorithm uniformly selects \(n\) configurations from \(\mathbb{C}\) (i.e., \(n\) points from \([0..1]\)). On average, \(n\) points partition \([0..1]\) into \(n+1\) equal-length segments. The \(k\)th-best configuration out of \(n\), denoted \(c_{k,n}\), has expected rank \(\frac{k}{n+1}\), where the \(k\)th \((\binom{n}{k})\) term below is a normalization constant [9, 126]:

\[
c_{k,n} = k \cdot \binom{n}{k} \cdot \int_0^1 x^{k-1} \cdot (1-x)^{n-k} \cdot dx = \frac{k}{n+1}
\]  

(3)

The expected **rank** or distance \(c_{n0}\) is from \(c_{\text{best}}\) (with rank 0), is:

\[
c_{1,n} = \frac{1}{n+1}
\]

(4)

Let’s pause to appreciate this result. The lone axis of Fig. 4 is the \(X\)-axis of any PCS graph. As the sample size \(n\) increases, \(c_{n0}\) progressively moves closer to \(c_{\text{best}}\) at \(X=0\), Fig. 4a→4c. If a sample size of 99 is used, \(c_{n0}\) will be 1%, on average, from \(c_{\text{best}}\) in ranking along the \(X\)-axis. **Note:** Eqns (3)-(4) do not reference \(|\mathbb{C}|\); \(|\mathbb{C}|\) disappeared when the limit was taken in (2). This means Eqns (3)-(4) predict \(c_{n0}\) \(X\)-axis ranks for an **infinite-sized configuration space**. Only for tiny spaces, \(|\mathbb{C}|\leq1000\), will predictions by Eqns (3)-(4) be low. See Appendix A.
How accurate is the \( \frac{1}{n+1} \) estimate? Answer: We can compute \( v_{1,n} \), the second moment of \( c_{1,n} \), and then the standard deviation \( \sigma_{1,n} \) of \( c_{1,n} \) [15, 81]:

\[
v_{1,n} = 1 \cdot \left( \frac{n}{1!} \right) \cdot \int_0^1 x^2 \cdot (1-x)^{n-1} \cdot dx = \frac{2}{(n+1) \cdot (n+2)}
\]

\[
\sigma_{1,n} = \sqrt{v_{1,n} - c_{1,n}^2} = \sqrt{\frac{2}{(n+1) \cdot (n+2)} - \left( \frac{1}{n+1} \right)^2}
\]

For large \( n \), Eqn (6) converges to \( \sqrt{\frac{2}{n^2} - \frac{1}{n^2}} = \frac{1}{n} \), which equals \( c_{1,n} = \frac{1}{n} \). Fig. 5 shows the convergence rate:

\[
\% \text{diff} = 100 \cdot \left( \frac{c_{1,n}}{\sigma_{1,n}} - 1 \right)
\]

When \( n=50 \), \( c_{1,n} \) is 2% larger than \( \sigma_{1,n} \). For \( n \geq 200 \), there is no practical difference between theoretical \( c_{1,n} \) and \( \sigma_{1,n} \) values, i.e., the standard deviation of \( c_{1,n} \) is small.

Readers may have noticed that our configuration ranking is along the X-axis, not the Y-axis. This is a percentile. In SPL, the goal is to be in the smallest percentile: \( \leq 1\% \) means “in the top 1 percentile”.

**Conclusion:** To find a \( c_{no} \) in a colossal product space, SRS takes a uniform sample of size \( n \), builds and benchmarks each configuration, and returns the best performing configuration, \( c_{no} \), that on average is the top \( \frac{100}{n+1} \) percentile of all products with a standard deviation of \( \frac{100}{n+1} \) percentile.

### 3.3 How to uniformly sample an SPL configuration space

Every SPL has a feature model \( F \). This model can be translated into a propositional formula \( \phi \) [8, 12, 13]. A \#SAT tool can count the number of solutions to \( \phi \) efficiently [111]. We know \( |\phi| = |C| \).

Let uc be the functionality constraints on \( \phi \). The predicate for a user-constrained space is \( \phi \land \text{uc} \).

Alg. 1\(^3\) samples a configuration by assigning a Boolean value to each feature \( f_1, f_2, \ldots, f_\omega \) in \( F \). First, \( f_1 \) is randomly assigned according to its probability \( p_1 = \frac{\#\phi \land f_1}{\#\phi} \) of being true in any configuration. Suppose \( f_1 \) is assigned to false. Then, \( f_2 \) is randomly assigned according to its probability \( p_2 \) of being true in a configuration conditioned to \( f_1 \)'s prior assignment: \( p_2 = \frac{\#\phi \land (f_1 \land f_2)}{|\phi \land \neg f_1|} \).

This procedure advances until the last feature \( f_\omega \) is assigned, and thus the random configuration is completed. A formal proof of Alg. 1's uniformity is given in Appendix B.

**BDDSampler.** A new tool, called BDDSampler [16, 50], implements an optimized version of Alg. 1 [89, 91]. BDDSampler is built on top of the CUDD [31] library for BDDs and is remarkably fast, even for colossal spaces. The last column in Table 2 shows the time BDDSampler needed to sample 1,000 configurations for different SPLs\(^4\) averaged over 100 executions.\(^5\) The 3\(^rd\) column in Table 2 lists the BDD synthesis times for feature models by the procedure of [35].\(^6\)

---

\(^2\) The integrals of this section were evaluated by Mathematica 12.1.1.0.

\(^3\) Knuth first sketched this algorithm in 2009 [66]. Batory reinvented it in 2016 unaware of his work. Oh was first to implement it with practical improvements: using Heule's cube-and-conquer algorithm to find an efficient ordering of features to partition the space [52], caching \#SAT computations to avoid repeated evaluations, replacing the remaining last g bits to assign when they are “don’t cares” with a random g-bit number, and (optionally) caching configurations to remove duplicates thereby achieving sampling-without-replacement [89, 91].

\(^4\) The BDDs of the SPLs in Table 2 are available at: https://doi.org/10.5281/zenodo.4514919

\(^5\) An Intel(R) Core(TM) i7-6700HQ, 2.60GHz, 16GB RAM, operating Linux Ubuntu 19.10 was used.

\(^6\) The tool used to synthesize the BDDs is available at: https://github.com/davidfa71/Extending-Logic
Algorithm 1: Uniform Random Sampling (URS)

1 Configuration URS(ϕ, F):

**Input**: ϕ feature model F’s propositional formula ∧ functionality constraints F = (f₁, f₂, ..., fₜ) be a static, arbitrarily-ordered list of all features in F

**Output**: a random configuration

2 |ϕ| ← #SAT computes the number of solutions to ϕ;
3 **for** each i in (1..ôte) **do**
4  |ϕ ∧ fᵢ| ← #SAT computes the number of solutions to ϕ ∧ fᵢ;
5  Generate a random value j ∈ [0,1];
6  if j ≤ |ϕ ∧ fᵢ| |ϕ| then
7      fᵢ ← true; ϕ ← ϕ ∧ fᵢ; |ϕ| ← |ϕ ∧ fᵢ|;
8  else
9      fᵢ ← false; ϕ ← ϕ ∧ ¬fᵢ;
10    |ϕ| ← #SAT computes the number of solutions to ϕ ∧ ¬fᵢ;
11 **return** (f₁, f₂, ..., fₜ);

SRS requires (i) building a BDD structure, (ii) sampling configurations, (iii) building products, and (iv) benchmarking products. Whereas (i)-(ii) can be done relatively quickly, (iii)-(iv) are computationally expensive, and that is why minimizing the sample size is critical to both SPL0 and ML performance [79]. For example, sampling all 26,256 configurations of JHipster with BDDSampler took 4.48 seconds. However, building and compiling all 26,256 configurations took 4,376 hours of CPU time (182 days approximately or 10sec/build-and-benchmark) and needed 5.2 terabytes of disk on the INRIA supercomputer Grid’5000 [48].

### 3.4 What Sample Size to Use?

A basic question for any SPL0 sampling method is: What sample size is needed to find a near-optimal solution for a given accuracy? As rigorous analyses are usually not cited by the authors of proposed non-URS methods (e.g., [30, 34, 40, 42, 49, 82]), this question may have no answer. URS does. Let ρ be the desired percentile of accuracy (e.g., top 1% sets ρ = .01). Each selected configuration is a Bernoulli trial. The confidence/probability ç that a uniform sample of size n returns a cᵢo in the top ρ accuracy is Eqn (8):

\[ ç = 1 - (1 - ρ)^n \]

Solving for n yields Eqn (9):

\[ n = \frac{\ln(1 - ç)}{\ln(1 - ρ)} \]
Table 3 lists the sample size to achieve given confidence (ç) and accuracy (p) for an infinite-sized space. Example: A configuration in the top 2% of C with 95% confidence is returned when n=148.

Other tables can be derived from Eqn (8) for accuracy (p) and confidence (ç). Example using Table 4a: A budget of 100 samples and 95% confidence returns a configuration in the top 2.95% of all solutions.

<table>
<thead>
<tr>
<th>ç = %confidence</th>
<th>n = sample size</th>
</tr>
</thead>
<tbody>
<tr>
<td>90.0%</td>
<td>8.80% 4.50% 2.28% 1.14% 0.57% 0.29% 0.14%</td>
</tr>
<tr>
<td>95.0%</td>
<td>11.29% 5.82% 2.95% 1.49% 0.75% 0.37% 0.19%</td>
</tr>
<tr>
<td>98.0%</td>
<td>14.49% 7.53% 3.84% 1.94% 0.97% 0.49% 0.24%</td>
</tr>
<tr>
<td>99.7%</td>
<td>20.73% 10.97% 5.64% 2.86% 1.44% 0.72% 0.36%</td>
</tr>
</tbody>
</table>

Table 5 lists t-values given ç and n.

<table>
<thead>
<tr>
<th>ç = %confidence</th>
<th>n = sample size</th>
</tr>
</thead>
<tbody>
<tr>
<td>90.0%</td>
<td>1.71 1.69 1.66 1.65 1.65 1.66 1.61</td>
</tr>
<tr>
<td>95.0%</td>
<td>2.06 2.01 1.98 1.97 1.97 1.96 1.96</td>
</tr>
<tr>
<td>98.0%</td>
<td>2.49 2.40 2.36 2.35 2.34 2.33 2.33</td>
</tr>
<tr>
<td>99.7%</td>
<td>3.3 3.12 3.04 3.04 3.04 2.99 2.98</td>
</tr>
</tbody>
</table>

3.5 Why is URS Important?

A common question is: what is an estimated mean µ of a configuration space property X? Answer: create a uniform sample of size n and benchmark each configuration to obtain its X value. Then compute the mean X̄ and standard deviation s of sampled X values. According to the Central Limit Theorem (CLT) [114], the true population mean µ is contained in the following confidence interval:

\[ \bar{X} - t \cdot \frac{s}{\sqrt{n}} \leq \mu \leq \bar{X} + t \cdot \frac{s}{\sqrt{n}} \]  

(10)

where t is determined from Student’s t-distribution given a desired confidence level ç and sample size n. Table 5 lists t values for some combinations of ç and n [114]. Note: A CLT precondition is that samples are uniform.

Example. Let µ be the average number of features that are present in a configuration. Fig. 6 plots µ estimates for two SPLs with different sampling methods and sample sizes. The X-axis is n, the sample size, and the Y-axis is µ estimates. The straight line (—) indicates the correct µ as these SPLs are small enough to enumerate and compute the correct answer. The dashed lines indicate the 95% confidence envelope for each µ estimate, Eqn (10). X marks estimates by URS. Sampling methods ▲ and ◊ are not uniform; ▲ is QuickSampler [34] and ◊ is DDbS [62]. Observe:

- All three µ estimates converge to an answer with increasing n;
- URS correctly estimates µ with increasing accuracy; other methods converge to different incorrect answers;
- Method ◊ selects different sample sets each time in Fig. 6b, but oddly the same number of features occurs in all samples. Thus, the estimate by ◊ is suspicious as it lacks variability.

Conclusion: Classical statistical methods assume URS as a precondition; if this precondition is violated, computed statistics are suspect [20]. Other benefits of URS include:

- Population statistics (like µ) can be predicted by probability analyses. URS can confirm the correctness of these predictions; and
• When analytical predictions are unavailable, URS can estimate population statistics that a correct analysis should return.

3.6 PCS Graphs of Enumerable and Non-Enumerable SPLs

What do real PCS graphs look like? This is not a fundamental question, but one asked of curiosity. Several small SPLs were enumerated and benchmarked by Siegmund et al. [105, 106], which took months to complete. From his data, we computed their unnormalized PCS graphs, Fig. 7.

- **Apache** is an open-source Web server [7]. With 9 features and 192 configurations, the maximum server load size was measured through autobench and httperf;
- **LLVM** is a compiler infrastructure in C++ [75]. With 11 features and 1024 configurations, test suite compilation times were measured;
- **H264** is a video encoder library for H.264/MPEG-4 AVC format written in C [45]. With 16 features and 1152 configurations, Sintel trailer encoding times were measured; and
• BerkeleyDBC is an embedded database system written in C [18]. With 18 features and 2560 configurations, benchmark response times were measured.

A complete PCS graph plots every point in $C$; this is possible when an SPL configuration space is enumerable. But what about spaces that are too large to enumerate? A number of techniques were tried, and the simplest worked best:

1. Take a uniform sample of size $n=100$ or $n=200$ as this, to us, yields a minimal fidelity PCS graph;
2. For each configuration $c$, build and benchmark it to yield $(c)$;
3. Sort the $(c, (c))$ tuples from best-performing to worst;
4. Let $y_i$ be the $i$th best performance. Plot a PCS graph using these points $\left\{ \left( \frac{1}{n+1}, y_i \right) \right\}_{i=1}^n$.

Example. uClibc-ng is a C library for embedded Linux systems with 269 features and $|C| \approx 8 \cdot 10^{26}$ [90]. A minimum fidelity ($n=200$) PCS graph of uClibc-ng is Fig. 8. Build size was measured.

3.7 SRS Answers to Central Questions

Section 1.1 listed three questions; SRS offers elegant answers for each:

1. How does one find a $c_{no}$ in an SPL configuration space? Answer: Take a uniform sample of size $n$, benchmark each configuration, and return the best-performing configuration, $c_{no}$;
2. How accurate (e.g., how near $c_{best}$) is the returned $c_{no}$? Answer: On average, the $c_{no}$ is $\frac{100}{n+1}$ percentiles from $c_{best}$ with standard deviation of $\frac{100}{n+1}$ percentiles; and
3. What sample size should be used? Answer: Choose a desired accuracy and confidence for a $c_{no}$, and use Table 3 to determine the sample size.

4 RECURSIVE RANDOM SEARCH (RRS)

We believe SRS offers a minimal performance bound for every SPL0 algorithm, as more sophisticated algorithms and those that exploit domain-specific knowledge should perform better. In this section, we review another promising random search algorithm. There are no replacements for SRS yet; a replacement would have an SPL0 statistical guarantee on its $c_{no}$s.

RRS. A $c_{no}$ will be in the top $\frac{1}{10^9} = 10\%$ percentile using a uniform sample of size 9. Increasing the solution precision to the top $\frac{1}{100} = 1\%$ requires a sample size of 99, $11 \times$ larger. Suppose from the first 9 configurations feature $f$ is inferred to be common to configurations in the top 10%. If the scope of the search is restricted to $(\phi \land F)$ and another uniform sample of size 9 is taken, a near-optimal solution would be within $\frac{1}{10^9} \cdot \frac{1}{100} = \frac{1}{10^8} = 1\%$, for a total of 18 configurations; a $5.5 \times$ improvement. This is Recursive Random Search (RRS).

Implementation. A good rule-of-thumb for $c_{best}$, the optimal configuration of a PCS graph, is that it contains some of the top performance-enhancing features of an SPL [21]. We call such features noteworthy. The difficulty is that some features become noteworthy only in the presence of other noteworthy features. Thus the order in which noteworthies are exposed is important.

Consider the PCS graph of LLVM, Fig. 9a. This graph is almost linear. Look how noteworthy features ($f$ or $\neg f$) present themselves in Fig. 9a-d, in order of most-influential to next-most-influential, and so on, recursively restricting the next subspace to search.
To mechanize this, let $N$ configurations be uniformly sampled per recursion. For $r$ recursions, the total number of configurations taken is $n = r \cdot N$. For every sampled configuration $c$, we know its features and its measured performance $S(c)$. Which features are noteworthy? Answer: Given $N$ configurations, compute the average performance $\bar{S}(f)$ of configurations with feature $f$, and the average performance $\bar{S}(\neg f)$ of configurations without $f$. Their difference:

$$
\Delta(f) = \bar{S}(f) - \bar{S}(\neg f)
$$

is the performance influence of $f$. The sign of $\Delta(f)$ indicates whether $f$ improves (negative value) or degrades (positive value) average performance. Further, a $t$-test [38] checks whether $\Delta(f)$ is statistically significant with 95% confidence.\(^8\)

RRS is Alg. 2. The set of features examined for being noteworthy are features common to the top two performing configurations in a recursion’s sample set. Experimentally we found using only the top configuration $T$ was misleading – some noteworthy features of $T$ do not belong to $c_{\text{best}}$ and by selecting them would assure RRS would never reach $c_{\text{best}}$. Shared features in the top two configurations was less misleading; shared features in the top three was too constraining – important features may not be in all configurations. Further, a SAT solver was unneeded to validate that a feature set satisfied feature model constraints: all features of a configuration satisfy feature model constraints; so too is any subset, and any shared subset among $k>1$ configurations.

Comparison. The accuracy of SRS and RRS can be compared by experiments that compute the average rank $\bar{\mu}$ of solutions returned by RRS (which is possible for enumerable SPLs) to the theoretical accuracy of SRS for a sample size $n$, $\frac{1}{n+1}$ a.k.a. Eqn (4). The experiment uses:

- $N$ as the number of configurations per RRS recursion; and
- $n$ as the total number of configurations taken by RRS.

Fig. 10 plots averages of 100 experiments for different SPLs and different $N$. While both $\bar{\mu}$ and $\frac{1}{n+1}$ decrease sharply with increasing $N$, $\bar{\mu}$ is on average better than $\frac{1}{n+1}$.

Key limitations of RRS are:

- It is not always better than SRS when the $N$ (configurations per recursion) is too small; and
- It lacks analyses like $c_{\text{no}}$ rank prediction (Eqn (4)) and confidence guarantees (Eqn (8)).

The next sections evaluate SRS and RRS. A solution to the above limitations is given in Section 7.

---

\(^8\) A typical rule-of-thumb [38] states that the Central Limit Theorem holds whenever the sample size is $\geq 30$. Accordingly, the distribution of the sample means is normal and thus the performance contribution of each feature is estimated by subtracting means and using a $t$-test. However, when $N<30$, a more robust estimator and a non-parametric test is required; in particular, the feature’s performance is calculated by $\Delta(f) = \text{median}(S(f)) - \text{median}(S(\neg f))$, and statistical significance checked with a Mann-Whitney U-test [77].
Algorithm 2: Recursive Random Search (RRS)

Configuration RRS(r, N, φ, NW):

1. **Input:**
   - \( r \): recursion number (initially 0)
   - \( N \): number of configurations per recursion
   - \( φ \): feature model propositional formula \( ∧ \) functionality constraints
   - \( NW \): set of noteworthy features (initially empty)

2. **Output:**\( c_{no} \) best configuration found (set of features)

   - sample \( \leftarrow \) randomly sample \( N \) configurations from \( φ ∧ NW \);
   - sort sample so that \( \text{sample}[0] \) has best performance, and \( \text{sample}[1] \) has next best;
   - commons \( \leftarrow \) negative or positive features common to \( \text{sample}[0] \) and \( \text{sample}[1] \);
   - for each \( f \) in commons do
     - if \( \Delta(f) < 0 ∧ \text{t-test}(f) \) then
       - add \( f \) to \( NW \);
   - if \( NW \) unchanged from previous recursion then
     - return \( \text{sample}[0] \);
   - else
     - return RRS \((r + 1, N, φ, NW)\);

5 EVALUATION USING ENUMERABLE SPLS

SPL researchers use enumerable SPLs (\(|C| ≤ 250K\)) as benchmarks with metrics of overall accuracy (MAPE, defined in Section 5.4) and, to a lesser extent, solution accuracy (average rank of returned \( c_{no} \)) and reliability (standard deviation of returned \( c_{no} \)) to compare different PM algorithms [42, 46, 47, 62, 87]. We adopt these guidelines.

Using the same sample size or smaller, an SPL0 algorithm is more accurate than others if it finds better solutions (\( c_{no} \)). And an SPL0 algorithm is more reliable than others if its solutions have a smaller standard deviation (\( σ \)). A higher \( σ \) means solutions vary more.

We ask the following research questions about SPL0 algorithms:

- **RQ1:** Which algorithm is the most accurate across selected SPLs?
- **RQ2:** Which algorithm is the most reliable across selected SPLs?
- **RQ3:** Are PM accuracy and PM solution accuracy correlated?

5.1 Evaluation Setup

Enumerated spaces allow us to (a) know the true PCS rank of a \( c_{no} \) and (b) compute the difference of a \( c_{no} \)'s true performance \( $(c_{no})$ \) from a PM's estimate \( \hat{(c_{no})}$ \). Taken from [105, 106], the SPLs are:

- BerkeleyDBC is an embedded database system with 18 features and 2,560 configurations [18]. Benchmark response times were measured;
- 7z is a file archiver with 44 features and 68,640 configurations [1]. Compression times were measured; and
- VP9 is a video encoder with 42 features and 216,000 configurations [116]. Video encoding times were measured. To our knowledge, VP9 is the largest SPL that has been enumerated.

Each successive SPL in the above list has a configuration space that is \( \sim 10 \times \) larger than its predecessor. Fig. 11 shows their unnormalized PCS graphs.

We compare SRS and RRS with two PMs: SPLConqueror [107] and DeepPerf [46]. DeepPerf is a state-of-the-art deep sparse neural network that out-performed other major PMs in 2019, including CART [43], DECART [42], Fourier [93], and SPLConqueror. We include SPLConqueror as it is the state-of-the-art in LR PMs, using linear regression as described in Section 2.

Recall the purpose of a PM is to predict the performance of any configuration in \( C \). It is not to find an optimal or near-optimal solution. That is the purpose of an optimizer. Earlier we explained that finding \( c_{\text{best}} \) by an optimizer is NP-hard. To discount this difficulty, we use a perfect optimizer that returns the best-performing configuration according to its PM for free. Of course, such an optimizer does not exist but can be emulated for enumerable configuration spaces. So the conclusions of this section favor PMs.

For SRS and DeepPerf, we ran experiments with sample sizes 50, 100, 200, 500, and 1000. DeepPerf asks for the sample size to use and the number of experiments; hyperparameters for its neural network are configured automatically. For RRS, we ran experiments with \( N \in \{15, 20, 30, 50, 100, 200\} \) configurations per recursion and summed the total number of configurations used after RRS terminates. Remember RRS does not perform well w.r.t. SRS when too few configurations per recursion are used. RRS has a minimum sample size (MinSS) whose value is revealed by experiments RQ1 and RQ2.

For SPLConqueror, the settings of Kaltenecker et al. were used [62]. All five sampling methods of SPLConqueror were evaluated, each producing a distinct PM. Diversified Distance-Based Learning, which we label as S2, was reported to have the best prediction accuracy.\(^9\) For each sampling method, three different sample sizes were used, corresponding to \( t \)-way population sizes \( t \in \{1, 2, 3\} \), although some SPLConqueror algorithms used additional configurations whose numbers we could not control but did report. See [62, 107] for more details.

Each experiment was repeated 100 times and averages are reported. 100 was chosen so that our evaluations would finish in two weeks of compute time. Statistical significance tests are reported in Appendix C for those interested. Our experimental data is at https://doi.org/10.5281/zenodo.7485062.

\(^9\) S1 is Distance-Based, S2 is Diversified Distance-Based, S3 is Solver-Based, S4 is uniform sampling from an enumerated configuration space, and S5 is Randomized Solver-Based.
Finding Near-Optimal Configurations in Colossal Spaces with Statistical Guarantees

Fig. 12. Average percentile rank ($\mu_x$) vs. Average Sample Size ($\bar{n}$) by SPL and SPL0.

ACM Transactions on Software Engineering and Methodology, Vol. 1, No. 1, Article 1. Publication date: January 2022.
5.2 RQ1: Which algorithm is the most accurate across selected SPLs?

Let \( n \) be the number of configurations benchmarked by a PM in an experiment; \( \bar{n} \) the average over 100 experiments. Let \( \mu_x \) be the percentile rank of its \( c_{no} \), also averaged over 100 experiments. \( \mu_x = 5\% \) says a PM returned \( c_{no} \) in the top 5\% (.05 percentile), on average, from \( c_{best} \).

The lines of Fig. 12 (next page) connect \((\bar{n}, \mu_x)\) points of each PM. SPLConqueror has 5 lines, one for each sampling method. Fig. 12a-c show the full results; Fig. 12d-f show a top 5\% (.05 percentile) magnified view. Tables (not graphics) for Fig. 12 are in our Zenodo download.

We found:

- SRS and RRS exhibited the overall best performance;
- When using \( \leq 20 \) configs/recursion, SRS dominates RRS in all but one point in 7z, Fig. 12e. When \( \geq 30 \) is used, RRS dominates SRS for all SPLs. This discussion continues in RQ2;
- When RRS uses \( >200 \) configurations total, it returns a \( c_{no} \) whose normalized rank is less than 0.2\% on average, compared to the theoretical SRS \( c_{no} \) normalized rank of 0.5\%, Eqn (4);
- The \( \mu_x \) of SPLConqueror PMs varied, depending on the sampling method and sample size. S2 dominated other SPLConqueror algorithms and outperformed RRS in BerkeleyDBC and 7z. However, no SPLConqueror algorithm outperformed SRS or RRS in VP9.
- DeepPerf under-performed SRS and RRS for all sample sizes and SPLs. DeepPerf dominated SPLConqueror on VP9, but under-performed BerkeleyDBC and 7z except on three points.

With respect to better \( c_{no} \) accuracy with larger sample sizes, we observed:

- SRS and RRS steadily improved \( \mu_x \) values with increasing sample sizes in all SPLs. SRS and RRS produced the most consistent results;
- More configurations did not assure better \( c_{no} \) for PMs. DeepPerf found progressively better \( c_{no} \) as sample sizes increased to 500 but did not consistently improve \( c_{no} \) afterward.
- SPLConqueror PM \( c_{no} \) varied considerably. Only two results, S2 and S3 in VP9, showed strictly improving \( c_{no} \) with increasing sample size.

With respect to theoretical predictions:

- Fig. 13(a) shows the \( \mu_x \) of SRS with sample size \( n \) matches Eqns (4)-(6). The SRS \( \mu_x \) of 7z and VP9 measurements are slightly lower than theoretical \( \mu_x \) because some configurations exhibit the same performance but the rank that we assigned measured the number of configurations that have better performance. This possibility is evident in the flat shelf of configurations approaching the origin in the PCS graphs of these SPLs (Fig. 11).

Summarizing:

- Fig. 12 shows that RRS generally outperforms SRS, DeepPerf and SPLConqueror over a wide range of different sample sizes in different SPLs.
- SRS and RRS \( c_{no} \) progressively move toward the origin \((c_{best})\) of each PCS graph as sample sizes increase. DeepPerf \( c_{no} \) plateau for BerkeleyDBC and 7z.
- We consider SRS as a "minimal performance bound" for SPL0s, as it relies only on URS. DeepPerf failed to outperform SRS for all plotted 45 points in Fig. 12. SPLConqueror failed to outperform SRS in 28-of-45=62\% plotted points. These results raise a general concern on the \( c_{no} \) accuracy of PMs.

---

10 SRS bettered SPLConqueror on 6 points of BerkeleyDBC, 7 in 7z, and 15 in VP9, for a total of 28-of-45=62\%. Some SPLConqueror experiments used smaller or larger numbers of samples compared to SRS experiments. For these cases, we used order statistics \((1/(n+1))\) to derive if SPLConqueror performed better than SRS or not.
• SPLConqueror outperformed RRS in 12-of-45=27% of the data points in Fig. 12.\textsuperscript{11} However, the sampling method and sample size that yielded these results were unknown before these experiments. A priori, it is not obvious which SPLConqueror algorithm to use ahead of time.
• A perfect optimizer was used, which biases the results of this section toward PMs.

**Conclusion:** *Sampling (esp. RRS) produced the best $\mu_x$ solutions in these experiments.*

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig13}
\caption{SRS Theoretical ($\mu_x, \sigma_x$) and Experimental ($\mu_x, \sigma_x$).}
\end{figure}

### 5.3 RQ2: Which algorithm is the most reliable across selected SPLs?

The standard deviation $\sigma_x$ of $\mu_x$ measures the reliability of solutions returned by SPL0 algorithms. The larger the $\sigma_x$, the less stable or more variable the result; the smaller the $\sigma_x$, the better.

The lines of Fig. 14 (next page) connect ($\bar{n}, \sigma_x$) points of each SPL0 algorithm. Fig. 14a-c are the full results; Fig. 14d-f show a magnified top 5% (.05 percentile) view. Tables (not graphics) for Fig. 14 are in our Zenodo download. We found:

- SRS and RRS demonstrated consistently small $\sigma_x$ below 1% for $\bar{n} \geq 200$ in all SPLs, matching the theoretical predictions of Fig. 5. Further the $\sigma_x$ of SRS and RRS decreased steadily – well below 1% – as the sample size increased;
- When using $\geq 30$ configs/recursion, the $\sigma_x$ of $\mu_x$ is clearly lower for RRS than SRS (see Fig. 14b-c). Henceforth, we use MinSS=30 configurations per recursion unless otherwise specified.
- DeepPerf reduced $\sigma_x$ with increasing sample sizes up to 500; but not consistently over 500 ($\sigma_x$ of 7z increased $\geq 500$). Further DeepPerf has a significantly higher $\sigma_x$ than SRS and RRS for all SPLs, doing no better than $\sigma_x=6\%$.
- $\sigma_x$ for SPLConqueror varies considerably. S3 and S5 had the lowest $\sigma_x$, as it approached 0. The S3 and S5 PMs were created by samples from a SAT solver, which are known to be biased [62]. We conjecture these PMs returned similar solutions. In general, larger sample sizes did not consistently lower $\sigma_x$ and SPLConqueror $\sigma_x$ were higher than those of SRS and RRS.
- The $\sigma_x$ of SRS matches the theoretical $\sigma_x$ for Order Statistics, Fig. 13(b). The SRS $\sigma_x$ for 7z and VP9 measurements are slightly lower than theory Eqns (4)-(6) for the reason given earlier.

**Conclusion:** *Sampling (esp. RRS) produced the lowest $\sigma_x$ values and were the most reliable in these experiments.*

**Additional Evidence.** In [91], we compared a draft of RRS (here called RRS\textsubscript{0}), with two PMs, one by Sarkar [101] and a precursor to SPLConqueror [106], on small SPLs explained earlier: Apache (|$\mathcal{C}$|=192), LLVM (|$\mathcal{C}$|=1024), and H264 (|$\mathcal{C}$|=1152). SRS dominated these PMs on all SPLs, and RRS\textsubscript{0} dominated SRS, consistent with results of this section.

\textsuperscript{11} SPLConqueror outperformed SRS in 7 points of BerkeleyDBC, 5 in 7z, and 0 in VP9, for 12-of-45=27%.
Fig. 14. Average Reliability ($\sigma_x$) vs. Average Sample Size ($\bar{n}$) by SPL and SPL0.
5.4 RQ3: Are PM accuracy and PM solution accuracy correlated?

An implicit assumption in the ML PM literature is “PM accuracy is correlated to PM solution accuracy” [42, 43, 106, 107, 109], which we denote as conjecture $K$. To quantify $K$, we use the Mean Absolute Percentage Error (MAPE), which is widely applied to measure PM accuracy [42, 46]. MAPE accounts for the difference between $c$’s predicted performance $\hat{S}(c)$ and $c$’s benchmarked performance $S(c)$.

For an enumerable space $\mathbb{C}$:

$$\text{MAPE} = 100 \sum_{c \in \mathbb{C}} \frac{|S(c) - \hat{S}(c)|}{S(c)}$$

The box-plots\textsuperscript{12} of Fig. 15a summarize MAPE values for the PMs obtained with DeepPerf and SPLConqueror. DeepPerf consistently produces more accurate and reliable PMs than SPLConqueror (i.e., the boxes are nearer to the X-axis and narrower, respectively).

Fig. 15. MAPE accuracy of DeepPerf and SPLConqueror (S1-S5).

However, DeepPerf’s predictions are not that good and worsen as $|\mathbb{C}|$ increases. Fig. 15b zooms MAPE values to DeepPerf’s scale. DeepPerf’s (a) accuracy decreases with increasing SPL size $|\mathbb{C}|$, as the median values are 3.72 (BerkeleyDBC), 9.73 (7z), and 17.5 (VP9) and (b) reliability also reduces with increasing SPL size $|\mathbb{C}|$, as the 25\textsuperscript{th} and 75\textsuperscript{th} percentiles are [2.61, 5.78] for BerkeleyDBC, [7.94, 19] for 7z, and [10.1, 44.0] for VP9. This suggests that although PMs for increasingly larger spaces can be created with small sample sizes, PM MAPE accuracy suffers.

PM solution accuracy is the rank of a $c_{no}$ returned by a PM in an RQ1 experiment. (Again, 100 such experiments were done per [PM, SPL, sample size] triplet). A pair (MAPE, $\beta$) can be defined for each PM per experiment. The scatter-plot in Fig. 16 shows the (MAPE, $\beta$) pairs collected from all PM experiments. Now conjecture $K$: If MAPE and $\beta$ are ideally correlated, there would be a 1-to-1

\textsuperscript{12} A box-plot encodes the values of five percentiles [121]. The lower-end of the thin vertical line is the 0\textsuperscript{th} percentile (or lowest value); the upper-end denotes the 100\textsuperscript{th} percentile (or highest value). The horizontal line in the box denotes the median; the box extends downwards to indicate 25\textsuperscript{th} percentile boundary and upwards the 75\textsuperscript{th} percentile boundary.
relationship between MAPE and β values; the points would follow a clear pattern, being aligned on a straight line or a curve. Further, if they were positively correlated, low MAPE values would correspond to low βs. Therefore, an optimizer should return better models with lower MAPE values.

Fig. 16 doesn’t show this: DeepPerf in BerkeleyDBC displays a wide range of βs for the same MAPE values (i.e., the points are vertically stacked). Inversely, for SPLConqueror S1 in VP9, PMs with very different MAPE values got roughly the same βs (i.e., the points are horizontally aligned at different heights).

Table 6 lists the dependency between MAPE and β estimated with Spearman’s ρ, Kendall’s τ, Hoeffding’s D [53], and Distance Correlation (dCor) [110]. The magnitude of these measures shows the strength of the dependency. The higher the magnitude, the more dependent are MAPE and β. Whereas the magnitude of ρ, τ, and dCor goes from 0 (no dependency) to 1 (total dependency), D magnitudes range from -0.5 (no dependency) to 1 (total dependency). To facilitate its comparison with the other
measures, $D$ was rescaled to $[0..1]$. Also, $\rho$ and $\tau$ might have had a negative sign if MAPE and $\beta$ had an inverse relationship ($\beta$ decreasing as MAPE increases), but this didn’t occur. In this table, a correlation measure $c$ can be interpreted as very weak if $c<0.2$, weak if $0.2\leq c<0.4$, moderate if $0.4\leq c<0.6$, strong if $0.6\leq c<0.8$, and very strong if $c\geq0.8$.

**Conclusion: An implicit assumption in ML PM literature is PM accuracy is correlated to PM solution accuracy. We found evidence to the contrary, as the correlation was weak in our experiments.**

5.5 Threats to Validity

There are two confounding factors: SPLs and sample size. We considered three enumerable SPLs whose sizes were $\sim 10\times$ larger than the next. Different performances might have resulted using other SPLs. However, SRS and RRS experimental results on an additional three enumerable SPLs in [91] (smaller than the SPLs used here) were consistent with this paper’s results (see end of Sect. 5.2.)

As for sample size, a goal or motivation of prior work was to use the smallest sample sizes possible to get accurate predictions. (A justification for this assertion was given in Sect. 3.3). We followed a standard evaluation procedure used in prior work to compare SRS and RRS with DeepPerf and SPLConqueror [46]. SRS and RRS consistently exhibited the smallest $\mu_x$ and smallest $\sigma_x$ of $c_{no}$'s returned across all sample sizes and SPLs considered. It is possible with larger sample sizes that DeepPerf and SPLConqueror might have performed better.

5.6 Summary

SRS and RRS consistently produced the lowest ranked and most stable $c_{no}$s (i.e., smallest $\mu_x$ and $\sigma_x$) across diverse enumerable SPLs of different sizes and sample sizes. We noticed in the ML PM literature an implicit assumption that a more accurate PM should produce more accurate $c_{no}$s, but the results of RQ1 and RQ2 suggested otherwise. Upon further investigation, we found the correlation of PM model accuracy is weak w.r.t. $c_{no}$ (solution) accuracy. **We again remind readers that we used a perfect optimizer to compute our PM results; an imperfect optimizer would unlikely improve PM performance.**

We offer the following explanation for these results. Creating an accurate PM for even small spaces is hard. Look carefully at Figs. 12-16 to see a recurring trend: as SPL size $|C|$ increases, performance graphs become progressively more wild. And greater overall PM accuracy does not necessarily lead to better solutions. We fail to see how small sample sizes can produce truly accurate PMs. It is asking too much. Others, prior to us, reached a similar conclusion [128, 130].

**Conclusion: Random sampling is a better technology match for SPL0 than ML PMs.**

6 EVALUATION OF SRS AND RRS ON KCONFIG SPLS

We evaluate SRS and RRS on two SPLs that, to our knowledge, have not been evaluated in PM work and $c_{no}$ estimates of $c_{best}$ were found. Both used the Kconfig configuration tool [65]. They are:

- axTLS 2.1.4 is a client-server library with 94 features and $2 \cdot 10^{12}$ configurations [10], and

![Fig. 17. PCS graph estimates using 200 configurations.](a) axTLS 10^{12} (b) ToyBox 10^{91}
• ToyBox 0.7.5 is a Linux command line utilities package with 316 features and $1.4 \cdot 10^8$ configurations [112].

Both were benchmarked for their build size. Fig. 17 shows their minimum fidelity PCS graphs. We ask:

**RQ4:** Does RRS consistently outperform SRS in colossal configuration spaces?

Unlike the SPLs from Section 5, we cannot measure the precise $X$-axis rank of configurations nor the value of $c_{best}$ as both require enumeration. We can compare the true build size of solutions of SRS and RRS from the same SPL to determine the best $c_{no}$.

We devised an experiment to address **RQ4** so that it could be completed within two weeks:

- Compare SRS and RRS with the same total number of samples $n = \{100, 200, 300, 400, 500\}$;
- SRS samples $n$ configurations and reports the minimum build size;
- RRS samples 30 configurations per recursion;
- RRS terminates once the total number of configurations it uses reaches $n$, or by not finding a noteworthy feature, or when the constricted configuration subspace is smaller than 30 and enumeration occurs; and
- All experiments are repeated 25 times.

Fig. 18 shows the results.

![Fig. 18. Optimization of Kconfig SPLs using different sample sizes.](image)

**Observations.** SRS generally found progressively better solutions as $n$ increased for both axTLS and ToyBox; solutions for axTLS seemed to reach a fixed point when $n>200$.

RRS terminated early for axTLS for $n \in \{300, 400, 500\}$, where the average number of configurations benchmarked was 210, 217, 220. At termination, the last constricted space was so small that it was enumerated. The odd shape of Fig. 18a is simply RRS repeatedly converging on a near-minimum build size after examining >200 configurations. Overall, RRS found $c_{no}$s with smaller build sizes than SRS.

**How good are these results?** In an abandoned experiment prior to **RQ4**, we uniformly sampled and benchmarked 46250 configurations each from axTLS and ToyBox (included in our Zenodo download). We were able to salvage this work for **RQ4** as a 46250 point PCS graph, Fig. 19. The best solution had the percentile rank of $1/(46250 + 1) \cdot 100\% = .22\%$ or 5-sigma ($\leq .23\%$), a high level of resolution [98, 119]. We then overlaid the results of Fig. 18 and Fig. 19 to produce Fig. 20.

Fig. 20 magnifies Fig. 19 to the top-performing percentiles. PCS$_{best}$ is the best-performing of all 46250 configurations. The dashed black line is the PCS$_{best}$ boundary. With increasing sample sizes,
SRS solutions approach PCS\textsubscript{best}, as expected, but are never below PCS\textsubscript{best}. RRS solutions appear as $\mathbf{X}$s; they are inside the .22% percentile, visually on the Y-axis of each PCS graph usually below PCS\textsubscript{best}. Overall, RRS solutions are better than PCS\textsubscript{best} once $n \geq 200$.

**Conclusion:** RRS finds better $\mu_y$ solutions than SRS. This has been a consistent result from small through colossal SPLs in our experiments.

### 7 FIXED BUDGET SPL0: THE ESSENTIAL PROBLEM

Sections 5–6 compared different SPL0 algorithms by averaging experiments that sampled tens of thousands of configurations per SPL. This extravagance is unlikely to be common in practice.

Instead, users are more likely to have a fixed budget (a maximum allotment of configurations for benchmarking) because of limited time, limited costs, or other reasons. The challenge is that no single SPL0 algorithm will outperform all other algorithms for all SPLs or sample sizes. We know that RRS-is-always-better-than-SRS is false: there are cases in this paper where SRS performs better than RRS. But if a statistical bound on the quality of a solution is needed, SRS is the only game in town.

Here is a solution to the fixed budget SPL0: run both SRS and RRS with the same number of configurations. Both are executed in steps of $N$ configurations, where $N \geq \text{MinSS}$.

The first step samples $N$ configurations using SRS. These samples are reused as the first $N$ configurations of RRS. At this point, both SRS and RRS return the same “near optimal” configuration.
In subsequent steps, SRS samples another N configurations from the entire space, while RRS samples a different set of N configurations from a noteworthy-constricted space. This last step is repeated until the allocation is exhausted. The best c_no returned by SRS or RRS is chosen, along with the statistical guarantees of the SRS c_no. SRS guarantees give a conservative bound on the goodness of the RRS c_no.

**Example.** Consider a budget of 450 configurations. The first 50 are used by both SRS and RRS; 400 configurations remain. 200 configurations are then allocated to both SRS and RRS, and are consumed in 4 additional rounds of 50 configurations each. A total of 450 configurations is consumed.

We repeat this 3 times, i.e., we conduct 3 identical experiments (R1–R3) whose results are not averaged. Fig. 21 shows all three experiments return essentially the same result. Each red dot on the Y-axis indicates the first round results for both SRS and RRS for an experiment. Each red dot attached to two lines: one for SRS and the other for RRS.

Table 7 tallies the results of (R1–R3). The c_no columns list the minimum build size found and “Best Alg” indicates which algorithm produced the solution. θ-Bound is a conservative theoretical bound on the goodness of that solution, derived from Eqn (8) using 250 configurations at 1% yields 91.9% confidence.

**Conclusion:** A solution to the fixed budget SPLO problem is to give the same number of configurations to SRS and RRS, and take the best solution of the two. The statistical quality of the SRS solution serves as a conservative bound for RRS.

### 8 RELATED AND FUTURE WORK

#### 8.1 Highly Configurable Systems

**Highly Configurable Systems (HCSs)** form a broader universe in which SPLs and SPLO reside. HCSs have configuration parameters that are real and/or binary variables called options or tuning knobs. Unlike SPLs, an HCS has no feature model.

A pipeline of t tools is an example. Each tool has k (e.g., command-line) options. Selecting any or all of the k options for a tool is possible, and selecting/deselecting an option for one tool has no effect on the selection or deselection of options of other tools. The configuration space size for
this problem is precisely \(2^{k \cdot t}\). HCSO, the HCS counterpart to SPL0, finds values for each of the \(k \cdot t\) options that work best together for a given workload and environment [57].

Another example is a database system with \(w\) real-valued tuning knobs [6]. A space of \(\mathbb{R}^w\) option combinations must be explored; the setting of one knob may trigger adjustments of other knobs. A key problem is to create ML models to understand the causal functional relationships among knobs [57]. HCSO finds a \(w\)-tuple that achieves a near-optimal performance [128, 130].

Yet another example is the algorithm configuration or parameter tuning problem [54], where the parameters of an algorithm are configured to achieve the algorithm’s optimal performance for a given set of problem instances.

At a high abstraction level, HCSO and SPL0 look alike. Unbeknown to us in 2003, Ye and Kalyanaraman developed an RRS-like algorithm (also named RRS) to search contour plots for minima in network parameter configurations [128]. They uniformly sampled an \(\mathbb{R}^2\) space, and used performance rankings to identify the top “noteworthy” 2D points. Then their RRS recursively drills down on areas surrounding these points to find minima. As there are no features (as in SPLs), the mechanisms of their RRS algorithm differ from ours. They also discovered Eqns (8)-(9) to guide their search and to choose sample sizes. Here again, their context and use of these equations differs from ours, but much is the same.

Fig. 22 is taken from [128]: the 2D contour is randomly sampled, and the top (in this case 3) performing regions in blue are “noteworthy” and RRS explores regions around these points.

The core differences between HCSSs and SPLs are:

- HCSSs have no feature model;
- Our SRS algorithm provides statistical guarantees on \(c_{no}\) it returns. Order statistics are also useful and may applicable to HCSO algorithms; and
- URS of SPL spaces is much harder as configurations are solutions to propositional formulas rather than points in continuous real 2D or n-D HCS spaces.

The rest of this section focuses on related work in the SPL domain.

### 8.2 Relevant Results in ML PMs

**Other PMs for SPL0s.** Guo et al. encoded a PM as a Classification and Regression Tree (CART) [43]. Sarkar et al. extended [43] with “projective sampling”, a technique that checks performance-estimation accuracy improvement with more samples [101]. Later, Guo et al. improved the efficiency of CART by resampling and automated parameter tuning techniques [44].

Zhang et al. used Fourier learning and incrementally sampled configurations until a PM achieved a desired accuracy [129]. Ha et al. combined Fourier learning with LASSO regression to improve the efficiency of learning Fourier coefficients for each feature [47]. Dorn et al. used probabilistic programming to derive a PM that captures the uncertainty from benchmarking configurations and reasoning with incomplete data [33]. Martin et al. compared different ML techniques and discovered that different methods work better for different SPLs and that feature selection techniques from ML can improve learning in general [80]. These papers were evaluated using relatively small SPLs with \(\leq 60\) features and \(|C|\leq 250K\).

**Scaling PMs.** As of today, PMs of SPLs with \(|C|>10^6\) are rare. When attempted, a non-URS sample is taken whose size ranged from 500-5000 configurations, i.e., the size of an enumerated SPL in this...
paper. Recently, PMs for Linux were created from 85K configurations [79]. Whether accurate PMs for colossal spaces can be learned from small samples (85K ≪ ∼104000) and be optimized efficiently is an open question beyond the scope of our paper.

**Improving PM Accuracy.** PMs are not very accurate [79, 115]. An SPL codebase can be carved into regions (methods or groups of methods) that have the same feature presence condition (i.e., a feature qualification that must be satisfied for the region to be present in a product). By selecting configurations that cover (almost) all execution paths, and partitioning the codebase into regions, a PM for each region is created, these PMs are then composed to produce a composite PM with improved accuracy. Fixed workloads are customized for each region [115].

**Transfer Learning.** A PM is created with a fixed workload. Should the workload change, the PM may need to be relearned (Section 2). An alternative is transfer learning [59]. Let \( \hat{\$}: \mathcal{C} \rightarrow \mathbb{R} \) be the performance estimation function of a PM for space \( \mathcal{C} \). A transfer function (TF) translates a \( \hat{\$} \) learned for workload \( w \) to another function \( \hat{\$}' \) with a different workload \( w' \). A linear TF, \( \hat{\$}'(c) = \alpha \cdot \hat{\$}(c) + \beta \), is postulated, \( \forall c \in \mathcal{C} \). The values of constants \( \alpha \) and \( \beta \) are learned. Linear TFs work well for small workload distortions, but existing evidence suggests otherwise for greater distortions [59, 79].

A recent paper by Martin et al. [79] presents a heterogeneous transfer learning method (tEAMS) that works surprisingly well to evolve PMs of progressive releases of Linux. MAPE values for newly learned PMs are in the 8.2%-9.2% range. When using the same budget, tEAMS produces transferred PMs with MAPE values 5.6%-7.1%. However, MAPE values tend to degrade after multiple transfers.

### 8.3 Optimizers

**Optimizers for SPLs.** Optimizers in the SPL0 literature have focused on multi-objective optimization using evolutionary algorithms [32, 68, 127], active learning [131], filtered Cartesian flattening [117, 118], and integer programming [127].

FLASH is the only paper known to us that uses ML PMs specifically for optimization [87]. It is based on Sequential Model-Based Optimization [56], a broad generalization of RRS for HCSO. To optimize a performance metric, FLASH builds a CART model with an initial learning set \( L \) of benchmarked configurations. Then another set \( S \) of configurations is chosen, CART estimates the performance of each \( s \in S \), and the best-performing configuration, \( c_{no} \), from \( S \) is returned. This \( c_{no} \) is then benchmarked, added to \( L \), and this cycle repeats for a budgeted number of iterations. FLASH was evaluated on tiny (<6 options w. \( |C|<4K \)) and small (<20 options w. \( |C|<240K \)) HCSs.

**Domain-Specific Optimizers.** Exploiting domain-specific knowledge can lead to better \( c_{no}s \). COZART [72] is a tool to find a Linux kernel configuration with minimum build size. With prior knowledge of which features are necessary for booting the Linux kernel and that build size decreases by deselecting features, COZART derives a configuration that selects the necessary features and excludes others as much as possible. COZART does not search for configurations, yet it finds a configuration smaller than sampling does.

**Random Search Optimizers.** Random Search is a family of numerical optimization algorithms for functions that are discontinuous and non-differentiable [17, 123]. SRS and RRS are examples. There is nothing preventing SRS or RRS to be used as an optimizer for a PM: replace the component that builds a configuration \( c \) and benchmarks it, with a component that calls a PM to return an estimate of \( c \)’s performance. The inaccuracy of PM predictions may limit the utility of statistical guarantees of SRS.
8.4 Sampling SPL Configurations

As late as 2020, it was believed that URS of non-enumerable SPL spaces was infeasible [62, 97]. Consequently, novel sampling algorithms were proposed as substitutes. Dutra et al. devised QuickSampler which randomly selects features and attempts to fix them using a MaxSAT solver [34], a solver that tries to maximize the number of satisfiable CNF clauses. Kallenberg et al. introduced Diversified Distance-based Sampling (DDbS) which treats configurations as vectors and derives configurations with maximum difference among them [62]. MaxSAT (and thus QuickSampler) does not achieve URS and DDbS is not scalable [92].

Some build tools offer their own sampling algorithm. The Kconfig language [65] is supported by the conf tool [36], that has the randconfig option to randomly generate configurations that are not uniform. randconfig assigns values to features in the order they appear in a Kconfig specification, so that a valid value for a feature may be constrained by the selection of prior features. Samples are therefore biased. Recently, another tool called KconfigSampler [37] supports the hierarchical random sampling of the Linux Kernel. This kind of sampling is not uniform but ensures that features at the same abstraction level in the Kconfig specification have the same probability of appearing in a random configuration. KconfigSampler is implemented as a net of interconnected BDDs.

Other tools partition the solution space into cells as evenly as possible using universal hashing functions. Then, they select one cell at random, and generate a solution with a SAT solver. UniWit [26] was the first sampler that implemented this idea, which guarantees uniformity but has serious scalability limitations. Two later iterations of UniWit, called Unigen [28] and Unigen2 [25], tried to improve scalability while keeping uniformity, with not much success [51, 97]. The last UniWit iteration is Unigen3 [76], which finally sacrifices uniformity to provide scalability.

Other work achieved URS by counting solutions of a propositional formula $\phi$. Oh et al. were first to experimentally demonstrate URS of large SPL spaces. They used a model counting BDD to count the exact number of solutions to $\phi$ and functionality-constrained versions of $\phi$ [91]. This work was later generalized with the Smarch tool, which uses #SAT and Alg. 1, Section 3.3.

Three other samplers based on counting are Spur [4], KUS [102], and BDDSampler [50]. Spur relies on #SAT technology, KUS on a knowledge compilation structure called Deterministic Decomposable Negation Normal Form (d–DNNF), and BDDSampler on BDDs. The evaluation of Unigen2, Smarch, Spur, KUS, and BDDSampler was reported in [50]; a variety of models, in terms of size (from 14 to 18,570 variables) and application domain (automotive industry, embedded systems, a laptop customization system, a web application generator, integrated circuits, etc.) were examined. Results showed that only BDDSampler currently provides both uniformity and scalability.

8.5 Feature Models and URS

Numerical Features. We focused on binary \{0,1\} features in this paper as this matches classical SPL feature models [8, 13]. However, the Linux build tool Kconfig [64] routinely has feature models with binary and numerical features (NFs). A NF is a numerical value within a bounded range, which can be approximated by an integer in a corresponding range. Bit-blasting is a technique to encode numerical values as bit vectors and arithmetic operations and constraints as propositional formulas [23]. Doing so allows NF propositional formulas to be directly analyzed “as is” by both SRS and RRS [83, 84]. As DeepPerf and SPLConqueror can handle NFs natively, future work should compare how SRS and RRS perform w.r.t. DeepPerf, SPLConqueror, and FLASH on NF models.

Scalability of URS. Our analysis of URS, Eqns (2)-(9), yields results for an infinite-sized configuration space. However, the best tools today [16] cannot analyze Linux, the largest known SPL. Extending today’s #SAT and BDD technologies to process Linux spaces remains an open problem.
Dimension Reduction. Not all features contribute to performance; most features of an SPL are of this type. There are several ways in which irrelevant features can be identified and removed from ML PMs [3, 46, 107]. SRS and RRS do something similar: they ignore non-noteworthy features. In contrast, how performance-irrelevant features can be eliminated from a feature model’s propositional formula \(\phi\) and still admit model counting is not obvious. If this could be done, it might solve the scalability problems that remain for URS, discussed above.

Tseitin’s Transformation. Not any translation of a feature model to propositional formula \(\phi\) and then to a CNF formula, \(\phi^{cnf}\), can be used with a \#SAT sampling tool. Some translations do not preserve the 1:1 correspondence between products and solutions of \(\phi\), resulting in an over-counting. Tseitin’s transformation is one of several transformations that preserve the required 1:1 correspondence for URS [113]. The check: if a translation of \(\phi\) to \(\phi^{cnf}\) adds no additional variables (features), then \(|C|=|\phi^{cnf}|\). BDDs do not have this problem. See Appendix D for more details.

RRS vs. SRS. A perfect RRS would constrain \(C\) in each recursive iteration by selecting a subspace that always contains \(c_{best}\). Currently, RRS uses a heuristic that chooses noteworthy features with the best contributing performance in a sample. This procedure works most times, but as we saw not always. An open problem remains: is there an improved RRS algorithm or analysis that always selects a subspace containing \(c_{best}\) with a computable degree of confidence?

9 CONCLUSIONS

ML is an alluring way to explore PMs for SPLs. But lacking a scalable way to uniformly sample highly-constrained spaces of colossal (\(\geq 10^{18}\)) SPLs had two consequences. (1) Serious efforts were spent on non-URS methods to find substitutes for URS [2, 24, 27, 34, 42, 49, 62, 65], but to properly evaluate their statistical behavior, a gold standard required URS. (2) Most PMs were not adequately evaluated for scalability; SPLs with enumerable spaces (\(\leq 250K\)) were common until recently (e.g., [79]). In Sect. 3.3, we diminished these problems by showing how to uniformly sample colossal SPL configuration spaces as large as \(10^{1441}\).

An initial motivation for PMs was to find SPL \(c_{no5}\) for a given workload. Typical PMs required an optimizer to find a \(c_{no}\): but the only way to determine the quality of \(c_{no5}\) (e.g., how near they are to optimal) required enumerable SPLs. In Sects. 3.1–3.2, we showed how order statistics with URS provided a needed statistical guarantee for colossal SPLs: a \(c_{no}\) is \(x\)% from optimal with \(y\)% confidence. Further, given any two of (accuracy \(x\)% confidence \(y\)% or sample size \(n\)) for a \(c_{no}\), the third is computed by an equation or found in a table.

Two random search algorithms that used URS were presented, SRS and RRS. With enumerated SPLs in Sect. 5, we compared them to state-of-the-art PMs, DeepPerf (a sparse neural network) and SPLConqueror (linear regression), on \(c_{no}\) accuracy (average distance \(\mu\) from optimal) and reliability (standard deviation of \(\mu\)). Experiments showed SRS dominated both PMs, and RRS dominated SRS. Further, a common belief in the PM literature is “a more accurate PM produces a more accurate \(c_{no}\)”. We found evidence to the contrary, where PM accuracy was weakly correlated to \(c_{no}\) accuracy.

In Sect. 6, we demonstrated the efficacy of RRS and SRS on two colossal SPLs: axTLS (\(|C|=10^{12}\)) and ToyBox (\(|C|=10^{81}\)). Sampling at most 500 configurations, RRS found \(c_{no5}\) that were inside \(.22\) percentile (or 5-sigma) of optimal for both SPLs. And in Sect. 7, we presented a fixed budget algorithm that gave the same sample size to both RRS and SRS, let each compute their \(c_{no5}\) where the best \(c_{no}\) was returned along with the statistical guarantees of SRS, as RRS has no guarantees.

Our work opens research topics of substance: (1) generalize URS to numerical features; (2) compare PMs with SRS and RRS on numerical feature models; (3) use URS to determine how well PMs scale to colossal SPLs; and (4) improve URS scalability to the largest known SPL: the Linux Kernel.
Finding Near-Optimal Configurations in Colossal Spaces with Statistical Guarantees

Acknowledgments. We thank the referees for their help to improve this paper. We also thank Prof. Marijn Heule (CMU), Daniel-Jesus Munoz (U. of Malaga), Prof. Maggie Myers (UT Austin), and Prof. Norbert Siegmund (U. Liepzig). Work by Oh and Batory was supported by NSF grants CCF1212683 and ACI-1550493. Work by Heradio was supported by the Universidad Nacional de Educacion a Distancia (projects 2021V/PUNED/008 and 2022V/PUNED/007).

REFERENCES


ACM Transactions on Software Engineering and Methodology, Vol. 1, No. 1, Article 1. Publication date: January 2022.

ACM Transactions on Software Engineering and Methodology, Vol. 1, No. 1, Article 1. Publication date: January 2022.
Finding Near-Optimal Configurations in Colossal Spaces with Statistical Guarantees


A

$|\mathbb{C}| > 1000$: When an Infinite Space Can Approximate a Discrete Space

$|\mathbb{C}|$ is big enough when Eqns (4)-(6) are satisfied, i.e., when the mean $(\bar{c}_{1,n})$ and standard deviation $(\sigma_{1,n})$ of many samples converge to their theoretical counterparts, $c_{1,n}$ and $\sigma_{1,n}$. Eqns (4)-(6). Fig. 23 shows the result of simulating 1,000 samples, with $|\mathbb{C}|$ configurations each, for different values of

A

|\mathbb{C}| > 1000: WHEN AN INFINITE SPACE CAN APPROXIMATE A DISCRETE SPACE

$|\mathbb{C}|$ is big enough when Eqns (4)-(6) are satisfied, i.e., when the mean $(\bar{c}_{1,n})$ and standard deviation $(\sigma_{1,n})$ of many samples converge to their theoretical counterparts, $c_{1,n}$ and $\sigma_{1,n}$. Eqns (4)-(6). Fig. 23 shows the result of simulating 1,000 samples, with $|\mathbb{C}|$ configurations each, for different values of

A

$|\mathbb{C}| > 1000$: WHEN AN INFINITE SPACE CAN APPROXIMATE A DISCRETE SPACE

$|\mathbb{C}|$ is big enough when Eqns (4)-(6) are satisfied, i.e., when the mean $(\bar{c}_{1,n})$ and standard deviation $(\sigma_{1,n})$ of many samples converge to their theoretical counterparts, $c_{1,n}$ and $\sigma_{1,n}$. Eqns (4)-(6). Fig. 23 shows the result of simulating 1,000 samples, with $|\mathbb{C}|$ configurations each, for different values of
For each $|\mathbb{C}|$, there is one point representing the mean, $\mu_{1,n}$, in Fig. 23a, and one point for the standard deviation, $\sigma_{1,n}$, in Fig. 23b. Red lines show the theoretical $\mu_{1,n}$ and $\sigma_{1,n}$ counterparts.

The vertical (blue) line of Fig. 23 shows the approximation works well for $|\mathbb{C}|=1024$ (a tiny SPL), i.e., for SPLs with 10 unconstrained optional features. A more conservative estimate was postulated $|\mathbb{C}|>2000$ in [91].

**B PROOF OF UNIFORMITY OF THE URS ALGORITHM**

Two kinds of probabilities need to be distinguished to prove the uniformity of Alg. 1:

1. The probability $P(c)$ that configuration $c$ is sampled; and
2. The probability $P(f)$ that feature $f$ belongs to a sampled configuration, i.e., $P(f) = \frac{|\phi \land f|}{|\phi|}$.

Uniformity means that every configuration has the same chance to be sampled. According to the probability definition, $\sum_{i=1}^{\omega} P(c_i) = 1$. Hence, uniformity is satisfied whenever $P(c) = \frac{1}{|\phi|}$ for any $c$.

Alg. 1 samples a configuration by incrementally assigning true or false to each of the $\omega$ features in a feature model. In Eqn (13) (next page), $a_i$ stands for the value assigned to feature $f_i$. Due to feature constraints, assignments depend on each other, and so feature values must be generated following the chain rule [124] to ensure the final configuration is valid, i.e., using feature conditional probabilities, Eqn (14). In each iteration $i$, the algorithm produces a random assignment $a_i$ by taking into account the probabilities of the previous assignments $a_1, a_2, \ldots, a_{i-1}$ (Eqn. 15). At the end, all features are assigned and $|\phi \land a_1 \land a_2 \land \ldots \land a_\omega| = 1$, since a complete feature assignment corresponds to a unique configuration. As a result, the probability of sampling the configuration is $P(c) = \frac{1}{|\phi|}$ (Eqn. 16), which guarantees the sampling procedure is uniform.
C STATISTICAL SIGNIFICANCE

Results of Sections 5 and 6 were analyzed to test their statistical significance. As usual in science, the confidence level was set to 95%.

A result is said to be statistically significant when it is unlikely to happen by chance. That is, Sections 5 and 6 answer RQ1–RQ4 by analyzing a sample of SPLs (BerkeleyDBC, 7z, VP9, axTLS, and ToyBox). However, we could have accidentally selected a very particular set of SPLs that does not reflect the characteristics of the whole population of SPLs. Statistical significance means rejecting that possibility, thus supporting the generality of our results.

C.1 RQ1 and RQ2

An ANOVA test is the standard way to check if the differences among each algorithm’s c_{no}S in Section 5 were statistically significant [38]. However, our experiments violated ANOVA preconditions:

- c_{no}S for each algorithm were not normally distributed. Table 8 summarizes Shapiro-Wilk tests [99] conducted per algorithm; as all p-values were ≤0.05, normality was rejected.
- The variance of c_{no}S returned by each algorithm was highly different. In particular, the Levene test [74] for variance homogeneity produced F=191.5 and p-value ~0. As p-value ≤0.05, variance homogeneity was rejected.

The Kruskal-Wallis test [70] was used as the non-parametric alternative to ANOVA. It raised H=3.050 and p-value ~0. As p-value ≤0.05, the test concluded that at least one of the algorithms achieved c_{no}S significantly different from at least one of the other algorithms.

To determine precisely for which algorithms the c_{no}S differ, all pairwise comparisons in Table 9 were performed following the method described in [104]. First, all c_{no}S were ranked (i.e., the smallest c_{no} scored a rank of 1, the second smallest one a rank of 2, and so on). Then, the mean of the c_{no} ranks was computed for each algorithm. The absolute value of the difference between the means of every pair of algorithms was calculated. These absolute values, called observed differences, were compared to thresholds, named critical differences and calculated from the number of experiments carried out per algorithm and the confidence level.

According to [104], observed differences should be considered statistically significant whenever they are ≥ than their corresponding critical differences. Therefore, all observed differences were statistically significant except when comparing DeepPerf to SPLConqueror S1, and SPLConqueror S2 to SPLConqueror S4.
To summarize:

- The Kruskal-Wallis and multiple comparison tests support the statistical significance of RQ1 (Section 5.2).
- Levene test supports the statistical significance of RQ2 (Section 5.3).

### C.2 RQ3

Table 10 summarizes the significance of the correlations between MAPE and β reported in Table 6 (Section 5.4). As all p-values ≤ 0.05, all correlation measures were statistically significant.

### C.3 RQ4

Analogous to Appendix C.1, a t-test would be the standard way [38] to check the significance of the SRS and RRS difference reported in Section 6; however, the experimental data violated t-test preconditions:

- The build sizes of the configurations obtained with SRS and RRS were not normally distributed. Table 11 summarizes Shapiro-Wilk tests [99] conducted per algorithm; as all p-values were ≤ 0.05, normality was rejected.
- The build size variance for each algorithm was heterogeneous. The Levene test [74] produced F = 28.453 and p-value = 1.46 × 10⁻⁷. As p-value ≤ 0.05, variance homogeneity was rejected.

### Table 9. Multiple comparison test.

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Observed difference</th>
<th>Critical difference</th>
<th>Statistically significant?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRS vs. RRS</td>
<td>1938.758</td>
<td>293.0368</td>
<td>yes</td>
</tr>
<tr>
<td>SRS vs. DeepPerf</td>
<td>2398.176</td>
<td>306.424</td>
<td>yes</td>
</tr>
<tr>
<td>SRS vs. SPLCon. S1</td>
<td>2994.880</td>
<td>350.495</td>
<td>yes</td>
</tr>
<tr>
<td>SRS vs. SPLCon. S2</td>
<td>1491.906</td>
<td>350.495</td>
<td>yes</td>
</tr>
<tr>
<td>SRS vs. SPLCon. S3</td>
<td>522.209</td>
<td>350.495</td>
<td>yes</td>
</tr>
<tr>
<td>SRS vs. SPLCon. S4</td>
<td>1230.784</td>
<td>350.495</td>
<td>yes</td>
</tr>
<tr>
<td>SRS vs. SPLCon. S5</td>
<td>811.818</td>
<td>350.495</td>
<td>yes</td>
</tr>
<tr>
<td>RRS vs. DeepPerf</td>
<td>4336.934</td>
<td>293.037</td>
<td>yes</td>
</tr>
<tr>
<td>RRS vs. SPLCon. S1</td>
<td>4033.646</td>
<td>338.854</td>
<td>yes</td>
</tr>
<tr>
<td>RRS vs. SPLCon. S2</td>
<td>3430.664</td>
<td>338.854</td>
<td>yes</td>
</tr>
<tr>
<td>RRS vs. SPLCon. S3</td>
<td>2460.968</td>
<td>338.854</td>
<td>yes</td>
</tr>
<tr>
<td>RRS vs. SPLCon. S4</td>
<td>1126.934</td>
<td>338.854</td>
<td>yes</td>
</tr>
<tr>
<td>RRS vs. SPLCon. S5</td>
<td>2042.603</td>
<td>338.854</td>
<td>yes</td>
</tr>
<tr>
<td>DeepPerf vs. SPLCon. S1</td>
<td>896.270</td>
<td>350.495</td>
<td>yes</td>
</tr>
<tr>
<td>DeepPerf vs. SPLCon. S2</td>
<td>1875.966</td>
<td>350.495</td>
<td>yes</td>
</tr>
<tr>
<td>DeepPerf vs. SPLCon. S3</td>
<td>1167.391</td>
<td>350.495</td>
<td>yes</td>
</tr>
<tr>
<td>DeepPerf vs. SPLCon. S4</td>
<td>261.122</td>
<td>350.495</td>
<td>yes</td>
</tr>
<tr>
<td>DeepPerf vs. SPLCon. S5</td>
<td>1491.906</td>
<td>350.495</td>
<td>yes</td>
</tr>
</tbody>
</table>
The Mann-Whitney U-test [38], also known as Wilcoxon signed-rank test, was used as the non-parametric alternative to t-test. It raised W = 46043 and p-value = 0.741. As p-value ≤ 0.05, the test concluded that SRS and RRS difference was statistically significant.

### Table 10. Significance tests of MAPE and β correlation.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeepPerf</td>
<td>2.988 · 10⁻⁶</td>
</tr>
<tr>
<td>SPLCon. S1</td>
<td>8.591 · 10⁻¹¹</td>
</tr>
<tr>
<td>SPLCon. S2</td>
<td>~ 0</td>
</tr>
<tr>
<td>SPLCon. S3</td>
<td>~ 0</td>
</tr>
<tr>
<td>SPLCon. S4</td>
<td>~ 0</td>
</tr>
<tr>
<td>SPLCon. S5</td>
<td>~ 0</td>
</tr>
</tbody>
</table>

### Table 11. Shapiro-Wilk’s normality tests for t-test.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>W</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRS</td>
<td>0.829</td>
<td>~ 0</td>
</tr>
<tr>
<td>RRS</td>
<td>0.741</td>
<td>~ 0</td>
</tr>
</tbody>
</table>

### D PROPOSITIONAL FORMULA φ TO CNF CONVERSION

SAT and #SAT solvers require a Conjunctive Normal Form (CNF) formula as input [19, 120]. Transforming φ into a CNF formula φ^cnf is straightforward with rules of logical equivalence. But doing so may increase the number of clauses exponentially [120] and simplifying φ^cnf to reduce the number of clauses is nontrivial [63, 86].

To avoid this, Equisatisfiable Transformations (ETs) are used. Two formulas are equisatisfiable when one formula is satisfiable only if the other is satisfiable, and vice versa [125]. ETs produce
a CNF formula $\phi^{cnf}$ that is equisatisfiable to $\phi$ [113]. There are many ETs [58, 96, 113] not all of which are suitable for URS.

Consider: $\phi = (a \land b) \lor (c \land d)$. An ET from Plaisted and Greenbaum [96] introduces additional variables $x_1$ and $x_2$ for the clauses of $\phi$:

$$\phi^{cnf} = (x_1 \lor x_2) \land (\neg x_1 \lor a) \land (\neg x_1 \lor b) \land (\neg x_2 \lor c) \land (\neg x_2 \lor d)$$

Each row of Table 12 is a solution of both $\phi$ and $\phi^{cnf}$. The last solution of $\phi$ corresponds to 3 solutions of $\phi^{cnf}$.

A problem for URS is exposed: using $\phi^{cnf}$ yields a biased sampling of $\phi$. Statistical predictions by URS of $\phi^{cnf}$ are distorted predictions about $\phi$:

- $|\phi^{cnf}|$ is 9 and $|\phi|$ is 7, a 28% over-estimation; and
- The percentage of products with feature $d$ in $\phi^{cnf}$ is $\frac{78}{9}$, whereas the correct answer in $\phi$ is $\frac{71}{7}$, a 10% over-estimation.

How do redundant solutions arise? We observed empirically that if $\phi^{cnf}$ adds no new variables to $\phi$ then all is OK: URS statistics about $\phi^{cnf}$ match $\phi$ because $|\phi^{cnf}| = |\phi|$.

Adding variables might not be a problem. Tseitin’s transformation [113], a well-known ET method, adds variables but does not increase the number of solutions. Tseitin’s transformation extends the Plaisted and Greenbaum transformation with blocked clauses [71]. The elimination of blocked clauses [61], which is a SAT preprocessing technique used in top-tier solvers, removes those clauses and introduces redundant solutions.

The example of Table 12 shows there are bad ETs that both add variables and distort statistical predictions. The pragmatic problem is this: Given a feature-model-to-propositional-formula tool, you may not know if the tool (nor the #SAT solver that uses the propositional formula) employs bad ETs if extra variables are used.

We used the Kmmax tool [39] in our work which avoids translation controversies as it adds no extra variables in translating $\phi$ to $\phi^{cnf}$.

Also, Projected Model Counters (#∃SAT) [11] can be used as an alternative to classical #SAT solvers to prevent miscounting. #∃SAT counts the solutions of $\phi^{cnf}$ with respect to an input set of relevant variables, called projection variables. If all variables in $\phi$ are specified as the projection variables, #∃SAT will ignore any other auxiliary variables in $\phi^{cnf}$, thus computing the right count. Further, sampling with BDDs avoids ET problems as BDDs don’t require the input formula to be in any particular form. This is an advantage of using BDDs.