Multi-Objective Optimization in Large Software Product Lines

JEHO OH, DON BATORY, University of Texas at Austin, USA
PAUL GAZZILLO, Stevens Institute of Technology

Finding products that balance multiple competing objectives is a fundamental and difficult optimization problem. Software product lines compound its difficulty as their product spaces are gigantic, sometimes exceeding $10^{60}$ distinct configurations.

We present new algorithms for (a) true random selection of configurations in gigantic product spaces, (b) discovering the shape of their Pareto fronts, (c) exploring regions of interest on these fronts with greater efficiency than offered by existing algorithms, and (d) using real and previously used systems in our experiments as supporting evidence.

CCS Concepts: • Software and its engineering → Software configuration management and version control systems; Search-based software engineering.

Additional Key Words and Phrases: software product lines, searching configuration spaces, multi-objective optimization

ACM Reference Format:

1 INTRODUCTION

A Software Product Line (SPL) is a family of related software products [2]. Each product represents a unique combination of features, where a feature is an increment in product functionality. If an SPL has n optional features, the number of distinct SPL products can be $O(2^n)$. Industrial SPLs, such as different versions of Linux, have thousands of features, giving rise to gigantic product spaces in excess of $10^{60}$ configurations [22, 34].

Finding near-optimal products in a large product space is hard [39]. Enumeration is infeasible, so heuristics are used. There are two possibilities. First, an SPL can have a performance model – meaning that the performance of any product can be estimated accurately, so some form of guided search, like evolutionary algorithms [12, 14, 15, 17, 29], is used to find near-optimal products. Performance models permit huge numbers of samples of a product space to be evaluated with little cost.

Second, an SPL may not have a performance model or one that is accurate enough for optimization, so selected products must be built and then benchmarked in a search for near-optimality [6, 26]. Minimizing the number of samples taken becomes paramount. Another common limitation of existing research is that product samples are not truly randomly selected and statistical guarantees on the quality of the recommended product are largely absent.

An important variant of finding near-optimal products is multi-objective optimization (MOO) or Pareto optimization where multiple competing metrics are optimized simultaneously [19]. In avionics, for example, maximizing aircraft
range, passenger volume, and cruise speed conflicts with minimizing fuel consumption and lifecycle costs [11]. Balancing trade-offs are tasks that avionics engineers do with care.

We explore MOO in SPLs with large product spaces in this paper. We show how, using true random sampling, an entire Pareto front can be revealed and how engineer-chosen regions of Pareto fronts can be explored – i.e. segments of a front that are of prime interest to an engineer, not wasting time exploring irrelevant segments. We demonstrate the practicality of our work with experiments using kConf iG-based systems with gigantic product spaces that require product benchmarking to extract performance metrics. We also compare our work with multi-objective optimization algorithms that use performance models to predict performance metrics, using our sampling algorithms as a search engine.

The novel contributions of our paper are:

- True random sampling of gigantic configuration spaces,
- Algorithm SX that finds Pareto near-optimals for an entire configuration space,
- Algorithm SR that finds Pareto near-optimals for a given subspace, and
- Experiments with real systems with gigantic configuration spaces ($>10^{68}$) that show SX and SR match or exceed existing algorithms in accuracy using fewer samples.

Our research generalizes Oh et al. [26] that showed how single objective optimization of SPL product spaces ($\leq 10^{12}$) could be searched efficiently by random sampling.

2 BACKGROUND

2.1 Limitations in Current Work

Although MOO involves two or more objective functions to be optimized simultaneously, sometimes it is mapped to a single function $S(x)$ to be optimized by weighting each distinct objective function $O_i(x)$ with a coefficient $w_i$ and summing:

$$S(x) = \sum w_i \cdot O_i(x)$$

We do not consider such models in this paper, as single-objective optimization is addressed elsewhere [12, 14, 15, 25, 26, 28–30, 40].

State-of-the-art approaches for MOO on Highly Configurable Systems (HCS) have used Multi-Objective Evolutionary Algorithms (MOEA) and Bayesian Optimization (BO) techniques [17, 20, 29, 42].

MOEA evolves configurations to find those with near-optimal performance. An initial set of “randomly” sampled configurations is iteratively evolved until objectives are satisfied or a time or sample limit has been reached. At each evolutionary step, a predefined number of better performing configurations among available samples go through the following transformations and replace worse performing configurations:

- **mutation**: randomly change the features of a configuration,
- **crossover**: arbitrarily combine feature selections of different configurations, and
- **selection**: replace poor-performing configurations with better-performing configurations from a) and b).

IBEA and NSGA-II are two state-of-art MOEA algorithms used for MOO in HCS [12, 14, 15, 17, 29]. MOEA has known drawbacks:

1. It is expensive as crossover and mutation creates huge numbers of invalid configurations.

1[12, 15] use a fitness function, which can be seen as a single weighted function. This function determines which configurations are better at each evolutionary step.

Manuscript submitted to ACM
(2) It relies on a performance model which MOEA uses to estimate the performance of valid and invalid configurations. (Of course, performance of invalid configurations cannot be measured or verified. Using models to estimate the performance of invalid configurations is part of the MOEA’s search strategy).

(3) MOEA results are sensitive to how mutation, crossover, and selection are done, which makes it nontrivial to be ‘tuned’ for different search spaces [27].

BO derives a performance model that predicts a Pareto front. A surrogate model is used to predict both the performance of any configuration and its prediction uncertainty. Based on an initial surrogate model constructed from sampled configurations, all configurations are evaluated for their performance and uncertainty. Then, configurations that are both likely to be on a Pareto front and most uncertain are benchmarked for their actual performance to revise the surrogate model. This process continues until the current estimated Pareto front is “accurate enough”. State-of-the-art approaches on BO for HCS use a Gaussian process model as the surrogate model [20, 42]. BO also has known drawbacks:

(1) It is not scalable. BO enumerates all configurations and estimates their performance using a surrogate model. Enumeration is infeasible for large HCS spaces. It is unclear how to reduce the size of HCS spaces to make BO usable.

(2) As with MOEAs, the accuracy of a Gaussian process model depends on proper setting of parameters, which is nontrivial.

Chen et al. [6] proposed SWAY as an alternative to MOEA and BO. SWAY starts with a large set of valid configurations. It randomly selects a configuration $r$ and partitions the set into two disjoint subsets based on features they have in common with $r$. From each partition, configurations $r_1$ and $r_2$ are chosen to have the most different feature decisions with respect to $r$. Both $r_1$, $r_2$ are then benchmarked, and the partition with the best performing $r_1$ is selected for recursive partitioning; the other is discarded. This continues until the remaining partition is smaller than a predefined size. This reduction in partition size reduces the number of configurations to evaluate.

To recap, current approaches have following limitations:

- It is unclear that the starting and ending set of samples are purely random; configurations are indeed sampled from the space, but not much more can be (or has been) said.
- MOEA and BO require a performance model. It was observed in [26] that creating performance models to find optimal configurations is neither as efficient nor as accurate as directly sampling a configuration space.
- Their goal is to find as many Pareto near-optimals as widely dispersed as possible, which we also adopt. However, no explicit effort is made in prior work to find extrema points on a front (Fig. 1a). Nor can they target a specific region of a front which is of prime interest to engineers (Fig. 1b). Both are common use-cases of MOO [19].

---

Fig. 1. Extrema and Restricted Pareto Fronts.
Our work addresses these limitations:

- We use true random sampling of valid configurations, which is scalable up to the limit of using a #SAT solver [35] on a single PC. This is inherently different from MOEAs, as only valid configurations are used for reasoning.
- True random sampling enables mathematical characterizations of the quality of our answers.
- We do not use performance models, but rather focus on the selection of features to navigate a configuration space.

We statistically infer the influence of each feature to shift sampling toward a certain region/extreme. This enables us to guide or focus a search without enumeration.

2.2 Recap of Near-Optimal Configurations [26]

Every SPL has a feature model (FM) that defines all legal combinations of features [2]. Each legal combination is a configuration. Let \( C \), called the product space, be the set of all configurations for FM.

Every classical FM can be mapped to a propositional formula \( \phi \) whose variables are SPL features and whose solutions are configurations [2].\(^2\) Let \(|C|\) be the size of \( C \).

**Randomly Selecting Configurations.** A BDD can encode \( \phi \) and count the number of its solutions. Uniform random sampling of \( C \) randomly selects a number \( c \in \{1 \ldots |C|\} \), and uses the (binary) number \( c \) to traverse the BDD via a binary-search to find the \( c^{\text{th}} \) configuration in \( C \).

[26] assumed that the SPL has no reliable cost function. For each sampled configuration \( c \), \( c \)'s SPL program is built and benchmarked. Let \( \$ (c) \) be its measured performance. Doing this for \( n \) distinct samples forms a set of ordered pairs \((c_i, \$ (c_i))\) for \( 1 \leq i \leq n \). Call this a sample set of size \( n \).

**PCS Graphs.** Suppose a sample set has all configurations in \( C \).\(^3\) By sorting all pairs by their \( \$ \)-coordinate from the best-performing configuration to the worst-performing, a Performance Configuration Space (PCS) graph is produced: the \( X \)-axis lists configurations and the \( Y \)-axis their benchmarked values. For all but the smallest configuration spaces (\(|C|<4000\)), the \( X \)-axis can be approximated by the unit interval \( I = [0,1] \) so that a PCS function is \( \$ : I \rightarrow R \), where \( R \) is the set of reals.

Although we know a configuration \( c \) and its performance \( \$ (c) \), we do not know where \( c \) is positioned along \( I \) in the \( X \)-axis of a PCS graph. All that we know is:

(i) The \( C \rightarrow I \) mapping is injective – no two configurations map to the same \( x \in I \), and

(ii) Randomly selecting a configuration \( c \in \{1 \ldots |C|\} \) maps to a random point \( x \in I \) on the \( X \)-axis of a PCS graph.

**Mathematics of Random Selection.** Suppose \( n \) points are uniformly selected from \( I \). Let \( c_{\text{best}} \) be the best performing point among the \( n \), which in a PCS graph means \( c_{\text{best}} \)'s \( X \)-axis value is the closest to \( 0 \). The probability density function for \( c_{\text{best}} \) is:

\[
\rho_n(x) = n \cdot (1 - x)^{n-1}
\]

\( n \) is the normalization term and \((1-x)^{n-1}\) is the probability that \( n-1 \) selections are greater than \( x \). The mean of \( \rho_n \) is:

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

Eq. (2) tells us a simple way to find a near-optimal configuration in \( C \): take \( n \) samples and return \( c_{\text{best}} \). This is Non-Recursive Searching (NRS). On average, \( c_{\text{best}} \) will be \( \frac{1}{n+1} \) from \( x=0 \) on the \( X \)-axis. So if \( n=9 \), \( c_{\text{best}} \) is \( \frac{1}{10} = 10\% \) from \( x=0 \); if \( n=99 \), \( c_{\text{best}} \) is \( \frac{1}{100} = 1\% \) from \( x=0 \), **no matter how big** \(|C|\) is. Supporting experimental evidence for this claim is

---

\(^2\)Non-classical feature models have numerical and textual constraints [24]. We deal with classical feature models in this paper [2].

\(^3\)Of course, this is possible only with unlimited time for large configuration spaces.
Multi-Objective Optimization in Large Software Product Lines

presented in Sec. 5.2. For later reference, the standard deviation $\sigma$ for $p_n$ is:

$$\sigma = \sqrt{\frac{2}{(n + 1)(n + 2)} - \left(\frac{1}{n + 1}\right)^2} \quad (3)$$

Lastly, the number of samples taken can be reduced. Recall the best configuration $c_{\text{best}}$ out of 9 random samples will be 10% from $x=0$. Taking 99 random samples (or 11× the previous number) closes the gap to 1%. But 90% of the additional 90 samples will not perform better than $c_{\text{best}}$. A better approach is to use the first 9 samples to identify the best 10% of $C$, and another 9 samples to reduce this smaller space by another 10% to the best 1% of $C$ for a total cost of 18 samples. This is Statistical Recursive Searching (SRS).

A sample set of size $s$ is used on each recursion. We can compute $s(F)$, the average performance of a configuration in this sample set that has feature $F$, and $s(\neg F)$, the average performance of a configuration in this sample set that does not have $F$. A feature $F$ or its negation is noteworthy if it increases a configuration’s performance with 95% confidence. Noteworthy features are added to a configuration on the next recursion. Recursion stops when no noteworthy features are found.

Results. Fig. 2a shows the PCS graph of BerkeleyDBC with 2560 configurations [31]. With 30 experiments and $n=50$ samples per experiment, we used the ideas of this section to compute the near-optimal configurations returned by NRS and SRS. Fig. 2b provides a close-up view showing the top 5% best-performing configurations.

3 MULTI-OBJECTIVE CONFIGURATION SPACES

3.1 Random Sampling with #SAT Tools

A BDD [1] encoded the propositional formula of a feature model as a hierarchical graph in [26]. A bottom-up traversal of the BDD counted the number of valid configurations, while a top-down traversal matched a valid configuration with a given random number. BDDs have well-known scalability issues [1]. Of the 15 Linux feature models studied in [3], only 2 could be analyzed by BDDs.

#SAT can be used as an alternative to BDD. #SAT extends SAT algorithms by associating the number of solutions to each truth assignment [5]. Among available #SAT tools, sharpSAT [35] is the state-of-the-art, widely used, and provides an efficient way to cache sub-formulas being counted.

Although #SAT does not generate a traversable graph like a BDD, it is possible to partition a space by selecting features in a binary-search way. Algorithm 1 iteratively adds a truth assignment to a variable as a constraint and counts the number of solutions of the revised formula. To improve efficiency, it is possible to assign multiple variables per iteration and to retain partitioning information across multiple calls. As an optimization, when all remaining variables are free, those variables can be assigned without further partitioning. Constraints can be also imposed by the user or search, by adding the constraints to the formula before counting.
Algorithm 1: Sampling Algorithm

1 Procedure Sample(r, FM):
   \[\text{Input : } r \text{ (random number from } [1, \text{ total # of configs}]) \]
   \[FM \text{ (feature model propositional formula)} \]
   \[\text{Output: config (sampled config. (set of features))} \]
   \[\text{while variables remain unassigned do} \]
   \[v \leftarrow \text{variable that not yet assigned}; \]
   \[nvC \leftarrow \text{number of solutions for } FM \land \neg v; \]
   \[\text{if } (nvC = 2^k \text{ of unassigned variables}) \text{ then} \]
   \[\text{for (each } f \text{ in unassigned variables) do} \]
   \[i \leftarrow r \mod 2; r \leftarrow r / 2; \]
   \[\text{if } (i = 0) \text{ then} \]
   \[\text{add } \neg f \text{ to config;} \]
   \[\text{else} \]
   \[\text{add } f \text{ to config;} \]
   \[\text{else} \]
   \[\text{if } (r \leq nvC) \text{ then} \]
   \[\text{add } \neg v \text{ to config;} \]
   \[FM \leftarrow FM \land \neg v; \]
   \[\text{else} \]
   \[\text{add } v \text{ to config;} \]
   \[FM \leftarrow FM \land v; \]
   \[r \leftarrow r - nvC; \]
   \[\text{return } \text{config}; \]

3.2 Pareto Graphs and PCS Graphs

A PCS graph plots one performance metric for each configuration. A Pareto graph plots multiple metrics [19].

Fig. 3 shows the classical outline of a Pareto graph: The X-axis denotes a normalized performance, where \(x=0\) is the best for metric \(X\) and \(x=1\) is the worst. The same for the Y-axis, except \(Y\) represents a different and competing metric. Each point on a Pareto graph is a \((x_p, y_p)\) performance pair for some configuration \(p \in C\). A configuration \(c\) dominates other configurations that are immediately above and/or to its right, because its \(x_c\) and \(y_c\) metrics are better than those it dominates. The regions of dominance for four points are shown in Fig. 3. The sequence of undominated points, or Pareto optimals, forms a Pareto front, the focus of MOO optimizations.

Fig. 4 shows how Pareto and PCS graphs are related. The upper-right quadrant is the Pareto graph of Fig. 3. The PCS graph for metric \(Y\) is the upper-left quadrant; the PCS graph for metric \(X\) is the bottom-right. Both PCS graphs impose unique orderings of configurations, where each configuration on the \(C_Y\) axis of PCS \(Y\) is also a configuration on the \(C_X\) axis of PCS \(X\), and vice versa.

We generalize the mathematics of Sec. 2.2 to sample the entirety of a PCS graph. The probability density function for the \(j^{th}\) largest value in \(n\) random selections from the unit interval \([0,1]\) is \(p_{j,n}(x)\):

\[p_{j,n}(x) = j \cdot \binom{n}{j} \cdot x^{j-1} \cdot (1 - x)^{n-j}\]

\(j \cdot \binom{n}{j}\) is for normalization, \(x^{j-1}\) is the probability that \(j-1\) values are less than \(x\), and \((1-x)^{n-j}\) is the probability that \(n-j\) values are more than \(x\). The mean \(\mu_{j,n}\) and
Multi-Objective Optimization in Large Software Product Lines

second moment $\mu_{j,n}$ of $p_{j,n}$ are:

$$\mu_{j,n} = \int_{0}^{1} x \cdot p_{j,n}(x) \cdot dx = \frac{j}{(1+n)} \quad (5)$$

$$\bar{\mu}_{j,n} = \int_{0}^{1} x^2 \cdot p_{j,n}(x) \cdot dx = \frac{j \cdot (j + 1)}{(n + 1) \cdot (n + 2)} \quad (6)$$

Eq. (5) says on average random selections in $I$ are evenly-spaced.

We expect variations from the mean $\mu_{j,n}$. Let an experiment choose $n$ random numbers from $I$, and let there be $k$ experiments. Let $S_{i,j}$ be the sample set for the $i^{th}$ experiment and $S_{1,j}$ be the $j^{th}$ largest value in $S_1$. The standard deviation $\bar{s}_{n,k}$ of the observed $j^{th}$ value and its expected value for all $j$ in all $k$ experiments is:

$$\bar{s}_{n,k} = \sqrt{\frac{1}{k \cdot n} \cdot \sum_{i=1}^{k} \sum_{j=1}^{n} (S_{i,j} - \mu_{j,n})^2} \quad (7)$$

Correspondingly, the theoretical standard deviation $\sigma_{j,n} = \sqrt{\bar{\mu}_{j,n} - \mu_{j,n}^2}$ of $p_{j,n}$ averaged over all $j$ is:

$$\sigma_n = \sqrt{\frac{1}{n} \sum_{j=1}^{n} \sigma_{j,n}^2} = \sqrt{\frac{1}{\delta \cdot (n + 1)}} \quad (8)$$

We experimentally verify theory Eq. (8) with observations Eq. (7) in Sec. 5.2.

4 FINDING PARETO NEAR-OPTIMALS

To see how Pareto near-optimals can be found by Random Sampling (RS), suppose a sampling event occurs with probability $\alpha$. On average, this happens after $1/\alpha = 10$ samples. To observe 4 such events, the average number of samples to take is $4 \cdot 10 = 40$.

Let $E(\alpha, j)$ be the expected number of samples to witness an event $j$ times when the event occurs with probability $\alpha$:

$$E(\alpha, j) = j/\alpha \quad (9)$$

The oval in Fig. 5 is the region in which configuration points appear. This region can have any shape within the unit square. All of its points have an equal probability of being selected.

We expect to find Pareto near-optimals in the shaded region, where $\alpha$ is the fraction of all points in the shaded region. In Fig. 5, if we wanted 3 points in the shaded region, we want $E(\alpha, 3)$. If $\alpha = 1/5$, we compute $E(1/5, 3) = 15$.

$E(\alpha, j)$ is the average number of random samples needed to select $j$ points in region $\alpha$. If $\alpha$ shrinks by a fraction $f$ (while retaining $j$ points in $f^{-1} \cdot \alpha$), we compute $E(f^{-1} \cdot \alpha, j)$. If we want $t$ times the number of points in $\alpha$, we compute $E(\alpha, t \cdot j)$. From Eq. (9):

$$E(f^{-1} \cdot \alpha, j) = f^{-1} \cdot E(\alpha, j)$$

$$E(\alpha, t \cdot j) = t \cdot E(\alpha, j) \quad (10)$$

To shrink $\alpha$ by half or to double the number of points in $\alpha$, we must double the number of samples, $E(\alpha/2, j) = E(\alpha, 2 \cdot j) = 2 \cdot E(\alpha, j)$. This is inefficient, as only a small fraction $(\alpha)$ of the additional samples are in the desired region, analogous to the situation that motivated statistical recursive searching in Sec. 2.2. In the next sections, we present recursive algorithms to:

- Set constraints on objectives and find Pareto near-optimals that satisfy those constraints (SR), and

---

4If configurations were uniformly distributed, $\alpha$ is the fraction of the oval that the shaded region occupies. How $\alpha$ is computed is not important; the conclusions that we draw from this analysis are important.
• Find as many and diverse Pareto near-optimals as possible (SX).

4.1 Finding Noteworthy Features

Central to our work is finding noteworthy features [26]. These are features that are certain to improve a designated objective. Using noteworthies as constraints in the next round sampling has two benefits: 1) the sample space is further restricted and 2) the resulting configurations, on average, have better a objective value.

Algorithm 2, GetNoteworthy, takes as input a sample set $S$ and an objective $\delta$ to improve (i.e. minimize), computes the average performances $\delta(F)$ and $\delta(\neg F)$ of configurations in $S$ with and without feature $F$. A feature is noteworthy if it improves performance with statistical certainty – passing Welch’s t-test [37] on the hypothesis that the average value of samples including the decision is better than not including it with 95% confidence.

Algorithm 2: GetNoteworthy

\begin{algorithm}
\caption{GetNoteworthy($S$, $\delta$)}
\begin{algorithmic}[1]
\Procedure{getNoteworthy}{$S$, $\delta$}\\
\Input{$S$ (an evaluated sample set)}\\
\Output{$\Delta N$ (new set of noteworthy features)}\\
\State $\Delta N = \emptyset$;
\State sort $S$ by $\delta$ so that $S[0]$ has the best value;
\State commons ← common feature decisions in $S[0]$ and $S[1]$;
\For{each feature $F$ in commons}
\State $\Delta(\delta) \leftarrow$ average $\delta$ difference between samples with and without $F$;
\If{$\Delta(\delta) < 0 \land \text{WelchTTest}(F)$}
\State add $F$ to $\Delta N$;
\EndIf
\EndFor
\State return $\Delta N$;
\EndProcedure
\end{algorithmic}
\end{algorithm}

4.2 SR: Finding Regions of Interest on Fronts

A common MOO use-case is trading-off different objectives, as no single configuration optimizes them all. This is the use-case that we call ‘finding regions of interest’.

We start with a sample set of size $s$ from the full population of configurations and plot its points on a Pareto graph showing which are currently Pareto optimal. We also mark the centroid (⋆) of these samples – the XY median of the $s$ points. The user then picks a goal (⊕), the centerpoint of a rectangular region, whose size is user-defined, where searching is to be focused (Fig. 6a). The objective is to move the centroid on each successive recursion, toward the true Pareto front (dark+thick line in Fig. 6) nearest the goal.

Let $(\Delta x, \Delta y)$ be the difference between ⋆ and ⊕ in Fig. 6. (In general, this is an $n$-tuple of $\Delta$s, one for each of the $n$ objectives or dimensions). The $\Delta$ with the greatest difference determines the objective in which the GetNoteworthy algorithm is invoked. Another sample set of size $s$ is taken from the noteworthy-restricted space, and the process recurses. For example, the first recursion transitions from Fig. 6a to Fig. 6b by optimizing $X$, moving the centroid ⋆ towards the Y-axis. The second recursion, Fig. 6c, optimizes $Y$, moving the centroid towards the X-axis. Recursion terminates when no new noteworthy features are found. See Algorithm 3 (SR).

Readers might ask “Can’t you go directly from ⋆ to ⊕ or as close as possible in one step?” Initially, that is what we thought we could do. In a nutshell, we do improve the targeted objective, but other objectives may be correlated. So

\footnote{\textit{takeSamplesAndEval}(s, $\mathcal{F}$, $N$) takes $s$ samples from the configuration space defined by predicate $\mathcal{F} \land N$, where $\mathcal{F}$ is the feature model and $N$ is the set of noteworthy features. Each sampled configuration is “evaluated” so that every objective is assigned a performance value. If the space is too small, fewer than $s$ samples may be returned.}
Algorithm 3: Search Region (SR)

1 Procedure searchRegion(\(F, D, s, G\)):

   Input : \(F\) (feature model)
   \(D\) (set of dimensions/objectives)
   \(s\) (sample set size per recursion)
   \(G\) (goal point to recurse toward)

   Output: \(P\) (found Pareto front configurations)

2 \(N \leftarrow \emptyset\);

3 repeat
4    samples ← takeSamples\((s, F, N)\); \(5\)
5       for each \(\delta\) in \(D\) do
6          \(\text{avg}_{\delta} \leftarrow \) average of dimension \(\delta\) from samples;
7          \(\text{std}_{\delta} \leftarrow \) standard deviation of \(\delta\) from samples;
8          \(\Delta_{\delta} \leftarrow |\text{avg}_{\delta} - G_{\delta}|/\text{std}_{\delta};\)
9          \(\Delta N \leftarrow \) getNoteworthy\((\text{samples}, \delta \text{ with largest } \Delta_{\delta})\);
10         add \(\Delta N\) to \(N\);
11         add samples to \(\text{samples}_{\text{total}}\);
12 until \(\Delta N = \emptyset \lor \text{samples} = \emptyset\);
13 \(P \leftarrow \) Pareto front configurations from \(\text{samples}_{\text{total}}\);
14 return \(P\);

changing one may change them all. Moreover, estimating how much the centroid will move is harder than knowing the direction it will move, especially with a small number of samples. Hence we recurse, selecting another \(s\) samples and repeating, shrinking the search space and converging on or near \(\oplus\).

4.3 SX: Finding Extremes on Pareto Fronts

Finding as many and diverse Pareto optimals as possible is another common MOO use-case. We begin with a sample set of size \(s\). We then separately optimize on each objective starting with this set. We take the union of the samples produced, derive the Pareto optimals from it, to yield our answer. Exploiting the distribution of points, especially correlation, in a search space can yield improved results. We also do this, as explained below.

Fig. 7 shows 2D spaces with different correlations; the dark + thick line represents the Pareto front. When correlation is positive, optimizing one objective also optimizes the other (Fig. 7a). In this case, optimizing on one objective may suffice to find the Pareto front. With little or negative correlation, optimizing one objective exposes only a segment of
the front (Fig. 7b-c). To get the remaining segment(s), optimization is performed on every other objective. Note that in such spaces, alternating objectives on consecutive recursions/steps (as does searchRegion) may not reach the extrema, as some feature decisions to improve an objective may hinder other objectives. See Algorithm 4 (SX).

Algorithm 4: Search Extremes (SX)

```plaintext
1 Procedure searchExtremes(F, D, s):
2     Input : F, D, s (same as searchRegion)
3     Output: P (found Pareto front configurations)
4     N ← ∅;
5     initSamples ← takeSamples(s, F, N); 5
6     for each pair ρ of dimensions in D do
7         corrρ ← correlation between pairs from initSamples;
8         if corrρ > 0.8 then
9             remove one objective in pair from D;
10            for each δ in D do
11                samples ← initSamples;
12                repeat
13                    ∆N← getNoteworthy(samples, δ);
14                    add ∆N to N;
15                    samples ← takeSamples(s, F, N); 5
16                    add samples to samplesTotal;
17                until ∆N = ∅ ∨ samples = ∅;
18     return P;
```

Table 1. Systems Used in Evaluation

| System Type | Name          | Domain                          | | | | Objectives to minimize | Objective evaluation |
|-------------|---------------|---------------------------------|---|---|------------------------|----------------------|
| Ground truth (GT) | BerkeleyDBC | Embedded database system | 18 | 2.5 × 10^8 | 3: # unselected features, response time, memory usage | Benchmarked all valid configurations |
|              | LLVM          | Compiler infrastructure         | 11 | 1.0 × 10^8 |                                |                      |
| Performance Model (PM) | Fiasco      | Linux microkernel                | 1638 | 3.6 × 10^14 | 4: # unselected features, # unused features, defect, cost | Artificial performance model sets values to add for each selected feature |
|              | uClinux       | Linux kernel for controllers     | 1850 | 1.6 × 10^8  |                                |                      |
|              | eshop         | B2C system                       | 330  | 5.0 × 10^14 |                                |                      |
| Benchmarked (BN) | axTLS        | Client/server library            | 93   | 5.1 × 10^8  | 2: # unselected features, build size | Benchmarked all sampled configurations |
|              | toybox        | Linux command line utility       | 295  | 1.0 × 10^5  |                                |                      |

5 EVALUATION

Three research questions evaluate our work:

RQ1: Do observations match theory?
RQ2: Is SX better for extremes?
RQ3: Is SR better for ranges?

Table 1 lists the systems that we use and their categories:

- **Ground Truth (GT)** systems have all valid configurations benchmarked [32]. GT configuration spaces are small, so it is possible to know their true Pareto optimals as well as the exact rank of any configuration. Every objective for GT systems was normalized by its smallest and largest benchmark measures.
- **Performance Model (PM)** are kConfig-based systems (a) used in prior work and (b) relied on artificial performance models instead of benchmarking. We used the performance models in [7] and [15] to compare with our results. Each objective (metric) was normalized with 0 as its minimum value and 1 as its maximum possible value. Feature models were dimacs files produced by the LVAT tool [3] available in [4].
- **Benchmarked (BN)** are kConfig-based systems that were randomly selected by sampling, built, and benchmarked. Feature models are dimacs files produced by the Kmax tool [10], a static analysis tool for kConfig-based systems. Each sampled configuration was converted into a .config file to build the targeted system. Objectives were normalized by deriving the minimum and maximum values observed from all experiments.

### 5.1 Experimental Set-Up Details

GT, PM, and BN systems require different ways to evaluate configurations because of their diversity. Consider PM systems. The SWAY [7] and SATIBEA [15] techniques both require a performance model. The fairest comparison required us to replicate their results:

- **SATIBEA** evolves an initial sample with a predefined rate for mutation and crossover. We replicated SATIBEA by using the source code from [16] with the same settings in [15]:
  - Sample size per evolution (including initial population): 300
  - Standard mutation rate: 0.001
  - Smart mutation rate: 0.98
  - Crossover rate: 0.05

- **SWAY** recursively reduces a large number of configurations until a predefined size is reached. We replicated SWAY by using the source code from [8] with the same settings in [7]:
  - Evaluations per partition: 10
  - Termination sample size: \( \sqrt{\text{initial sample size}} \)

We did not create performance models for GT and BN systems – as this is both difficult and beyond the scope of our work.

Wang et al. [36] guided our selection of metrics. For each experiment, we collected:

- **Number of samples (N)** is the total number of samples taken per experiment, a key indicator of required effort.
- **Hypervolume (HV)** is the area dominated by a Pareto front where all metrics are normalized. HV is a number where \( 0 \leq HV \leq 1 \) [41]. Higher HV is better because the identified front is closer to optimal and covers wider range of Pareto

---

6 LLVM did not have memory usage data for all configurations, hence this objective was not used.

7 We used the newest version of BN systems for benchmarking: v2.1.4 for axTLS and v0.7.5 for Toybox. Note that LVAT systems are older versions. We took them from the LVAT repository [4] and used them as is.

8 We set some configuration options for BN systems to ensure builds without errors. For axTLS, these were features related to building a Linux platform (PLATFORM_LINUX), problems with linking (CONFIG_HTTP_BUILD_LUA and CONFIG_HTTP_ENABLE_LUA), and broken builds (LANGUAGE_BINDINGS). For Toybox, they were pending features (PENDING...) that could break builds and that relied on a library (TOYBOX_SMACK). The table reflects the number of configurations after applying these constraints.

9 BO enumerates and evaluates all valid configurations, and was not used.
optimals. HV is the main metric to compare Pareto front estimates in the literature and was computed using the jMetal library [9].

- **Pareto Front Size (PFS)** is the number of Pareto near-optimals identified. The higher PFS the better, as a user has more configurations from which to choose.
- **Run Time (RT)** is the time spent running an experiment in seconds. This includes time for sampling and evaluating configurations, and reasoning over samples.

All experiments were performed on a Intel Core i7-6700 @ 3.40GHz Linux machine with 16GB of RAM. Experiments were repeated 20 times and the results averaged.

### 5.2 RQ1: Do observations match theory?

Fig. 8 plots the observed $\sigma_{n,k}$ and theoretical $\sigma_n$, Eqs (7, 8), for all metrics in $\Gamma$ systems with $k=100$ and different $N$. These graphs confirm a close agreement of theory and observations. Minor differences are due to tiny configuration spaces [26].

For RQ1, our theory matches observations independent of axis metrics (response time, memory usage, and number of unselected features), confirming that our samples are chosen from a uniform distribution.

### 5.3 RQ2: Is SX better for extremes?

RQ2 is about finding an entire Pareto front. We performed experiments to observe how HV, PFS, and RT change with increasing $N$. Here are some set-up details:

**SATIBEA.** We conducted experiments with different run times to gather results for different $N$. SATIBEA creates invalid configurations in its evolutionary steps. As invalids are also used for evolution, we counted them towards $N$. We did not consider invalids for deriving other metrics (HV, PFS), as their configurations do not exist.

Manuscript submitted to ACM
SWAY. We used different initial sample sizes to gather results for different $N$. $N$ is the total number of samples evaluated during an experiment, which includes evaluations used in partitioning its initially large sample set. HV and PFS were measured from the configurations in the last partition.

SX. Results for different $N$ were gathered by using different numbers of samples $s$ per recursion, namely $s \in \{5, 8, 10, 15, 20, 30, 50\}$. In general, taking more samples per recursion makes statistical inferences more reliable.

RS. Non-recursive sampling, or simply random sampling (RS), is plotted as a sanity check and as a minimal performance bound for all approaches.

Graphs. Fig. 10 is a matrix of graphs: rows are performance metrics (HV, PFS, RT) and columns are different systems grouped by category (GT, PM, BN). We consider each column-category in turn.

5.3.1 GT Systems

GT allows us to see how the results of RS and SX compare to the true Pareto front of a configuration space. Fig. 10a reveals SX to produce larger HV and PFS values than RS, indicating that SX finds better results with less effort – provided $s \geq 10$ was sufficient to demonstrate key distinctions in performance.

Fig. 9 plots all 1024 configurations of LLVM and the results of RS and SX for a single experiment with $N=180$. An HV difference of $.04$ separates the true Pareto front from that produced by SX, and an additional $.04$ separates the fronts of SX and RS. While the number ‘.04’ is unintuitive, Fig. 9 visually shows the improvement that SX exhibits over RS in finding better configurations.
SATIBEA, SATIBEA exhibited a clearly poorer or occasionally comparable HV performance to SX. SATIBEA was dominated by SX on PFS performance. SATIBEA produced no valid configurations for values of N smaller than plotted in Fig. 10b. Even when N<1000, it often yielded no valid configurations in 300 evolved samples.

SWAY. SWAY performed generally worse and less consistently on HV than SX. SWAY was comparable to SX on Fiasco, better than SX for eshop only when N<800, and performed poorly on uClinux. Further, as N increased, HV at best increased minimally for SWAY. In contrast SX improved consistently.

Further, SWAY was dominated by SX on PFS: PFS clearly increases as N increases for SX. Not always for SWAY: SWAY showed no increase of PFS with greater N for uClinux. From these experiments, it is unclear to us how the partitioning of SWAY improves its set of Pareto near-optimals or their numbers.

RT Performance. SATIBEA win hands-down over SX on RT performance; SWAY is roughly comparable to SX. The reason is that true random sampling has overhead – we need repeated calls to sharpSAT to convert random numbers into configurations. This takes time. SWAY uses a modified SAT-solver to find its configurations, so it too takes time. Neither our sampling approach nor SWAY’s has dominant performance. SATIBEA is unburdened using incorrect configurations.¹⁰

5.3.3 BN Systems

Fig. 10c presents the first results on axTLS and Toybox, which to our knowledge are systems whose objective values have not been explored previously. Fig. 10c complements our GT graphs (Fig. 10a) in that it reinforces our earlier

¹⁰In general, SX was as fast or faster than RS as recursive constriction of sample space made sharpSAT to run faster. One exception was uClinux, which had large number of free variables. Algorithm 1 will not call sharpSAT further if all variables are free. Adding constraints through recursion reduced the number of free variables, so that more sharpSAT calls were made per sample.
observations that SX outperforms RS. But axTLS and Toybox are quite different from GT systems in that they are 5 orders of magnitude larger for axTLS and 62 orders for Toybox.

We do not yet know how close we are to the true Pareto front of either system; our graphs are not yet approaching a limit (compare Fig. 10a with Fig. 10c), so additional experiments to extend these graphs to several thousand samples may be required. The longest experiment for Toybox collected 1.4K samples and took 2.5 hours to run (×20 for twenty experiments); the longest experiment for axTLS collected ~500 samples and took ~30 minutes to run (×20 for twenty experiments). Future papers can use this data to compare our results with newer approaches.

Finally, the RT of Fig. 10c looks different than Fig. 10a-b. The reason is that we included the time to build axTLS and Toybox systems. Although SX indeed has sampling overhead, it is a relatively small contributor to system build and benchmarking.

5.3.3 Recap of Results of RQ2 Experiments

Our conclusions for RQ2 are:
• SX converges faster and finds more Pareto near-optimals than RS on ground truth systems, even though SX and RS execution times are comparable.
• Using artificial performance models instead of benchmarking, SX performed better than SWAY and SATIBEA by using fewer samples and yielding more Pareto near-optimals.
• SX hyper-volume and Pareto front size metrics improved as N increased, and did so as fast or faster with more consistently than SWAY or SATIBEA. SX’s performance is more predictable.
• SX and SWAY have non-trivial overheads: the time to sample legal configurations using SAT technologies. SATIBEA, a multi-objective evolutionary algorithm, is indifferent to the legality of configurations and has the lowest overhead.

5.4 RQ3: Is SR better for ranges?

RQ3 is about finding Pareto near-optimals within a designated range or ‘region of interest’ along a Pareto front.

We selected a region that has a balance between objectives, which is nontrivial but likely to be meaningful for users. We collected all Pareto optimals POj from all RQ2 experiments of each system j and derived from POj:
The goal point $G$ was the point in $PO_j$ with the smallest normalized Euclidean distance to $M$. In effect, $G$ is an ideal Pareto optimal that minimizes all objectives. $G$ is at the center of the goal range $R$, which extended $\pm 10\%$ of $\Delta_{\ell}$ around $G$ in each $\ell$ direction.

To see how well SR searches within $R$, metrics \textit{HV} and PFS for different $N$ were measured, but were computed \textit{only from points from within} $R$.\textsuperscript{11} To get results for different $N$, we performed experiments on SR with a different number of samples per recursion, $s$, the same as we did with SX for RQ2. To compare with other approaches, we measured the same metrics as RQ2, again computing metrics only from points within $R$. Fig. 11 is a matrix of graphs for RQ3, as Fig. 10 was for RQ2.

### 5.4.1 GT Systems

SR dominates RS and SX on both \textit{HV} and PFS in Fig. 11a, indicating that SR can indeed focus on $R$ using a smaller number of samples. All algorithms have the same run-times.

Fig. 12 is the dual of Fig. 9. It shows the result of a SR experiment on LLVM. RS found one point and SX found no points in $R$. SR took less than half of the samples required for SR and RS, and found three Pareto near-optimals in $R$, including $G$. Note that SR does focus on a smaller range than other algorithms; no samples were collected near the extremes and were focused on the desired region.

### 5.4.2 PM Systems

We split SR’s experiments into two groups: uClinux+eshop, and then Fiasco. Fig. 11b shows the experimental results.

**uClinux+eshop.** SR was able to find more and better Pareto near-optimals within $R$ in less time than RS, SWAY, and SATIBEA. When the number of samples per recursion $s \geq 10$, SR showed a rapid increase of both \textit{HV} and PFS with increasing $N$ far beyond any other approach.

SATIBEA found configurations within range for eshop only, while SWAY was unable to find any configurations for both systems. This is consistent with Fig. 10b, where PFS values were small for both SATIBEA and SWAY across the entire configuration space.

**Fiasco.** A different story: although SR took fewer samples than SX,\textsuperscript{12} SWAY had the best \textit{HV}, and RS performed better than SR and SX, which is contrary to our other experimental results. The PFS of SR and SX both increased rapidly once they could reach $R$, with $s \geq 30$. However, both needed to take far more samples than they did to match the performance of SWAY and SATIBEA.

So what is going on? Fig. 13 shows a negatively-correlated space and the centroid (⋆) of an initial set of samples. SR must locate a region close to the centroid in Fig. 13a, whereas it must move the centroid far in Fig. 13b. Moving far means more recursions — but if the number of samples per recursion is too small, we discovered that SR may not make good decisions in selecting noteworthy features, causing it not to reach $R$.

\textsuperscript{11}Stated differently, \textit{HV} and PFS statistics would be greatly distorted if irrelevant points outside $R$ were included.

\textsuperscript{12}That SR took fewer samples than any other approach is indicated by the length of its (purple) line. Each dot in Fig. 11 represents the result for a particular value of $s \in \{5, 8, 10, 15, 20, 30, 50\}$

Manuscript submitted to ACM
A second factor is also involved. If a space is under-constrained, selecting noteworthy features is less likely to affect the selection of other features in future recursions. Thus, a configuration space constricts gradually (see Fig. 13c). In highly constrained spaces, selecting a noteworthy feature may lead to the inclusion or exclusion of larger number of other features. Noteworthy feature selections becomes a more subtle process, because making the wrong decision will indeed constrict the space (even faster because of constraints) to yield a subspace that does not include $\mathcal{R}$ (see Fig. 13d).
Among PM spaces, Fiasco is by far the most constrained, 2–3× more than that of uClinux and eshop. This makes Fiasco more difficult to search than other PM systems. Fig. 11b shows at least 100 samples are required per recursion to get to R. But once it is there, PFS shows that SR can find more configurations in range than other approaches.

SATIBEA and SWAY have no relation to these factors, as they do not consider where configurations may reside. Consequently, they may get good results as in Fiasco, but also get poor results as in uClinux and eshop. Such fine control is not a characteristic of these approaches.

5.4.3 BN Systems

Both BN systems showed results that were consistent with all non-BN systems except Fiasco, so that SR is best at searching over a region. This suggests that SR can be used for optimizing real large systems as well.

5.4.4 Recap of Results of RQ3 Experiments

We make the following conclusions for RQ3:

- SR indeed focuses on a given region R and finds more Pareto near-optimals faster and with less work than other algorithms.
- SR has weaknesses. It will perform better when G is close to the initial centroid and when the search space is less constrained. This leaves some room for SR to improve, which is a subject for future work.

5.5 Threats to Validity

Internal Validity. Ground-truth (GT) data and performance models for PM systems were created by others [6, 15, 31, 33]. We reused these artifacts to control the threat of comparing different approaches.

For BN systems, we verified that all selected configurations could be built. In doing so we discovered some features led to builds with errors (see footnote 8). Eliminating these few configurations should not impact our results.

We performed 20 experiments and averaged their results to control randomness. Certainly there are outliers, but the trends we reported were consistent across systems. Increasing the number of experiments would not have significantly changed the trends.

External Validity. We used 7 real-world systems with different domains, numbers of features, and configuration evaluation criteria. The objectives of systems had diverse correlations ranging from -0.9 to 0.9, indicating that the configuration spaces were not similar. We are aware that the performance SR and SX may not generalize to all systems, but identical trends from our evaluations across systems gives confidence that our conclusions should hold for other SPLs.

6 OTHER RELATED WORK

Sec. 2.1 covered the core papers; here are a few others.

Optimizing Highly Configurable Systems. Hierons et al. [18] set the validity of configurations as the highest objective to optimize, to yield as many valid configurations as possible. Henard et al. [15] extended IBEA with a SAT solver to generate random configurations and to minimize invalid configurations during evolution. Guo et al. [13] augmented IBEA with SMT solvers to deal with non-boolean features (e.g. optimizing configuration constants), a topic that we do not address. As an alternate to BO and MOEA, White et al. [38] proposed an approach based on linear programming, which solves a knapsack problem with constraints on two objectives.

Manuscript submitted to ACM
Sampling Configurable Systems. Efficient testing strategies for configurable systems is another area that relies on sampling. Liebig et al. [21] and Medeiros et al. [23] compared different sampling algorithms for fault detections, and concluded that randomly selecting features, not configurations, leads to large number of invalid configurations.

7 CONCLUSIONS

Multi-objective optimization of SPL products is both a fundamental and important problem. We have shown (1) how true random sampling, using #SAT solvers, can go a long way to create satisfactory solutions to sample gigantic configuration spaces, (2) using sensible strategies to explore the extremes of a space (algorithm SX) and (3) focusing on subranges of spaces (algorithm SR) can noticeably outperform today’s one-size-fits-all algorithms using the same ground truth data, performance models, and performance metrics by which prior researchers evaluated their algorithms.

Our results also suggest next steps for future research: (1) reducing the time to map random numbers to configurations in gigantic search spaces, (2) carefully studying the relationship of selecting noteworthy features when a configuration space is highly constrained, and (3) more experiments using real systems, because without them, we would not have imagined weaknesses in our algorithms and weaknesses in others.

This is a rich area for research: taking advantage of the distribution of points in a space (i.e. correlations) and using mathematics to guide the development of new search algorithms are keys to its future success.

Acknowledgements. We acknowledge and appreciate the support of NSF on grant CCF1421211.

REFERENCES

[15] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. 2015. Combining multi-objective search and constraint solving for configuring large software product lines. In ICSE.