ConEx: Efficient Exploration of Big-Data System Configurations for Better Performance

Rahul Krishna†, Chong Tang†, Kevin Sullivan, and Baishakhi Ray

Abstract—Configuration space complexity makes the big-data software systems hard to configure well. Consider Hadoop, with over nine hundred parameters, developers often just use the *default* configurations provided with Hadoop distributions. The opportunity costs in lost performance are significant. Popular learning-based approaches to auto-tune software does not scale well for big-data systems because of the high cost of collecting training data. We present a new method based on a combination of *Evolutionary Markov Chain Monte Carlo (EMCMC)* sampling and cost reduction techniques to find better-performing configurations for big data systems. For cost reduction, we developed and experimentally tested and validated two approaches: using scaled-up big data jobs as proxies for the objective function for larger jobs and using a dynamic job similarity measure to infer that results obtained for one kind of big data problem will work well for similar problems. Our experimental results suggest that our approach promises to improve the performance of big data systems significantly and that it outperforms competing approaches based on random sampling, basic genetic algorithms (GA), and predictive model learning. Our experimental results support the conclusion that our approach strongly demonstrates the potential to improve the performance of big data systems significantly and frugally.

Index Terms—Performance Optimization, MCMC, SBSE, Machine Learning.

1 Introduction

The use of Big-data frameworks such as Hadoop and Spark has become a de-facto standard for developing large scale data-driven applications. These frameworks are highly configurable and can be tailored to meet a diverse set of needs. In practice, however, it is hard to fully exploit such configurability. Instead, off-the-shelf, or *default*, configurations are most commonly used [1]. This often leaves significant performance potential unrealized [2]. Configuring for performance is important especially for big data because "even a small performance improvement translates into significant cost savings due to the scale of computations involved [3]".

Finding high-performing (or *good*) configurations for big data systems is hard. Their configuration spaces are vast and their configuration-to-performance functions are complex. First of all, they have multiple configurable subsystems [4]. Hadoop, for example, has about 900 parameters across 4 sub-systems. Some parameters, e.g., numeric ones, have many values. Secondly, parameters also have diverse types, including optional and dependent substructures. For example, setting one Boolean parameter to *true* can enable a feature, requiring values for all of its parameters. Further, parameters can also be constrained by external values. For example, in a multi-core system, one cannot set the *number-of-core* value to a number larger than the number of available cores. Also, due to its discrete nature, typical mathematical optimization methods do not apply [5]. Finding good configurations ends up as a black-art [6].

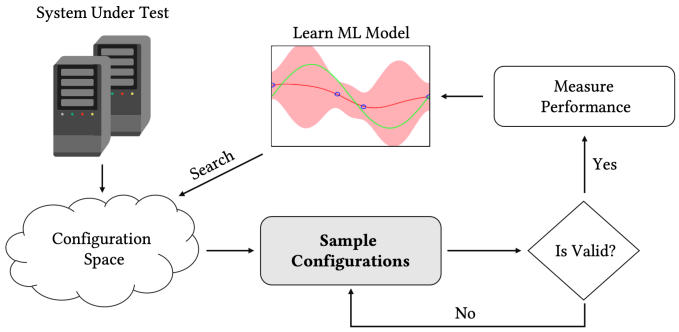


Figure 1. A typical framework for machine learning based automatic configuration of software sytems.

For traditional software, the problem of finding better configuration has given rise to a significant body of work [7], [8], [9], [10], [11], [12]. This research share a common general framework shown in Figure 1. They involve the following steps: (i) deploy different sampling strategies to select a set of valid and representative configurations [13], [14], [15], [16], [17], [18], (ii) use the sampled configurations to measure the performance of the system, (iii) use Machine Learning (ML) models to learn the mapping between configuration and performance [7], [8], [12], [19], [20], and finally, (iv) use the learned models to find better configurations. Most notably, **the success of these learning-based approaches is contingent on the size and the quality of the sampled configurations used for training**.

For big-data systems, existing techniques often struggle to scale due to two reasons. First, the cost of collecting training data in big data systems is prohibitively large [21]. For example, in a typical Hadoop system, a single run of a

- †*Krishna, R., and Tang, C., are joint first authors.*
- *Tang, C. is with Walmart Labs, Mountain View, CA USA. E-mail: ct4ew@virginia.edu*
- *Sullivan, K. is with the Department of Computer Science, University of Virginia, Charlottesville, VA USA. E-mail: sullivan@virginia.edu*
- *Ray, B. and Krishna, R. are with the Department of Computer Science, Columbia University, New York City, NY USA. E-mail: rayb@cs.columbia.edu and i.m.ralk@gmail.com*

simple sorting task for a typical workload (data size=3.2GB) can take up to 14 minutes. Typical learning-based approaches need about 2000 samples [22]. For the sorting task in Hadoop, this would take 19 days to obtain! Therefore, we seek multiple orders of magnitude cost reduction for such a method to be practical. Second, The configuration space of big data systems is complex. The configuration-finding problem for big data systems involves a significantly larger number of dimensions than addressed in most previous work. Nair *et al.* [19] showed that for complex, configurable systems the measurement data often do not generalize well making it almost impossible to find the "best" configuration.

To address this issue, Nair *et al.* [19] proposed a rank-based learning approach. Instead of focusing on constructing an accurate model, they use a random sampling strategy to build a "bad" model that can learn to predict whether one given configuration is better than another. They showed such predictors can be trained easily with significantly fewer samples. For big-data systems, as we will show in §6, it is hard to learn even such rank-preserving models with high accuracy. One might try Neural Networks (NNs) [23]. However, NNs require large amounts of high-quality samples data for training, and the cost of even collecting such data for big-data systems would be prohibitively high [24].

Given the number of samples needed to train a good model and the cost involved in collecting them for big-data systems, we cannot rely on the popular random sampling+learning based approaches. They must be eschewed in favor of better methods that (a) can give us *near-optimal* configurations within a *sampling budget* and more importantly (b) can be scaled much more easily.

To achieve these objectives, in this paper **we argue that random sampling is inadequate and we need smarter sampling strategies that can explore diverse configurations with a high discriminatory power**. In particular, we show that Evolutionary Markov Chain Monte Carlo (EMCMC) sampling strategy is best suited for this purpose. A nice property of EMCMC is that, unlike random sampling, it tries to estimate the target distribution, *i.e.* draws more samples from important regions while maintaining sample diversity.

Overall, our work makes the following contributions:

- We implement a configuration exploration framework for big-data systems called CONEX. We demonstrate experimentally that CONEX outperforms learning-based methods in the case of big-data systems.
- We make available a replication package for CONEX to accompany this paper at git.io/Jv958.
- We show that Evolutionary Markov Chain Monte Carlo (EMCMC) based strategy can be effective in finding the high-performing configurations in a complex and high-dimensional configuration space of big-data systems. In general, EMCMC outperforms random and evolutionary (*i.e.* genetic algorithm based) sampling strategies.
- We find compelling evidence that CONEX can *scale-up*, *i.e.*, good configurations found with small workloads work well for significantly larger workloads and saves significant experimental cost.
- We find compelling evidence that CONEX can *scale-out*, *i.e.*, good configurations for one kind of job would yield improvements for other *dynamically similar* jobs, saving further sampling cost.

The rest of this paper is organized as follows: § 2 provides background. §3 discusses MCMC. § 4 introduces CONEX. § 5 presents the experimental design and evaluation. § 6 highlights our experimental results. § 7 samples some previous related work in this area. § 8 highlights some threats to the validity. § 9 offers some concluding remarks.

2 FORMALIZATION AND BACKGROUND

2.1 Terminologies and Example

The following are some of the frequently used terms:

- Configuration *parameter* p_i: Is a configuration variable whose value is set by a system installer or user to invoke certain desired system property.
- Configuration *type* t: Is an N-element record type $[p_1, \ldots, p_N]$, where each element p_i is a configuration parameter and N is the number of parameters representing the dimensionality of the space.
- *Configuration* c: Is a configuration type, t, in which valid values are assigned to the configuration parameters p_i.
- Configuration *space* ζ: Is the set of all valid configurations for a given system. The definition of $valid$ varies from system to system. If there are no constraints on a configuration c, and if N is the number of parameters and M is the average number of possible values of each parameter, then the size of configuration space ζ is roughly equal to M^N.
- *Performance* Y: The measured performance of the software system given that it is configured according to c. A number of performance measurements can be made for a system. For example, we may want to *maximize* performance measures such as *throughput* or *minimize* measures such as *latency*.
- Target distribution *P(Y | c)*: The conditional distribution that models the performance of the software system given the configuration c.

Table 1 presents two excerpts of sample configurations, $\mathbf{c_1}$ and $\mathbf{c_2}$, for Hadoop. In practice, configurations have hundreds of parameters, of varying types: Boolean, integer, categorical, string, etc. In total, as per Table 2, Hadoop has 901 and Spark has 212 configuration parameters giving rise to $3 * 10^{28}$ and $4 * 10^{16}$ total configurations respectively.

Table 1
Two Hadoop configuration examples

Parameters	$\mathbf{c_1}$	$\mathbf{c_2}$
dfs.blocksize	3	2
mapred.job.ubertask.enable	FALSE	True
mapred.map.java.opts	-Xmx1024m	-Xmx2048m
...	...	...
mapred.shuffle.merge.percent	0.66	0.75

2.2 Identifying optimal configurations with Stochastic Approximation

In the most general sense, the goal of finding the best configuration for a configurable software system can be understood as a search problem. The goal of this search problem would be to identify a configuration that maximizes (or minimizes) an objective function. More formally, given a space of all possible configurations ζ, and valid configuration from that space c, let us represent the software

system as a function $S : c \rightarrow Y$. The function S consumes an input c and returns the performance Y. The goal of finding the optimal configuration can be viewed as a search for a configuration $c^* \in \zeta$ such that we obtain the best performance Y^*. This can be generalized as follows:

$$\begin{cases} Y^* = \min_{c \in \zeta} Y \equiv \min_{c \in \zeta} S(c) & Y = \text{Latency, etc.} \\ Y^* = \max_{c \in \zeta} Y \equiv \max_{c \in \zeta} S(c) & Y = \text{Throughput, etc.} \end{cases} \quad (1)$$

However, the space of configurations, ζ, is exponentially large. Further, the software system S is quite complex and each of the configuration $c \in \zeta$ is highly nonlinear, high dimensional, and are otherwise inappropriate for deploying deterministic optimization strategies. Therefore, we seek alternative strategies to find optimum configurations [25].

A prominent alternative to deterministic optimization is the use of machine learning models in conjunction with stochastic optimization methods to solve the search problem of finding the optimum configuration for a given software system [7], [8], [19]. These methods use a three-step approach described below:

(i) *Stochastic Sampling*. This step attempts to overcome the issue of exponentially large space of possible configurations $c \in \zeta$. The sample of configurations c are drawn from an underlying probability distribution $f(c) = f(p_1, p_2, ..., p_N) \ \forall \ c \in \zeta$. This distribution is almost always assumed to be uniform in nature, *i.e.*, $f_\zeta(c) \rightarrow \mathcal{U}(p_1, p_2, ..., p_N) \ \forall \ c \in \zeta$. The total number of samples that are drawn are limited by a sampling budget.

(ii) *Modeling with machine learning*. Even with a limited number of samples, the time and cost overhead of having to measure the true performance can be exorbitant. Therefore, from among the sampled configurations, a few representative samples are used to construct a machine learning model to approximate the behavior of the software system. More formally, the machine learning model can be represented as a function ML that takes as input a configuration c and returns an estimated performance measure $\hat{Y}$, *i.e.*, $ML : c \rightarrow \hat{Y}$ where ML is a machine learning model and $\hat{Y}$ is the performance predicted by the ML model.

(iii) *Identifying the best configuration*. With the machine learning model from above, the performances of the remaining configuration samples are predicted. The configuration yielding the best performance is returned as being the optimum setting for the given software system.

2.2.1 Challenges with the current state-of-the-art

There are several challenges associated with using the methodology discussed above. Foremost among them is due to the stochastic sampling step. Since this sampling step precedes the construction of an ML model, the quality of samples directly affects the subsequent steps.

The problem with sampling arises due to the assumption that the configuration parameters p_i follow a uniform distribution. This assumption is fraught with risks—

1) Most of the time modeling might be spent exploring sub-optimal regions of the configuration space ζ.
2) The machine learning model build with these samples tend to be severely biased. As a result, they often fail to identify the best configuration if it exists in the regions of the configuration space that are previously unseen.

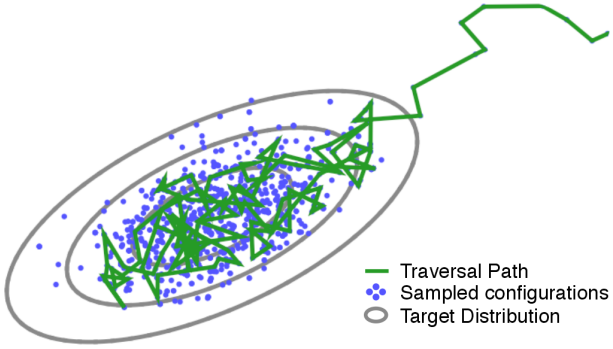


Figure 2. An example of *MCMC* traversal. The **green line** represents the state transition paths of the MCMC algorithm, the **blue dots** represents the samples and the **gray ellipses** represent the boundaries of the target distribution.

3) The optimum configuration identified by the ML model is at best only an estimate of the local optima.

The aforementioned problems can be a major limiting factor for big data systems such as Hadoop and Spark where both the configuration space and the cost of dynamic sampling are often significantly large. Also, it usually takes a longer time to run big data jobs; hence, generating enough dynamic samples within a limited exploration budget is challenging. Thus, getting stuck to some local region is a common problem. This paper aims to address these issues with Evolutionary Markov Chain Monte Carlo (EMCMC) based stochastic search strategy, as discussed next.

3 Evolutionary Markov Chain Monte Carlo

The problem of finding optimal configuration would be trivial if one knew how the configurations (c) affect the performance of the software system (Y), *i.e.*, if the distribution $P(Y \mid c)$ was known beforehand. When $P(Y \mid c)$ is unknown, one workaround could be to randomly draw enough samples from the configuration space to approximate the function. However, given the vastness of the configuration space of the big data system, a random draw is unlikely to find an optimal solution without some additional guidance. To overcome this, we use a class of algorithms based on Evolutionary Markov Chain Monte Carlo (EMCMC). While MCMC based methods are used to generate configurations from the unknown distribution $P(Y \mid c)$, the evolutionary component provides some additional guidance towards reaching the optima faster.

MCMC works by constructing a Markov Chain to generate a sequence of configuration samples where a new sample ($c_{t+1} \rightarrow Y_{t+1}$) depends only on its previous sample ($c_t \rightarrow Y_t$). The process of generating a next sample given a current sample is called a state-transition. We approximate the unknown distribution by recording all the state transitions; as the number of state transitions increases the distribution of the new samples converge more closely to the desired distribution (as illustrated in Figure 2).

The rest of this section discusses the use of MCMC to find optimal configurations. Specifically, our preferred MCMC algorithm (Metropolis-Hastings Algorithm) is discussed in §3.1, the evolutionary variant used for optimization (EMCMC) and its benefits are explained in §3.2.

3.1 Metropolis-Hastings Algorithm

A number of MCMC based methods exists in literature [26], [27], [28], [29]. From these, we choose the Metropolis-Hastings algorithm [27], [30] as they are best suited for deriving samples from high-dimensional spaces such as those observed in configuring big-data systems.

Given a configuration $c = [p_1, p_2, p_3, ..., p_N]$, with N configurable parameters denoted by p_i and the performance measure Y (*e.g.* execution time), there exists an unknown conditional distribution function $P(Y|c)$ that informs us of the performance Y of the job under test given a configuration c. However, without exploring the entire configuration space, $P(Y|c)$ cannot be inferred.

The Metropolis-Hastings algorithm attempts to generate a new configuration from the unknown distribution function $P(Y|c)$ using an *approximate function* $Q(Y|c)$ to draw samples, where $Q(Y|c)$ is proportional. but may not be identical, to the original distribution.

From $Q(Y|c)$, the Metropolis-Hastings algorithm generates a sequence of configurations such that the probability of selecting the next configuration (c_{t+1}) is dependent only on the current configuration (c_t). Since the configuration c_{t+1}, which is generated at step $t+1$, depends only on its immediate predecessor c_t, the sequence of samples belong to a first order Markov chain.

At each step, the algorithm measures an *Acceptance Probability* $A(c_{t+1}|c_t)$ that determines if the newly sampled candidate (c_{t+1}) will be accepted or rejected. The acceptance probability compares the performance of the newly generated candidate (*i.e.*, $Q(Y|c_{t+1})$) with respect to where the current sample lies (*i.e.*, $Q(Y|c_t)$). It is given by:

$$A(c_{t+1}|c_t) = min\left(1, \frac{Q(Y|c_{t+1})}{Q(Y|c_t)}\right) \tag{2}$$

Based on the acceptance probability the new candidate configuration is either accepted (in which case the candidate value is used in the next iteration) or rejected (in which case the candidate value is discarded, and the current value is reused in the next iteration).

The key component of the acceptance probability is that it is determined by the ratio $Q(Y|c_{t+1})/Q(Y|c_t)$. If the new configuration c_{t+1} is *more* likely to produce a better performance than the current configuration c_t, then $Q(Y|c_{t+1}) > Q(Y|c_t)$ and $Q(Y|c_{t+1})/Q(Y|c_t) > 1$. Consequently, according to Equation 2, $A(c_{t+1}|c_t) = 1$ and we will always accept the the new configuration.

On the other hand, if the new configuration c_{t+1} is *less likely* to produce a better configuration than the current configuration, then $Q(Y|c_{t+1}) < Q(Y|c_t)$ and $Q(Y|c_{t+1})/Q(Y|c_t) < 1$. In this case, $A(c_{t+1}|c_t)$ is less than 1 and we will sometimes reject the new configuration. However, since the acceptance probability $A(c_{t+1}|c_t)$ is non-zero, depending on the probability value, we may sometimes accept new configurations that perform worse than the previous configuration. The poorer the new configuration performs, the smaller the value of $A(c_{t+1}|c_t)$ will be, thereby making it less likely for us to accept a configuration with a very bad performance score. Using acceptance probability maintains the diversity among newly generated samples instead of always greedily choosing the better performants, and thus, slowly moves towards the target distribution.

The MCMC algorithm is designed to spend most of its time in the high-density region of the target distribution. Consequently, the samples of configurations obtained using MCMC are highly likely to contain the global optima among them.

3.2 EMCMC: An Evolutionary Extension to MCMC

Traditionally, MCMC uses some arbitrary distribution to generate the next sample configuration c_{t+1}, given the current sample c_t. The initial candidate configurations are mostly non-performant and are often discarded. Thus, although MCMC converges to the global optima eventually, it does so very slowly. This is detrimental because we may quickly exhaust our computational budget before finding a high-performant configuration.

To expedite the convergence, we ought to modify the existing MCMC to offer some additional guidance during the generation step in the Metropolis-Hastings algorithm discussed above (*i.e.*, Step-1). In this paper, we propose a novel *Evolutionary-MCMC* algorithm (or *EMCMC* for short). Like evolutionary algorithms (such as genetic algorithms [31], [32], [33]), in EMCMC we start with an initial *population* of $N > 1$ configurations. We then repeat the following steps until the allocated budget is exceeded:

1) **Evolutionary Generation**: We randomly choose a subset of initial configurations. For each configuration in the subset (c_t), we *generate* a new configuration c_{t+1} by applying (a) mutation; and (b) cross-over operations. That is:
   - *Mutation*: As described in §2.1, a configuration c is comprised many parameters p_i. During mutation, some of these parameters are randomly changed (with allowed values) to form a new configuration, say c_{t+1}.
   - *Cross-over*: We randomly selecting two parent configurations, from among all the c_t^i and c_t^j. Each configuration is bisected, *i.e.*, divided in 2-parts. Then the first part of c_t^i is spliced with the second part of c_t^j and vice versa, generating two offspring configurations c_{t+1}^i and c_{t+1}^j.
2) **Acceptance or Rejection**: For each new configuration generated with the previous step, we compute the acceptance probability according to Equation 2.
3) **Transition**: Using the acceptance probability for each c_{t+1}, we either retain those configurations, or we reject them. All the retained configurations represent the next state of the EMCMC algorithm.

3.2.1 Generalizability of EMCMC

The rest paper uses EMCMC strategy as part of the ConEx framework to generate new samples. It is worth noting that EMCMC can be applied to any domain where one not only needs to generate samples that belong to an unknown distribution but also requires the newly generated samples to exhibit some desired property (such as optimizing performance). Accordingly, EMCMC based approaches offer some marked benefits over both traditional MCMC and genetic algorithms.

Benefits over traditional genetic algorithms. Ordinary genetic algorithms are susceptible to getting trapped at local

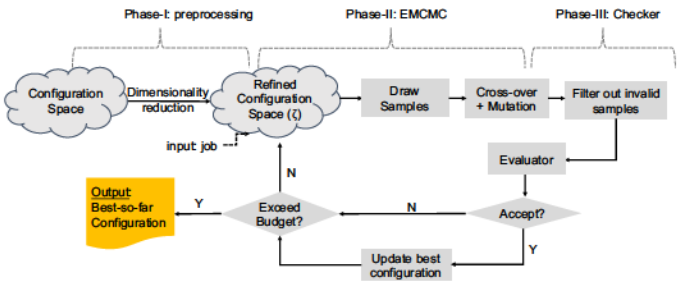


Figure 3. Workflow of the ConEx framework.

optima [34]. This is because, in regular genetic algorithms, a new configuration will never be accepted even if its performance is only a little worse than that of its parents and any worse configuration will always be discarded. In contrast, the acceptance or rejection of new configurations in EMCMC is contingent on the acceptance probability which sometimes accepts inferior configurations. This avoids a quick convergence to local optima and increases the chances of eventually reaching the global optima.

Benefits over traditional MCMC. As mentioned previously, pure MCMC based optimization algorithms converge very slowly. In contrast, by using principles of evolutionary algorithms, EMCMC ensures that MCMC has some guidance as it approaches the global optima.

4 ConEx: Configuration Explorer

Our approach is to use an EMCMC algorithm to sample Hadoop and Spark configuration spaces in search of high-performing configurations. We have implemented our approach in a tool called ConEx. The rest of this section describes our approach in detail.

Figure 3 presents an overview of ConEx. ConEx takes a big data job as input and outputs the best configuration it found before exceeding a sampling cost budget. ConEx works in three phases. In Phase-I, it reduces the feature space by filtering out the configuration parameters that are not relevant to performance. In Phase-II, it uses EMCMC sampling to find better configurations. Sampling starts with the default system configuration as the seed value. While sampling, ConEx discards invalid configurations generated during sampling using a checker developed by Tang et al. *et al.* [35] (Phase-III). If a configuration is valid, ConEx runs a benchmark job using it and records the CPU time of the execution. It then compares the result with that of the best configuration seen so far, updating the latter if necessary, per our acceptance criterion (see Equation (2)). Accepted configurations are subjected to cross-over and mutation, as described in Section 3.2, to produce configurations for the next round of sampling. Once ConEx exceeds a pre-set sampling budget, it outputs the best configuration found so far. We now describe each of these steps in greater detail.

4.1 Phase-I: Pre-processing the configuration space

Hadoop and Spark have 901 and 212 configuration parameters, respectively (Table 2). Yet most do not affect performance. In this step, we reduce the dimensionality of the configuration space by filtering out the parameters that do not affect the performance—this is similar to standard feature selection technique in Statistics or Machine Learning [36]. We reduce the dimension two ways: we consider only parameters relevant to performance using our domain knowledge; and we select only a few values for sampling for each parameter.

The first part is manual, and based on a study of technical manuals and other work [37]. For example, we removed Hadoop parameters related to version (*e.g.*, java.runtime-.version), file paths (*e.g.*, hadoop.bin.path), authentication (*e.g.*, hadoop.http.authentication.kerberos.keytab), server-address, and ID/names (*e.g.*, mapreduce.output.basename). For Spark, we selected parameters related to the runtime environment, shuffle behavior, compression and serialization, memory management, execution behavior, and scheduling. In general, we err on the side of caution– to make sure that we cover all parameters related to runtime performance, we select all parameters that have somewhat impact on the final performance. If we are not sure about a parameter, we include it in the sampling space. Overall, the selecting the features during the pre-processing required a few man-hours worth of manual inspection. After feature subset selection, we ended up with a total of 44 and 27 parameters for Hadoop and Spark respectively (summarized in table 2) that may possibly impact the performance. The remaining parameters are related to versions, paths, etc. and they tend not to affect the performance.

We then finitely sub-sampled the ranges of integer, float, and string types. In particular, we sub-sample the configuration space by defining small ranges of values for each parameter, varying by parameter type. Boolean parameters are sampled for true and false values. Numerical parameters are sampled within a certain distance from their default. Even these reduced configuration spaces are several orders of magnitude larger than those studied in previous work. For example, most systems studied by Nair et al. [19] have only thousands or at most a few million configurations. Table 2 summarizes the resulting configuration spaces that we dealt with for Hadoop and Spark.

4.2 Phase-II: Finding better configurations

This phase is driven by an EMCMC sampling strategy and implemented by Algorithms 1 and 2. Algorithm 1 is the main driver; Line 1 lists inputs and outputs. The algorithm takes a reduced configuration space (ζ) and a given job as inputs. ConEx samples configurations from

Table 2
Configuration Space Characteristics

System	Total Parameters	Total	Bool	Int	Float	Categorical	String	Approx. Total
Hadoop v2.7.4	901	44	4	26	6	3	5	$3 * 10^{28}$
Spark v2.2.0	212	27	7	14	4	2	0	$4 * 10^{16}$

ζ and evaluates their performance *w.r.t.* the input job. The routine also requires a seed configuration ($conf_{seed}$), and a termination criterion based on a maximum number of generations (max_gen). We choose $max\_gen = 30$ in our experiment. The tool then outputs are the best configuration found ($conf_{\text{best}}$) and its performance ($perf_{\text{best}}$).

Lines 2, 4 and 5 initialize some parameters including setting the best configuration and performance to the respective seed values. Line 6 gets the first generation of configurations by randomly sampling n items from ζ. We choose $n = 4D$ where D is the number of parameters, but it could be set to any reasonable value. Lines 9 to 18 are the main procedure for evaluating and evolving *w.r.t.* each configuration. Given a configuration $conf_p$, Line 10 records the job's performance ($perf_p$) and Line 11 decides whether to accept it based on Equation (2). If accepted, Line 13 stores the accepted configuration to a list $confs_{accepted}$, which later will be used in generating next-generation configurations (Line 20). If the accepted one is better than the best previously found, Lines 14 to 16 update the state accordingly.

Once all configurations in the first generation are processed, Line 19 computes the performance improvement achieved by this generation *w.r.t.* the seed performance. Next in Line 20, the algorithm prepares to enter the next generation by generating offspring configurations using cross-over and mutation operations (see Algorithm 2). Line 21 updates the generation number. This process repeats until the termination criterion is satisfied (Line 22). Finally, the last line returns the best found configuration and the corresponding performance.

Algorithm 2 is the evolution sub-routine of the EMCMC algorithm, as described in Section 3.2. For preparing configurations of the next generation, it takes the best configuration found so far and a list of parent configurations as inputs. There are two main steps: cross-over and mutation. From the best configuration, Line 3 selects half of all parameters as cross-over parameters ($P_{crossover}$), and Line 4 identifies 6% of parameters as mutation parameters (P_{mutate}). Next, for each parent configuration, Line 6 exchanges the values of same parameters in two parents with $P_{crossover}$. Note that since the values of the same parameter is exchanged, their types are automatically preserved. It then randomly mutates the values of the mutation parameters at Line 7. The resulting offspring is added into the children set at Line 8. A set of new offspring configurations is returned at Line 10.

Algorithm 1: Explore Configuration Space

```
1  Function EMCMC()
   Input  : Refined Configuration Space ζ, job,
            seed conf_seed, threshold max_gen
   Output: Best configuration conf_best
2  perf_seed ← run job with conf_seed
3  conf_best, perf_best ← conf_seed, perf_seed
4  generation ← 1
5  Δperf ← 0
6  conf_parents ← sample n random configurations from ζ
7  while generation < max_gen do
8      confs_accepted ← EmptyList
9      foreach parent conf_p ∈ conf_parents do
10         perf_p ← run job with conf_p configuration
11         accepted ← Accept(perf_best, perf_p) # Eq. 2
12         if accepted then
13             confs_accepted.add(conf_p)
14             if perf_p > perf_best then
15                 conf_best, perf_best ← conf_p, perf_p
16             end
17         end
18     end
19     Δperf ← (perf_best − perf_seed)/perf_seed
20     conf_parents ← evolve(conf_best, confs_accepted)
21     generation ← generation + 1
22 end
23 return conf_best
```

Algorithm 2: The evolutionary sub-routine of EMCMC

```
1  Function Evolve
   Input  : conf_best, confs_accepted
   Output: conf_children
2  conf_children ← EmptyList
3  P_crossover ← randomly select 50% parameters from
     conf_best
4  P_mutate ← randomly select 6% parameters of conf_best
5  foreach conf_p ∈ confs_accepted do
6      conf_new ← crossover(conf_best, conf_p, P_crossover)
7      conf_new ← mutate(conf_new, P_mutate)
8      conf_children.add(conf_new)
9  end
10 return conf_children
```

4.3 Phase-III: Configuration Validity Checking

Configuration spaces are often subject to constraints on one or more parameters. For example, Hadoop's *JVM options* parameter is of *string* type, but not any string value will work. These constraints must be met to generate valid configurations. This is typical in domains such as software product line optimization [38], where we have access to the constraints in the form of feature models. Unfortunately, for big-data systems, such as Hadoop and Spark, such validity constraints are neither well documented nor strictly enforced. Thus, the likelihood to making configuration mistakes is increased.

For big data system such mistake is costly as it increases the cost of dynamic sampling. In a previous work, we[1] have developed and employed a configuration constraint checker developed with COQ theorem prover [39] to express and check constraints [35]. In this work we extend the open-source instrumentation[2] of the checker to ensure that the newly generated constraints are valid.

The checker provides expressive means for documenting properties and constraints on the configuration parameters, and the type checker checks that all constraints are satisfied. We express all the configuration constraints in the form of COQ rules. Note that, as we leverage an existing checker, here we just need to write the rules, which requires manual effort of a single day. Such expressiveness cannot be offered by just the documentation. For example, Hadoop informally documents (but does not enforce) that certain configuration parameter values should be multiple of the hardware page

1. Tang, C and Sullivan, K
2. https://github.com/ChongTang/SoS_Coq

size. Our checker generates a type error if a violation occurs.

4.4 Using ConEx in Production scale jobs

The proposed framework for ConEx may operate with any workload size. However, in order to extend its usability to production scale jobs, we employ the following two strategies:

1) **Scale-up**: For cost-effectively sampling performance as a function of configuration during sampling processes, we run Hadoop and Spark jobs using inputs that are several orders of magnitude smaller (here, 100X) than those we expect to run in production.
2) **Scale-out**: To amortize the cost of sampling and profiling, we use a dynamic similarity measure of big data jobs to decide when good configurations found for one kind of job might be used for another kind without any additional sampling activity.

The above strategies do not form part of the core ConEx framework. Instead, they are used when ConEx in deployed in a production environment.

5 Experimental Design

We implemented ConEx with about 4000 lines of Python code[3]. Our experiments were conducted in our in-house Hadoop and Spark clusters. Each had one master node and four slaves, each with an Intel(R) Xeon(R) E5-2660 CPU, 32GB memory, and 2 TB local SSD storage. We assigned 20GB of memory for Hadoop on each node in our experiments. We also made sure that no other programs were running except core Linux OS processes.

5.1 Study Subject and Platform

Table 2 summarizes the parameters and their types that we have studied for Hadoop $v2.7.4$ and Spark $v2.2.0$.

To evaluate ConEx, we selected big-data jobs from HiBench [40], a popular benchmark suite for big data frameworks. It provides benchmark jobs for both Hadoop and Spark. For Hadoop, we selected five jobs: WordCount, Sort, TeraSort, PageRank, and NutchIndex. Of these, `nutchindex` and `pagerank` are used in websearch; `sort` and `terasort` sort data; and finally `wordcount` operates on text data. These jobs only need Hadoop to execute. For Spark, we selected five Spark jobs: WordCount, Sort, TeraSort, RF, and SVD. Here, `svd` and `rf` are machine learning jobs that are unique to Spark.

HiBench has six different sizes of input workload for each type of job, from "tiny" (30KB) to "Bigdata" (300GB). For our experiments, we used "small (320MB)", "large (3.2GB)", and "huge (32GB)" data inputs. We ignore the `tiny` workloads since the size (32kb) is too small to accurately model I/O and memory requirements for larger workloads such as `large` and `huge`. We note that our smallest baseline workload (`small`) is 3MB and the largest workload (`huge`) is 3GB. Although, the memory and I/O requirements for `small` and `huge` are vastly different, they can still be accommodated within our hardware.

3. Replication package is available at https://git.io/Jv958

Table 3
Example Tuple representing resource usage of a job

Example System Call Sequence	$foo(1,"b"), bar("b", True), foo(2,"b"), foo(3,"c")$
A	$\{foo, bar, foo, foo\}$
B	$\{foo : ("b","c"), bar : ("b")\}$
C	$\{foo : ["b" : 0.66, "c" : 0.33], bar : ["b" : 1.0]\}$
D	$\{foo : ["1^{st}arg" : 2.0]\}$

Table 5 shows the CPU times taken by the Hadoop jobs running with default configurations. We used small inputs while sampling but then evaluated the resulting configurations using huge (100X larger) workloads. A HiBench execution has two steps: data preparation and job execution. We prepared the data manually to control the data used for each job.

5.2 Job Classification

To test our *scale-out* hypothesis we needed a measure of job similarity. We settled on resource usage patterns rather than HiBench job types for this purpose. HiBench classifies jobs by business purpose (*e.g.* indexing and page rank jobs fall in *Websearch* category), which does not necessarily reflect the similarity in resource usage patterns. Our approach is based on the profiling of run-time behavior using system call traces. Similar approaches have been widely used in the security community [41], [42], [43]. We use a Unix command-line tool, `strace`, to capture system call traces for this purpose.

Based on the system call traces, we represent each job by a four-tuple, $< A, B, C, D >$, where A is a system call sequence, B is a set of unique string and categorical arguments across all system calls, C expresses term frequencies of string and categorical arguments captured per system call, and D is the mean value of the numerical arguments per system call. Table 3 shows an example tuple.

To compute the similarity between two jobs, we calculate similarities between corresponding tuple-elements separately. Each contributes equally to the overall similarity measure. For A, we use pattern matching—we slice the call sequences and compute the similarity between them. To find the similarity between B elements, we compute the Jaccard Index, which is a common approach to compute the similarity of two sets. For C elements, we compute the average difference of each term frequency. Finally, for D elements, we compute the similarity of mean value of numerical arguments as $1 - abs(mean1, mean2)/max(mean1, mean2)$. We take the average value of these four scores as the final similarity score between two jobs. We consider two jobs to be *similar* if their similarity score is above 0.77 (*i.e.* from third quartile (Q3) of all the similarity scores).

5.3 Comparing with Baselines

We compare ConEx's performance with three potentially competing approaches: (i) a random sampling strategy, (ii) a genetic algorithm-based optimization strategy, and (iii) a learning-based approach. The first one evaluates whether

EMCMC is a good sampling strategy, the second one checks EMCMC's ability to find a near-optimal configuration. The last one evaluates the choice of EMCMC over a model-learning approach. Here, we compare CONEX with Nair *et al.*'s ranking based approach [44], which is most relevant for our problem.

5.4 Evaluation Criterion

Typically, performance optimization models are evaluated using *performance gain*. Let the performances of a job with the default configuration be is $perf_{\text{default}}$, and the best discovered configuration be defined as $perf_{\text{best}}$. Then the performance gain Δ_{gain} measures the absolute percentage improvement in the performance. It is computed as below:

$$\Delta_{\text{gain}}\% = \left| \frac{perf_{\text{default}} - perf_{\text{best}}}{perf_{\text{default}}} \right| \times 100$$

Note that while it may make intuitive sense to compare $perf_{\text{best}}$ to the true-best configuration, the best configuration is unknown as the configuration space is extremely large. In contrast with machine learning based methods of Nair *et al.* [44] that attempt to predict for the best configuration using a learning based approach, we use a evolutionary search to estimate the best configuration within a given budget. If we have infinite budget, CONEX can theoretically converge on the true-best configuration. Therefore, we compare, the best configuration found within a given budget (*i.e.*, $perf_{\text{best}}$) with the default configuration (*i.e.*, $perf_{\text{default}}$).

6 EXPERIMENTAL RESULTS

RQ1. Can CONEX find configurations that are better than the baseline default configurations?

The first research question seeks to provide a summary of the performance gains that can be achieved with the use of CONEX for three workloads: small, large, and huge of Hadoop and Spark jobs. Specifically, we explore different configurations using CONEX with smaller workloads (in RQ1-1); then we use the configurations obtained from the small workloads to check for performance gains of larger workloads, *i.e.*, through scale-up hypothesis (in RQ1-2); and finally for different jobs that share similar dynamic characteristics (in RQ1-4).

RQ1-1. How effective is CONEX in finding optimal configurations for small workloads? We investigate this RQ for Hadoop by exploring the configuration space using *small* workloads. HiBench generates a detailed report after each run, with performance information including CPU time. For Spark jobs, we intentionally refrained from using a small workload. This is because, for small workloads, the runtime in Spark was negligibly low compared to Hadoop. Therefore, to enforce a fair comparison, we used Spark "large" workloads, tailored to take as long as small jobs did in Hadoop (about 30 seconds per run). This ensures the cost of sampling is comparable between both systems.

Our findings are tabulated in Table 4. The best configurations found by CONEX achieved between 7% to 72% improvements over the default configurations for five Hadoop jobs. On average, we notice a performance gain of 30.3% for Hadoop jobs operating on a small workload. For Spark jobs, CONEX produced 0.4% to 40.4% performance improvements for all five jobs with an average improvement of 10.6%. Thus, we conclude that:

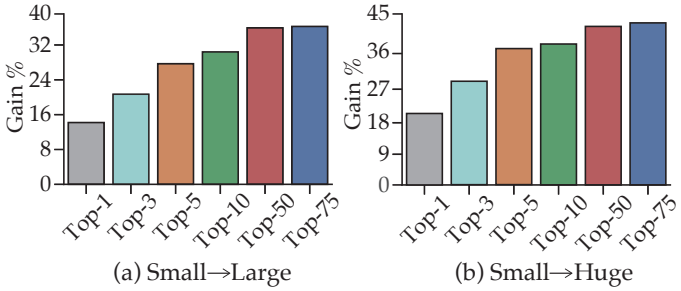
Table 4
Performance improvement offered by CONEX for *Scale-up*. Note: the numbers reported below for scale-up were obtained using the Top-1 configuration from the baseline.

HADOOP

Small (Baseline)					
WordCount	Sort	TeraSort	NutchIndex	PageRank	Average
12.50%	72.10%	27.40%	7.09%	32.70%	30.30%
Small ⟶ Large					
15.60%	7.70%	16.00%	18.70%	44.70%	20.50%
Small ⟶ Huge					
21.50%	15.80%	18.30%	14.20%	25.20%	19.00%

SPARK

Large (Baseline)					
WordCount	Sort	TeraSort	RF	SVD	Average
2.70%	3.28%	40.41%	6.35%	0.38%	10.60%
Large ⟶ Huge					
5.80%	1.60%	16.80%	7.10%	1.90%	6.60%



(a) Small→Large (b) Small→Huge

Figure 4. Performance gains over to default configuration for scale-up in Hadoop WordCount using top-1, top-3, top-5, top-10, top-25, top-50, and top-75 best configurations from the small workload. In both small→large and small→huge we notice even with top-1 we achieve a performance improvement over default. From top-3 to top-50 the performance gains increases. Beyond top-50 the improvements are marginal.

> **Result:** For Hadoop and Spark jobs with the small workload, CONEX can find configurations that produce up to 72% and 40% performance gains over the default configurations in Hadoop and Spark respectively.

Even if CONEX manages to find a better configuration with smaller workloads, CONEX will be most effective if it can improve performance for larger workloads. It will obviate the need for making additional performance measurements thereby saving significant cost.

RQ1-2. How well does CONEX *Scale-Up* when exposed to much larger workloads? We investigate this RQ in three steps: (i) Evaluate the performance gain of Hadoop and Spark jobs for large and huge workloads when using the best configuration obtained for a small workload, (ii) Assess if any of the top-1 to top-75 configurations for a small workload achieve notable performance gains for large and huge workloads; and (iii) Perform a cost analysis to assess

whether Scale-Up can save configuration exploration cost.

(i) Evaluating Performance Gain.

Here, we determine whether good configurations found for `small` jobs provide comparable benefits for much larger jobs. Accordingly, with no additional sampling, we reuse the best configuration (*i.e.*, Top-1) obtained for `small` workload (from *RQ1-1*) on `Large` (10× larger) and `Huge` (100× larger) inputs for Hadoop jobs. For Spark, we use the best configurations (*i.e.*, Top-1) obtained from `large` jobs to evaluate `Huge` inputs.

For Hadoop, in all five jobs, using the Top-1 best configuration, we see an average of 20.5% and 19% improvements under large and huge workloads respectively. We observe that, for WordCount, PageRank, and NutchIndex jobs, larger workloads exhibited better performance gains than the baseline improvements under the small workloads (see Table 4), *e.g.*, in Hadoop's WordCount, using CONEX for small workloads (baseline) results in a performance gain of 12.5% whereas small→large results in a performance gain of 15.6% while small→huge results in a gain of 21.5%.

For Spark, using the Top-1 best configuration, we observed performance improvements for all five jobs ranging from 1.6% to 16.8%. However, for Sort and SVD, there were limited improvements: 1.6% and 1.9% respectively. Spark jobs run very fast compared to Hadoop jobs. Although we have scaled up the workload 10 times, it seemed to be hard to achieve significant gains. Additional research is needed to better understand the scale-up potential of Spark jobs. Nevertheless, we saw an average performance improvement of 6.6%, which we still believe is significant if it holds in practice at this scale.

(ii) Assessing variations in scale-up performance. Larger workloads present several memory constraints in addition to other I/O overheads. Therefore, there is a potential for numerous variations in the scale-up performance (how well gains on small jobs scale up to similar gains on large jobs). To evaluate this, we assess how well each of the Top-1, Top-3, Top-5, Top-10, Top-50, and Top-75 best-performing configurations found in the small inputs performed on the much larger data sets.

We observe that better performance gain is usually achieved for configurations that are among the Top-3, Top-5, Top-10, and Top-50 for small workloads as shown in Figure 4. We also observed that, beyond Top-50 configurations in the small workload (*e.g.*, Top-75), there were no further improvements in scale-up. This was true for both Large and Huge workloads.

While it is true that running fifty production-scale jobs (instead of Top-1) as a final step of our approach would incur significant additional cost and compute resources, we find that doing so may produce a notable performance improvement (as illustrated in Figure 4). This is a trade-off that depends on the available budget. If the budget is very low, industrial practitioners may use just to Top-1 best configuration from the small workload and still obtain improvements reported in Table 4. However, if additional budget is available, practitioners may choose to evaluate the Top-3 to Top-50 configurations to find larger improvements.

(iii) Cost Analysis. The first three columns in Table 5 show the total execution time (in seconds) across the master-slave nodes for each Hadoop job while our test-bed is configured with default configurations. For example, *WordCount* under a small workload takes about 33 seconds per cluster (166.2/5, where 5 is the number of nodes of our cluster). However, it takes around 1873 seconds with "huge" data, with the time difference of 1840 seconds. The last column shows the number of dynamic evaluations of sampled configurations before CONEX achieves the best configuration. Thus, for *WordCount* job, our scale-up strategy saves 1656 hours ($(1873 - 33) * 3241$ seconds) to find a better configuration using scale-up strategy. In total, for all the five jobs, the scale-up strategy saves about $9,600$ hours or 39.6 times. In monetary terms, that amounts to about $12,480 on the AWS EMR service with m4.xlarge EC2 instances.

Table 5
Average CPU time (secs) for default configuration in Hadoop for each of the studies job under three workload sizes.

	Small	Large	Huge	#EXP
WordCount	166	862	9367	#3241
Sort	133	869	9891	#3318
Terasort	115	1056	8751	#2876
Pagerank	300	5657	13096	#3177
NutchIndex	477	6596	11215	#4685

Table 6
Performance gains offered by CONEX with the *Scale-out* Hypothesis.

HADOOP

Src. / Tgt.	Similar Jobs: WordCount	Sort	TeraSort	PageRank	Diff. Job: Nutch
WordCount	*21.5%*	10.7%	28.1%	20.6%	7.1%
Sort	11.4%	*15.8%*	21.1%	18.4%	5.7%
TeraSort	10.4%	1.8%	*18.3%*	29.9%	3.8%
PageRank	20.8%	23.8%	23.4%	*25.2%*	16.8%
Nutch	12.2%	27.6%	15.5%	10.4%	*14.2%*

SPARK

Source / Target	Similar Jobs: WordCount	Sort	TeraSort	RF	SVD
WordCount	*5.8%*	50.8%	22.8%	5.9%	1.8%
Sort	3.5%	*1.6%*	22.3%	9.9%	4.7%
TeraSort	5.1%	17.8%	*16.7%*	9.9%	3.1%
RF	4.3%	20.1%	10.0%	*7.2%*	2.5%
SVD	2.2%	23.8%	13.4%	23.4%	*1.9%*

Result: Configurations found using small workloads as proxies for sampling production-scale performance do tend to produce significant performance improvements (from 10% to 20%, on average) for much larger workloads and thus, save a significant amount of exploration cost.

RQ1-3. After how many optimized runs of a job does CONEX offer break-even benefits? Previously, we showed that scale-up with both top-1 and top-50 configurations produces notable performance gains over default configurations. However, such gains come with some initial exploration cost. For instance, to find the optimal configuration, CONEX had to run a job a few iterations as per the allowed

Table 7
Most Influential Parameters for Hadoop Jobs. The numbers in the top row indicate the total performance gain achieved by the corresponding job. The numbers in the bottom row represent the percentage performance gain achieved by the corresponding parameters.

WordCount (22.34%)	**Sort** (19.77%)	**TeraSort** (18.45%)	**PageRank** (19.56%)	**NutchIndex** (19.04%)
memory.mb: 12.61%	input.buffepercent: 12.13%	map.java.opts: 8.65%	memory.mb: 10.82%	map.java.opts: 9.44%
sort.spill.percent: 3.76%	memory.mb: 4.29%	input.buffepercent: 7.22%	input.buffepercent: 5.38%	task.io.sort.mb: 6.19%
input.buffepercent: -0.48%	io.seqfile.compress.blocksize: 1.83%	task.io.sort.mb: 2.84%	java.opts: 3.10%	reduce.memory.mb: 4.92%
job.max.split.locations: -0.67%	io.file.buffesize: 1.58%	memory.mb: 1.75%	task.io.sort.mb: 2.38%	map.memory.mb: 1.83%
yarn.app.aresource.mb: -1.54%	java.opts: 1.11%	io.seqfile.compress.blocksize: 1.58%	memory.mb: 1.95%	yarn.Rscheduleclass: 0.39%

Note: Due to the page limit, we only list five most influential parameters.

time budget, which is around 250 iterations with a small workload for word count job. Further, for top-50 setting, we need to run an additional 50 production scale jobs with a large or a huge workload. In this RQ, we investigate how many production scale jobs we need to run (with an unoptimized setting) to amortize this cost.

To answer this question, we use Hadoop's WordCount as an example. First, we run CONEX on a small workload for 12 hours (approx. 250 iterations). Second, we deploy WordCount for Large and with 3 different configuration settings:

- *Configuration-1 (Baseline)*: Here we use the default configuration. We do not deploy CONEX, therefore, this job incurs no initial overhead.
- *Configuration-2 (Top-1)*: Here we use the best configuration obtained by running CONEX on a small workload for 12 hours. This incurs an initial overhead of 12 hours.
- *Configuration-3 (Top-50)*: Here we first run CONEX for 12 hours to find the top-50 configurations on a small workload. Next, we run the top 50 configurations on large/huge workload to find the best configuration from among the top 50. Thus, in addition to the 12 hours taken to run CONEX, this job incurs an additional initial overhead of having to run large/huge workloads 50 times. For large workload in hadoop WordCount, this adds 12 more hours of initial overhead. For Huge workload, this adds an initial overhead of 103 hours.

Figure 5 shows how many times we need to run a job in these experimental settings to amortize the initial exploration overhead with a trade-off curve. The x-axis shows job iterations in large/huge workload, and y-axis is the gain *w.r.t.* baseline. During the initial exploration phase, the gain will be negative because, during that time, one can run production-scale jobs. Once a better configuration is selected, the job's performance starts gaining (reflected by positive slopes) *w.r.t.* baseline. However, it takes some initial runs to amortize the overhead cost.

The initial exploration to find top-1 configuration requires around 12 hours, which is our overhead. On average, a large workload takes 862 seconds to run. In 12 hours, we could have run the large workload around 48 times at the default configuration (see ① in Figure 5(a)). To amortize the overhead at the top-1 setting, we need to run the large job additional 370 times. Thus, we obtain a *break-even* performance gain if the large workload were to be run more than 418 times (as shown by ③ in Figure 5(a)). In contrast, Top-50 incurs a total of 23 hours of initial overhead because we run CONEX for 12 hours and then to run 50 configurations it takes another 11 hours after that to find the

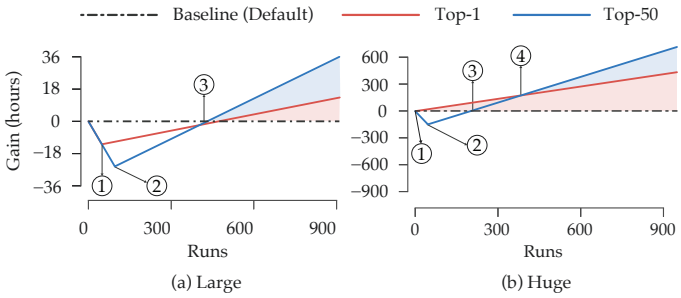


Figure 5. Break-even analysis of scale out (top-1 and top-50) vs. default configuration.

best configuration to use. This incurs a total cost equivalent to 98 runs of a large workload (shown by ② on Figure 5(a)). In However, as noted in Figure 4, using top-50 offers an overall gain of 32% as opposed to the 16% offered by top 1. Consequently, using top-50 after break-even offers more performance gain than top-1 (the region shaded blue is larger than baseline compared the region shaded by pink). Thus, to amortize the overhead at the top-50 setting, we need to run the large job additional 306 times; we reach a break-even performance gain with default at 404 iterations. It is, therefore, generally better to expend the additional initial overhead to find the best configuration from among top-50.

For huge workloads, finding top-1 configuration with scale-up incurred an initial overhead of 12 hours. A huge workload, on average, takes 3 hours to run. Therefore, in 12 hours, we can deploy 4 instances of the huge workload with the default configuration before we can find a top-1 configuration with scale-up. Therefore, we achieve *break-even* using top-1 almost immediately (after 4 runs) as shown by ① and the shaded pink region in Figure 5(b). On the other hand, Top-50 incurs a total of 12 hours of initial exploration overhead and then an additional 103 hours to run the top 50 configurations from scale-up to find the best configuration from among them to use we. This accounts for an equivalent of 54 runs of the huge job (shown by ② in Figure 5(b)). However, after as few as 250 runs, with using the configuration from top-50 we achieve break-even performance (see ③ in Figure 5(b)). In huge, the performance gain obtained by using the best configuration from top-50 is 41% (from Figure 4) compared to 19% from top-1. This additional gain of using top-50 offers an overall break-even performance gain over top-1 if the huge job runs more than 380 times (see ④ and region shaded by blue in Figure 5(b)).

Table 8 highlights the break-even point for various

Table 8
Amortization costs for various workloads. The more compute intensive the workload is, the sooner we achieve the break-even point. Note: in each of the following setting, CONEX was deployed for 12 hours.

Job	Avg. Small Runtime (seconds)	Workload	Default Runtime (seconds)	Gain (%)	Setting	Overhead	Break-Even Point (wrt. Baseline)
Word Count	166	Large	897	16	Top-1	48	418
		Huge	9367	21	Top-1	4	23
		Large	897	32	Top-50	98	404
		Huge	9367	41	Top-50	54	185
Sort	133	Large	869	7	Top-1	49	749
		Huge	9891	15	Top-1	4	30
TeraSort	115	Large	1056	16	Top-1	40	290
		Huge	8751	18	Top-1	4	26
NutchIndex	300	Large	6596	18	Top-1	6	39
		Huge	11215	14	Top-1	3	24
PageRank	300	Large	5657	44	Top-1	7	22
		Huge	13096	25	Top-1	3	15

workloads. Firstly, we notice that are compute intensive (NutchIndex and PageRank) achieve break-even *w.r.t.* using CONEX much sooner. Specifically,

1) For Large workloads: It takes 39 and 22 default large runs before using CONEX offers better amortized performance gain in NutchIndex and PageRank.
2) For Huge workloads: It takes 24 and 15 default large runs before using CONEX offers better amortized performance gain in NutchIndex and PageRank.

Secondly, we observe that, in all cases, CONEX offers break-even benefits much earlier for Huge workload. This is because, for huge workloads, the initial overhead of having to find a good configuration is far less detrimental than running unoptimized (default) configurations.

Typically a server (*e.g.* AWS) is configured with CONEX, where a job is expected to run numerous times. A web server, fo instance, is expected to process Millions of queries once it is configured. Hence, we hope the initial amortization cost is acceptable, given its benefits in the long run. Also, this cost is no more than the other learning-based approaches that need to collect samples to train a model.

Result: CONEX offers break-even performance gains over default configurations if a big data job runs more than 420 times (for large workloads). For workloads of size Huge, CONEX offers break-even performance gain after only 4 runs. Overall, if a big data job runs more than 420 times (for a large workload) or 250 times (for a huge workload), then it is better to use the best configuration from top-50 even if the initial overhead is larger.

RQ1-4. How well does CONEX *Scale-Out* when exposed to different job types?

To further reduce exploration cost we assess if our scale-out hypothesis holds, *i.e.* configuration found for one kind of job, A (e.g., Word Count), will also produce performance gains for a similar kind of job, B (e.g., Sort). We test this by evaluating performance gains for jobs of some job type, B, using configurations found for a job of some type, A, where the similarity between A and B is measured using Section 5.2. Among the five Hadoop jobs, we found *WordCount*, *Sort*, *TeraSort* to be highly similar, that *PageRank* is somewhat similar, and that *NutchIndex* has low similarity with this group. Table 6 shows the results of the Scale-Out hypothesis.

For example, Table 6 shows that CONEX found a configuration for *WordCount (WC)* that improves its performance by 21.5%. When the same configuration is used for the similar target jobs: *Sort*, *TeraSort*, *PageRank*, the performance gains achieved (10.7%, 28.1%, and 20.6% respectively) are close to the improvements found by their own best configurations. However, for *NutchIndex*, which is not so similar, we see a performance gain of only 7.1%, while it achieved 14.2% gain while experimenting with its own best configuration. Similar conclusions can be drawn for Spark jobs (Table 6). A surprising outcome was that, in few cases, a better configuration found for one job, e.g., NutchIndex, yielded greater gains for another job, e.g., Sort (27.6%) than the gain achieved by its improved configuration (15.8% for Sort). We have left the analysis of such surprising behavior for future work.

Result: Our scale-out hypothesis holds good, i.e., the configuration found with a representative job can bring significant performance gain for other similar jobs.

RQ1-5. Which parameters have the highest impact on improving performance gain of CONEX? Here we study how sensitive performance gains are *w.r.t.* individual configuration parameters. From each best-found configuration, we set the value of each parameter back to its default value leaving all other improved parameter values unchanged and check to see how much the performance reverts to the baseline.

For example, if $perf_{\text{def}}$ and $perf_{\text{best}}$ are the default and best performances (*i.e.* CPU times) obtained by CONEX for a job, then, the performance improvement is

$$\Delta_{\text{best}} = (perf_{\text{def}} - perf_{\text{best}}) / perf_{\text{def}}$$

Next, to measure how sensitive the gain is *w.r.t.* a parameter p_i, we set p_i's value back to default without changing the other parameter values from the best configurations. We measure the new performance *w.r.t.* to the default; Thus,

Table 9
Performance* gain of EMCMC over Genetic Algorithm (GA) and Random Sampling for Hadoop jobs

	Genetic Algorithm			Random		
	Small	Large	Huge	Small	Large	Huge
WordCount	92.31%	58.38%	84.61%	123.21%	61.66%	47.26%
Sort	2.12%	-32.81%	53.75%	18.98%	44.53%	16.96%
TeraSort	26.85%	-6.96%	15.09%	136.21%	85.96%	64.72%
NutchIndex	23.95%	62.26%	5.95%	-31.83%	29.31%	18.96%
PageRank	-29.57%	155.80%	18.89%	-0.67%	272.52%	77.75%
Average	0.63%	52.20%	31.16%	25.33%	105.34%	45.39%

*Percentage performance improvement is computed as $\frac{\text{perf}_{emcmc} - \text{perf}_{ga/random}}{\text{perf}_{ga/random}}$

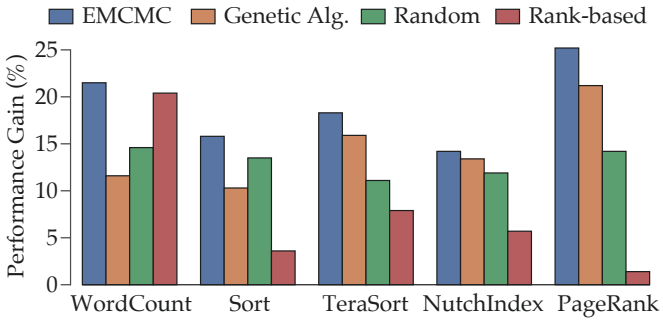


Figure 6. EMCMC compares with other approaches in performance improvement for Hadoop Huge Workload.

$\Delta_i = (perf_{\text{def}} - perf_i)/perf_{\text{def}}$. Then the sensitivity of parameter p_i is the difference of performance improvement:

$$sensitivity_i = \Delta_{\text{best}} - \Delta_i$$

We conducted this analysis for all the parameters one by one for the Hadoop benchmark jobs using "huge" workloads. Table 7 shows the results. The second row is the overall performance gain[4]. The results suggest that performance improvement is sensitive to only a few parameters. However, no single parameter is responsible for most of the improvement. That is,

Result: The influences of individual parameters are limited and the overall improvements arise from the combinations of, or interactions between, multiple parameters in the configuration.

These results suggest that, at least for Hadoop, higher-order interactions are present in the objective function and that these will need to be addressed by algorithms that seek high performing configurations. Also, further improvements in sampling efficiency might be possible by focusing on a smaller subset of performance-critical parameters. However, we leave this to be explored in our future work.

RQ2. How does the EMCMC sampling strategy perform compared to random and evolutionary sampling?

Here we compare EMCMC with (i) random and (ii) genetic algorithm (GA) based evolutionary sampling strategies. A random approach samples a parameter value from the uniform distribution, *i.e.* each value in the value range has the same probability to be selected. We have also implemented a GA based optimization strategy with the same cross-over and mutation strategies and the same fitness function as of EMCMC (See Section 3.2). For comparison, we run the baseline strategies to generate the same number of configurations and profile their performances with "Small" data sets. We then conduct the scale-out validation to evaluate the performance gain in larger workloads. Table 9 shows the performance gain of EMCMC over these two strategies.

In general, for all the jobs, EMCMC based sampling performed better. For example, on average, EMCMC performed 52.20% and 31.16% better than GA for large and huge jobs. In comparison to random sampling, EMCMC performed 105.34% and 45.39% better, on average. Figure 6 pictorially represents the results for "Huge" workload. The improvement of the performance of EMCMC over GA also gives us an estimate of how much the evolutionary part of EMCMC contributes to ConEx's performance.

Result: EMCMC based sampling strategy, on average, outperforms random (by up to 105%) and genetic algorithm (by up to 52%) based evolutionary sampling strategies to find better performing configurations for Hadoop.

RQ3. How does ConEx perform compared to the state-of-the-art Machine Learning based configuration optimization approaches?

To compare our approach with learning-based approaches, we choose the previous work of Nair *et al.* [19] published in FSE 2017. We carefully choose this work as they also intended to find a near-optimal configuration rather than the best one, which is the most practical approach for a big-data job. They used a rank-based performance prediction model to find a better configuration. The authors argued that such a model works well when the cost of sampling is high, such as ours. They showed that compared to residual-based approaches, their model saves a few orders of magnitude of sampling cost and achieves similar and even better results.

In their experiments with larger systems having millions of configurations (*e.g.* SQLite), the training pool covered 4500 configurations, including 4400 feature-wise and pair-wise sampled and extra 100 random configurations. We used the same approach—we randomly collected the same number of configurations as ConEx to profile their performances (similar to RQ4) and used them as training. We reused the model published by Nair *et al.*[5]. As they did, we ran each model 20 times to find improved configurations.

For a fair comparison, following Nair et al., we evaluated both approaches by measuring rank difference (RD) between the predicted rank of a configuration and the rank of the training data (the profiled performance in our case). Table 10 shows the result. Here we ran each model 1000 times to get enough data for the descriptive analysis. The results show that although the minimum RD is 0, the

4. We used a different cluster to do sensitivity analysis. So the overall performance could be slightly different from those in Table 4.

5. https://github.com/ai-se/Reimplement/tree/cleaned_version

Table 10
Descriptive rank differences of 1000 tests

Job	Mean	Std	Min	Max
WordCount	13.2	24.4	0	408
Sort	28.7	42.6	0	391
TeraSort	14.3	19.1	0	171
NutchIndex	16.4	24.0	0	296
PageRank	9.5	16.7	0	158

average and maximum RDs are 13.2 and 408 respectively, and the standard deviation is 24.4. It means that this model could be largely wrong when trying to find high-performing configurations.

None of the approaches we evaluated guarantee to find truly optimal configurations. So we discuss which approach can find the best candidate from all configurations checked. As we see from Table 10, although a learning-based approach can find good configurations, it cannot guarantee the resulting one is the best. In some cases, the ranking mistake could be as large as 408. On the other hand, our sampling-based approach can accurately find the best thanks to the dynamic evaluation and guided sampling strategy.

How much performance improvement one can gain by using Nair *et al.*'s approach? While our final goal is to improve system performance, we studied which approach can find better configurations, concerning how much performance one can gain. Suppose an engineer wants to use their approach to find a good configuration. She knows that all learning-based approaches have prediction errors. One possible way is to run such a model multiple times to rank configurations and then find the one with the best ranking on average across all tries. In this paper, we modified the tool released by Nair et al. to get the predicted ranking of configurations. We ran the above-described procedure 20 times and find out the configuration with the highest rank in average. The last bar in Figure 6 shows the performance improvement of the rank-based approach *w.r.t.* the default configuration. CONEX performs 5.4% to $1,700\%$ better than the ranked-based approach across five Hadoop jobs.

To understand why Nair *et al.*'s approach doesn't perform well in finding good Hadoop configurations, we studied the accuracy of the trained models. In their implementation, the ranked-based model wraps a decision tree regressor as the under-hood performance prediction model. We checked the R^2 scores of these regressors, and it turns out that all scores are negative for all five jobs. It means that the trained model performs arbitrarily worse. This is not surprising because Hadoop's configuration space is complex, hierarchical, and high-dimensional; it is hard to learn a function approximating the objective function for such a space. A neural network-based regression model might work better. However, that would incur more sampling costs to gather adequate training samples.

Result: Compared to Nair et al's learning-based approach, our approach finds configurations with higher (from 5.4% to 1,700%) performance gains.

7 Related Work

The related work broadly falls under two categories: (i) tuning big-data systems, and (ii) tuning traditional software.

(i) Tuning big data framework.

Starfish [45] is one of the initial works on Hadoop auto-tuning. It tunes parameters based on the predicted results of a cost model. However, the success of such a model depends largely on the underlying cost model, which is often hard to build. Liao *et al.* [46] have already proved that the predicted results of Starfish's cost model could vary largely under different task settings. They used a vanilla GA with only six important parameters to identify high-value configurations and beat [45]. We empirically showed EMCMC strategy performs better than a GA based approach. We further selected all parameters related to performance tuning, as without knowing how parameters interact, we cannot exclude any relevant parameter. Another line of work by Babu et al. [47] tune MapReduce parameters by sampling data from actual production, and thus, they optimize a job given a cluster and a fixed workload. Yu [48] optimize in-memory cluster over various data set sizes using hierarchical modeling and genetic algorithms. In contrast, CONEX is more suitable to configure clusters where a diverse set of jobs are running under various sizes. Our assumptions are more generic and allow for learned configurations to be applied to a larger workload size (scale-up) and across similar jobs (scale-out).

A more recent line of research has explored multi-objective performance optimization focusing on performance goals such as throughput, latency, running time, etc. For example, Mishra *et al.* [49], propose LEO which uses an online probabilistic graphical model to learn a Pareto-optimal power/performance trade-off given a configuration. Compared to offline learning-based methods, LEO was demonstrated to perform better. More recently, Zhu *et al.* [50] proposed BestConfig that uses divide-and-diverge sampling in conjunction with recursive bound and search to perform a multi-objective performance optimization with the help of a utility function to combine multiple objectives. CONEX, on the other hand, is used to perform a search over a single objective and shows better performance gain (for runtime) compared to BestConfig on Hadoop's PageRank (the framework/job common to both the works).

(ii) Tuning Generic Software. A large body of research exists on configuring generic software that uses different sampling+learning strategies. The main challenge to apply them directly to our case is the cost of dynamic profiling at the scale of big-data and the complexity of the configuration space. Here we systematically summarize these related work.

Configuration Sampling. This step selects a subset of configurations based on different sampling strategies. For example, variations of random [12], [13], [25] sampling are used to draw configurations uniformly from the configuration space. We have shown that CONEX works much better than random sampling. Researchers also sampled test inputs targeting different regions [14], [51], or covering all the configurations satisfying some predefined constraints [15], [16], [17]. Kaltenecker *et al.* [18] further proposes a distance-based sampling to sample the whole space uniformly. Since big-data configuration space is quite complex and huge than

previously studied systems, partitioning the configuration space is challenging and will require a significant amount of dynamic traces. Further, uniform sampling from different regions may not be necessary if the configurations that will lead to better performance is sparse. Instead, EMCMC based sampling strategy theoretically can approximate the global configuration space, and we showed that the guided approach can help to find a near-optimal configuration. Sampling-based approaches often select invalid configurations [52]. To handle this problem, researches used constraint solvers to sample valid configurations [14], [53]. Instead, we used an off-the-shelf configuration constraint checker [35] (see Section 4.3).

Learning-based approaches. A large body of previous works estimates system performance by learning prediction models [7], [12], [19], [54], [55]. The accuracy of these models depends on the representativeness of training data. As shown in RQ5, for big-data systems, because of the complex high-dimensional configuration space, it is challenging to find a representative model. Also, collecting training samples is costly [21]. Existing logs from industrial uses of such systems are not necessarily useful as users tend to use the default, or at least very few, configuration settings [1].

Previous approaches also rely on the degree of parameter interactions. For example, Zhang et al. [9] assume all parameters are independent boolean variables and formulate the performance prediction problem as learning Fourier coefficients of a Boolean function. In contrast, Meinicke et al. [56] studied parameter interactions by running multiple configurations and comparing differences in control and data flow. They discovered that interactions are often less than expected but still complex enough to challenge search strategies. Siegmund et al. [8] learned how parameter interactions influence system performance by adding interaction terms in learning models. This approach combines domain knowledge and sampling strategies to generate effective training data. We have also seen evidence of parameter interactions in RQ2. However, unlike the predicting models, our search strategy is less affected by the parameter interactions as we have made no assumptions about such interactions. Thus, our work complements such previous efforts and present a novel search-based strategy for tuning big-data frameworks.

Other Applications. Many software engineering applications use sampling and optimization strategies in the past. For example, researchers used automated search to find valuable test cases [57], [58] and increase test coverage [59].

Weimer et al. [60] used genetic programming for program repair. Le [61] used MCMC techniques to generate program variants with different control- and data-flows. Whittaker and Thomason [62] used a Markov Chain model to generate test inputs to study software failure causes. Oh, et al. [25] worked to find good configurations for software product lines. Vizier [63] was developed at Google for optimizing various systems like machine learning models. Our work demonstrates the promise of similar approaches for the performance tuning of critical big-data systems.

There are some researches related to detecting software performance issues [64], [65]. However, finding a better configuration for performance improvement and identifying performance issues in software are orthogonal problems. Ţăpuş *et al.* [66] propose a distributed tool to optimize a system's resource utilization. We are interested in gaining higher performance given such resources.

8 Threats to Validity

Internal Validity. Threats to the internal validity arise from the experimental setup. First, the experimental results may be affected by uncontrolled factors on hardware platforms. In our experiments, we adopted some strategies to mitigate such unseen factors. For example, we make sure that no other programs are running while we are running experiments. We also choose a subset of all parameters to study with domain knowledge but we may have inadvertantly missed some important ones. To mitigate this threat, we referred to many previous works cited in this review paper [37] on Hadoop, and have included all parameters studied by other researchers in our parameter set. It's very expensive to run ConEx many times because of the nature of "big" data. Thus, running our algorithm enough times to get its statistic characteristics is an impossible mission. However, our two scale-up and scale-out hypotheses are created to mitigate this limitation. Although we didn't conduct a statistical analysis, we tested our method on multiple jobs at different scales to demonstrate that our method works.

External Validity. We report results only for two big-data frameworks namely, Hadoop and Spark framework. Our results may not generalize to other frameworks. That said, Hadoop and Spark are among the most widely used big-data frameworks in HiBench, and we believe that the results are representative in other settings.

For scale-out we choose 10 representative jobs. The findings of this work may not apply to other job types. However, this paper chooses a diverse set of job types operating in different domains, *e.g.*, `nutchindex` and `pagerank` are used in websearch; `svd` and `rf` are machine learning jobs; `sort` and `terasort` sort data; and finally `wordcount` operates on text data.

For the scale up hypothesis, our findings may vary for workloads that have significantly different memory consumption constraints and I/O overhead trade-offs. To address this threat we choose a diverse set of workloads with different memory consumption and I/O overheads. In particular, we choose a range of workload sizes that fit within our available hardware (Intel(R) Xeon(R) E5-2660 CPU, 32GB memory, and 2TB of SSD storage)—our smallest baseline workload (`small`) is 3MB and the largest workload (`huge`) is 3GB. Note that, the memory and I/O requirements for `small` and `huge` are vastly different. We also pick a diverse set of workloads. These diverse job types ensure that the scale-up may hold under different domains as well as under different workloads.

Finally, due to the nature of dynamic evaluation, the experimental results may be affected by uncontrolled factors on hardware platforms. In our experiments, we adopted some strategies to mitigate such unseen factors. For example, we make sure that no other programs are running on the experimental platform while we are running experiments. We also run each dynamic evaluation three times to get average performance as a final result.

Construct Validity. At various places in this paper, we made different engineering decisions, *e.g.*, the range of values for each configuration parameter (from Phase-I in §4), maximum number of generations (max_{gen}), etc. While these decisions were made using advice from the literature [37] or based on engineering judgments, we acknowledge that other constructs might lead to other conclusions.

Evaluation Bias. In RQ2 and RQ3, we have shown the performance of CONEX by comparing against Genetic Algorithms, Random Sampling, and Rank Based methods of Nair *et al.* to draw our conclusions. In choosing the comparisons, we chose those methods that (1) focus on single objective performance optimization, and (2) make available a replication package. The reported results hold for the software systems, evaluation metrics, and other performance optimization methods used for comparison in this paper. It is possible that with other Big-Data software systems and performance optimization methods methods, there may be slightly different conclusions. This is to be explored in future research.

9 CONCLUSIONS

In this work, we proposed an EMCMC-based sampling strategy in combination with scale-up and scale-out tactics to cost-effectively find high-performing configurations for big-data infrastructures. We conducted and have reported results from carefully designed, comprehensive, and rigorously run experiments. The data that emerged provides strong support for the hypothesis that our approach has strong potential to significantly and cost-effectively improve the performance of real-world big data systems. The data also strongly support the hypothesis that our approach outperforms several competing approaches.

In this work, we had a single scalar objective function for each system: reducing CPU time for Hadoop and wall-clock time for Spark. However, in reality, there might be tradeoffs between performance improvements and other constraints (*e.g.* cost). For example, user has to pay more money to Amazon EC2 for renting high performing systems. Whether techniques such as ours can be adapted to work in such situations remains a question for further study.

REFERENCES

[1] K. Ren, Y. Kwon, M. Balazinska, and B. Howe, "Hadoop's adolescence: an analysis of hadoop usage in scientific workloads," *Proceedings of the VLDB Endowment*, vol. 6, no. 10, pp. 853–864, 2013.

[2] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. Acm, 2015, pp. 307–319.

[3] Z. Jia, C. Xue, G. Chen, J. Zhan, L. Zhang, Y. Lin, and P. Hofstee, "Auto-tuning spark big data workloads on power8: Prediction-based dynamic smt threading," in *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*. Ieee, 2016, pp. 387–400.

[4] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[5] J. J. Louviere, D. Pihlens, and R. Carson, "Design of discrete choice experiments: a discussion of issues that matter in future applied research," *Journal of Choice Modelling*, vol. 4, no. 1, pp. 1–8, 2011.

[6] S. B. Joshi, "Apache hadoop performance-tuning methodologies and best practices," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ser. Icpe '12. New York, NY, USA: Acm, 2012, pp. 241–242. [Online]. Available: http://doi.acm.org/10.1145/2188286.2188323

[7] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *Proceedings of the 34th International Conference on Software Engineering*, ser. Icse '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 167–177.

[8] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. Acm, 2015, pp. 284–294.

[9] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki, "Performance prediction of configurable software systems by fourier learning (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, ser. ASE '15, Ieee. Washington, DC, USA: IEEE Computer Society, 2015, pp. 365–373.

[10] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: efficient performance prediction for large-scale advanced analytics," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 363–378.

[11] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-efficient sampling for performance prediction of configurable systems (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. Ieee, 2015, pp. 342–352.

[12] J. Guo, K. Czarnecki, S. Apely, N. Siegmundy, and A. Wasowski, "Variability-aware performance prediction: A statistical learning approach," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2013, pp. 301–311.

[13] V. Gogate and R. Dechter, "A new algorithm for sampling csp solutions uniformly at random," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2006, pp. 711–715.

[14] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, "Distribution-aware sampling and weighted model counting for sat," in *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[15] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "Ipog/ipog-d: efficient test generation for multi-way combinatorial testing," *Software Testing, Verification and Reliability*, vol. 18, no. 3, pp. 125–148, 2008.

[16] M. F. Johansen, Ø. Haugen, and F. Fleurey, "An algorithm for generating t-wise covering arrays from large feature models," in *Proceedings of the 16th International Software Product Line Conference-Volume 1*. Acm, 2012, pp. 46–55.

[17] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu, "Practical pairwise testing for software product lines," in *Proceedings of the 17th international software product line conference*. Acm, 2013, pp. 227–235.

[18] C. Kaltenecker, A. Grebhahn, N. Siegmund, J. Guo, and S. Apel, "Distance-based sampling of software configuration spaces," in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE). ACM*, 2019.

[19] V. Nair, T. Menzies, N. Siegmund, and S. Apel, "Using bad learners to find good configurations," *ArXiv e-prints*, Feb. 2017.

[20] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel, "Finding faster configurations using flash," *IEEE Transactions on Software Engineering*, 2018.

[21] G. M. Weiss and Y. Tian, "Maximizing classifier utility when there are data acquisition and modeling costs," *Data Mining and Knowledge Discovery*, vol. 17, no. 2, pp. 253–282, 2008.

[22] T. Ye and S. Kalyanaraman, "A recursive random search algorithm for large-scale network parameter configuration," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1. ACM, 2003, pp. 196–205.

[23] H. Ha and H. Zhang, "Deepperf: Performance prediction for configurable software with deep sparse neural network," in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE). ACM*, 2019.

[24] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. Acm, 2009, pp. 165–178.

[25] J. Oh, D. Batory, M. Myers, and N. Siegmund, "Finding near-optimal configurations in product lines by random sampling," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. Acm, 2017, pp. 61–71.

[26] C. Ritter and M. A. Tanner, "Facilitating the gibbs sampler: the gibbs stopper and the griddy-gibbs sampler," *Journal of the American Statistical Association*, vol. 87, no. 419, pp. 861–868, 1992.

[27] W. K. Hastings, "Monte carlo sampling methods using markov chains and their applications," *Biometrika*, vol. 57, no. 1, pp. 97–109, 1970.

[28] J. Goodman and J. Weare, "Ensemble samplers with affine invariance," *Communications in applied mathematics and computational science*, vol. 5, no. 1, pp. 65–80, 2010.

[29] M. M. Tibbits, C. Groendyke, M. Haran, and J. C. Liechty, "Automated factor slice sampling," *Journal of Computational and Graphical Statistics*, vol. 23, no. 2, pp. 543–563, 2014.

[30] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *The journal of chemical physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[31] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[32] R. Akbari and K. Ziarati, "A multilevel evolutionary algorithm for optimizing numerical functions," *International Journal of Industrial Engineering Computations*, vol. 2, no. 2, pp. 419–430, 2011.

[33] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.

[34] M. Srinivas and L. M. Patnaik, "Genetic algorithms: A survey," *computer*, vol. 27, no. 6, pp. 17–26, 1994.

[35] C. Tang, K. Sullivan, J. Xiang, T. Weiss, and B. Ray, "Interpreted formalisms for configurations," *arXiv preprint arXiv:1712.04982*, 2017.

[36] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of machine learning research*, vol. 3, no. Mar, pp. 1157–1182, 2003.

[37] A. S. Bonifacio, A. Menolli, and F. Silva, "Hadoop mapreduce configuration parameters and system performance: a systematic review," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2014, p. 1.

[38] A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar, "Scalable product line configuration: A straw to break the camel's back," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. Ieee, 2013, pp. 465–474.

[39] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy *et al.*, "The coq proof assistant reference manual: Version 6.1," Ph.D. dissertation, Inria, 1997.

[40] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*. Ieee, 2010, pp. 41–51.

[41] N. Khadke, M. P. Kasick, S. Kavulya, J. Tan, and P. Narasimhan, "Transparent system call based performance debugging for cloud computing." in *Mad*, 2012.

[42] E. Yoon and A. Squicciarini, "Toward detecting compromised mapreduce workers through log analysis," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. Ieee, 2014, pp. 41–50.

[43] M. P. Kasick, K. A. Bare, E. E. Marinelli III, J. Tan, R. Gandhi, and P. Narasimhan, "System-call based problem diagnosis for pvfs," Carnegie-mellon Univ Pittsburgh Pa Dept Of Electrical And Computer Engineering, Tech. Rep., 2009.

[44] V. Nair, T. Menzies, N. Siegmund, and S. Apel, "Faster discovery of faster system configurations with spectral learning," *Automated Software Engineering*, pp. 1–31, 2017.

[45] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: a self-tuning system for big data analytics." in *Cidr*, vol. 11, no. 2011, 2011, pp. 261–272.

[46] G. Liao, K. Datta, and T. L. Willke, "Gunther: Search-based autotuning of mapreduce," in *European Conference on Parallel Processing*. Springer, 2013, pp. 406–419.

[47] S. Babu, "Towards automatic optimization of mapreduce programs," in *Proceedings of the 1st ACM symposium on Cloud computing*. Acm, 2010, pp. 137–142.

[48] Z. Yu, Z. Bei, and X. Qian, "Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 564–577.

[49] N. Mishra, H. Zhang, J. D. Lafferty, and H. Hoffmann, "A probabilistic graphical model-based approach for minimizing energy under performance constraints," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 267–281, 2015.

[50] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "Bestconfig: tapping the performance potential of systems via automatic configuration tuning," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 338–350.

[51] T. Y. Chen, H. Leung, and I. Mak, "Adaptive random testing," in *Annual Asian Computing Science Conference*. Springer, 2004, pp. 320–329.

[52] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-oriented software product lines*. Springer, 2016.

[53] C. Henard, M. Papadakis, M. Harman, and Y. Le Traon, "Combining multi-objective search and constraint solving for configuring large software product lines," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 517–528.

[54] S. Apel, A. Von Rhein, T. ThüM, and C. KäStner, "Feature-interaction detection based on feature-based specifications," *Computer Networks*, vol. 57, no. 12, pp. 2399–2409, 2013.

[55] P. Jamshidi, M. Velez, C. Kästner, N. Siegmund, and P. Kawthekar, "Transfer learning for improving model predictions in highly configurable software," in *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press, 2017, pp. 31–41.

[56] J. Meinicke, C.-P. Wong, C. Kästner, T. Thüm, and G. Saake, "On essential configuration complexity: measuring interactions in highly-configurable systems," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. Acm, 2016, pp. 483–494.

[57] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, 2009.

[58] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, 2004.

[59] S. Park, B. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie, "Carfast: Achieving higher statement coverage faster," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. Acm, 2012, p. 35.

[60] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Software Engineering (ICSE), 2012 34th International Conference on*. Ieee, 2012, pp. 3–13.

[61] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *ACM SIGPLAN Notices*, vol. 50, no. 10. Acm, 2015, pp. 386–399.

[62] J. A. Whittaker and M. G. Thomason, "A markov chain model for statistical software testing," *IEEE Transactions on Software engineering*, vol. 20, no. 10, pp. 812–824, 1994.

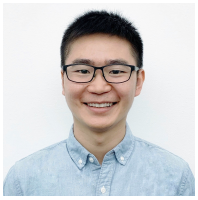
[63] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Acm, 2017, pp. 1487–1495.

[64] Q. Luo, A. Nair, M. Grechanik, and D. Poshyvanyk, "Forepost: Finding performance problems automatically with feedback-directed learning software testing," *Empirical Software Engineering*, vol. 22, no. 1, pp. 6–56, 2017.

[65] C. Yilmaz, A. S. Krishna, A. Memon, A. Porter, D. C. Schmidt, A. Gokhale, and B. Natarajan, "Main effects screening: a distributed continuous quality assurance process for monitoring performance degradation in evolving software systems," in *Proceedings of the 27th international conference on Software engineering*. Acm, 2005, pp. 293–302.

[66] C. Ţăpuş, I.-H. Chung, J. K. Hollingsworth *et al.*, "Active harmony: Towards automated performance tuning," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 2002, pp. 1–11.

Rahul Krishna is a post doctoral researcher in Computer Science at Columbia University. He received his Ph.D. in NC State University. His current research explores ways to use machine learning to generate actionable insights for building reliable software systems. His other research interests include program analysis, artificial intelligence, and security. See http://rkrsn.us for more details.



Chong Tang is a senior software engineer at Microsoft. He has received his Ph.D. degree from the University of Virginia. Chong's research interests include Software Synthesis, Search-based Software Engineering, and Software Analysis.

Kevin Sullivan is an Associate Professor in the Department of Computer Science at the University of Virginia. His research activities span software engineering, systems engineering, and cyber-social systems. His current research aims to establish foundations for the definitions of physical semantics for cyber-physical systems (CPS) code grounded in the formalization of mathematical physics within a modern constructive logic proof assistant and in the mapping of everyday CPS code to such formalized mathematics. Kevin has his Ph.D. and M.S in Computer Science and Engineering from the University of Washington and his undergraduate degree from Tufts University.

Baishakhi Ray is an Assistant Professor in the Department of Computer Science, Columbia University, NY, USA. She has received her Ph.D. degree from the University of Texas, Austin. Baishakhi's research interest is in the intersection of Software Engineering and Machine Learning. Baishakhi has received Best Paper awards at FASE 2020, FSE 2017, MSR 2017, IEEE Symposium on Security and Privacy (Oakland), 2014. Her research has also been published in CACM Research Highlights and has been widely covered in trade media. She is a recipient of the NSF CAREER award, VMware Early Career Faculty Award, and IBM Faculty Award.