ConEx: Searching Configuration Spaces to Improve the Performance of Big Data Systems

Chong Tang
University of Virginia
Charlottesville, VA USA
ct4ew@virginia.edu

Kevin Sullivan
University of Virginia
Charlottesville, VA USA
sullivan@virginia.edu

Baishakhi Ray
Columbia University
New York City, New York USA
rayb@cs.columbia.edu

***Abstract*—Configuration space complexity makes many modern software systems hard to configure well. Consider Hadoop as an example, with over nine hundred configuration parameters, developers often just use the *default* configurations provided with Hadoop distributions. The opportunity costs are substantial. We present a novel method based on *Evolutionary Markov Chain Monte Carlo (EMCMC)* techniques to find better configurations. The challenge is to manage the cost of sampling, which involves dynamic profiling of big data jobs under different configurations. We developed and evaluated two approaches, each of which we validated experimentally. First, we use scaled-down big data problems as proxies for sampling. Second, we use a similarity measure to infer that results obtained for one kind of problem will probably work well for similar problems. Our experimental results suggest that our approach promises to significantly improve the performance of big data systems 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 has strongly demonstrated potential to significantly and cost-effectively improve the performance of big data systems.**

***Index Terms*—configuration, metaheuristic, optimization, MCMC, evolutionary algorithm**

I. INTRODUCTION

Many software systems are highly configurable and can be tailored to meet different needs in various environments. However, in practice, users often find it hard to understand how to take advantages of rich configurability, and just use pre-packaged or *default* configurations [1]. As a result, they leave significant performance potential unrealized. Configuring a system to achieve better performance is important [2], in particular, for big data systems because "even a small percentage of performance improvement immediately translates to huge cost savings because of the large scale" [3].

However, automatically finding high-performing configurations for big data systems are often challenging because: (i) A system usually contains many configurable sub-systems, which increase the complexity of the whole configuration space [4]. For example, Hadoop has around 900 parameters across four sub-systems: CORE, HDFS, MAPREDUCE, and YARN. (ii) The configuration parameters can have fields with diverse types, as well as optional and dependent substructures. For example, setting one Boolean parameter to *true* can enable an entire subsystem, requiring many more configuration values. Further, some parameters, especially the numeric ones, can have thousands of alternative values. (iii) The parameters can be parameterized by external constraints: e.g., one cannot set a Hadoop *number-of-CPUs* parameter to a number larger than the available number of CPUs. (iv) Finally, analysts often do not know key properties of configuration-to-performance functions: e.g., how parameters interact, whether the functions are convex, linear, and differentiable, etc. For such complex and discrete problems, standard mathematical optimization methods are known not to apply well [5]. Therefore, finding good configurations ends up as a "black-art" [6].

Finding high-performing configurations for *traditional software systems* is known to be hard and has given rise to a significant body of work [7], [8], [9], [10], [11], [12]. Much work focuses on learning generalized models to predict a given configuration's performance. To learn such a model, one must profile a system under many configurations to sample the objective function under widely varying conditions. However, the cost of sampling real-world systems are usually high [13] and they often are not applicable to complex configurable systems [14]. To address these issues, Nair *et al.* [14] proposed a rank-based model. Instead of predicting the performance of a configuration, they answered whether one configuration is better than others. They showed that such a rank-based technique requires fewer samples than residual-based models (their main target is to minimize mean squared error).

The configuration-finding problem for big data systems involves much higher-dimensional spaces than addressed in most previous work. For these systems it is hard to learn even rank-preserving models with high accuracy (see in Section V). One might try Neural Networks (NNs). However, NNs require large amounts of data for training, and the cost of collecting such data for big-data systems would be very high [15]. Thus, instead of finding a near optimal configuration like previous works, we reduce the problem as:

- Given a limited computation budget, find a *good configuration* that achieves better performance than others.
- Find the good configuration with *as little cost as possible*.

Our idea is that learning accurate predictive models over entire configuration spaces might be unnecessary if one's aim is just to find *good* configurations—that perform better than those engineers use today. Our approach is to use direct meta-heuristic search (*e.g.* Markov Chain Monte Carlo (MCMC), genetic algorithms (GA)) of configuration spaces augmented by two search-cost-reduction techniques to find good configurations cost-effectively. Such models tend to converge more directly on good configurations through guided search around seed configurations, without the need to learn generalized models over the entire configuration space. In this paper we present results for such an approach based on Evolutionary Markov Chain Monte Carlo (EMCMC) methods, in particular.

The need to reduce the search costs is essential in our case. For example, it takes one thousand runs of a production-scale big data job—perhaps one run once a day—for a search to find a configuration that yields a ten percent reduction in runtime costs. It would then take ten thousand production runs, or more than twenty-five years, just to break even. Multiple orders of magnitude of cost reduction would be needed for such a search method to be practical.

To this end, we devised, implemented, and evaluated two ideas. First, we hypothesized that jobs that are one hundred times smaller than those used in production can be used as proxies for sampling configuration-to-performance objective functions and that improvements found using these smaller jobs would still hold for much larger ones. We call this our *scale-up* hypothesis. Second, we hypothesized that better configurations for one kind of job would yield improvements for *dynamically similar* jobs, entirely avoiding the need for redundant searches. We call this as our *scale-out* hypothesis.

We conducted experiments using the Hadoop and Spark big data frameworks as experimental testbeds, and five canonical kinds of big data problems defined by the HiBench [16] benchmark suite. For each kind of problem, HiBench provides a range of job sizes. We measured improvements by comparing CPU time under default (experimental control) configurations with (treatment) configurations found using (a) our approach, and (b) three competing alternatives. The evaluation results showed that our approach gains performance from 14.2% to 25.2% for tested jobs. It outperforms a same-cost *random* search by 17% to 125%, a basic GA by 6% to 85%, and the learning-based approach of Nair et al. from 5% to 1700%.

These improvements, combined with an assumed 100X reduction in search costs based on our scale strategy suggest that, our approach could produce break-even benefits for daily big data jobs in just a few months. This ignores the possibility that results from one search could produce immediate benefits for similar jobs. Our conclusion then, and the main contribution of this work, is that idea, supported by experimental evidence, the combining evolutionary Markov Chain Monte Carlo meta-heuristic search techniques with scale-up and scale-out cost-reduction methods has strongly demonstrated potential to produce industrially important performance benefits to users of big data systems.

II. Background

This section presents the relevant background. First, we define some key terms that we use in this work.

Definition II.1. *Configuration Parameter (p_i).* A configuration parameter is a variable, the value of which is set by the installer and/or user to specify how to configure a particular property of a system.

Definition II.2. *Configuration (c).* A *configuration tuple*, or simply a *configuration*, $c = [p_0, \ldots, p_N]$, is an N tuple, where each element p_i is a configuration parameter and N is the total number of parameters, i.e., the dimensionality of the space.

Definition II.3. *Configuration Space (ζ).* A configuration space, $\zeta = \{c|valid(c)\}$, is the set of all valid configuration tuples for a given system. The definition of $valid$ varies from system to system. If there are no constraints on what it means to be valid, and if N is the number of parameters and M is the average number of values of each parameter, then the size of ζ will be roughly M^N.

Parameter	Tuple_1	Tuple_2
dfs.blocksize	3	2
mapreduce.job.ubertask.enable	FALSE	True
mapreduce.map.java.opts	-Xmx1024m	-Xmx2048m
...	...	...
mapreduce.reduce.shuffle.merge.percent	0.66	0.75

The above table shows two sample configurations for Hadoop. We list only four parameters due to space limitations. In practice, configurations have many parameters, of varying types: boolean, integer, categorical, string, etc. For parameters with many values (e.g., integer- or string-valued parameters), the size of the configuration space can be vast even when there are only a few parameters. Table I summarizes the configuration space of Hadoop and Spark.

Table I: **Configuration Space Characteristics**

System	Total Parameters	Studied Parameters						Total Configurations
		Total	Bool	Int	Float	Categorical	String	
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	$4 * 10^{16}$

Note that, although we have studied a small subset of the total available parameters, the studied configuration space is still huge.

A. Heuristic Optimization

A promising approach to find high-performing configuration can be based on meta-heuristic search and sampling, as shown by Oh *et al.* [17] for software product lines. Since our goal is to find a *good enough* configuration within a given computation budget, the choice of heuristic optimization strategy becomes important for faster convergence. There are many optimization techniques one could try. Gradient descent, for example, is useful if error functions are differentiable, but that is not the case here. Coordinate descent and other derivative-free techniques, e.g., stochastic cyclic coordinate descent [18], are often useful, but can get stuck in local optima [19], as we saw when we tried these methods in our domain. Having explored a range of such methods, this work has adopted an approach based on MCMC methods, as explained below.

B. Markov Chain Monte Carlo (MCMC)

Monte Carlo is a sampling method to draw samples from a probability distribution $P(.)$ defined on a high dimensional space [20]. Markov Chain assumes that a configuration c_t, given all its previous instances $\{c_0, c_1, ..., c_{t-1}\}$, only depends on its immediate neighbour c_{t-1}, *i.e.* $P(c_t|c_0,c_1,...,c_{t-1})$=$P(c_t|c_{t-1})$. MCMC combines the two methods; since the target distribution is unknown, MCMC operates on a proposal distribution. Given a configuration c_i, MCMC draws a neighbor c^* from the proposal distribution and evaluates its fitness value (i.e., better performance). Then, it decides to accept/reject this neighbor based on the acceptance probability of the fitness value. The acceptance probability can be computed using widely used Metropolis algorithm [21]:

$$A(c_{new}|c_{curr}) = min(1, \frac{P(c_{new})}{P(c_{curr})}) \tag{1}$$

If a neighbor is accepted, MCMC transits to the next state (c_{new}) from current state (c_{curr}). Thus, MCMC repeatedly samples a candidate next state, c_{new}, from a proposal distribution, accepts it with an acceptance probability, and repeats until a specified computing budget is reached [22]. Note that, due to the probabilistic nature of acceptance, it can accept some samples with worse fitness than c_{curr}, introducing diversity that mitigates problems due to noise and local minima.

As the number of accepted samples increases, the MCMC sampling distribution approaches the target configuration distribution. A good MCMC algorithm is designed to spend most of the time in the high-density region of the target distribution. Thus, MCMC is often used to sample approximate global optima. A detailed description of MCMC can be found in [23].

C. Evolutionary MCMC (EMCMC)

MCMC algorithms use a single Markov chain, often initially discarding most candidate states, thus converging slowly [20]. The *Evolutionary-MCMC* algorithm addresses this problem. Like evolutionary (e.g., genetic) algorithms (GA) [24], [25], [26], EMCMC starts with a population of $N > 1$ states, selects a subset of high fitness states, and then obtains a population of states for the next round by applying mutation and cross-over operations to the selected states.

Cross-over operations work by randomly selecting parent configurations, c_i and c_j. Each configuration is divided in 2-parts (1-point cross-over). Then the first part of c_i is mixed with the second part of c_j and vice versa, generating two offspring configurations. Given a configuration created by the cross-over operation, c_k, the mutation operation updates the value of some random parameters to generate a new one, c_k^*. A key difference between ordinary GAs and an EMCMC approach is that in GAs, only strictly better states are accepted, whereas EMCMCs can accept slightly worse states, as explained previously.

III. Technical Approach

Our approach is to use EMCMC algorithms to sample Hadoop and Spark configuration spaces in search of high-performing configurations. To reduce the cost of, or even need for, costly EMCMC runs, we employ two additional tactics.

1) **Scale-up**: We run Hadoop and Spark using inputs that are 100X smaller than those that we want to run in production for sampling performance as a function of configuration.

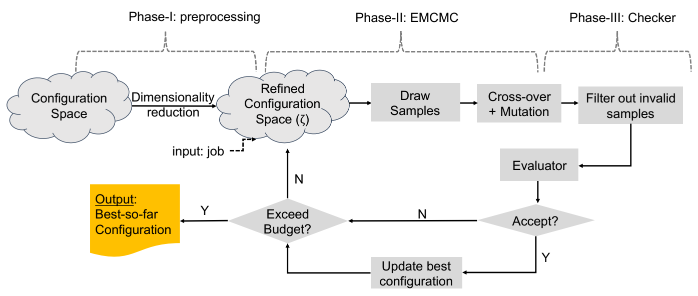

Figure 1: **ConEx Workflow**

2) **Scale-out**: We use a measure of the similarity of big data jobs to enable what amounts to a kind of transfer learning, wherein good configurations for one job are used for another without change.

We have implemented our approach in a tool called ConEx. The rest of this section describes our approach in detail.

Overview

Figure 1 presents an overview of ConEx. ConEx takes a big data job as input and outputs the best configuration it found before exceeding a search cost budget. ConEx works in three phases. In Phase-I, it eliminates configuration parameters 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.* [27] (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 these with those of best configuration seen so far, updating the latter if necessary, as per our acceptance criterion (see Equation (1)). Accepted configurations are subjected to cross-over and mutation 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.

Phase-I: Pre-processing the configuration space

The total number of configuration parameters for Hadoop and Spark are 901 and 212 respectively (see Table I). However, the majority are unrelated to performance. Also, all values of the related parameters are not equally relevant when searching for good results. Thus, we reduce the dimensionality of the search space in two ways: (i) we consider only the parameters relevant to performance ,and (ii) for each selected parameters, we select only a few values for sampling.

The first part is manual; we referred to official document and other related works [28]. For example, we removed parameters related to version, file paths, authentication, server-address, and ID/names. These parameters have negligible impacts on the CPU time. For Spark, we selected parameters related to the runtime environment, shuffle behavior, compression and serialization, memory management, execution behavior, and scheduling. In this way, we select 44 and 27 parameters to study further. Table I summarizes these parameters.

Note that, although this step significantly reduces the number of configurations, since some of the parameter domains are large (*e.g.* integer, float, string), the resulting configuration space is still huge: $3 * 10^{28}$ and $4 * 10^{16}$ for Hadoop and Spark respectively. Thus, even the reduced configuration space is several orders of magnitude larger than those studied in previous work. For example, most systems studied by Nair et al. [14] only have thousands of configurations, with only a few systems like SQLite have millions of configurations.

We further discretize the configuration space by defining sampling 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. String parameters (*e.g.*, as Java virtual machine settings) are sample from all valid values.

Phase-II: Finding better configurations

This phase is driven by an EMCMC search strategy and implemented by Algorithms 1 to 3. 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 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 the best configuration found ($conf_{best}$) and its performance ($perf_{best}$).

Algorithm 1: Explore Configuration Space with EMCMC

```
1  Function EMCMC()
     Input  : Refined Configuration Space $\zeta$,
              job,
              seed configuration $conf_{seed}$,
              threshold max_gen
     Output: Best configuration $conf_{best}$,
              Corresponding performance $perf_{best}$
2  $perf_{seed} \leftarrow$ run $job$ with $conf_{seed}$
3  $conf_{best} \leftarrow conf_{seed}$
4  $perf_{best} \leftarrow perf_{seed}$
5  $generation \leftarrow 1$
6  $\Delta perf \leftarrow 0$
7  $conf_{parents} \leftarrow$ sample $n$ random configurations from $\zeta$
8  do
9      $confs_{accepted} \leftarrow EmptyList$
10     foreach parent $conf_p \in conf_{parents}$ do
11         $perf_p \leftarrow$ run $job$ with $conf_p$ configuration
12         $accepted \leftarrow$ Accept($perf_{best}, perf_p$) # based on Eq. 1
13         if accepted then
14             $confs_{accepted}.add(conf_p)$
15             if $perf_p > perf_{best}$ then
16                 $conf_{best} \leftarrow conf_p$
17                 $perf_{best} \leftarrow perf_p$
18             end
19         end
20     end
21     $\Delta perf \leftarrow (perf_{best} - perf_{seed})/perf_{seed}$
22     $conf_{parents} \leftarrow evolve(conf_{best}, confs_{accepted})$
23     $generation \leftarrow generation + 1$
24 while generation < max_gen;
25 return $conf_{best}, perf_{best}$
```

Lines 2 to 6 initialize some parameters including setting the best configuration and performance to the respective seed values. Line 7 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 10 to 20 are the main procedure for evaluating and evolving *w.r.t.* each configuration. Given a configuration $conf_p$, Line 11 records the job's performance ($perf_p$) and Line 12 decides whether to accept it based on Equation (1). This function is implemented in Algorithm 3. If accepted, Line 14 stores the accepted configuration to a list $confs_{accepted}$, which later will be used in generating next-generation configurations (Line 22). If the accepted one is better than the best previously found, Lines 15 to 18 update the state accordingly.

Once all configurations in the first generation are processed, Line 21 computes the performance improvement achieved by this generation *w.r.t.* the seed performance. Next in Line 22, the algorithm prepares to enter the next generation by generating offspring configurations using cross-over and mutation operations (see Algorithm 2). Line 23 updates the generation number. This process repeats until the termination criterion is satisfied (Line 24). Finally, the last line returns the best found configuration and the corresponding performance.

Algorithm 2: The evolutionary sub-routine of EMCMC

```
1  Function Evolve
     Input  : $conf_{best}, confs_{accepted}$
     Output: $conf_{children}$
2  $conf_{children} \leftarrow EmptyList$
3  $P_{crossover} \leftarrow$ randomly select 50% parameters from $conf_{best}$
4  $P_{mutate} \leftarrow$ randomly select 6% parameters of $conf_{best}$
5  foreach $conf_p \in confs_{accepted}$ do
6      $conf_{new} \leftarrow crossover(conf_{best}, conf_p, P_{crossover})$
7      $conf_{new} \leftarrow mutate(conf_{new}, P_{mutate})$
8      $conf_{children}.add(conf_{new})$
9  end
10 return $conf_{children}$
```

Algorithm 2 is the evolution sub-routine of the EMCMC algorithm. It is adapted from a standard Genetic Algorithm [24]. 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 the cross-over parameters with $P_{crossover}$. 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 3 computes acceptance probabilities as described in Equation (1). It takes two performances as input: a current candidate under test and the best found so far, and returns whether to accept/reject the current candidate. First, Line 2 computes the performance improvement ($\Delta perf$) between the two. If the current configuration is worse than the best value, $\Delta perf$ will be negative. Next, Line 3 computes the acceptance probability of the configuration using an exponential function, which always returns a positive value even if $\Delta perf$ is

Algorithm 3: Acceptance sub-routine of EMCMC

```
1 Function Accept
  Input  : $perf_{best}, perf_p$
  Output: bool: accept/reject $conf_p$
2 $\Delta perf \leftarrow \frac{perf_{best} - perf_p}{perf_{best}}$
3 $accept\_prob \leftarrow exp(50 * \Delta perf)$
4 if $Rand(0,1) < accept\_prob$ then
5     return True
6 end
7 return False
```

negative. Finally, the acceptance probability is compared with a random number sampled between 0 and 1 exclusively. Thus, even if a configuration is slightly worse than the best one, it still has some chance of being accepted.

Phase-III: Configuration Validity Checking

Configuration spaces are often not complete cross-product spaces. Rather, they are subsets defined by constraints on parameter values. A problem we encountered is that validity constraints on Hadoop and Spark configurations are not well documented, nor statically enforced. For example, the type of Hadoop's *JVM options* parameter is *string*, but not any string will do. The lack of constraint documentation and enforcement makes it easy to make configuration mistakes and also vastly increases search space sizes. To reduce the search cost, CONEX leverages a configuration checker developed by Tang *et al.* [27]. It allowed us to avoid costly dynamic profiling of about 8% of all of the configurations that we sampled.

IV. EXPERIMENTAL DESIGN

We implement CONEX with about 4000 lines of Python code. The tool is available in a public Github repository: https://www.github.com/username/blind.

A. Study Subject and Experimentation Platform

We developed CONEX by focusing on Hadoop[4] and used Spark[29] for validating the approach. Hadoop and Spark are the two most popular big data frameworks today. The main difference between them is that Spark uses memory as much as possible in lieu of mass storage to reduce execution times.

Table I summarizes the parameters and their types that we have studied for Hadoop $v2.7.4$ and Spark $v2.2.0$. Hadoop has 901 parameters across four sub-systems. After Phase-I, we identified 44 parameters relevant to performance. Similarly, 27 out of 212 Spark parameters are selected. These parameters, in total, produces about $3 * 10^{28}$ and $4 * 10^{16}$ configurations respectively. Such configuration spaces are several orders of magnitude larger and complex than those studied before [14]. To further reduce the search space, for numeric parameters, we select $\pm 10\%$ values around the default and all valid values for boolean, string, and categorical types.

To evaluate CONEX, we select big-data jobs from HiBench [16], a popular benchmark suite for big data framework evaluation. It provides benchmark jobs for both Hadoop and Spark. For Hadoop, we selected five jobs: WordCount, Sort, TeraSort, PageRank, and NutchIndex from the Micro and Websearch categories. These jobs only need a core Hadoop system to execute as opposed to other categories that require Hive or Spark. For Spark, we selected five Spark jobs: WordCount, Sort, TeraSort, RF, and SVD. HiBench has six different sizes of input workload, from "tiny" to "Bigdata". For our experiments, we used "small", "large", and "huge" data inputs. Table V shows the CPU times taken by the Hadoop jobs running with default configurations. As the running time of the larger jobs much higher compared to the small jobs, we experimented with smaller jobs and validated with the larger workloads (*Scale-up* hypothesis).

We conducted our experiments in our in-house Hadoop and Spark clusters having one master node and four slave nodes. Each node has a Intel(R) Xeon(R) E5-2660 CPU and 32GB memory. 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.

B. Job Classification

To reduce sampling cost, we proposed *scale-out* hypothesis, *i.e.* configurations found to be good for one kind of job might also work well for other jobs with similar resource usage pattern. To test this hypothesis we need a way to cluster jobs by their resource usage. HiBench classifies jobs by their business purposes (*e.g.* web page indexing and ranking related jobs fall in the *Websearch* category). However, such a classification does not necessarily reflect their true resource usage. We thus developed an approach to clustering jobs by profiling their run-time behaviors based on system call traces. Similar approaches have been widely used in the security community [30], [31], [32].

A Unix command line tool `strace` captures system call traces of a process, typically running on a single machine, and logs how the process has used system resources. We configure Hadoop and Spark to run on the single-node model; here, a task runs as a single Java process. We then collect call traces of all studied jobs.

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

Table II: **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	$\{foo : ["1^{st} arg" : 2.0]\}$

To compute similarity between two jobs, we calculate the similarities between each tuple-element separately, which contributes equally to the overall similarity estimation. For tuple elements A, we use pattern matching—we slice the

Table III: **Performance improvement for Hadoop jobs from three sampling strategies**

	EMCMC			**Genetic Algorithm**			**Random Sampling**		
	Exploration Phase	**Scale-up Phase**		**Exploration Phase**	**Scale-up Phase**		**Exploration Phase**	**Scale-up Phase**	
Job	**Small**	**Large**	**Huge**	**Small**	**Large**	**Huge**	**Small**	**Large**	**Huge**
WordCount	12.5%	15.6%	21.5%	6.5%	9.8%	11.6%	5.6%	9.7%	14.6%
Sort	72.1%	7.7%	15.8%	70.6%	11.4%	10.3%	60.6%	5.3%	13.5%
TeraSort	27.4%	16.1%	18.3%	21.6%	17.2%	15.9%	11.6%	8.6%	11.1%
PageRank	32.7%	44.7%	25.2%	46.4%	17.5%	21.2%	32.9%	12.0%	11.2%
NutchIndex	7.1%	18.7%	14.2%	5.7%	11.5%	13.4%	10.4%	14.4%	12.0%

call sequences and compute the similarity between them. To find similarity between two 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).

C. Comparing with Baselines

We compare ConEx's performance with three baselines: (i) Random sampling, (ii) Genetic algorithm based evolutionary sampling, and (iii) a cutting-edge learning-based approach. The first two evaluate whether EMCMC is a good sampling strategy over other sampling strategies. The last baseline evaluates the choice of meta-heuristic search over popular learning-based approach. In particular, we compare ConEx with Nair *et al.*'s ranking based approach [33] as they showed that their rank-based approach requires fewer samples over other residual-based approaches.

V. Experimental Results

We evaluated ConEx using Hadoop jobs. To test the generalizability of our approach, we also tested it experimentally using the Spark framework. We start our experiments with the basic question:

RQ1. Can ConEx find better configurations than the baseline configuration within a given computation budget?

We investigate this RQ by exploring the configuration space using small workloads. First, ConEx runs the benchmark jobs by setting the workload size "Small" in the HiBench configuration file. HiBench generates a detailed report after each run, with diverse performance information including CPU time. For Spark jobs, we use larger dataset instead of the small workload because spark jobs run very fast under small workload—it is difficult to observe any meaningful performance gain. Thus, we use a larger workload for exploration, which is about the same size as "large" workload defined in HiBench, tailored to take about 30 seconds per run, making the cost of exploration comparable to that of Hadoop. Each execution in HiBench contains two steps: data preparation and job execution. However, we manually prepare the data so that each job under test can be run with the same workload. Hence, we modify HiBench to bypass its first step and run jobs with our data. Given performance for the default and best discovered configurations, denoted as $perf_d$ and $perf_b$, we characterized the performance improvement produced by a search as $\frac{perf_d - perf_b}{perf_d}$.

Table III shows the results: configurations found by ConEx with EMCMC approach achieve 7% to 72% better performances than default configurations for five Hadoop jobs. For Spark jobs, ConEx finds 2.7% to 40.4% performance improvements for all five jobs (see "Exploration" column of Table IV). The highest improvement is for TeraSort (40.4%).

Result 1: *For Hadoop and Spark jobs with small workload,* ConEx *can find configurations that produce up to 72% and 40% performance gains respectively over the default configurations.*

Even if ConEx manages to find a better configuration with small workloads, ConEx will be most effective if it can improve performance for larger workloads and thus save significant cost. This leads us to question:

RQ2. Scale-Up: Do configurations found with small workloads also produce significant improvements in performance for much larger workloads?

Here, we run the same HiBench jobs as used in RQ1 with "Large" and "Huge" inputs—10X and 100X times larger workloads respectively (*i.e.* more than 3-GB and 30-GB inputs for jobs in the "Micro" category). For a given job, we choose top 50 best performing configurations from RQ1 and profile them along with the default configuration and record the CPU times. We then compare the performance gains *w.r.t.* the baseline performance.

Table III presents the results for Hadoop jobs. For all five jobs, we see an average of 20.6% and 19% improvements under large and huge workloads. In fact, for WordCount, PageRank, and NutchIndex, large workloads achieve better performance gain than the improvement under the small ones.

We note that running fifty production-scale jobs as a final step of our approach would significantly increase its costs and push out the point of break-even by many production runs. We therefore conducted a limited sensitivity analysis, plotting performance gains for each of the top fifty jobs. What we found was basically noise around the mean improvement

Table IV: **EMCMC results for Spark jobs**

Job	Exploration	Scale-Up
WordCount	2.7%	5.8%
Sort	3.3%	1.6%
TeraSort	40.4%	16.7%
RandomForest	6.4%	7.2%
SVD	0.4%	1.9%
Average	10.64%	6.7%

with a low variance. The conclusion we reach based on our limited data is that simply picking the top candidate would not incur much opportunity cost, and in any case one could find a configuration very close to the best of fifty by sampling a small handful of top configurations. In fact, with the best configurations found with smaller workload ended up achieving 20.1%, 15.7%, 14.5%, 11.2%, 22.9% for five jobs respectively (slightly lower than the best in top fifty).

Similarly, we see performance improvements for all Spark jobs (see Table IV). Here, we use huge workload for scale-up evaluation, since we use large data in the exploration phase. We saw performance improvements for all five jobs, ranging from 1.6% to 16.7%. However, for Sort and SVD, there are limited improvements: 1.6% and 1.9% respectively. Spark jobs run very fast compared to Hadoop jobs. Thus, although we have scaled up the workload 10 times, it becomes difficult to achieve significant gain. Additional research is needed to better understand the scale-up potential of Spark jobs. Nevertheless, we saw an average performance improvement of 6.7%, which we believe is non-trivial at this scale.

Table V: **CPU time (secs) of default configuration for each Hadoop job under three data inputs**

	Small	Large	Huge	#EXP
WordCount	166.2	862.6	9367.1	#3241 (Gen17)
Sort	133.4	869.8	9891.7	#3318 (Gen17)
Terasort	115.7	1056.6	8751.6	#2876 (Gen15)
Pagerank	300.0	5657.2	13096.1	#3177 (Gen16)
NutchIndex	477.9	6596.5	11215.7	#4685 (Gen24)

Sensitivity Analysis. Here we study how sensitive the performance gain is *w.r.t.* individual configuration parameter. From the best-found configuration, we set the value of each parameter back to its default and check how much the performance has changed. For example, say $perf_{def}$ and $perf_{best}$ are the default and best performances (*i.e.* CPU times) obtained by ConEx for a job. Then, the performance improvement *w.r.t.* the default configuration is $\Delta_{best} = \frac{perf_{def} - perf_{best}}{perf_{def}}$. Note that, this is the best gain observed by ConEx. Next, to measure how sensitive the gain is *w.r.t.* a parameter c_i, we set c_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, $\Delta_i = \frac{perf_{def} - perf_i}{perf_{def}}$. Then the sensitivity of parameter c_i is the difference of performance improvement: $sensitivity_i = \Delta_{best} - \Delta_i$.

We conduct this analysis for all the parameters one by one for the Hadoop jobs with "huge" workload. Table VI shows the results. The second row is the overall performance gain. [1] It shows that performance improvement is sensitive to only few parameters. However, no single parameter is responsible for most of the improvement. Instead, the data seem to indicate that the influences of individual parameters are limited and that overall improvements come from combinations of, or interactions among, multiple parameters. 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.

Cost saving. The first three columns in Table V show the total execution time (in seconds) across the master-slave nodes for each Hadoop job under three workload sizes on our test platform. For example, *WordCount* under 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 step-up hypothesis saves 1656 hours ($(1873 - 33) * 3241$ seconds) to find a better configuration. In total, for all the five jobs, the scale-up hypothesis 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. If one adopts the resulting configurations of our approach, she could start to save the cost of running jobs after reaching the break-even points. We got 21.5% improvement for WordCount using about 277 hours in searching. A huge WordCount runs about 2.6 hours according to Table V. That means each "huge" run saves about 0.52 hours, and one will start to save cost after only 500 runs.

Result 2: *Our data support the scale-up hypothesis: configurations found using small workloads as proxies for sampling production-scale performance do tend to produce significant performance improvements for much larger workloads.*

Further exploration cost can be saved if our scale-out hypothesis holds good, *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). Thus, we ask:

RQ3. Scale-Out: Do configurations found for one kind of job produce significant performance improvements for similar types of jobs?

We test this RQ by evaluating performance gains for job B using configurations found for job A, where the similarity between A and B is measured using Section IV-B. Among the five Hadoop jobs, we find that *WordCount*, *Sort*, *TeraSort* are highly similar, and *PageRank* is somewhat similar to them, whereas *NutchIndex* has low similarity with this group. Table VII shows the results of Scale-Out hypothesis testing.

[1]We used a different cluster to do sensitivity analysis. So the total performance could be slightly different from those in Table III.

Table VI: **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%)
MR.M.memory.mb: 12.61%	MR.R.input.buffer.percent: 12.13%	MR.map.java.opts: 8.65%	MR.M.memory.mb: 10.82%	MR.map.java.opts: 9.44%
MR.M.sort.spill.percent: 3.76%	MR.M.memory.mb: 4.29%	MR.R.input.buffer.percent: 7.22%	MR.R.input.buffer.percent: 5.38%	MR.task.io.sort.mb: 6.19%
MR.R.input.buffer.percent: -0.48%	io.seqfile.compress.blocksize: 1.83%	MR.task.io.sort.mb: 2.84%	MR.M.java.opts: 3.10%	MR.reduce.memory.mb: 4.92%
MR.job.max.split.locations: -0.67%	io.file.buffer.size: 1.58%	MR.R.memory.mb: 1.75%	MR.task.io.sort.mb: 2.38%	MR.map.memory.mb: 1.83%
yarn.app.MR.am.resource.mb: -1.54%	MR.M.java.opts: 1.11%	io.seqfile.compress.blocksize: 1.58%	MR.R.memory.mb: 1.95%	yarn.RM.scheduler.class: 0.39%

Due to the page limit, we only list five most influential parameters. The capital “MR” is a substitution of “mapreduce”, “M” is “map”, and “R” is “reduce”.

Table VII: **Scale-out Hypothesis Testing.**

(a) **Hadoop**

Rep. Job	**Similar Jobs**				**Diff. Job**
	WC	**Sort**	**TeraSort**	**PageRank**	**Nutch**
			EMCMC		
WC	*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%*

(b) **Spark**

Rep. Job	**Similar Jobs**				
	WC	Sort	TeraSort	RF	SVD
WC	*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%*

The row header (first column “Rep.Job”) indicates the representative job, and the column headers represent the target jobs. The numbers indicate the performance gains achieved by a target job while running with the best configuration of Rep.Job for huge workload. The underlined numbers highlight the best performance in each column. The numbers in the diagonal show the performance achieved by a job when tested with its own best configuration (representative job=target job).

For example, Table VIIa 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.

A surprising aspect of these results is that, in few cases, a better configuration found for one job, e.g., Nutch, yielded greater gains for another job, e.g., Sort (27.6%) than the gain achieved by its own best configuration (15.8% for Sort). We speculate that dynamic characteristics of jobs such as Nutch might have exerted greater evolutionary pressure than those of other jobs. But we have not adequately explored the causes of these surprising results, and plan to do so in future work.

Result 3: *Our scale-out hypothesis holds good, i.e., the configuration found with a representative job can bring significant performance gain for other similar jobs.*

Finally, we evaluate ConEx *w.r.t.* the baseline approaches as described in Section IV-C. We present the results for only Haddop jobs here. In particular we check:

RQ4. How does EMCMC sampling strategy perform compared to alternative sampling strategies?

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 sampling strategy with the same cross-over and mutation strategies and the same fitness function as of EMCMC. 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.

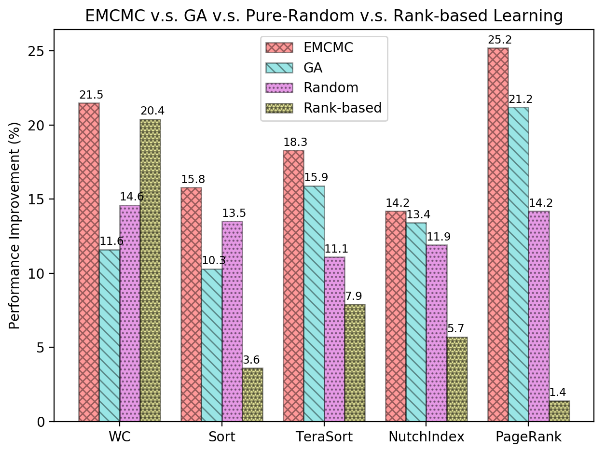

Figure 2: **EMCMC compares with other approaches in performance improvement for Hadoop Huge Workload.**

Table III shows the detailed results for different workloads. Overall, for all the jobs, EMCMC based sampling performs better. Figure 2 pictorially represents the results for “Huge” workload. EMCMC outperforms the random strategy from 17% to 125% under Huge workload across all the studied Hadoop jobs. EMCMC based sampling strategy also achieves better gain than the genetic algorithm (GA) based evolutionary search. EMCMC performs 6% to 85% better than GA for all the five jobs. The improvement of performance of EMCMC over GA also gives us an estimate of how much the evolutionary part of EMCMC contributes to ConEx's performance.

Result 4: *EMCMC based sampling strategy outperforms random and genetic algorithm based evolutionary sampling strategies to find better performing configurations.*

RQ5. How does search-based approach perform compared to learning-based approaches?

To compare our approach with learning-based approaches, we choose the state-of-the-art work of Nair *et al.* [14] published in FSE 2017. They used a rank-based performance prediction model to find 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.

Their training pool covers 4500 configurations, including 4400 feature-wisely and pair-wisely sampled and 100 random configurations, for large systems with millions of configurations (*e.g.* SQLite). 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 directly used the model published by Nair *et al.* As they did, we ran each model 20 times.

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 VIII 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 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.

Table VIII: **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

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

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 in 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 2 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 cost to gather adequate training samples.

Result 5: *Compared to Nair et al's learning-based approach, our approach finds configurations with higher performance gains.*

VI. Related Work

Learning-based approaches. A large body of work estimates system performance by learning prediction models [7], [12], [34], [14], [35]. The accuracy of these models depends on the quality of training data. As shown in RQ5, due to big-data systems have complex configuration space, it is challenging to find a representative model. Also, collecting training data is costly [13]. Existing logs from industrial uses of such systems are not necessarily useful as users tend to use default, or at least very few, configuration settings [1]. Previous approaches also rely on 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. Meinicke et al. [36] 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 performance by adding interaction terms in learning models. This approach combines domain knowledge and sampling strategies to generate training data. We have also seen the evidence of parameter interactions in RQ2. However, 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.

Metaheuristic search. Meta-heuristic methods are popular approaches to solve complex problems in other engineering domains. Zhang et al. [37] used such algorithms to find better settings to avoid power-consuming cache flushing. For robot motion planning, Jaillet et al. [38] used stochastic sampling to find globally low-cost paths for robotics with arbitrary user-given cost functions. Burns et al. [39] used heuristic sampling to solve the connectivity problem in robotic motion planing. Baltz et al. [40] presented their *Optometrist* algorithm to find configurations for a plasma fusion reactor. Unique characteristics of our work include use of an EMCMC sampling strategy and the applicatoin of validated scale-up and scale-out methods to reduce the cost of sampling/search.

Search-based software engineering (SBSE). In software engineering, many works have utilized heuristic algorithms with promising results. Jia and Harman [41] used automated search to find valuable test cases. McMinn [42] surveyed the application of search techniques for automated test data generation. Weimer et al. [43] used genetic programming to produce new code variants as candidate repairs, with test-suite-passing as an objective function. Le [22] used MCMC techniques to generate program variants with different control- and data-flows. Whittaker and Thomason [44] used a Markov Chain model to generate test inputs to study software failure causes. Oh et al. [17] worked to find good configurations for software product lines. Vizier [45] was developed at Google for optimizing various systems like machine learning models. The work presented here demonstrates the promise of SBSE-like approaches for tuning critical big-data systems.

Auto-tuning big data framework. A significant amount of work has been done to auto-tune different types of big data frameworks. Starfish [46] is one of the initial works on Hadoop auto-tuning. It tunes parameters by the predicted results of a cost model. As we discussed, accurate performance models are hard to build. If the accuracy of a model is low, it could introduce large errors. Actually, Liao *et al.* [47] have shown that the predicted results of Starfish's cost model could vary largely under different tasks settings. Starfish is targeted to the MapReduce framework. Our tool is intended to be general and we have presented results suggesting that it can be profitably applied on other systems (as we did for Spark). Babu et al. [48] tune MapReduce parameters by evaluating a job with sampled data from production data that the job will process ultimately. Our assumptions are much looser, and the configurations that we found will work on other datasets too. For example, we explored with a smaller dataset, A, but validated with another large dataset, B. Liao *et al.* [47] uses vanilla GA plus memoization to identify high-value configurations. But they selected only six important parameters to tune. In contrast, we selected all parameters that related to our objective performance. Without knowing how parameters interact, we cannot exclude any relevant parameter.

VII. Threats to Validity

Internal Validity. A number of threats to the internal validity arise from the experimental setup. First, due to the nature of dynamic evaluation, it is possible that the experimental results are 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 ran each dynamic evaluation three times to get average performance as a final result. We also choose a subset of all parameters to study with domain knowledge. It is possible that we missed some important ones. To mitigate this threat, we referred to many previous works cited in this review paper [28] on Hadoop, and have included all parameters studied by other researchers in our parameter set.

External Validity. In this work, we report results only for Hadoop and Spark framework using 10 representative jobs. Since these two are most widely used big-data frameworks and we used a popular benchmark tool from Intel, we believe the results will be representative in other settings.

VIII. Future Work

In this work, we adopt domain knowledge to reduce the dimension of a configuration space. In future, we hope to learn how to infer the modularity properties of configuration spaces so that we can decompose a large space into decoupled sets of parameters. For example, we can try to group parameters based on the computation phase they are involved in, e.g., parameters that are only involved in "map" steps can form an independent cluster. Such a decomposition would enable order-of-magnitude reductions in search costs by reducing a geometric problem to an additive one.

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, use 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.

We do not claim that our sampling strategy is truly optimal. In fact, no meta-heuristic algorithm guarantees to find the optimal solution. We leave the design and evaluation of more sophisticated strategies for future work. Our initial results show that EMCMC strategy can produce significant performance improvement. In future work, we plan to generalize our framework to use semantically informed sampling, constraint-based sampling, sampling from distributions that justified using domain knowledge, etc.

IX. Conclusions

In this work, we proposed to use meta-heuristic search employing an EMCMC 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 potential to significantly and cost-effectively improve the performance of real-world big data systems. The evaluation results show that our approach outperforms other competing search-based approaches based on random sampling and genetic algorithm. Thus, the results indicate that an EMCMC-based approach is justified by the characteristics of big-data infrastructures. Our data further support the hypothesis that our search-based strategy outperforms a state-of-the-art model-learning approach. We emphasize once again that this work is distinguished in part from earlier work by the much larger sizes of the configuration spaces we address. Given the large-scale economic and environmental costs of big data computing, our approach thus appears to have the potential to produce significant payoffs for users of big data systems and for our broader society.

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] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] A. A. Canutescu and R. L. Dunbrack, "Cyclic coordinate descent: A robotics algorithm for protein loop closure," *Protein science*, vol. 12, no. 5, pp. 963–972, 2003.

[19] P.-L. Loh and M. J. Wainwright, "Regularized m-estimators with nonconvexity: Statistical and algorithmic theory for local optima," in *Advances in Neural Information Processing Systems*, 2013, pp. 476–484.

[20] M. M. Drugan and D. Thierens, "Evolutionary markov chain monte carlo," in *International Conference on Artificial Evolution (Evolution Artificielle)*. Springer, 2003, pp. 63–76.

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

[22] 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.

[23] C. Andrieu, N. De Freitas, A. Doucet, and M. I. Jordan, "An introduction to mcmc for machine learning," *Machine learning*, vol. 50, no. 1-2, pp. 5–43, 2003.

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

[25] 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.

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

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

[28] 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.

[29] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] C. Zhang, F. Vahid, and R. Lysecky, “A self-tuning cache architecture for embedded systems,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 2, pp. 407–425, 2004.

[38] L. Jaillet, J. Cortés, and T. Siméon, “Sampling-based path planning on configuration-space costmaps,” *IEEE Transactions on Robotics*, vol. 26, no. 4, pp. 635–646, 2010.

[39] B. Burns and O. Brock, “Toward optimal configuration space sampling.” in *Robotics: Science and Systems*. Citeseer, 2005, pp. 105–112.

[40] E. Baltz, E. Trask, M. Binderbauer, M. Dikovsky, H. Gota, R. Mendoza, J. Platt, and P. Riley, “Achievement of sustained net plasma heating in a fusion experiment with the optometrist algorithm,” *Scientific Reports*, vol. 7, 2017.

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

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

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