An Uncertainty-Aware Approach to Optimal Configuration of Stream Processing Systems

Pooyan Jamshidi
Imperial College London, UK
Email: p.jamshidi@imperial.ac.uk

Giuliano Casale
Imperial College London, UK
Email: g.casale@imperial.ac.uk

***Abstract*—Finding optimal configurations for Stream Processing Systems (SPS) is a challenging problem due to the large number of parameters that can influence their performance and the lack of analytical models to anticipate the effect of a change. To tackle this issue, we consider tuning methods where an experimenter is given a limited budget of experiments and needs to carefully allocate this budget to find optimal configurations. We propose in this setting Bayesian Optimization for Configuration Optimization (BO4CO), an auto-tuning algorithm that leverages Gaussian Processes (GPs) to iteratively capture posterior distributions of the configuration spaces and sequentially drive the experimentation. Validation based on Apache Storm demonstrates that our approach locates optimal configurations within a limited experimental budget, with an improvement of SPS performance typically of at least an order of magnitude compared to existing configuration algorithms.**

I. Introduction

We live in an increasingly instrumented world, where a large number of heterogeneous data sources typically provide continuous data streams from live stock markets, video sources, production line status feeds, and vital body signals [12]. Yet, the research literature lacks automated methods to support the configuration (*i.e.*, auto-tuning) of the underpinning SPSs. One possible explanation is that, "big data" systems such as SPSs often combine emerging technologies that are still poorly understood from a performance standpoint [18], [40] and therefore difficult to holistically configure. Hence there is a critical shortage of models and tools to anticipate the effects of changing a configuration in these systems. Examples of configuration parameters for a SPS include buffer size, heap sizes, serialization/de-serialization methods, among others.

Performance differences between a well-tuned configuration and a poorly configured one can be of orders of magnitude. Typically, administrators use a mix of rules-of-thumb, trial-and-error, and heuristic methods for setting configuration parameters. However, this way of testing and tuning is slow, and require skillful administrators with a good understanding of the SPS internals. Furthermore, decisions are also affected by the nonlinear interactions between configuration parameters.

In this paper, we address the problem of finding optimal configurations under these requirements: (i) a configuration space composed by multiple parameters; (ii) a limited budget of experiments that can be allocated to test the system; (iii) experimental results affected by uncertainties due to measurement inaccuracies or intrinsic variability in the system processing times. While the literature on auto-tuning work is abundant with existing solutions for databases, e-commerce and batch processing systems that address some of the above challenges (*e.g.*, rule-based [21], design of experiment [35], model-based [24], [18], [40], [31], search-based [38], [27], [34], [10], [1], [39] and learning-based [3]), this is the first work to consider the problem under such constraints altogether.

In particular, we present a new auto-tuning algorithm called BO4CO that leverages GPs [37] to continuously estimate the mean and confidence interval of a response variable at yet-to-be-explored configurations. Using Bayesian optimization [30], the tuning process can account for all the available prior information about a system and the acquired configuration data, and apply a variety of kernel estimators [23] to locate regions where optimal configuration may lie. To the best of our knowledge, this is the first time that GPs are used for automated system configuration, thus a goal of the present work is to introduce and apply this class of machine learning methods into system performance tuning.

BO4CO is designed keeping in mind the limitations of sparse sampling from the configuration space. For example, its features include: (i) sequential planning to perform experiments that ensure coverage of the most promising zones; (ii) memorization of past-collected samples while planning new experiments; (iii) guarantees that optimal configurations will be eventually discovered by the algorithm. We show experimentally that BO4CO outperforms previous algorithms for configuration optimization. Our real configuration datasets are collected for three different SPS benchmark systems, implemented with Apache Storm, and using 5 cloud clusters worth several months of experimental time.

The rest of this paper is organized as follows. Section II discusses the motivations. The BO4CO algorithm is introduced in Section III and then validated in Section IV. Finally, Section V discusses the applicability of BO4CO in practice, Section VI reviews state of the art and Section VII concludes the paper.

II. Problem and Motivation

A. Problem statement

In this paper, we focus on the problem of optimal system configuration defined as follows. Let X_i indicate the i-th configuration parameter, which takes values in a finite domain $Dom(X_i)$. In general, X_i may either indicate (i) integer variable such as "level of parallelism" or (ii) categorical variable such as "messaging frameworks" or Boolean parameter such as "enabling timeout". Throughout the paper, by the term option, we mean possible values that can be assigned to a parameter. The *configuration space* is thus $\mathbb{X} = Dom(X_1) \times \cdots \times Dom(X_d)$, which is the Cartesian product of the domains

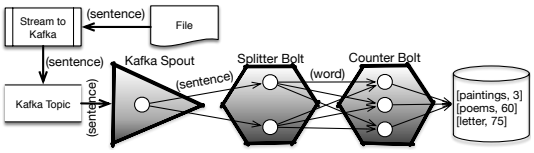

Fig. 1: WordCount topology architecture.

of d parameters of interest. We assume that each configuration $\boldsymbol{x} \in \mathbb{X}$ is valid and denote by $f(\boldsymbol{x})$ the response measured on the SPS under that configuration. Throughout, we assume that f is latency, however other response metrics (*e.g.*, throughput) may be used. The graph of f over configurations is called the *response surface* and it is partially observable, *i.e.,* the actual value of $f(\boldsymbol{x})$ is known only at points $\boldsymbol{x}$ that has been previously experimented with. We here consider the problem of finding an optimal configuration $\boldsymbol{x}^*$ that minimizes f over the configuration space $\mathbb{X}$ with as few experiments as possible:

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}\in\mathbb{X}} f(\boldsymbol{x}) \tag{1}$$

In fact, the response function $f(\cdot)$ is usually unknown or partially known, *i.e.*, $y_i = f(\boldsymbol{x}_i), \boldsymbol{x}_i \subset \mathbb{X}$. In practice, such measurements may contain noise, *i.e.*, $y_i = f(\boldsymbol{x}_i) + \epsilon_i$. Note that since the response surface is only partially-known, finding the optimal configuration is a blackbox optimization problem [23], [29], which is also subject to noise. In fact, the problem of finding a optimal solution of a non-convex and multi-modal response surface (cf. Figure 2) is $\mathcal{NP}$-hard [36]. Therefore, on instances where it is impossible to locate a global optimum, BO4CO will strive to find the best possible local optimum within the available experimental budget.

B. Motivation

1) A running example: WordCount (cf. Figure 1) is a popular benchmark SPS. In WordCount a text file is fed to the system and it counts the number of occurrences of the words in the text file. In Storm, this corresponds to the following operations. A Processing Element (PE) called Spout is responsible to read the input messages (tuples) from a data source (*e.g.,* a Kafka topic) and stream the messages (*i.e.*, sentences) to the topology. Another PE of type Bolt called Splitter is responsible for splitting sentences into words, which are then counted by another PE called Counter.

2) Nonlinear interactions: We now illustrate one of the inherent challenges of configuration optimization. The metric that defines the surface in Figure 2 is the *latency* of individual messages, defined as the time since emission by the Kafka Spout to completion at the Counter, see Figure 1. Note that this function is the subset of wc(6D) in Table I when the level of parallelism of Splitters and Counters is varied in $[1, 6]$ and $[1, 18]$. The surface is strongly *non-linear* and *multi-modal* and indicates two important facts. First, the performance difference between the best and worst settings is substantial, 65%, and with more intense workloads we have observed differences in latency as large as 99%, see Table V. Next, non-linear relations between the parameters imply that the optimal number of counters depends on the number of Splitters, and vice-versa. Figure 3 shows this *non-linear interaction* [31] and

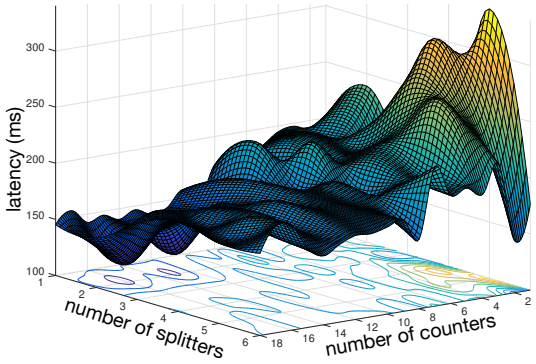

Fig. 2: WordCount response surface. It is an interpolated surface and is a projection of 6 dimensions, in wc(6D), onto 2D. It shows the non-convexity, multi-modality and the substantial performance difference between different configurations.

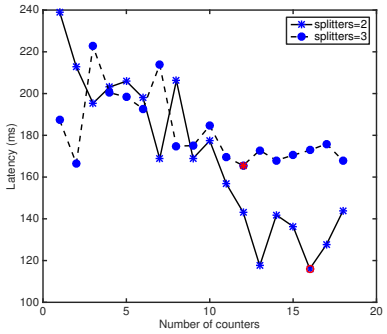

Fig. 3: WordCount latency, cut though Figure 2.

demonstrates that if one tries to minimize latency by acting just on one of these parameters at the time, the resulting configuration may not lead to a global optimum, as the number of Splitters has a strong influence on the optimal counters.

3) Sparsity of effects: Another observation from our extensive experiments with SPS is the *sparsity of effects*. More specifically, this means low-order interactions among a few dominating factors can explain the main changes in the response function observed in the experiments. In this work we assume sparsity of effects, which also helps in addressing the intractable growth of the configuration space [19].

Methodology. In order to verify to what degree the sparsity of effects assumption holds in SPS, we ran experiments on 3 different benchmarks that exhibit different bottlenecks: WordCount (wc) is CPU intensive, RollingSort (rs) is memory intensive, and SOL (sol) is network intensive. Different testbed settings were also considered, for a total of 5 datasets, as listed in Table I. Note that the parameters we consider here are known to significantly influence latency, as they have been chosen according to professional tuning guides [26] and also small scale tests where we varied a single parameter to make sure that the selected parameters were all influential. For each test in the experiment, we run the benchmark for 8 minutes including the initial burn-in period. Further details on the experimental procedure are given in Section IV-B. Note that the largest dataset (*i.e.,* rs(6D)) has required alone $3840 \times 8/60/24 = 21$ days, within a total experimental time of about 2.5 months to collect the datasets of Table I.

TABLE I: Sparsity of effects on 5 experiments where we have varied different subsets of parameters and used different testbeds. Note that these are the datasets we experimentally measured on the benchmark systems and we use them for the evaluation, more details including the results for 6 more experiments are in the appendix.

	Topol.	Parameters	Main factors	Merit	Size	Testbed
1	wc(6D)	1-spouts, 2-max_spout, 3-spout_wait, 4-splitters, 5-counters, 6-netty_min_wait	{1, 2, 5}	0.787	2880	C1
2	sol(6D)	1-spouts, 2-max_spout, 3-top_level, 4-netty_min_wait, 5-message_size, 6-bolts	{1, 2, 3}	0.447	2866	C2
3	rs(6D)	1-spouts, 2-max_spout, 3-sorters, 4-emit_freq, 5-chunk_size, 6-message_size	{3}	0.385	3840	C3
4	wc(3D)	1-max_spout, 2-splitters, 3-counters	{1, 2}	0.480	756	C4
5	wc(5D)	1-spouts, 2-splitters, 3-counters, 4-buffer-size, 5-heap	{1}	0.851	1080	C5

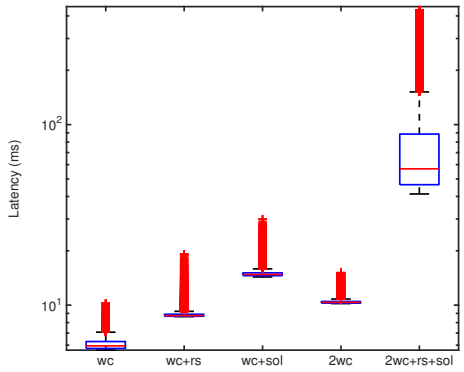

Fig. 4: Noisy experimental measurements. Note that + here means that wc is deployed in a multi-tenant environment with other topologies and as a result not only the latency is increased but also the variability became greater.

Results. After collecting experimental data, we have used a common correlation-based feature selector[1] implemented in Weka to rank parameter subsets according to a heuristic. The bias of the merit function is toward subsets that contain parameters that are highly correlated with the response variable. Less influential parameters are filtered because they will have low correlation with latency, and a set with the main factors is returned. For all of the 5 datasets, we list in Table I the main factors. The analysis results demonstrate that in all the 5 experiments at most 2-3 parameters were strongly interacting with each other, out of a maximum of 6 parameters varied simultaneously. Therefore, the determination of the regions where performance is optimal will likely be controlled by such dominant factors, even though the determination of a global optimum will still depends on all the parameters.

4) Measurement uncertainty: We now illustrate measurement variabilities, which represent an additional challenge for configuration optimization. As depicted in Figure 4, we took different samples of the latency metric over 2 hours for five different deployments of WordCount. The experiments run on a multi-node cluster on the EC2 cloud. After filtering the initial burn-in, we computed averages and standard deviation of the latencies. Note that the configuration across all 5 settings is similar, the only difference is the number of co-located topologies in the testbed. The data in boxplots illustrate that variability can be small in some settings (*e.g.,* wc), while they can be large in some other experimental setups (*e.g.,* 2wc+rs+sol). In traditional techniques such as design of experiments, such variability is addressed by repeating experiments multiple times and obtaining regression estimates for the system model across such repetitions. However, we here pursue the alternative approach of relying on GP models to capture both mean and variance of measurements within the model that guides the configuration process. The theory underpinning this approach is discussed in the next section.

[1]The most significant parameters are selected based on the following merit function [9], also shown in Table I:

$$m_{ps} = \frac{n\overline{r_{lp}}}{\sqrt{n + n(n-1)\overline{r_{pp}}}}, \tag{2}$$

where $\overline{r_{lp}}$ is the mean parameter-latency correlation, n is the number of parameters, $\overline{r_{pp}}$ is the average feature-feature inter-correlation [9, Sec 4.4].

III. BO4CO: Bayesian Optimization for Configuration Optimization

A. Bayesian Optimization with Gaussian Process prior

Bayesian optimization is a sequential design strategy that allows us to perform global optimization of blackbox functions [30]. The main idea of this method is to treat the blackbox objective function $f(\boldsymbol{x})$ as a random variable with a given prior distribution, and then perform optimization on the posterior distribution of $f(\boldsymbol{x})$ given experimental data. In this work, GPs are used to model this blackbox objective function at each point $\boldsymbol{x} \in \mathbb{X}$. That is, let $\mathbb{S}_{1:t}$ be the experimental data collected in the first t iterations and let $\boldsymbol{x}_{t+1}$ be a candidate configuration that we may select to run the next experiment. Then BO4CO assesses the probability that this new experiment could find an optimal configuration using the posterior distribution:

$$\Pr(f_{t+1}|\mathbb{S}_{1:t}, \boldsymbol{x}_{t+1}) \sim \mathcal{N}(\mu_t(\boldsymbol{x}_{t+1}), \sigma_t^2(\boldsymbol{x}_{t+1})),$$

where $\mu_t(\boldsymbol{x}_{t+1})$ and $\sigma_t^2(\boldsymbol{x}_{t+1})$ are suitable estimators of the mean and standard deviation of a normal distribution that is used to model this posterior. The main motivation behind the choice of GPs as prior here is that it offers a framework in which reasoning can be not only based on mean estimates but also the variance, providing more informative decision makings. The other reason is that all the computations in this framework are based on *linear algebra*.

Figure 5 illustrates the GP-based Bayesian optimization using a 1-dimensional response surface. The curve in blue is the unknown true posterior distribution, whereas the mean is shown in green and the 95% confidence interval at each point in the shaded area. Stars indicate measurements carried out in the past and recorded in $\mathbb{S}_{1:t}$ (*i.e.,* observations). Configuration corresponds to $\boldsymbol{x}_1$ has a large confidence interval due to lack of observations in its neighborhood. Conversely, $\boldsymbol{x}_4$ has a narrow confidence since neighboring configurations have been experimented with. The confidence interval in the neighborhood of $\boldsymbol{x}_2$ and $\boldsymbol{x}_3$ is not high and correctly our approach does not decide to explore these zones. The next configuration $\boldsymbol{x}_{t+1}$, indicated by a small circle right to the $\boldsymbol{x}_4$, is selected based on a criterion that will be defined later.

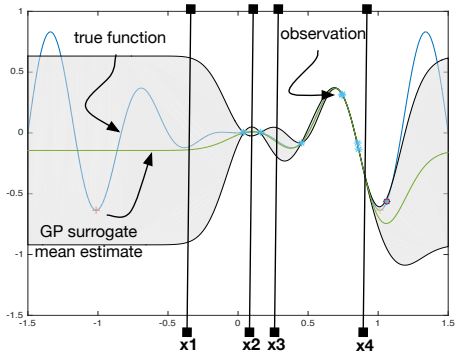

Fig. 5: An example of 1D GP model: GPs provide mean estimates as well as the uncertainty in estimations, *i.e.*, variance.

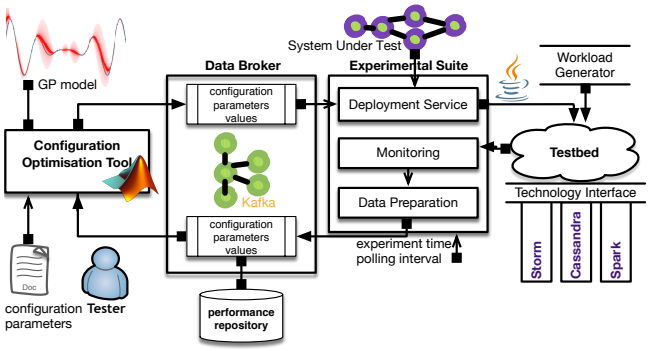

Fig. 6: BO4CO architecture: (i) optimization and (ii) experimental suite are integrated via (iii) a data broker. The integrated solution is available: https://github.com/dice-project/DICE-Configuration-BO4CO.

B. BO4CO algorithm

BO4CO's high-level architecture is shown in Figure 6 and the procedure that drives the optimization is described in Algorithm. We start by bootstrapping the optimization following Latin Hypercube Design (lhd) to produce an initial design $\mathcal{D} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ (cf. *step 1* in Algorithm 1). Although other design approaches (*e.g.*, random) could be used, we have chosen lhd because: (i) it ensures that the configuration samples in $\mathcal{D}$ is representative of the configuration space $\mathbb{X}$, whereas traditional random sampling [22], [11] (called brute-force) does not guarantee this [25]; (ii) another advantage is that the lhd samples can be taken one at a time, making it efficient in high dimensional spaces. After obtaining the measurements regarding the initial design, BO4CO then fits a GP model to the design points $\mathcal{D}$ to form our belief about the underlying response function (cf. *step 3* in Algorithm 1). The while loop in Algorithm 1 iteratively updates the belief until the budget runs out: As we accumulate the data $\mathbb{S}_{1:t} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{t}$, where $y_i = f(\boldsymbol{x}_i) + \epsilon_i$ with $\epsilon \sim \mathcal{N}(0, \sigma^2)$, a prior distribution $\Pr(f)$ and the likelihood function $\Pr(\mathbb{S}_{1:t}|f)$ form the posterior distribution: $\Pr(f|\mathbb{S}_{1:t}) \propto \Pr(\mathbb{S}_{1:t}|f)\Pr(f)$.

A GP is a distribution over functions [37], specified by its mean (see Section III-E2), and covariance (see Section III-E1):

$$y = f(\boldsymbol{x}) \sim \mathcal{GP}(\mu(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x}')), \tag{3}$$

Algorithm 1 : **BO4CO**

Input: Configuration space $\mathbb{X}$, Maximum budget N_{max}, Response function f, Kernel function $\boldsymbol{K}_\theta$, Hyper-parameters $\boldsymbol{\theta}$, Design sample size n, learning cycle N_l
Output: Optimal configurations $\boldsymbol{x}^*$ and learned model $\mathcal{M}$

1: choose an initial sparse design (*lhd*) to find an initial design samples $\mathcal{D} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$
2: obtain *performance measurements* of the initial design, $y_i \leftarrow f(\boldsymbol{x}_i) + \epsilon_i, \forall \boldsymbol{x}_i \in \mathcal{D}$
3: $\mathbb{S}_{1:n} \leftarrow \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{n}; t \leftarrow n+1$
4: $\mathcal{M}(\boldsymbol{x}|\mathbb{S}_{1:n}, \boldsymbol{\theta}) \leftarrow$ fit a $\mathcal{GP}$ model to the design ▷ Eq.(3)
5: **while** $t \leq N_{max}$ **do**
6: if $(t \mod N_l = 0)$ $\boldsymbol{\theta} \leftarrow$ *learn* the kernel hyper-parameters by maximizing the likelihood
7: find *next configuration* $\boldsymbol{x}_t$ by optimizing the selection criteria over the estimated response surface given the data, $\boldsymbol{x}_t \leftarrow \arg\, max_{\boldsymbol{x}} u(\boldsymbol{x}|\mathcal{M}, \mathbb{S}_{1:t-1})$ ▷ Eq.(9)
8: obtain performance for the *new configuration* $\boldsymbol{x}_t$, $y_t \leftarrow f(\boldsymbol{x}_t) + \epsilon_t$
9: Augment the configuration $\mathbb{S}_{1:t} = \{\mathbb{S}_{1:t-1}, (\boldsymbol{x}_t, y_t)\}$
10: $\mathcal{M}(\boldsymbol{x}|\mathbb{S}_{1:t}, \boldsymbol{\theta}) \leftarrow$ *re-fit* a new GP model ▷ Eq.(7)
11: $t \leftarrow t+1$
12: **end while**
13: $(\boldsymbol{x}^*, y^*) = \min \mathbb{S}_{1:N_{max}}$
14: $\mathcal{M}(\boldsymbol{x})$

where $k(\boldsymbol{x}, \boldsymbol{x}')$ defines the distance between $\boldsymbol{x}$ and $\boldsymbol{x}'$. Let us assume $\mathbb{S}_{1:t} = \{(\boldsymbol{x}_{1:t}, y_{1:t})|y_i := f(\boldsymbol{x}_i)\}$ be the collection of t observations. The function values are drawn from a multivariate Gaussian distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{K})$, where $\boldsymbol{\mu} := \mu(\boldsymbol{x}_{1:t})$,

$$\boldsymbol{K} := \begin{bmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & \ldots & k(\boldsymbol{x}_1, \boldsymbol{x}_t) \\ \vdots & \ddots & \vdots \\ k(\boldsymbol{x}_t, \boldsymbol{x}_1) & \ldots & k(\boldsymbol{x}_t, \boldsymbol{x}_t) \end{bmatrix} \tag{4}$$

In the while loop in BO4CO, given the observations we accumulated so far, we intend to fit a new GP model:

$$\begin{bmatrix} \boldsymbol{f}_{1:t} \\ f_{t+1} \end{bmatrix} \sim \mathcal{N}(\boldsymbol{\mu}, \begin{bmatrix} \boldsymbol{K} + \sigma^2 \boldsymbol{I} & \boldsymbol{k} \\ \boldsymbol{k}^\intercal & k(\boldsymbol{x}_{t+1}, \boldsymbol{x}_{t+1}) \end{bmatrix}), \tag{5}$$

where $\boldsymbol{k}(\boldsymbol{x})^\intercal = [k(\boldsymbol{x}, \boldsymbol{x}_1) \quad k(\boldsymbol{x}, \boldsymbol{x}_2) \quad \ldots \quad k(\boldsymbol{x}, \boldsymbol{x}_t)]$ and $\boldsymbol{I}$ is identity matrix. Given the Eq. (5), the new GP model can be drawn from this new Gaussian distribution:

$$\Pr(f_{t+1}|\mathbb{S}_{1:t}, \boldsymbol{x}_{t+1}) = \mathcal{N}(\mu_t(\boldsymbol{x}_{t+1}), \sigma_t^2(\boldsymbol{x}_{t+1})), \tag{6}$$

where

$$\mu_t(\boldsymbol{x}) = \mu(\boldsymbol{x}) + \boldsymbol{k}(\boldsymbol{x})^\intercal (\boldsymbol{K} + \sigma^2 \boldsymbol{I})^{-1} (\boldsymbol{y} - \boldsymbol{\mu}) \tag{7}$$

$$\sigma_t^2(\boldsymbol{x}) = k(\boldsymbol{x}, \boldsymbol{x}) + \sigma^2 \boldsymbol{I} - \boldsymbol{k}(\boldsymbol{x})^\intercal (\boldsymbol{K} + \sigma^2 \boldsymbol{I})^{-1} \boldsymbol{k}(\boldsymbol{x}) \tag{8}$$

These posterior functions are used to select the next point $\boldsymbol{x}_{t+1}$ as detailed in Section III-C.

C. Configuration selection criteria

The selection criteria is defined as $u : \mathbb{X} \rightarrow \mathbb{R}$ that selects $\boldsymbol{x}_{t+1} \in \mathbb{X}$, should $f(\cdot)$ be evaluated next (*step 7*):

$$\boldsymbol{x}_{t+1} = \underset{\boldsymbol{x} \in \mathbb{X}}{\operatorname{argmax}}\, u(\boldsymbol{x}|\mathcal{M}, \mathbb{S}_{1:t}) \tag{9}$$

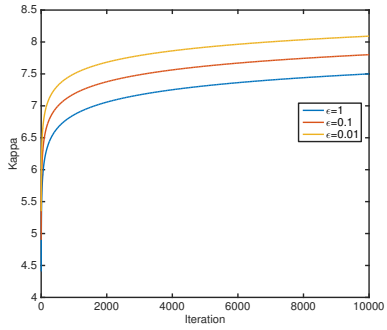

Fig. 7: Change of κ value over time: it begins with a small value to exploit the mean estimates and it increases over time in order to explore.

Although several different criteria exist in the literature (see [30]), BO4CO uses *Lower Confidence Bound* (LCB) [30]. LCB selects the next configuration by trade-off between exploitation and exploration:

$$u_{LCB}(\boldsymbol{x}|\mathcal{M}, \mathbb{S}_{1:n}) = \underset{\boldsymbol{x}\in\mathbb{X}}{\operatorname{argmin}}\, \mu_t(\boldsymbol{x}) - \kappa\sigma_t(\boldsymbol{x}), \tag{10}$$

where κ can be set according to the objectives. For instance, if we require to find a near optimal configuration quickly we set a low value to κ to take the most out of the initial design knowledge. However, if we want to skip local minima, we can set a high value to κ. Furthermore, κ can be adapted over time to benefit from the both [17]. For instance, κ can start with a reasonably small value to exploit the initial design and increase over time to do more explorations (cf. Figure 7).

D. Illustration

The steps in Algorithm 1 are illustrated in Figure 8. Firstly, an initial design based on lhd is produced (Figure 8(a)). Secondly, a GP model is fit to the initial design (Figure 8(b)). Then, the model is used to calculate the selection criteria (Figure 8(c)). Finally, the configuration that maximizes the selection criteria is used to run the next experiment and provide data for refitting a more accurate model (Figure 8(d)).

E. Model fitting in BO4CO

In this section, we provide some practical considerations to make GPs applicable for configuration optimization.

1) Kernel function: In BO4CO, as shown in Algorithm 1, the covariance function $k : \mathbb{X} \times \mathbb{X} \rightarrow \mathbb{R}$ dictates the structure of the response function we fit to the observed data. For integer variables (cf. Section II-A), we implemented the Matérn kernel [37]. The main reason behind this choice is that along each dimension in the configuration response functions different level of smoothness can be observed (cf. Figure 2). Matérn kernels incorporate a smoothness parameter $\nu > 0$ that permits greater flexibility in modeling such functions [37]. The following is a variation of the Matérn kernel for $\nu = 1/2$:

$$k_{\nu=1/2}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \theta_0^2 \exp(-r), \tag{11}$$

where $r^2(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i - \boldsymbol{x}_j)^\intercal \boldsymbol{\Lambda} (\boldsymbol{x}_i - \boldsymbol{x}_j)$ for some positive semidefinite matrix $\boldsymbol{\Lambda}$. For categorical variables, we implemented the following [14]:

$$k_\theta(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp(\Sigma_{\ell=1}^{d}(-\theta_\ell \delta(\boldsymbol{x}_i \neq \boldsymbol{x}_j))), \tag{12}$$

where d is the number of dimensions (*i.e.*, the number of configuration parameters), θ_ℓ adjust the scales along the function dimensions and δ is a function gives the distance between two categorical variables using Kronecker delta [14], [30]. TL4CO uses different scales $\{\theta_\ell, \ell = 1 \ldots d\}$ on different dimensions as suggested in [37], [30], this technique is called Automatic Relevance Determination (ARD). After learning the hyper-parameters (*step 6*), if the ℓ-th dimension turns out to be irrelevant, then θ_ℓ will be a small value, and therefore, will be discarded. This is particularly helpful in high dimensional spaces, where it is difficult to find the optimal configuration.

2) Prior mean function: While the kernel controls the structure of the estimated function, the prior mean $\mu(\boldsymbol{x}) : \mathbb{X} \rightarrow \mathbb{R}$ provides a possible offset for our estimation. By default, this function is set to a constant $\mu(\boldsymbol{x}) := \mu$, which is inferred from the observations [30]. However, the prior mean function is a way of incorporating the expert knowledge, if it is available, then we can use this knowledge. Fortunately, we have collected extensive experimental measurements and based on our datasets (cf. Table I), we observed that typically, for Big Data systems, there is a significant distance between the minimum and the maximum of each function (cf. Figure 2). Therefore, a linear mean function $\mu(\boldsymbol{x}) := \boldsymbol{a}\boldsymbol{x} + b$, allows for more flexible structures, and provides a better fit for the data than a constant mean. We only need to learn the slope for each dimension and an offset (denoted by $\mu_\ell = (\boldsymbol{a}, b)$).

3) Learning parameters: marginal likelihood: This section describe the *step 7* in Algorithm 1. Due to the heavy computation of the learning, this process is computed only every N_l iterations. For learning the hyper-parameters of the kernel and also the prior mean functions (cf. Sections III-E1 and III-E2), we maximize the marginal likelihood [30] of the observations $\mathbb{S}_{1:t}$. To do that, we train GP model (7) with $\mathbb{S}_{1:t}$. We optimize the marginal likelihood using multi-started quasi-Newton hill-climbers [28]. For this purpose, we use the off-the-shelf gpml library presented in [28]. Using the kernel defined in (12), we learn $\boldsymbol{\theta} := (\theta_{0:d}, \mu_{0:d}, \sigma^2)$ that comprises the hyper-parameters of the kernel and mean functions. The learning is performed iteratively resulting in a sequence of $\boldsymbol{\theta}_i$ for $i = 1 \ldots \lfloor \frac{N_{max}}{N_\ell} \rfloor$.

4) Observation noise: The primary way for determining the noise variance σ in BO4CO is to use historical data: In Section II-B4, we have shown that such noise can be measured with a high confidence and the signal-to-noise ratios shows that such noise is stationary. The secondary alternative is to learn the noise variance sequentially as we collect new data. We treat them just as any other hyper-parameters, see Section III-E3.

IV. Experimental Results

A. Implementation

From an implementation perspective, BO4CO consists of three major components: (i) an *optimization component* (cf. left part of Figure 6), (ii) an *experimental suite* (cf. right part of Figure 6) integrated via a (iii) *data broker*. The optimization component implements the model (re-)fitting (7) and criteria optimization (9) steps in Algorithm 1 and is developed in Matlab 2015b. The experimental suite component implements the facilities for automated deployment of topologies, performance measurements and data preparation and is developed in Java.

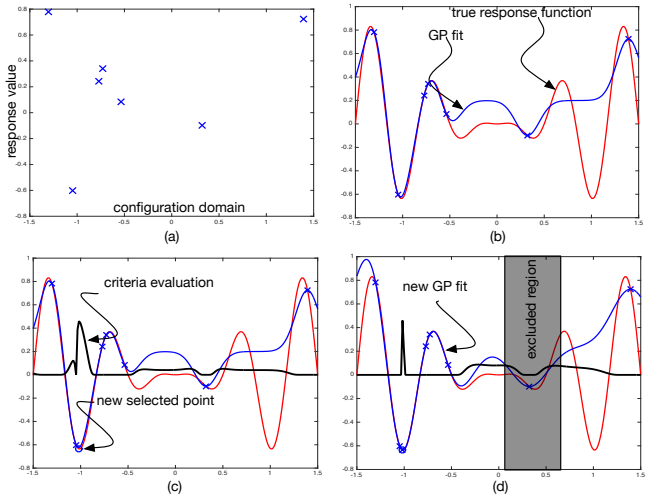

Fig. 8: Illustration of configuration parameter optimization: (a) initial observations; (b) a GP model fit; (c) choosing the next point; (d) refitting a new GP model.

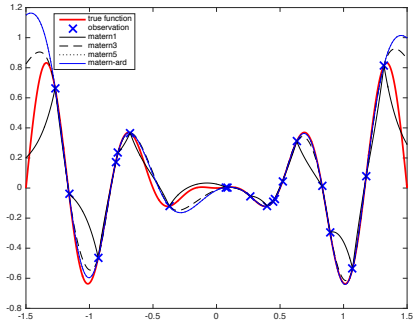

Fig. 9: The effect of changing the kernel in estimations.

The optimization component retrieves the initial design performance data and determines which configuration to try next using the procedure explained in III-C. The suite then deploys the *topology under test* on a testing cluster. The performance of the topology is then measured and the performance data will be used for model refitting. We have released the *code* and *data*: https://github.com/dice-project/DICE-Configuration-BO4CO.

In order to make BO4CO more practical and relevant for industrial use, we considered several implementation enhancements. In order to perform efficient GP model re-fitting, we implemented a covariance wrapper function that keeps the internal state for caching kernels and its derivatives, and can update kernel function by a single element. This was particularly helpful for learning the hyper-parameters at runtime.

B. Experimental design

1) Topologies under test and benchmark functions: In this section, we evaluate BO4CO using 3 different Storm benchmarks: (i) WordCount, (ii) RollingSort, (iii) SOL. RollingSort implements a common pattern in real-time data analysis that performs rolling counts of incoming messages. RollingSort is used by Twitter for identifying trending topics. SOL is a *network intensive* topology, where the incoming messages will be routed through an inter-worker network. WordCount and RollingSort are standard benchmarks and are widely used in the community, *e.g.*, research papers [7] and industry scale benchmarks [13]. We have conducted all the experiments on 5 cloud clusters and with different sets of parameters resulted in datasets in Table I.

We also evaluate BO4CO with a number of benchmark functions, where we perform a synthetic experiment inside MATLAB in which a measurement is just a function evaluation: Branin(2D), Dixon-Szego(2D), Hartmann(3D) and Rosenbrock(5D). These benchmark functions are commonly used in global optimization and configuration approaches [38], [34]. We particularly selected these because: (i) they have different curvature and (ii) they have multiple global minimizers, and (iii) they are of different dimensions.

2) Baseline approaches: The performance of BO4CO is compared with the 5 outstanding state-of-the-art approaches for configuration optimizations: SA [8], GA [1], HILL [38], PS [34] and Drift [33]. They are of different nature and use different search algorithms: simulated annealing, genetic algorithm, hill climbing, pattern search and adaptive search.

3) Experimental considerations: The performance statistics regarding each specific configuration has been collected over a window of 5 minutes (excluding the first two minutes of burn-in and the last minute of cluster cleaning). The first two minutes are excluded because the monitoring data are not stationary, while the last minute is the time given to the topology to fully process all messages. We then shut down the topology, clean the cluster and move on to the next experiment. We also replicated each runs of algorithms for 30 times in order to report the comparison results. Therefore, all the results presented in this paper are the mean performance over 30 runs.

4) Cluster configuration: We conducted all the experiments on 5 different multi-node clusters on three various cloud platforms, see A for more details. The reason behind this decision was twofold: (i) saving time in collecting experimental data by running topologies in parallel as some of the experiments supposed to run for several weeks, see Section II-B3. (ii) replicating the experiment with different processing node.

C. Experimental analysis

In the following, we evaluate the performance of each approach as a function of the number of evaluations. So for each case, we report performance using the absolute distance of the *minimum function value* from the global minimum. Since we have measured all combinations of parameters in our datasets, we can measure this distance at each iteration.

1) Benchmark functions global optimization: The results for Branin in Figure 10(a) show that BO4CO outperforms the other approaches with three orders of magnitude, while this gap is only an order of magnitude for Dixon as in Figure 10(b). This difference can be associated to the fact that Dixon surface is more rugged than Branin [36]. For Branin, BO4CO finds the global minimum within the first 40 iterations, while even SA stalls on a local minimum. The rest, including GA, HILL, PS perform similarly to each other throughout the experiment and reach a local minimum, which BO4CO finds only within the first 10 iterations. For Dixon, BO4CO gets close to the global minimum within the first 20 iterations, while the best performers (*i.e.*, PS and HILL) approach to points an order of

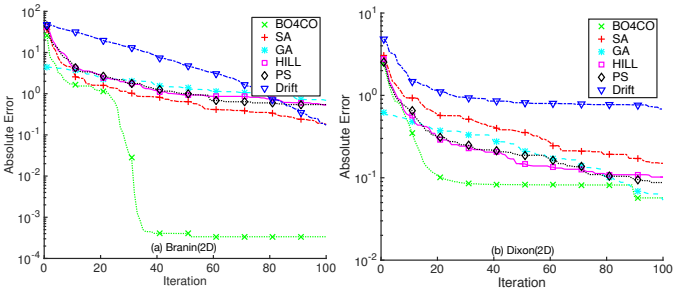

Fig. 10: Branin(2D), Dixon(2D) test function optimization.

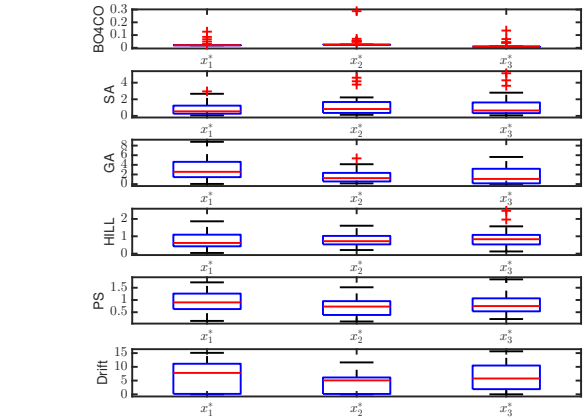

Fig. 11: Distance to the three Branin's minimizers.

magnitude away comparing with BO4CO. Branin function has 3 global minimizers at $x_1^* = (\pi, 12.27), x_2^* = (\pi, 2.27), x_3^* = (9.42, 2.47)$, interestingly comparing to baselines, BO4CO gets close to all minimizers (cf. Figure 11). Therefore, BO4CO gains information on all minimizers as opposed to baselines.

The results for Hartmann in Figure 12(a) show that BO4CO decreases the absolute error quickly after 20 iterations, but only approaches to the global minimum after 120 iterations. Neither of the baseline approaches get close to the global minimum even after 150 iterations. The good performance of BO4CO is also confirmed in the case of Rosenbrock, as shown in Figure 12(b). BO4CO finds the optimum in such large space only after 60 iterations, while GA, HILL, PS and Drift perform poorly with an error of three orders of magnitude higher than our approach. However, SA performs well with an order of magnitude away from the ones found by BO4CO.

2) Storm configuration optimization: We now discuss the results of BO4CO on the Storm datasets in Table I: SOL(6D), RollingSort(6D), WordCount(3D,5D).

The results for SOL in Figure 13(a) show that BO4CO decreases the optimality gap within the first 10 iterations and decreases this gap until iteration 200 and does not get trapped into a local minimum. Instead, baseline approaches like Drift and GA get trapped into a local minimum in early iterations, while HILL and PS get stuck some iterations later at 120. Among the baselines, SA performs the best and it decreases the optimality gap in the first 70 iterations, however, it gets stuck to a local optimum thereafter.

The results for RollingSort in Figure 13(b) are similar to the SOL ones. BO4CO decreases the error considerably in

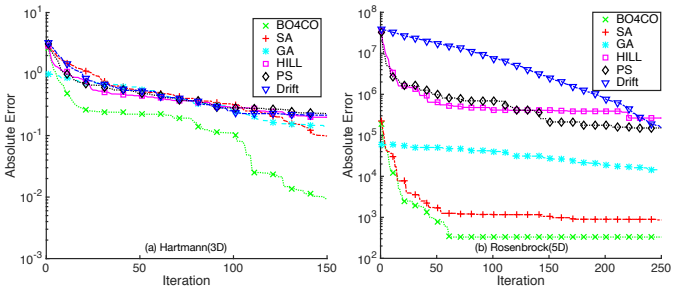

Fig. 12: Hartmann(3D), Rosenbrock(5D) optimization.

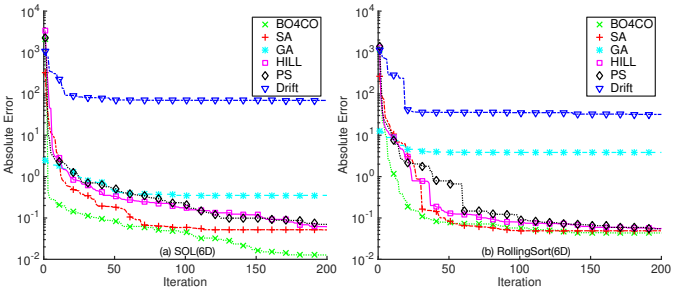

Fig. 13: SOL(6D), RollingSort(6D) optimization.

the first 50 iterations, while the baseline approaches, except SA and HILL, perform poorly in that period. However, during iterations 100-200, the rests, except GA and Drift, find configuration with close performance as the ones BO4CO finds.

For WordCount (Figure 14(a,b)) the results are different. BO4CO outperforms the best baseline performer, *i.e.*, SA, by an order of magnitude, while the others by at least two orders of magnitude. Among the baselines, SA performs the best for WordCount(3D), while for WordCount(5D) dataset, HILL and PS performs better in the first 50 iterations.

Summarizing, while the results for the Storm benchmarks are consistent with the ones we observed for the benchmark functions, it shows a clear gain in favor of BO4CO, with at least an order of magnitude in the initial iterations. In each case, BO4CO finds a better configuration much more quickly than baselines. As opposed to the benchmark functions, SA consistently outperforms the rest of baseline approaches. To highlight this achievement, note that 50 iterations for a dataset like SOL (6D) is only 1% of the total number of possible tests for finding the optimum configurations and identifying such configurations with a latency close to the global optimum can save a considerable time and cost.

D. Sensitivity analysis

1) Prediction accuracy of the learned GP model: Since BO4CO does not uniformly sample the configuration space (cf. Figure 5,8), we speculated that the GP models trained in BO4CO are not useful for predicting the performance of configurations that have not been experimented. However, when we compared the GP model on the WordCount, it was much more accurate than the ones of polynomial regressions (see Figure 15). This shows a clear advantage over design of experiments (DoE), which normally uses first-order and second-order polynomials. The root mean squared error

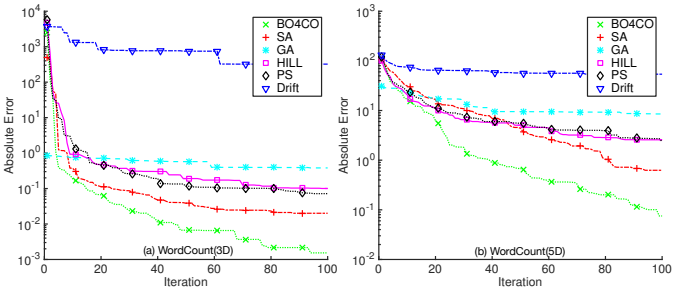

Fig. 14: WordCount(3D,5D) configuration optimization.

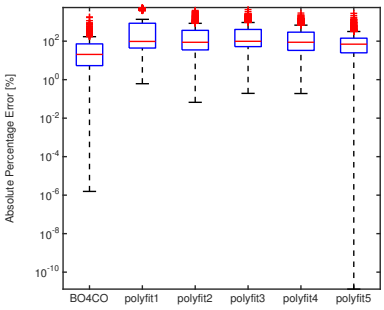

Fig. 15: Absolute percentage error of predictions made by BO4CO's GP fit after 100 iterations vs multivariate polynomial regression models for WordCount(3D) dataset.

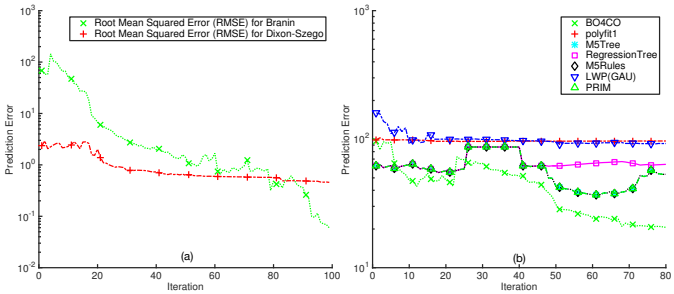

Fig. 16: (a) improving accuracy of GP models over time, (b) comparing prediction accuracy of GPs with other machine learning models on wc(6D).

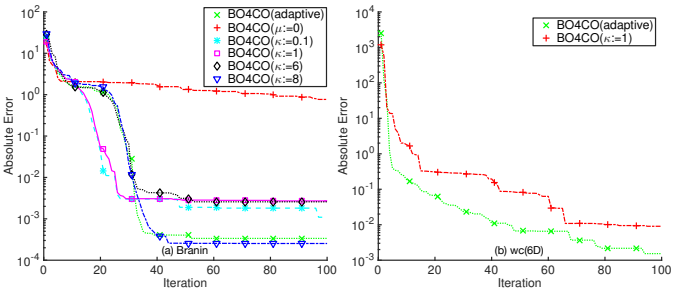

Fig. 17: Exploitation vs exploration (a) Branin, (b) wc(6D).

(RMSE) for Branin and Dixon in 16(a) clearly show that the GP models can provide accurate predictions after 20 iterations. This fast learning rate can be associated to the power of GPs for regressions [37]. We further compared the prediction accuracy of the GP models trained in BO4CO with several machine learning models (including M5Tree, Regression Tree, LWP, PRIM [37]) in Figure 16(b) and we observed that the GP model predictions were more accurate, while the accuracy of other models either did not improve (*e.g.*, M5Tree) or was deteriorated (*e.g.,* PRIM, polyfit5) due to over-fitting.

2) Exploitation vs. exploration: In (10), κ adjusts the exploitation-exploration: small κ means high exploitation, while a large κ means a high exploration. The results in Figure 17(a) show that using a relatively high exploration (*i.e.*, $\kappa = 8$) performs better up to an order of magnitude comparing with high exploitations (*i.e.*, $\kappa = 0.1$). However, for $\kappa = 0.1, 1$ exploiting the mean estimates improves the performance at early iterations comparing with higher explorations cases as in $\kappa = 6$. This observation motivated us to tune κ dynamically by using a lower value at early iterations to exploit the knowledge gained through the initial design and set a higher value later on, see Section III-C. The result for WordCount in Figure 17(b) confirms that adaptive κ improves the performance considerably over constant κ. Figure 18 shows that when we increase κ with a higher rate (cf. Figure 7), it will improve the performance. However, the results in Figure 17(a) suggest using a high value of exploration, this should not be set to an extreme where this makes the mean estimates ineffective. As in Figure 17(a), it performs 4 orders of magnitude worse when we ignore the mean estimates (*i.e.*, $\mu_t := 0$).

3) Bootstrapping vs no bootstrapping: BO4CO uses lhd design in order to bootstrap the search process. The results for Hartmann and WordCount in Figure 19(a,b) confirms that this choice provides a good opportunity in order to explore along all dimensions and not to trap into local optimum thereafter. However, the results in Figure 19(b) suggest that a high number of initial design may deteriorate the goal of finding the optimum early.

V. Discussions

A. Computational and memory requirements

The exact inference BO4CO uses for fitting a GP model to the t observed data is $O(t^3)$ because of inversion of kernel $\boldsymbol{K}^{-1}$ in (7). We could in principle compute the Cholesky decomposition and use it for subsequent predictions, which would lower the complexity to $O(t^2)$. However, since in BO4CO we learn the kernel hyper-parameters every N_ℓ iterations, Cholesky decomposition must be re-computed, therefore the complexity is in principle $O(t^2 \times t/N_\ell)$, where the additional factor of t/N_ℓ counts the expected number of iterations. Figure 20 provides the computation time for finding the next configuration in Algorithm 1 for 5 datasets in Table I. The time is measured running BO4CO on a MacBook Pro with 2.5 GHz Intel Core i7 CPU and 16GB of Memory. The computation time in larger datasets (RollingSort(6D), SOL(6D), WordCount(6D)) is higher than those with less data and lower dimensions (WordCount(3,5D)). Moreover, the computation time increases over time since the matrix size for Cholesky inversion gets larger.

BO4CO requires to store 3 vectors of size $|\mathbb{X}|$ for mean, variance and LCB estimates and a matrix of size $|\mathbb{S}_{1:t}| \times |\mathbb{S}_{1:t}|$ for $\boldsymbol{K}$ and of size $|\mathbb{S}_{1:t}|$ for observations, making the memory requirement of $O(3|\mathbb{X}| + |\mathbb{S}_{1:N_{max}}|(|\mathbb{S}_{1:N_{max}}| + 1))$ in total.

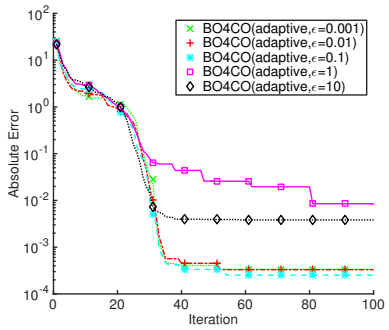

Fig. 18: Exploitation vs. exploration: compare the rate of changes in κ with different value of ϵ in Figure 7.

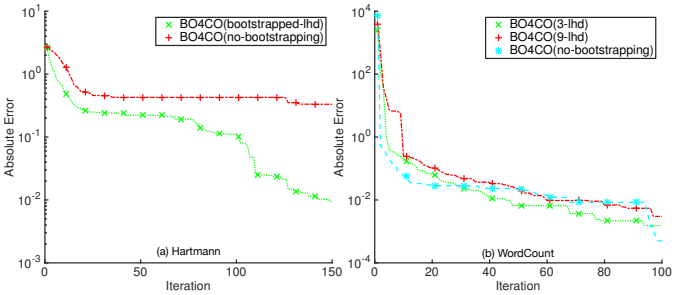

Fig. 19: Bootstrapping vs no bootstrapping: acceleration of performance due to bootstrapping.

B. BO4CO in practice

Extensibility. We have integrated BO4CO with continuous integration, delivery, deployment and monitoring tools in a DevOps pipeline as a part of H2020 DICE project. BO4CO performs as the configuration tuning tool for Big Data systems.

Usability. BO4CO is easy to use, end users only need to determine the parameters of interests as well as experimental parameters and then the tool automatically sets the optimized parameters. Currently, BO4CO supports Apache Storm and Cassandra. However, it is designed to be extensible.

Scalability. The scalability bottleneck is experimentation. The running time of the cubic algorithm is of the order of milliseconds (cf. Figure 20). Each of these experiments takes more than 10 minutes, orders of magnitude over BO4CO.

VI. Related Work

There exist several categories of approaches to address the system configuration problem, as listed in Table II.

Rule-based: In this category, domain experts create a repository of rules that is able to recommend a good configuration. *e.g.*, IBM DB2 Configuration Advisor [21]. The Advisor asks administrators a series of questions, *e.g.*, does the workload is CPU or memory intensive? Based on the answers, it recommends a configuration. However, for multi-dimensional spaces such as SPS in which the configuration parameters have unknown non-linear relationship, this approach is naive [34].

Design of experiments: DoE conducts exhaustive experiments for different combinations of parameters in order to find influential factors [9]. Although DoE is regarded as a classical approach for application configuration, in multi-dimensional

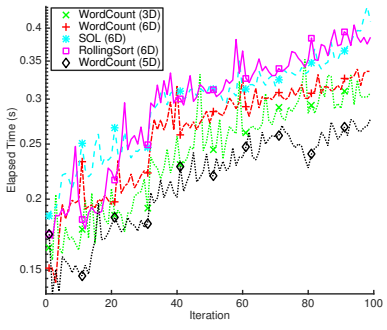

Fig. 20: Runtime overhead of BO4CO (excluding the experiment time) is in the scale of few hundred milliseconds.

spaces performing naive experiments without any sequential feedback from real environment is infeasible and costly.

Model-based: This category conducts a series of experiments where each runs the system using a chosen configuration to observe its performance. Each experiment produces a $(x, f(x))$ sample. A (statistical) model can then be trained from these samples and used to find good configurations. However, an exhaustive set of experiments, usually above the limited budget, need to be conducted to provide a representative data sets, otherwise the prediction based on the trained model will not be reliable (cf. Figure 15). White box [24] and black box models [18], [40], [31] have been proposed.

Search-based: In this approach, also known as *sequential design*, experiments can be performed sequentially where the next set of experiments is determined based on an analysis of the previous data. In each iteration a (statistical) model is fitted to the data and this model guides the selection of the next configuration. Evolutionary search algorithms such as simulated annealing, recursive random search [39], genetic algorithm [1], hill climbing [38], sampling [31] and Covariance Matrix Adaptation [29] have been adopted.

Learning-based: There exists some approaches that employ offline and online learning (*e.g.*, reinforcement learning) to enable online system configuration adaptation [3]. The approaches in this category, as opposed to the other approaches, try to find optimum configurations and adapt it when the situations has been changed at runtime. However, the main shortcoming is the learning that may converge very slowly [17]. The learning time can be shortened if the online learning entangled with offline training [3]. This can be even further improved if we discover the relationship between parameters (*e.g.*, [41], [5]) and exploit such knowledge at runtime.

Knowledge transfer: There exist some approaches that reduce the configuration space by exploiting some knowledge about configuration parameters. Approaches like [5] use the dependence between the parameters in one system to facilitate finding optimal configuration in other systems. They embed the experience in a well-defined structure like Bayesian network through which the generation of new experiments can be guided toward the optimal region in other systems.

Concluding remarks: Software and systems community is not the only community that has tackled such problem. For instance, there exists interesting theoretical methods, *e.g.* best arm identification problem for multi-armed bandit [4], that has been applied for optimizing hyper-parameters of machine

TABLE II: Systems configuration (auto-tuning) approaches.

Category	Empirical	Black box	Interactions	Approaches
Rule-based	No	No	No	[21]
DoE	Yes	Yes	Yes	[35]
Model (white-box)	Partially	No	No	[24]
Model (blackbox)	Yes	Yes	Yes	[18], [40], [31]
Search (sequential)	Yes	Yes	Yes	[38], [27], [8]
Search (evol.)	Partially	Yes	Yes	[10], [1], [39]
Space reduction	Yes	Yes	Yes	[41]
Online learning	Yes	Yes	No	[3]
Knowledge transfer	Yes	Yes	Yes	[5]

Empirical column describes whether the configuration is based on real data. Interactions describes whether the non-linear interactions can be supported.

learning algorithms, *e.g.* supervised learning [16]. More sophisticated methods based on surrogate models and meta-learning have reported better results in different areas, *e.g.*, in propositional satisfiability problem [15], convolutional neural networks [32], vision architectures [2], and more recently in deep neural networks [6].

VII. Conclusions

This paper proposes BO4CO, an approach for locating optimal configurations using ideas of carefully choosing where to sample by sequentially reducing uncertainty in the response surface approximation in order to reduce the number of measurements. BO4CO sequentially gains knowledge about the posterior distribution of the minimizers. We experimentally demonstrate that BO4CO is able to locate the minimum of some benchmark functions as well as optimal configurations within real stream datasets accurately compared to five baseline approaches. We have carried out extensive experiments with three different stream benchmarks running on Apache Storm. The experimental results demonstrate that BO4CO outperforms the baselines in terms of distance to the optimum performance with at least an order of magnitude. We have also provided some evidence that the learned model throughout the search process can be also useful for performance predictions. As a future work, since in the DevOps context several versions of a system are continuously delivered, we will use the notion of knowledge transfer [20], [5] to accelerate the configuration tuning of the current version under test.

Acknowledgment

The research leading to these results has received funding from the European Commission as part of the DICE action (H2020-644869).

References

[1] B. Behzad et al. Taming parallel I/O complexity with auto-tuning. In *Proc. ACM SC*, 2013.
[2] J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. *ICML (1)*, 28:115–123, 2013.
[3] X. Bu, J. Rao, and C.-Z. Xu. A reinforcement learning approach to online Web systems auto-configuration. *In Proc. ICDCS*, 2009.
[4] S. Bubeck, R. Munos, G. Stoltz, and C. Szepesvari. X-armed bandits. *Journal of Machine Learning Research*, 12(May):1655–1695, 2011.
[5] H. Chen et al. Experience transfer for the configuration tuning in large scale computing systems. *SIGMETRICS*, 37(2):51–52, 2009.
[6] T. Domhan, J. T. Springenberg, and F. Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *IJCAI*, 2015.
[7] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. Bigbench: towards an industry standard benchmark for big data analytics. In *SIGMOD*, pages 1197–1208. ACM, 2013.
[8] J. Guo et al. Evaluating the role of optimization-specific search heuristics in effective autotuning. Technical report, 2010.
[9] M. Hall. Correlation-based feature selection for machine learning. *Thesis, Waikato*, 1999.
[10] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.
[11] C. Henard, M. Papadakis, M. Harman, and Y. Le Traon. Combining multi-objective search and constraint solving for configuring large software product lines. In *ICSE*, pages 517–528. IEEE, 2015.
[12] M. Hirzel et al. A catalog of stream processing optimizations. *ACM Computing Surveys*, 46(4):46, 2014.
[13] S. Huang, J. Huang, Y. Liu, L. Yi, and J. Dai. Hibench: A representative and comprehensive hadoop benchmark suite. In *Proc. ICDE*, 2010.
[14] F. Hutter. Automated configuration of algorithms for solving hard computational problems. *Thesis*, 2009.
[15] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *LION*, pages 507–523. Springer, 2011.
[16] K. Jamieson and A. Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. *Preprint available a t*, 2015.
[17] P. Jamshidi et al. Self-learning cloud controllers: Fuzzy Q-learning for knowledge evolution. In *Proc. ICCAC*, 2015.
[18] T. Johnston et al. Performance tuning of mapreduce jobs using surrogate-based modeling. *ICCS*, 2015.
[19] J. P. C. Kleijnen. Response surface methodology. *Wiley Interdisciplinary Reviews*, 2010.
[20] M. Kurek, M. P. Deisenroth, W. Luk, and T. Todman. Knowledge transfer in automatic optimisation of reconfigurable designs. In *FCCM*, 2016.
[21] E. Kwan et al. Automatic database configuration for DB2 universal database: Compressing years of performance expertise into seconds of execution. In *Proc. BTW*, volume 20, 2003.
[22] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *FSE*. ACM, 2013.
[23] D. J. Lizotte et al. An experimental methodology for response surface optimization methods. *Journal of Global Optimization*, 53:699–736, 2012.
[24] D. A. Menascé et al. Preserving QoS of e-commerce sites through self-tuning: a performance model approach. In *Proc. EC*, pages 224–234. ACM, 2001.
[25] D. C. Montgomery. *Design and analysis of experiments*. John Wiley & Sons, 2008.
[26] Z. Nabi, E. Bouillet, A. Bainbridge, and C. Thomas. Of streams and storms. *IBM White Paper*, 2014.
[27] T. Osogami and S. Kato. Optimizing system configurations quickly by guessing at the performance. In *Proc. SIGMETRICS*, volume 35, 2007.
[28] C. E. Rasmussen and H. Nickisch. Gaussian processes for machine learning (gpml) toolbox. *JMLR*, 11, 2010.
[29] A. Saboori et al. Autotuning configurations in distributed systems for performance improvements using evolutionary strategies. *In Proc. ICDCS*, 2008.
[30] B. Shahriari et al. Taking the human out of the loop: a review of bayesian optimization. Technical report, 2015.
[31] N. Siegmund et al. Performance-influence models for highly configurable systems. In *Proc. FSE*, 2015.
[32] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *NIPS*, pages 2951–2959, 2012.
[33] J. Sun et al. Random drift particle swarm optimization. *arXiv preprint arXiv:1306.2863*, 2013.
[34] R. Thonangi et al. Finding good configurations in high-dimensional spaces: Doing more with less. In *Proc. MASCOTS*. IEEE, 2008.
[35] T. Ustinova and P. Jamshidi. Modelling multi-tier enterprise applications behaviour with design of experiments technique. In *Proc. QUDOS*. ACM, 2015.
[36] T. Weise. Global optimization algorithms-theory and application. *Self-Published*, 2009.
[37] C. K. Williams and C. E. Rasmussen. Gaussian processes for machine learning. *MIT Press*, 2006.
[38] B. Xi et al. A smart hill-climbing algorithm for application server configuration. In *WWW*, 2004.
[39] T. Ye and S. Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. *SIGMETRICS*, 31(1):196–205, 2003.
[40] N. Yigitbasi et al. Towards machine learning-based auto-tuning of mapreduce. In *Proc. MASCOTS*, 2013.
[41] W. Zheng, R. Bianchini, and T. D. Nguyen. Automatic configuration of internet services. *ACM SIGOPS Operating Systems Review*, 41(3):219, 2007.

APPENDIX

In this extra material, we briefly describe additional details about the experimental setting and complementary results that were not included in the main text.

A. Code and Data

https://github.com/dice-project/DICE-Configuration-BO4CO

B. Documents

https://github.com/dice-project/DICE-Configuration-BO4CO/wiki

C. Configuration Parameters

The list of configuration parameters in Apache Storm that we have used in the experiments (cf. Table IV):

- max_spout (topology.max.spout.pending). The maximum number of tuples that can be pending on a spout.
- spout_wait (topology.sleep.spout.wait.strategy.time.ms). Time in ms the SleepEmptyEmitStrategy should sleep.
- netty_min_wait (storm.messaging.netty.min_wait_ms). The min time netty waits to get the control back from OS.
- spouts, splitters, counters, bolts. Parallelism level.
- heap. The size of the worker heap.
- buffer_size (storm.messaging.netty.buffer_size). The size of the transfer queue.
- emit_freq (topology.tick.tuple.freq.secs). The frequency at which tick tuples are received.
- top_level. The length of a linear topology.
- message_size, chunk_size. The size of tuples and chunk of messages sent across PEs respectively.

D. Benchmark Settings

Table III represent the infrastructure specification we have used in the experiments (cf. testbed column in Table IV).

TABLE III: Cluster specification

Cluster	Specification
C1	OpenNebula, 3 Sup, 1 ZK, 1 Nimbus, N: (1CPU, 4GB Mem)
C2	EC2, 3 Sup, 1 ZK, 1 Nimbus, N: m1.medium (1 CPU, 3.75GB)
C3	OpenNebula, 3 Sup: (3CPU,6GB Mem), 1 ZK: (1CPU,4GB Mem), 1 Nimbus: (2CPU,4GB Mem)
C4	EC2, 3 Sup, 1 ZK, 1 Nimbus, N: m3.large (2CPU, 7.5GB)
C5	Azure, 3 Sup: Standard_D1(1CPU, 3.5GB) , 1 ZK, 1 Nimbus, N: Standard_A1(1CPU, 1.75GB)

E. Datasets

Note that the parameters with $\star$ shows the interacting parameters. After collecting experimental data, we have used a common correlation-based feature selector implemented in Weka to rank parameter subsets according to a correlation based on a heuristic. The analysis results demonstrate that in all the 10 experiments at most 2-3 parameters were strongly interacting with each other, out of a maximum of 6 parameters varied simultaneously. Therefore, the determination of the regions where performance is optimal will likely to be controlled by such dominant factors, even though the determination of a global optimum will still depend on all the parameters.

TABLE IV: Experimental datasets, note that this is the complete set of datasets that we experimentally collected over the course of 3 months (24/7) for evaluating BO4CO.

	Dataset	Parameters	Size	Testbed
1	wc(6D)	$\star$1-spouts: {1,3}, $\star$2-max_spout: {1,2,10,100,1000,10000}, 3-spout_wait:{1,2,3,10,100}, 4-splitters:{1,2,3,6}, $\star$5-counters:{1,3,6,12}, 6-netty_min_wait:{10,100,1000}	2880	C1
2	sol(6D)	$\star$1-spouts:{1,3}, $\star$2-max_spout:{1,10,100,1000,10000}, $\star$3-top_level:{2,3,4,5}, 4-netty_min_wait:{10,100,1000}, 5-message_size: {10,100,1e3,1e4,1e5,1e6}, 6-bolts: {1,2,3,6}	2866	C2
3	rs(6D)	1-spouts:{1,3}, 2-max_spout:{10,100,1000,10000}, $\star$3-sorters:{1,2,3,6,9,12,15,18}, 4-emit_freq:{1,10,60,120,300}, 5-chunk_size:{1e5,1e6,2e6,1e7}, 6-message_size:{1e3,1e4,1e5}	3840	C3
4	wc(3D)	$\star$1-max_spout:{1,10,100,1e3, 1e4,1e5,1e6}, $\star$2-splitters:{1,2,3,4,5,6}, 3-counters:{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18}	756	C4
5	wc+rs	$\star$1-max_spout:{1,10,100,1e3, 1e4,1e5,1e6}, $\star$2-splitters:{1,2,3,6}, 3-counters:{1,3,6,9,12,15,18}	196	C4
6	wc+sol	$\star$1-max_spout:{1,10,100,1e3, 1e4,1e5,1e6}, $\star$2-splitters:{1,2,3,6}, 3-counters:{1,3,6,9,12,15,18}	196	C4
7	wc+wc	$\star$1-max_spout:{1,10,100,1e3, 1e4,1e5,1e6}, $\star$2-splitters:{1,2,3,6}, 3-counters:{1,3,6,9,12,15,18}	196	C4
8	wc(5D)	$\star$1-spouts:{1,2,3}, 2-splitters:{1,2,3,6}, 3-counters:{1,2,3,6,9,12}, 4-buffer-size:{256k,1m,5m,10m,100m}, 5-heap:{"-Xmx512m", "-Xmx1024m", "-Xmx2048m"}	1080	C5
9	wc-c1	$\star$1-spout_wait:{1,2,3,4,5,6,7,8,9,10,100,1e3,1e4}, $\star$2-splitters:{1,2,3,4,5,6}, 3-counters:{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18}	1343	C1
10	wc-c3	$\star$1-spout_wait:{1,2,3,4,5,6,7,8,9,10,100,1e3,1e4,6e4}, $\star$2-splitters:{1,2,3,4,5,6}, 3-counters:{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18}	1512	C3

F. Performance gain

The performance gain between the worst and best configuration settings are measured for each datasets in Table V.

G. How we set κ

We set the exploration-exploitation parameter κ (cf. Figure 7 and boLCB.m in the github repository)) as:

$$\kappa_t = \sqrt{2\log(|\mathbb{X}|\zeta(r)t^r/\epsilon)}, \zeta(r) = \sum_{n=1}^{\infty}\frac{1}{n^r} \qquad (13)$$

where $0 < \epsilon < 1$ and $r \in \mathbb{N}, r \geq 2$, $\zeta(r)$ is Riemann zeta.

TABLE V: Performance gain between best and worst settings.

	Dataset	Best(ms)	Worst(ms)	Gain (%)
1	wc(6D)	55209	3.3172	99%
2	sol(6D)	40499	1.2000	100%
3	rs(6D)	34733	1.9000	99%
4	wc(3D)	94553	1.2994	100%
5	wc(5D)	405.5	47.387	88%