Noname manuscript No.
(will be inserted by the editor)

CyberCon: Efficient Goal Oriented Configuration of Complex Cyber-Systems

S. Sondur · K. Kant · A. Alazzawe

Received: date / Accepted: date

Abstract Most systems have numerous configuration parameters that must be set properly for acceptable performance, however, numerous dependencies often make proper setting very difficult. In this paper, we propose an approach that combines a machine learning to capture the impact of the parameters along with domain knowledge direct search of the parameter space to efficiently determine the configuration settings. We develop our technique in the context of Cloud Storage Gateway, but the techniques are applicable generally. An extensive evaluation using real world vendor provided workloads shows that the proposed intelligent meta-heuristics reduces the number of iterations by 20-32% in 90% of the cases while yielding a solution with very similar cost.

Keywords Cloud Storage Gateway · Configuration Management · Principal Component Analysis · Decision Trees · Meta-heuristics · Simulated Annealing · Dynamically Dimensioned Search

1 Introduction

The behavior of all cyber-systems in a data center or an enterprise system largely depends on their *configuration*, which describe the settings of various parameters in all of the relevant hardware and software modules. The ill-effects of misconfiguration has been widely articulated as unavailability [19], financial burden [8], security breach [6], etc. However, configuration settings cannot be classified as simply "correct" or "incorrect"; instead, the overall performance of a subsystem and the entire system depends in complex ways on the interactions between values of different configuration parameters. Such dependencies make the performance a non-convex function of the configuration parameters, thereby ruling out simple optimization methods that exploit convexity (e.g., hill climbing). Meta-heuristics are often used to solve

Sanjeev Sondur, Anis Alazzawe, Krishna Kant, Temple University, Philadelphia, PA 19122, USA.
Tel.: +1 215-204-7000, E-mail: sanjeev.sondur|kkant|aalazzawetemple.edu

such optimization problems; however, the objective function in this case is not known explicitly. Therefore, we exploit machine learning to characterize the objective function and then couple it with efficient meta-heuristics to obtain desirable configurations for given performance and/or cost objectives. The key issue in applying meta-heuristics in such a case is the exploitation of domain knowledge to steer the solution towards optimality without being trapped in local optima, and this is an aspect that we focus upon in this paper. It is also worth noting here that while traditional optimization methods provide a single solution to the problem, multiple solutions are often necessary in practice so that the administrator can choose from them based on considerations that are difficult to formalize.

In our previous work [25], we showed that it is possible to use machine learning techniques specifically guided by the domain knowledge to learn these relationships well enough to generate highly accurate performance predictions for given configuration settings (henceforth referred as *forward* problem). We also show that the typical machine learning techniques are unable to achieve good accuracy on the *backward* problem of recommending suitable configuration for given performance or cost targets. Furthermore, a direct machine learning model would be specific to a particular combination of configuration parameters.

The approach explored in this paper rests on two key ideas: (1) use the solution to the *forward problem* as an "oracle" in an combinatorial optimization methodology to search for the most suitable configuration(s) to satisfy the performance and cost objectives, and (2) exploit application domain knowledge to guide the search and thereby achieve significant speedups in the search. Although we explore generally applicable ways of exploiting the domain knowledge, the details, by definition, must be problem specific. Thus, we use the running example of the configuration of a real commercial system, namely a cloud storage gateway (CSG), that essentially creates an impression of unlimited storage with only very limited on-site storage. Commercial CSGs are nontrivial to configure due to many configuration parameters and a wide variety of applications, especially in an edge computing environment that we study here. We explain the complexity of *resource allocation* in CSG, however similar resource allocation challenges arise in configuration Edge and Fog computing, Storage as a Service (SaaS), Containers (VMimages, Dockers), Edge solutions, IoT/remote gateways, etc. These domains exhibit similar behavior and have modeling challenges as highlighted in our work.

To the best of our knowledge this is the first study in devising efficient method for recommending a configuration for complex cyber systems. Our approach is based on a statistical machine learning model for the performance with configuration parameters as input since we find that it is possible to build an accurate model for performance prediction. *The key innovation in our work is to use such a model for the backward problem of setting configuration parameters. We show that suitably chosen combinatorial optimization techniques, enhanced by domain knowledge, can efficiently solve this complex problem.*

In designing *CyberCon*, we use problem specific domain knowledge to enhance stochastic processes and aid the algorithm to converge at a solution faster (i.e. configuration state). In CyberCon, we explore two well studied algorithms -Dynamically Dimensioned Search (DDS), and Simulated Annealing (SA) based meta-heuristics,

and find that modified knowledge enhanced processes outperform their generic counterparts about 32% faster in 90% of the cases.

Our exhaustive search of prior-art showed that earlier research has addressed issues relating to resource provisioning and resource management, while we address the configuration problem as finding (or recommending) the required compute plus storage resource to satisfy a given condition (workload, performance, size, energy, etc.). We exploit the domain knowledge into meta-heuristics algorithm to reduce the (configuration) search space and converge at the required solution.

The remainder of this paper is organized as follows. We start with a discussion of the state of art in configuration setting in section 2. We describe the combinatorial optimization methods and their enhancements to include domain knowledge in section 3. We then describe the characteristics of CSG and the corresponding configuration problem in section 4. Section 4.2 gives details of the experiments and data collection. Section 5 discusses the evaluation methodology, implementation, and comparison of enhanced algorithms against their baseline versions. We conclude the paper in section 6.

2 Current Art on Configuration Selection

Studying the influence of relevant parameters on the performance of a specific system is a standard problem in computer science with essentially unlimited literature; therefore, we review here only generalized approaches for selecting configuration variables. In a recent survey, Zhang [34] categorizes various techniques in resource management in Cloud and Edge computing as latency optimization, shorter task completion time, container placement, data replication, and Edge caching strategies. Wang [31] presents a survey on the impact of Edge caching capacity, delay, and energy efficiency on system performance in a mobile Edge servers. Meta-heuristics based work has been proposed to solve Cloud resource provisioning problems by allocating applications among the virtual machines to satisfy user QoS [15]. In CHOPPER [11], authors present a resource scheduling and provisioning scheme based on QoS metrics (execution time, execution cost, energy consumption, and waiting time).

Klimovic and Costa [7, 14] confirm our observations regarding the difficulty of proper configuration in Cloud storage systems. Rao [23] show that a traditional control theoretic framework is inadequate to capture the complexities of resource allocation for VMs. Ofer [20] study comes close to our work, but their study applies deep learning to cache eviction/refresh techniques in Object-store, rather than setting configuration parameters. Ularu [29] use Decision Trees to configure an application and highlight the use of Decision Trees for solving the configuration problem because of the wide solution space to be explored. Authors in [24] show that managing dispersed Cloudlet infrastructure is very challenging because of the many unknowns pertaining to the software mechanisms and controls.

3 Combinatorial Optimization Based Configuration Selection

The configuration of a cyber-system is defined by a set of user-settable parameters $\vec{x}$ and vendor-selected (usually hidden) parameters $\vec{y}$. These along with the workload parameters parameters $\vec{w}$ determine the desired behavioral parameter ϕ (e.g., throughput, latency, energy consumption, etc.) of the system. A configuration of a system has an associated cost ω that depends on the configuration. That is,

$$\phi = g(\vec{w}, \vec{x}, \vec{y}) \quad (1)$$

$$\omega = f(\vec{x}, \vec{y}) \quad (2)$$

where $f()$ and $g()$ are appropriate functions. For the rest of this paper, we will ignore hidden vendor parameters ($\vec{y}$) as these are not under the control of the user. In general, both $f()$ and $g()$ could be quite complex and thus require a non-explicit model (e.g., a neural net or even a simulation), but we assume here that the cost model $g()$ is given explicitly.

3.1 Basic Approach

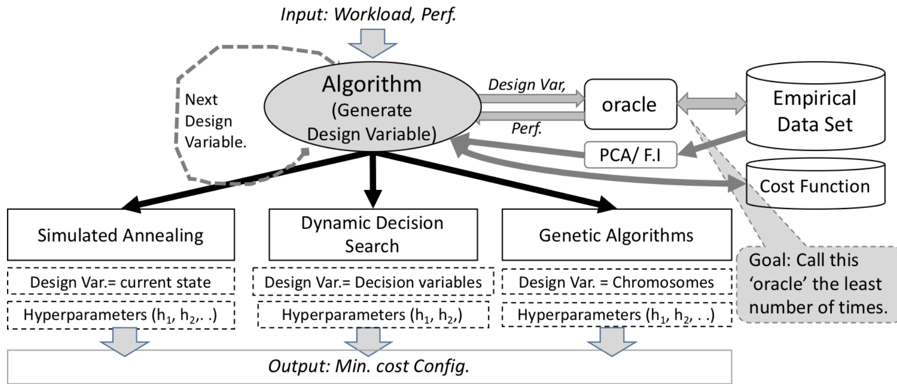


Fig. 1 Design of Algorithms.

Fig. 1 shows the overall scheme explored in this paper. Given a set of configuration variables, the first step is to define an "oracle", or a model for the *forward problem* of performance prediction. Because of complex interdependencies and influences of many configuration parameters, we do this based on machine learning. Such a model will require training on many configurations that encompass at least the practically important ranges of all configuration variables. We take advantage of this data to determine a relative ranking of importance of various parameters via Principal Component Analysis (PCA) or similar analysis. This helps in both confirming and applying the domain knowledge to the solution of the *backward problem* of configuration selection. The same goes for the relationships between various configuration parameters which may be well-understood facts (e.g., higher CPU speed requires lower memory latency), generic rules of thumb (e.g., additional 64MB of memory per additional VDI client), or system specific ones that have been observed or can be deduced from the training data. It is important to underscore the importance of domain knowledge here, since a blind application of these techniques is likely to yield incorrect or misleading results.

The problem to be solved is now to select a configuration $\vec{x}$ (or a few configurations) that provide a performance p above the lower bound (user desired performance p_{user}) while minimizing the cost of the solution. We define a constraint function as:

$$p \geq p_{user} \tag{3}$$

where p is the expected performance from configuration $\vec{x}$. The objective to find such a configuration $\vec{x}$ at a minimum cost.

$$\min(f(\vec{x})) \tag{4}$$

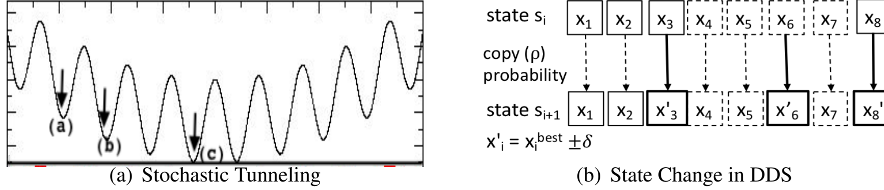
The objective can be expressed as the deployment cost, power consumption, cooling requirements, etc. The cost function is represented as the *normalized* cost of a configuration $\vec{x}$ based on the design variables.

Given the non-convex and non-monotonic influences of various parameters, the use of combinatorial optimization is natural for solving the *backward problem*. In general, this optimization could be either *deterministic* or *stochastic*, where the latter allows for uncertainty in the objective function. Although our interest is in the deterministic case, uncertainties arise naturally in real-world problems (e.g., the cost of the solution better described by a distribution rather than a single value). In the stochastic case, the objective is generally to use a statistical measure (e.g., expected value), so that essentially the same methods apply in both cases. A survey of many popular methods is contained in [4, 12]. The methods include many algorithms inspired by natural phenomenon (e.g., evolution, collective behavior of animals or particles, annealing of metals, quantum annealing, etc.) and some that are not (e.g., stochastic hill climbing or taboo search).

All methods explore a sequence of next states to find a better solution with different techniques to make a trade-off between the expense of exploration (i.e., number of iterations) and the quality of the solution. For the latter, the algorithm must necessarily consider states where the objective function is worse than the optimum found so far, which means that a monotonic convergence is generally not possible. Furthermore, if the algorithm finds a local optimum, it needs to move sufficiently further away so that it does not get trapped in that optimum. An intelligent method to do so is called *"tunneling"* [18], which works in two steps. Suppose that our objective is to minimize an original objective function $f(x)$ where x is the input vector (i.e., the configuration vector in our case). Then starting with the current point, find the local minima, say x^*, using steepest decent algorithm from the current point. Now instead of minimizing the original objective function $f(x)$, minimize a suitably scaled version of $[f(x) - f(x^*)]/\|x - x^*\|$ so as to more strongly reject nearby points where $f(x) > f(x^*)$ (and pick up points with $f(x) < f(x^*)$ with a higher probability). Many different approaches have been proposed in the literature [18] to do this and could be application dependent.

Because of their stochastic nature, these algorithms do not have any guarantee that the result will be optimal for every run. For example, Arsenault [1] compared combinatorial optimization algorithms in hydrological models and showed that different models performed better for different dimensionality and parameter space. Many comparative studies have been put forward to prove or disprove the efficiency of a particular method [10, 2]. Therefore, the focus of our research work is *not* to compare algorithms, rather we show that embedding domain knowledge into stochastic methods

can help the algorithm to converge into a solution faster (than an uninformed algorithm). Our work focused on studying the algorithms for their convergence speed and their capacity to find good objective function minima. In this front, we adapted our solution to two familiar stochastic methods, wiz. Dynamically Dimensioned Search (DDS) and Simulated Annealing (SA). We explain such modifications in the following sections.



(a) Stochastic Tunneling (b) State Change in DDS

Fig. 2 Stochastic Tunneling and Design State Change

3.2 Tunneling through Local Minima

The idea in stochastic optimization process is to "randomly" jump from current state x_i to new state x_{i+1} based on some probability factor ρ, with an aim to find a local minima x^* that minimizes the objective function $f(x)$. Enabling the stochastic algorithm to circumvent the local minima points and rapidly move from an area of "shallow" minima (point (a) in Fig. 2(a)) to a region of "deeper" minima (point (c) in same figure), allows for faster exploration of solution space and faster convergence to a good solution.

We explore ways to "tunnel" through the shallow minima (point (b) in same figure) nd avoid the slow dynamics of complex objective function. Such tunneling mechanisms should invariably use the domain knowledge to intelligently jump the local barriers and avoid uninteresting space. We use PCA metrics (section 5.3) and grouping of design variables as discrete space tunneling mechanism, so as to aid the algorithm from being trapped at a local minimum and (jump through or) tunnel out of the minimum. We present the details in the next sections in context to the two algorithms we explore.

3.3 Modified Simulated Annealing (**mSA**)

Simulated annealing is a general probabilistic local search algorithm, proposed by Cerny and Kirkpatrick et al. [3] to solve difficult optimization problems. The pseudocode [9] for generic SA (gSA) as given in Algorithm 1. In SA, a state refers to a set of design variables and a neighboring state refers to a set of values "relatively" closer to current design variables. In SA, entropy is represented as the cost function that has to be minimized [3]. An acceptable "state" is a solution to the problem that is being solved.

The SA method has been widely used since the cost function can be easy to put into practice [9]. Our SA algorithm uses design variables from the configuration ($\vec{x}$) to represent the state. The entropy of the system is defined as the cost of the current state (i.e cost of the configuration $f(\vec{x})$). The gSA steps in Algorithm 1 can be summarized as follows: (i) we first start with an initial annealing temperature (T_0) and a random design state (line 1), (ii) we search for the next state depending on annealing

Algorithm 1: Pseudocode for Simulated Annealing [9]

```
1  initialize(temperature T, random starting point);
2  while (coolIteration ≤ maxIterations) do
3      coolIteration = coolIteration + 1;
4      select a new point from the neighborhood ■;
5      compute currentCost(at this point);
6      δ = currentCost - previousCost;
7      if δ ≤ 0 then
8          accept neighbor;
9      else
10         accept with probability exp(-δ/T) ;
11     T = β * T ■ ;
```

temperature T_k and a random distribution (line 4), (iii) we compute the difference in entropy (δ) between the current state and past state (line 5,6), and probabilistically accepting the current state depending on Boltzmann probability factor (line $7 \cdots 10$). The annealing scheme is defined in line 11. The algorithm stops after reaching a defined cooling temperature (line 2).

Our solution is based on very fast simulated annealing (VFSA) presented by Xu [32], that enhances both the annealing temperature (line 11) and the perturbation model (line 4). To speed up the convergence rate of SA, VFSA uses Cauchy distribution function as the perturbation [16, 35] which is able to realize a narrower search as the iterative solution approaches an optimum solution, which accelerates the convergence speed [32]. We discuss the supporting functions of gSA & mSA in Table 2 using the following notations: k is the current iteration, n is the number of design variables, T_0 is the initial annealing temperature, α is the damping coefficient ($0 < \alpha < 1$), μ & U are uniform random variable between 0 and 1, $(B_j - A_j)$ is the range of jth design variable ($1 \leq j \leq n$), $\vec{x_i}$ is the configuration (design variables) at i^{th} state, c_i is the configuration cost at i^{th} state, p_i is the predicted performance of configuration $\vec{x_i}$ at i^{th} state, and p_u is user given performance.

mSA uses the annealing scheme T_k and Cauchy distribution perturbation model ζ from VFSA (see Table 2). For acceptance probability ρ, mSA makes a slight modification to accommodate the case where the next solution has the same performance but lower cost. If the acceptance probability for the current state is 1, a new random state is chosen (to avoid getting stuck in local minima) else a new state in the neighborhood is chosen.

Group	**Design Attribute Pairs**
Group G1	Number of Cores, Memory capacity
Group G2	Core speed, Memory bandwidth
Independently varied	Data cache, Disk IO rate

Table 1 Grouping Design Variables

Adding Domain Knowledge to mSA: We incorporate domain knowledge in the algorithm by dividing the design variables into groups based on their level of interdependencies. That is, the design variables within a group show strong interdependence and thus should be set collectively, where the settings across groups can be done independently. In theory, such grouping can be done purely using clustering tech-

Entity	Generic SA	Modified SA
Annealing temp T_k	$T_0 * exp(-\alpha(k-1)^{1/n})$	
Entropy change δ	$exp(\frac{c_i - c_{i-1}}{T_k})$	$(\frac{c_i - c_{i-1}}{T_k})^2$
Acceptance Prob. ρ	1,if $p_i \geq p_u$ 1,if $\delta \leq U(0,1)$ 0,otherwise	1, if$p_i \geq p_u \& c_i \leq c_{i-1}$ 1 ,if $\delta \leq U(0,1)$ 0 ,otherwise
Perturbation Model ζ_j	$T_k(\mu - 0.5)\left[\left(1+\frac{1}{T_k}\right)^{\lvert 2\mu-1\rvert} - 1\right](B_j - A_j)$	
Selecting neighboring state (s_{i+1})	$\begin{cases} \text{random_new_state()}, & \text{if } \rho = 1 \\ \zeta_j, & \text{otherwise} \end{cases}$	
Design Variables	Individually varied	Varied as a group
Evaluation Results	Fig 5	Fig 6

Table 2 Very Fast Simulated Annealing Functions

niques, but this would require a large amount of data and may still result in some unexpected groups. However, by applying the domain knowledge, the grouping can be either done entirely manually, or by coercing the clustering algorithm to prefer certain groupings over others.

For example, in the context of a computer system, it is well understood that a faster CPU should be paired with a faster DRAM, else the CPU will simply stall waiting for the memory. A faster disk is also important, but much less so, since the IOs involve a context switch whereas a memory access does not. Similarly, more CPU cores doing independent work will likely need more memory, and for workloads involving remote IO, both network and IO speeds must increase in tandem. *Grouping* of configuration variables based on insights avoids exploration of states that are unlikely to useful and thus is expected to both speed up the convergence and lead to better solutions within a given number of iterations.

Note that the entropy in a given state s_i is calculated as the cost of the configuration in state s_i, i.e cost($\vec{x_i}$). In the gSA algorithm, acceptance of a solution is defined by Boltzmann probability factor (line $7 \cdots 9$), and represented as a (negative) exponential function of the entropy change δ. Since the negative exponential function tends to zero very rapidly as the entropy increases, it explores only a very small neighborhood in the vicinity of the current solution. We thus choose a function with a longer tail, and found that the square function works quite well, i.e., better than cubic or linear function (Table. 2, entity δ). The evaluation results from the changes to the entities in mSA is discussed in Fig. 5 and Fig. 6.

3.4 Modified Dynamically Dimensioned Search (**mDDS**)

DDS is a greedy algorithm that searches globally at the start of the search and becomes a more local as the number of iterations approaches the iteration limit. The adjustment from global to local search is achieved by dynamically (line 15,19 in Algo. 2) and probabilistically (line 6 in Algo. 2) reducing the number of dimensions in the neighborhood (i.e., the set of decision variables or parameters modified from their best value). The decision variables are the model parameters being calibrated, and the dimension being varied is the number of model parameter values being changed to generate a new search neighborhood. Tolson [28] presented a detailed discussion on DDS. We injected domain knowledge in DDS and evaluated it

for solution convergence. For completeness, we present the pseudo-code of DDS in Algorithm 2. An illustration of design state change is given in Fig. 2(b).

Candidate solutions are created by perturbing the current solution values in the randomly selected dimensions only (line 6,12-21 in Algo. 2 and Fig. 2(b)). These perturbations magnitudes are randomly sampled from a normal distribution with a mean of zero. The algorithm design choices to select a subset of dimensions for perturbation completely at random and *without* reference to sensitivity information.

Algorithm 2: Pseudocode for DDS [28]

```
1  Initialize (perturbation size r, max. evaluation m, vector of lower x^min and upper limits x^max
     of decision variables D);
2  Initial solution x^0 = [x_1, x_2 ··· x_n]; i = 0; x^best = x^0;
3  Eval. init. fitness function F^best = F(x_0);
4  for i in (1 ··· m) do
5  |   // Randomly select J of D decision variables for inclusion in neighborhood {N} based on
   |     probability function of current iteration i P(i) = 1 − ln(i)/ln(m) ■;
6  |   for d in D do
7  |   |   add d to {N} with probability P(i)
8  |   if D is empty then
9  |   |   add one random d to D;
10 |   for j in N decision variables do
11 |   |   σ_j = r(x_j^max − x_j^min) ;
12 |   |   x_j^new = x_j^best + σ_j * RND ;
13 |   |   if x_j^new < x_j^min then
14 |   |   |   // reflect perturbation...■; x_j^new = x_j^min + (x_j^min − x_j^new);
15 |   |   |   if x_j^new > x_j^max then
16 |   |   |   |   x_j^new = x_j^min;
17 |   |   if x_j^new > x_j^max then
18 |   |   |   // reflect perturbation...■; x_j^new = x_j^max − (x_j^new − x_j^max);
19 |   |   |   if x_j^new < x_j^min then
20 |   |   |   |   x_j^new = x_j^max;
21 |   // Eval. new fitness function ;
22 |   if F(x^new) < F(x^best) then
23 |   |   // and update current best if necessary; x^best = x^new; F(x^best = F(x^new);
24 print Output x^best and F^best ;
```

Adding Domain Knowledge in mDDS: We modified the probability criteria in DDS in algorithm Algo. 2 with the probability metrics obtained from PCA. Instead of using the *variable insensitive* ($ln(i)/ln(m)$ at line 5) probability factor, our modified algorithm derives the probability factor defined by the PCA metric. This aids the algorithm to intelligently move the new state based on the influence or the contribution of the variable to the solution. Next, highly influential variables are switched at a higher rate compared to variables of lesser influence. These modification are marked as ■ in line 6, 15 & 19 of the algorithm (Algo. 2). The fitness function in line 6 & 23 is based on Eq. 3 and Eq. 4. That is, we select decision variables x_j that satisfy the constraint and gives the minimum possible cost. At every iteration, the algorithm updates the best fit state F^{best} with the design state x^{best} the satisfies Eq. 3 at the minimum cost (i.e. cost less than previous cost).

4 Configuration Modeling Issue

Given the importance of exploiting the domain knowledge in addressing the configuration management problem, it is important to understand some architectural details of the cloud storage gateway (CSG) that we experimented with and analyzed extensively for this work.

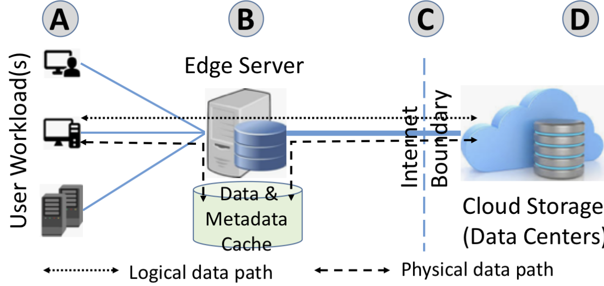


Fig. 3 CSG Architecture

4.1 Configuration Modeling for CSG

A CSG provides a small amount of local storage which is accessed by the applications using traditional block access model. This storage acts like a transparent cache to the vast cloud storage on the backend over the Internet. The cloud storage interface is invariably Object-based because of well-known advantages of the object storage [22]. Fig. 3 shows an abstract representation of CSG. The local storage must provide space for not only the required data but also for relevant metadata needed to locate and access the data. A more detailed description of CSG can be found in our earlier work [26]; here, it suffices to note that CSG is a complex entity and requires proper configuration of computing power, storage, and network elements. It also needs to deal with the uncertainty of transfer rate of objects over the Internet to/from the cloud and the mismatch between the object and block level transfers. Furthermore, the CSG is normally used by many workloads covering a wide range of characteristics and their own requirements in terms of storage access latency and ability to cope with IO timeouts.

We represent the CSG configuration as a combination of required compute and storage resource - number of cores (nc), core speed cs , memory capacity mc, memory bandwidth bw, and disk IO rate di. Workload can be defined by the request arrival rate ar, request size rs, and the metadata size ms. Now, in context of ESI system being studied, we can express Eq. 2 more clearly as:

$$\vec{x} = \{nc, cs, mc, bw, di\} \tag{5}$$

$$\vec{w} = \{ar, rs, ms\} \tag{6}$$

Note that in postulating these functions, we have included only a subset of the parameters that could potentially be relevant. As shown in section 5.2, since simply throwing in arbitrary platform parameters may actually dilute the model and lead to worse results,

We can now represent the configuration problem as achieving the right combination of resources $\vec{x}$ to meet the user defined constraint (Eq. 3) under given condition (Eq. 4).

4.2 Workloads for Evaluating CSG

We used a commercially available CSG to serve a satellite office connected in the back to vendor's cloud storage.[1] Obtaining real workloads is always challenging; therefore, we used a set of vendor provided workload patterns (shown in Table 3), that are reflective of real-world ESI user population. Meta-data shown in the table refer to attributes that influence both the meta-data operations and the performance. These are shown in the meta-data column as ownership (O), sub-directory depth (F=Flat,D=Deep), and object-permission (P).

Request Id	#users x objects x object_size	Meta-Data	Sample Application [30, 21, 33]
W1	25 x 10,000 x 4 KB	O,D,F,P	Health Monitors
W2	25 x 10,000 x 256 KB	O,D,F,P	MRI/ CT Scans/ Traffic Images
W3	5 x 10,000 x 1 MB	O,D,F,P	DICOM Visible Light
W4	5 x 1,000 x 10 MB	O,D,F,P	Mammography/ Street Video(1 min.)
W5	2 x 200 x 1 GB	O,D,F,P	Pathology

Table 3 Sample Workload Type and Applications.

For example, workload patterns for a smart-health monitoring system is characterized as "W1" defined by image size of 4KB, about 10,000 images/24 hrs, with the associated meta-data on date, ownership, location etc. Another workload pattern for health-care (e.g. Pathology) is characterized as "W5" with image size 1GB, about 200 images/24hrs, with meta-data about patient ID, hospital ID, etc.

In order to examine the influence of configuration parameters (compute power, memory resource, cache disk space, etc.) on the final outcome (performance), we executed our tests on different hardware servers of varied configurations given earlier. Disk space r on each of these machines was partitioned for different data cache (dc) size from 25 GB to 1000 GB. Hard disk (i.e. the data cache) in the servers was nfs mounted and connected to Object-store on Cloud OSS. Each of the servers had Ubuntu 14.04, required tools to run the experiments, and collect the metrics. For each of the test execution, the scripts collected the design variables used and performance observed. We defined the cost function (Eq. 2) as a normalized value based on the hardware manufacturer's server specification data, such as power rating, cooling BTU, size, etc. For example, cost C_{ij} = 0.475 is the cost for hardware server h_i (e.g. 2 x 1.8GHz, 32GB mem, 100K IOPS, etc.) and resource r_j (e.g. 500GB data cache, 100GB meta-data).

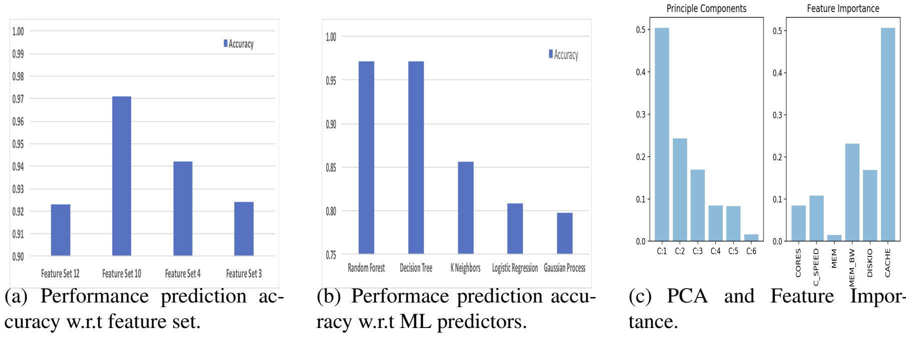
5 Evaluation Results

In this section, we discuss experimental results for both the forward and backward prediction problems discussed earlier.

[1] Due to confidentiality reasons, the vendor name or CSG model is not disclosed here.

5.1 Hyper-parameters of the Stochastic Process

For stochastic process like DDS, or SA, there is no way of knowing a priori which hyper-parameters will secure an optimal solution search [13]. For example, hyper-parameters like (i) initial annealing temperature T_0, damping coefficient α describe a SA process, and (ii) perturbation factor r and maximum iterations m describe a DDS algorithm. We choose the hyper-parameters after several execution, and choose the values that gave the best results. For DDS & mDDS, we used perturbation factor $r = 0.35$ and number of iterations $m = 500$. For gSA & mSA, our initial annealing temperature T_0 is set at 500 and damping coefficient α at 0.15. To enable meaningful comparison of different algorithms, we count the number of calls to *performance-prediction-oracle*, and as explained earlier, a good solution would result in a smaller number of calls to such an oracle. The count of calls to the oracle by different algorithms is shown as iterations (y-axis) in the results.



(a) Performance prediction accuracy w.r.t feature set.

(b) Performace prediction accuracy w.r.t ML predictors.

(c) PCA and Feature Importance.

Fig. 4 PCA, Feature Importance and ML Predictions)

5.2 Influence of Chosen Feature Sets on Performance

As stated earlier, including the configuration parameters (or feature set) without a proper consideration of their relevance not only makes the model more complex but also interferes with the accuracy of the model. We demonstrate this in the following by studying the performance p as a function of configuration parameters ($\vec{x}$). Fig. 4(a) shows the prediction accuracy results for various choice of attributes. In Fig. 4(a), Feature Set 3 includes high level attributes {server, workload, resource} and Feature Set 4 includes additional attributes by expanding the resource {server, workload, data-cache space, meta-data space}. Performance prediction accuracy using these two limited feature-set is about 93%. Feature Set 10 comprise of (Eq.5) $\{cs, nc, mc, bw, di, ar, rs, ms, ds, md\}$, ($ds$ refers to data-cache & md is meta-data space) this results in a higher prediction accuracy of 97%. We verified the results by expanding the feature set with additional attributes.

A blind inclusion of more attributes is labeled as Feature Set 12, which includes additional attributes of network bandwidth (nw) and logfile size (ls). These additional attributes add undesired noise in the data and results in poorer prediction accuracy (down to 91%). Based on our extensive experience and domain knowledge with Edge Storage, we know that this noise is the result of adding unpredictable network bandwidth (nw) and log-file size (ls), both of which do not contribute to the ESI

performance. These results reinforce our earlier comments regarding the selection parameters in Eq. 5 and Eq. 6.

There is no auto-solution for improving the efficiency of ML algorithms, as they depend on the application domain, careful selection of attributes (feature set), and hyper-parameters like regularization parameters, learning rate, etc [27]. Therefore, we tried several types of models and ultimately settled on Decision Tree (DT) for Eq.1, as it consistently performed the best (see Fig.4(b)). Building a performance prediction model for Eq.2 based on Decision Trees yielded an accuracy around 97% for various test-train data combinations (k-fold validation, k=5). An extensive analysis of the model ensured that it does not suffer from under-fit or over-fit.

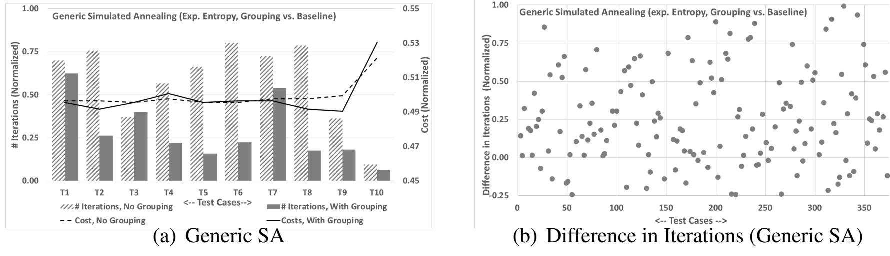
5.3 Extracting Feature Importance

Principal Component Analysis (PCA) is a dimensionality reduction technique that projects the data from its original p-dimensional space to a smaller k-dimensional subspace. PCA maximizes the variance accounted by the first k components and thereby attempts to include those components that have the most influence on the output. The k-dimensional subspace considered by PCA involves components that are linear combinations of the original variables; therefore, we still need to identify the most relevant original variables. In PCA terminology, the contribution of each variable to each principal component is described by *Loadings* [17], which can be easily extracted. Large loadings (positive or negative) indicate that a particular variable has a strong relationship with a particular principal component. The sign of a loading indicates whether a variable and a principal component are positively or negatively correlated.

Feature ablation is a technique for calculating feature importance (FI) that works for all machine learning models. A feature with a high importance has a greater impact on the target variable. We compared both FI from the DT model and PCA & Loadings from the PCA objects to gain confidence in ranking the predominant attributes that contribute to ESI performance. The *scree plot* of PCA and FI for our data-set is given in Fig. 4(c). In the figure, the left sub-graph shows PCA values for different orthogonal components ($C1 \cdots C6$) on x-axis, and the right sub-graph shows FI values for the design variables (on x-axis). Based on the above metrics, PCA & FI provided a reasonable metric to understand the variance of a parameter and its relative contribution towards the performance. Next, we discuss the solution for the *backward problem*, relate to finding the near-optimal configuration $\vec{x}$ that satisfies a given workload/performance criteria (Eq. 3) at minimal possible cost (Eq. 4). We compare the functions against baseline and validate how quickly an algorithm converges at such a required configuration state.

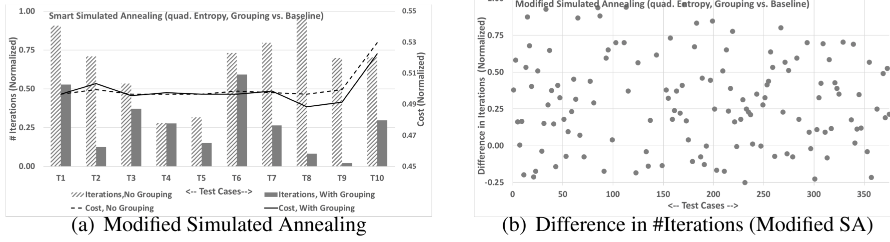
5.4 Recommending a Configuration using mSA

The design variables (CPU, memory, IO bandwidth, etc.) and the workload (file size, meta-data, etc.) formed the "state" of the system in SA. Obviously, the workload variables ($\vec{w}$) are not changed. Initially, we start with a random design variables to represent a configuration state $\vec{x}$, and consult the DT from section 5.2 to verify if the



(a) Generic SA (b) Difference in Iterations (Generic SA)

Fig. 5 Comparing gSA Test Results (Iterations and Cost) with/without Design Attr. Grouping



(a) Modified Simulated Annealing (b) Difference in #Iterations (Modified SA)

Fig. 6 Comparing mSA Test Results (Iterations and Cost) with/without Design Attr. Grouping

current configuration state satisfies the user defined performance (Eq. 3), and the cost of configuration state is computed.

In our mSA approach, we have two variations of choosing the design variables either by domain based grouping or individual perturbation (Table. 1) and two variations of annealing functions (Table. 2). We use gSA as a *baseline*, and present the effect of grouping design variables in Fig. 5. We used the same test cases T1$\cdots$T10 for both SA & DDS, and the y-axis represents the normalized iterations (calls to oracle) required to reach the minimum cost satisfying configuration ($\vec{x}$). Fig. 5(a) shows that a solution with grouping design variables discovers the satisfying configuration faster than a standard approach. The minimal cost of the discovered configuration validates that grouping design variable algorithm (in most cases) matches (or better) the minimal cost compared to generic version (without design variable grouping). In Fig. 5(b), we show more comprehensive results, with each dot referring a test case with different parameter. The figure clearly shows that mSA with grouping outperforms the gSA in about 90% of the test cases. Each dot on the positive side refers to the case where the mSA algorithm reaches the solution faster than mSA.

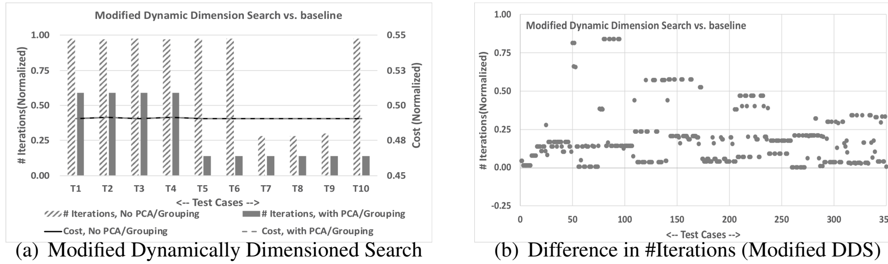
We now focus on mSA algorithm with modifications as explained in section 3.3. While the baseline gSA computes the delta entropy δ based on exponential function, mSA algorithm computes the same entropy as a quadratic function. Again, mSA is compared based on two criteria, (i) with individual perturbation of each design variable, and (ii) perturbation of design variables as groups (Table.1). The evaluation results of mSA is shown in Fig. 6 for same test cases T1$\cdots$T10 and y-axis shows iterations as a normalized metric. In each of these test cases, mSA with grouping of the design variables performs better compared to non-grouping, and reaches the desired configuration faster (see Fig. 6(a)), and simultaneously matching or outperforming the minimal cost function.

We summarize the results from several other tests in Fig. 6(a), with each dot representing a test case on the x-axis and the normalized difference in #iterations on the y-axis. As seen from the figure, mSA outperforms gSA in 90% of the cases. For test cases where mSA performs better than gSA, it takes 32% fewer iterations than gSA on the average, i.e. on an average, mSA reaches the required solution about 32% faster than gSA.

5.5 Recommending a Configuration using mDDS

The current state x_j^{new} in Algo. 2 is a combination of workload variables ($\vec{w}$, kept constant as explained earlier) and the configuration state defined by the design variables ($\vec{x}$). We use x_j^{new} to consult the oracle from section 5.2 to validate the fitness function. This steps predicts the achievable performance (Eq. 3) and then the cost of configuration state is computed. In mDDS, we apply an alternative probability distributions using PCA metrics as given in section 3.4. The algorithm finds the best configuration x_{best} that satisfies the user required performance at a minimum cost.

DDS is a simple greedy algorithm defined by only two hyper-parameters (perturbation size r & #iterations m), and hence easy to tune. We evaluated the same set of test cases used earlier and present the results in Fig.7. Fig.7(a) shows the convergence results of the 10 test cases $T1 \cdots T10$ (used earlier). In mDDS, with the PCA driven probability factor, the decision variables find the satisfying fitness function earlier compared to gDDS. We computed the difference in algorithm convergence for an extended test case. An extended study of over 350+ test cases at various hyper-parameter r values is shown in Fig.7(b). mDDS converges about 20% faster in more than 90% of the cases.



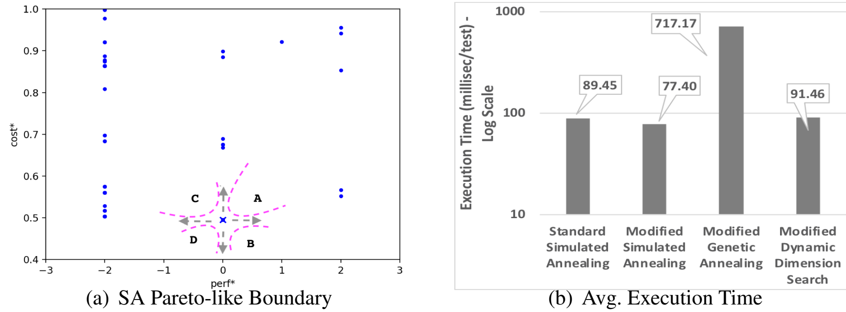
(a) Modified Dynamically Dimensioned Search (b) Difference in #Iterations (Modified DDS)

Fig. 7 Comparing mDDS Test Results (Iterations and Cost) with/without Domain Knowledge.

5.6 Discussion on Results

We executed all the algorithms for several test cases and captured the execution time per test case. All metrics were captured under identical conditions, i.e. the tests running on the same server with minimal overhead from unwanted OS processes. The average execution time per test case is shown in Fig. 8(b). The data shows that mSA is about 10% faster than gSA, and GA is considerably slow. Similar findings is reported by Keer [13] and Ferentinos [9], who found that for each iteration of genetic algorithm is considerably more expensive than simulated annealing. DDS and SA closely match each other wrt execution time.

Our interest in this work is to satisfy a user constraint on given performance, rather than trying to solve for two objectives (i.e. a maximum performance and minimal cost). A design is considered Pareto optimal [5] if there does not exist any other design which improves the value of any of its objective criteria deteriorating at least one other criterion. Using *similar concept*, we present a Pareto "like" boundary to show that the end results satisfy constraints (Eq. 3) and the objective (Eq. 4).



(a) SA Pareto-like Boundary

(b) Avg. Execution Time

Fig. 8 Function Boundary and Exec Times.

Pareto "like" boundary from one of our test cases is shown in Fig. 8(a), with performance class on the x-axis and cost on the y-axis. Each point in the graph refers to a configuration point from the mSA algorithm. Our optimal point is shown as "x" in the figure, with *A,B,C,D* showing other possible solution "areas". Any point towards A would refer to the required performance (or higher), but at higher costs - hence not a desired solution. Any points towards C or D is undesired since it refers to lesser than desired performance. *If the algorithm found* any points in B (other than x), it would be desirable since it refers to both satisfying performance and lesser costs. However, a rightful solution like mSA would find the most optimal solution **at** point x, and no points in B area (and rightfully so).

6 Conclusions

In this paper, we presented *CyberCon*, an efficient goal oriented methodology to recommend an optimal configuration for a complex cyber-system and illustrated its application to configuring a cloud storage gateway. We proposed a meta-heuristics based approach aided by domain knowledge (in this case DDS & SA) and machine learning techniques (Decision Tree). We have shown that the proposed approach can robustly identify suitable values of configuration parameters that satisfy the target performance and cost constraints. The results show that the approach yields results closer to the optimal value about 30% faster than the standard versions of the algorithms.

We envision that our approach can provide good configurations in almost any other context where one can establish the relative importance of various configuration parameters and relationships between them (via automated techniques like PCA or based on the experience with system).In the future, we will examine formal ways of encoding such domain knowledge so it can be exploited easily in a large variety of applications.

References

1. Arsenault R, Poulin A, Côté P, Brissette F (2014) Comparison of stochastic optimization algorithms in hydrological model calibration. Journal of Hydrologic Engineering 19(7):1374–1384
2. Barrette M, Wong T, De Kelper B, Côté P (2008) Statistical multi-comparison of evolutionary algorithms. Bioinspired Optimizaiton Methods and their Applications p 71
3. Ben-Ameur W (2004) Computing the initial temperature of simulated annealing. Computational Optimization and Applications 29(3):369–385
4. Bianchi L, Dorigo M, Gambardella LM, Gutjahr WJ (2009) A survey on metaheuristics for stochastic combinatorial optimization. Natural Computing: An International Journal 8(2):239–287
5. Chinchuluun A, Pardalos PM (2007) A survey of recent developments in multiobjective optimization. Annals of Operations Research 154(1):29–50
6. Chirgwin R (2017) Suspicious BGP event routed big traffic sites through Russia
7. Costa LB, Ripeanu M (2010) Towards Automating the Configuration of a Distributed Storage System. In: 2010 11th IEEE/ACM International Conference on Grid Computing, pp 201–208, DOI 10.1109/GRID.2010.5697971
8. Elliot S (2017) Amazon. com goes down, loses $66,240 per minute
9. Ferentinos KP, Arvanitis KG, Sigrimis N (2002) Heuristic optimization methods for motion planning of autonomous agricultural vehicles. Journal of Global Optimization 23(2):155–170
10. Franchini M, Galeati G, Berra S (1998) Global optimization techniques for the calibration of conceptual rainfall-runoff models. Hydrological Sciences Journal 43(3):443–458
11. Gill SS, Chana I, Singh M, Buyya R (2018) Chopper: an intelligent qos-aware autonomic resource management approach for cloud computing. Cluster Computing 21(2):1203–1241
12. Gutjahr W (2011) Recent trends in metaheuristics for stochastic combinatorial optimization. Open Computer Science 1(1):58–66
13. Kerr A, Mullen K (2019) A comparison of genetic algorithms and simulated annealing in maximizing the thermal conductance of harmonic lattices. Computational Materials Science 157:31–36
14. Klimovic A, Litz H, Kozyrakis C (2018) Selecta: heterogeneous cloud storage configuration for data analytics. In: 2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18), pp 759–773
15. Kumar M, Sharma S, Goel S, Mishra SK, Husain A (2020) Autonomic cloud resource provisioning and scheduling using meta-heuristic algorithm. Neural Computing and Applications
16. Lee CY (2015) Fast simulated annealing with a multivariate cauchy distribution and the configuration's initial temperature. Journal of the Korean Physical Society 66(10):1457–1466
17. Legendre P, Legendre LF (2012) Numerical ecology. Elsevier
18. Levy AV, Montalvo A (1985) The tunneling algorithm for the global minimization of functions. SIAM J Sci and Stat Comput 6(1):15–29

19. Newman LH (2017) How a tiny error shut off the internet for parts of the US
20. Ofer E, Epstein A, Sadeh D, Harnik D (2018) Applying deep learning to object store caching. In: Proceedings of the 11th ACM International Systems and Storage Conference, ACM, New York, NY, USA, SYSTOR '18, pp 126–126
21. Oracle (2010) Performance Evaluation of Storage and Retrieval of DICOM Image Content ...
22. Prahlad A, Muller MS, Kottomtharayil Re (2012) Data object store and server for a cloud storage environment, including data deduplication and data management across multiple cloud storage sites. US Patent 8,285,681
23. Rao J, Bu X, Xu CZ, Wang L, Yin G (2009) VCONF: A Reinforcement Learning Approach to Virtual Machines Auto-configuration. In: Proceedings of the 6th International Conference on Autonomic Computing, ACM, New York, NY, USA, ICAC '09, pp 137–146
24. Satyanarayanan M (2017) The emergence of edge computing. Computer 50(1):30–39
25. Sondur S, Kant K (2019) Towards automated configuration of cloud storage gateways: A data driven approach. In: International Conference on Cloud Computing, Springer, pp 192–207
26. Sondur S, Kant K, Vucetic S, Byers B (2019) Storage on the edge: Evaluating cloud backed edge storage in cyberphysical systems. In: 2019 IEEE 16th International Conference on Mobile Ad Hoc and Sensor Systems (MASS), pp 362–370
27. Sorower MS (2010) A literature survey on algorithms for multi-label learning. Oregon State University, Corvallis 18
28. Tolson BA, Shoemaker CA (2007) Dynamically dimensioned search algorithm for computationally efficient watershed model calibration. Water Resources Research 43(1)
29. Ularu EG, Puican FC, Suciu G, Vulpe A, Todoran G (2013) Mobile Computing and Cloud maturity-Introducing Machine Learning for ERP Configuration Automation. Informatica Economica 17(1)
30. Varma R (2008) Storage media for computers in radiology. The Indian journal of radiology and imaging 18:287–9
31. Wang S, Zhang X, Zhang Y, Wang L, Yang J, Wang W (2017) A survey on mobile edge networks: Convergence of computing, caching and communications. IEEE Access 5:6757–6779
32. Xu Y, Ye Q, Meng G (2018) Hybrid phase retrieval algorithm based on modified very fast simulated annealing. International Journal of Microwave and Wireless Technologies 10(9):1072–1080
33. Yi S, Li C, Li Q (2015) A survey of fog computing: Concepts, applications and issues. In: Proceedings of the 2015 Workshop on Mobile Big Data, ACM, New York, NY, USA, Mobidata '15, pp 37–42
34. Zhang Y, Xu K (2020) A survey of resource management in cloud and edge computing. In: Network Management in Cloud and Edge Computing, Springer, pp 15–32
35. Zhao LS, Sen MK, Stoffa P, Frohlich C (1996) Application of very fast simulated annealing to the determination of the crustal structure beneath Tibet. Geophysical Journal International 125(2):355–370