Effective Configuration Optimization of Large Scale Software Systems

Negar Mohammadi Koushki, Sanjeev Sondur, and Krishna Kant
Computer and Information Sciences
Temple University
Philadelphia, USA
{koushki|sanjeev.sondur|kkant}@temple.edu

***Abstract*—Most large-scale applications define an extensive set of run-time configuration variables, which must be set properly for the application to behave well and satisfy the performance requirements. Due to the complex interdependencies between parameters, a proper setting is often quite challenging. In this paper, we propose a modified meta-heuristic based discrete combinatorial optimization technique for setting the configuration parameters. The proposed method has two unique elements: (a) efficient use of implicit performance function (such as the one produce by machine learning), and (b) exploitation of the application domain knowledge in grouping and choosing the parameters for perturbation. Our extensive evaluation using available workload traces that do include configuration information shows that the proposed technique can provide a lower cost solution (by $\sim$62%) with faster convergence (by $\sim$45%) as compared to the traditional meta-heuristic algorithms. Also, our solution succeeds in finding an average of 30% *more additional* solutions than the baseline.**

***Index Terms*—Configuration Modeling, Resource Allocation, Resource Provisioning, Machine Learning, Meta-heuristics, Simulated Annealing**

I. Introduction

Configurations are parameter name-value settings that can be changed at run-time and impact many aspects of the system behavior, including its performance and cost. Although any particular entity in the system may have only a few configuration parameters, there is a large number of configurable entities starting with physical components and moving up the hierarchy into virtualized entities, service layers, libraries, middleware, etc. A net result is a huge number of configuration parameters that must be set properly for a performant system without unreasonable resource usage. Although many of the parameters may be "baseboard" and not changeable, this still leaves anywhere from 10s to 1000s of settable parameters [1]. It is also important to note that while in this paper we only consider performance and cost, many other parameters also influence these indirectly. For example, strong encryption or frequent authentication can affect performance significantly. Similarly, a limit on power consumption can degrade the performance.

The impact of configurable parameters on the system is often quite complex with numerous dependencies which are either not known or not well understood. The problem only gets worse with high level configuration parameters that are often provided to ease the configuration job. In most cases, the effect of the user configurable parameters on the behavior of a system cannot be easily expressed as an analytical function because of the non-linearity and interdependencies [2]. An ill configured system can exhibit undesired symptoms such as sub-optimal utilization of resources, poor performance, low availability, etc. Thus, finding a suitable configuration for a system (i.e. the combination of *name:value* pairs) to satisfy the desired behavior is the main operational challenge sought out by the data center operators.

As stated above, in this paper, we focus on configuration from the performance perspective; other goals, not addressed here, may include security, availability, maintainability, etc. The performance is obviously limited by the resources used, which can be expressed in form of a generalized cost metric. For example, in a virtualized environment, the cost may refer to either the actual cost charged by the provider (e.g., AWS) or a cost that we decide to assign to the use of various resources. We will primarily speak of determining values of configuration parameters that minimize the cost subject to achieving a given performance threshold, but one could similarly also consider the problem of maximizing performance subject to a cost threshold.

We approach the problem as a discrete combinatorial optimization problem due to the lack of an explicit function to express performance as a function of configuration parameters. The conflicting impacts of various parameters and the dependencies across them invariably make the problem non-convex, thereby ruling out simple optimization methods such as hill climbing. Meta-heuristics are often used to solve such problems and numerous methods have been developed [3]. All methods aim to explore the state space efficiently while avoiding being trapped in the local minima, of which there could be many. Fig. 1 illustrates the search process pictorially with x-axis representing the iterations and y-axis the solution obtained in each iteration. The algorithm will keep track of the minima obtained so far, and may or may not discover the global minima until the maximum iteration count (a hyperparameter of the algorithm) is reached.

Since our problem involves constrained optimization, we also need to ensure that any accepted solution is feasible. In such a setting, it is always helpful to avoid generating infeasible solutions in the first place, but this is not always possible. This is where domain knowledge is crucial. Often, the domain knowledge consists of an abstract relationship between configuration parameters or rules of thumb that can be evaluated easily. However, since they are fuzzy, and not strictly required, they cannot be used as formal constraints. Another kind of domain knowledge concerns the varying influence of parameters that can guide which parameters need to be perturbed and by how much to get to the next proposed solution. Yet another aspect concerns an estimate of the amount by which one needs to move to get out of the region of local optimality to land in another region that can possibly provide a lower local optimum.

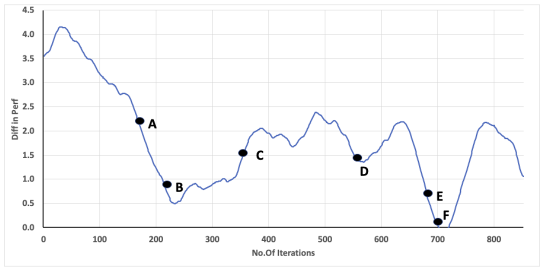


Fig. 1: Stochastic Tunneling.

A useful notion in stochastic optimization is *tunneling* [4] illustrated in Fig. 1, where we "tunnel" from a local minima to a deeper local minima directly [5]. Traditional tunneling works in the continuous parameter state space with explicit objective function $f(x)$ where x is the input vector (i.e., configuration vector in our case). It also assumes that $f(x)$ has first two derivatives. The method works in two steps: (a) find the local minima, say x^*, using the steepest decent algorithm from the current point, (b) minimize the modified function $h(x) = [f(x) - f(x^*)]/\|x - x^*\|^\alpha$ with $\alpha \geq 1$ to determine the next solution, say x'. It can be seen that with larger α, nearby points are penalized. (The choice of α can be problematic and other approaches have been investigated [13]). Thus if we choose x' based on the gradient of $h(x)$ away from x^*, we are more likely to go towards a deeper minima than at x^*.

Although such a mechanism cannot be applied this technique directly to our problem, due to implicit $f(x)$ and discrete parameter space, but we show that such a technique can improve the solution performance considerably.

The approach explored in this paper rests on three key ideas: (1) use the solution to the *behavior prediction problem* as an "oracle" in a combinatorial optimization to pick a feasible solution, (2) exploit application domain knowledge to guide the search, and (3) use an analogue of tunneling in the discrete space of our problem. Although the domain knowledge exploitation methods used in our approach are applicable generally, it is imperative that the modeler must have both the working knowledge and a decent amount of configuration related data to test the methodology.

Our research proposes an efficient method for recommending a configuration for software systems and makes the following contributions:

1) In the absence of a clear analytical function, we use machine learning (ML) based behavior prediction model as a surrogate to the objective and/or the constraint functions,
2) We exploit the domain knowledge that might be fuzzily stated, with metrics from ML model to reduce examination of the undesired portions of the search space,
3) We significantly reduce exploration of the states near a local minima by perturbing the design variables at relative rates based on their influence on the outcome, and "tunnel" to a probable area in search of a better solution, thereby making the search more efficient.

II. Current Art on Configuration Selection

Current state of art explored during our work shows that configuration issues are related to resource provisioning and resource management [6] techniques to optimize latency, task completion time, data replication, and impact on cache capacity, delay, and energy consumption, however our work addresses configuration management as *recommending* a suitable resource allocation (e.g. storage, compute, bandwidth) to achieve the desired goal (e.g. workload performance, energy, cost, size, etc). Meta-heuristics approach has been used to provision Cloud resources for satisfying QoS [7].

To overcome the difficultly in characterizing the behavior outcome (e.g predicting performance), several studies have used Classification Regression Trees (CART)-based model [8] and ML techniques [9] to design a performance influencing model (PIM) [10]. In our study, PIM is only the first step to build a surrogate function to solve the combinatorial optimization problem. Our work focuses on *choosing a set of configuration parameters* that satisfy user workload/performance demands under given constraints.

III. Combinatorial Optimization Based Configuration Selection

Let $\mathcal{C}$ denote the configuration defined as the vector of configuration variables $\vec{x}$ and their values $\vec{y}$. These along with the workload parameters $\vec{w}$ determine the desired objective function ϕ subject to some constraints. That is,

$$\vec{\mathcal{C}} = \{\vec{x}, \vec{y}\} \tag{1}$$

$$\phi = f(\vec{w}, \vec{\mathcal{C}}) \tag{2}$$

$$\omega_i^{(l)} \leq g_i(\vec{\mathcal{C}}) \leq \omega_i^{(h)} \qquad i = 1, 2, .., K \tag{3}$$

where the functions $f()$ and $g_i()$'s are usually quite complex and may not be expressible explicitly. Our study is supported by Ref [11], wherein authors state that performance modeling is complex since the running time (performance or throughput) is affected by the amount of resources in the cloud configuration in a non-linear way and performance under a cloud configuration is not deterministic. It is also worth noticing that the configuration space is often discrete with intermediate values being practically infeasible, even if they are conceptually meaningful. For example, if the memory modules for the systems at hand have a minimum granularity of 16GB, an installed memory of 24GB is infeasible. Thus, defining continuous or differentiable extensions of the functions $f()$ and $g_i()$ is neither straightforward, nor meaningful. Thus, the traditional tunneling structure is not possible; also, while one could estimate the local gradient by evaluating the functions at nearby feasible points, the value of local search is less clear.

Behavior $f()$ is expressed as the user expectation and can refer to performance, latency, throughput, etc. The constraints can represent the cost factor of such a system, heat dissipated or cooling needs, energy consumed, physical size, etc. Often, it is desirable to optimize multiple parameters simultaneously; however, in this paper, we consider objectives and constraints as a singular function.

A. Basic Approach

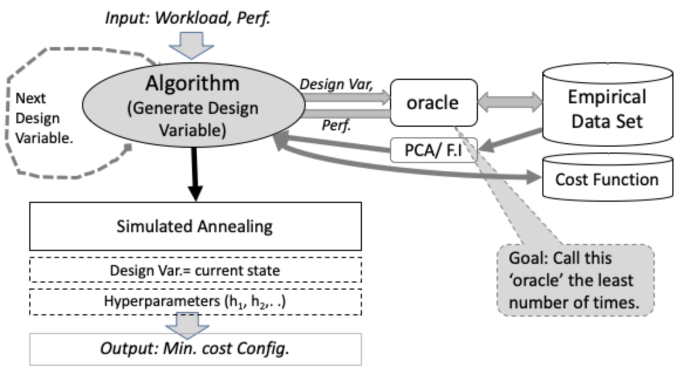

Input: Workload, Perf.
Next Design Variable.
Algorithm (Generate Design Variable)
Design Var,
Perf.
oracle
Empirical Data Set
PCA/ F.I
Cost Function
Simulated Annealing
Design Var.= current state
Hyperparameters (h1, h2,. .)
Output: Min. cost Config.
Goal: Call this 'oracle' the least number of times.

Fig. 2: Design of Algorithms.

Fig. 2 shows the overall scheme explored in this paper. Given a set of configuration variables (CVs), the first step is to define an "oracle", or a model for the **forward problem** of performance prediction based on the settings of CVs. As discussed in [17], statistical machine learning (ML) techniques work quite well for this. We have also shown that the ML techniques do not work well for the **backward problem** of configuration selection and require large amounts of training data [22]. We take advantage of the corresponding training 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 such as various relationships (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_u) while minimizing the cost of the solution. The choice of parameter values in $\vec{x}$ can directly or indirectly affect the objective function, i.e. ω cost of the configuration.

We now define the constraint as the desired performance p_u:

$$p = \phi \geq p_u \tag{4}$$

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

$$\min(g(\vec{x})) \tag{5}$$

The "cost" of a configuration can represent a user desired metric such as the deployment cost, power consumption, cooling requirements, etc. Data for cost function can be derived from vendor specification for hardware server and allocated resources (e.g., disk capacity) or other suitable function $g()$. For example, k^{th} configuration $\vec{x_k}$ for some choice of parameters such as number of CPU cores, core speed, memory bandwidth, IO bandwidth, storage capacity, etc. has a cost $g(\vec{x_k})$.

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.

All stochastic 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, cost of evaluation) 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.

Because of their stochastic nature, these algorithms do not have any guarantee that the result will be optimal for every run. Many comparative studies have been put forward to prove or disprove the efficiency of a particular method [12]. 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 a familiar stochastic method, wiz. Simulated Annealing (SA). We explain such modifications in the following sections.

B. Emulating Tunneling

To apply Tunneling like approach to our context, we first seek the local minima (step 1) followed by an intelligent perturbation based on the sensitivity to take us further away from the current minima (step 2). To prepare for this, we first determine the relative ranking of each CV in terms of its influence on the objective function and the complex constraints. A purely data-driven method for doing this is the standard principal component analysis (PCA), and should work well assuming a sufficiently large data-set. Starting with the most dominant parameter, we form a group by pulling in other parameters that are known to be related to it (based on domain knowledge). We then start with the next most dominant parameter and form the next group, until we have covered all the parameters. Then our approach with "smart" tunneling can be described by the following two steps:

1) We approach the local minima by considering the variation with respect to the leader of the first group. The gradient needs to be determined by taking a few samples.
2) The tunneling phase then perturbs each group leader in proportion of the PCA metric and adjusts all others in the group based on the known relationships

A stochastic optimization process works by randomly jumping from the current state x_i to a 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 $f(x)$. We apply the above approach with tunneling by enabling the stochastic algorithm to circumvent the local minima points and rapidly move from an area of shallow minima (point (a) in Fig. 1) to a region of deeper minima (point (d) in the same figure), thereby allowing for faster exploration of solution space and faster convergence to a good solution. With a configuration problem at hand, as our objective function cannot be characterized by direct analytical function, we use the performance prediction oracle (a black box) functions as a objective function.

We explore ways to tunnel through the shallow minima (point (b) in the same figure) and avoid the slow dynamics of the complex objective function. Such tunneling mechanisms should invariably use the domain knowledge to intelligently jump the local barriers and avoid uninteresting space (i.e avoid point (d and e) in the same figure). We group the design variables together as discrete space tunneling mechanism, 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 algorithm we explore.

C. Generic Design Approach Summarized

The approach described above can be generalized independent of the data-set and domain as:

1) Run experiments to collect the data with relevant configurable parameters and observable outcome.
2) In absence of a clear analytical function to describe the relationship between the configurable parameters to outcome, use suitable a ML model to design an "oracle" as a prediction engine (a.k.a PIM).
3) Use PCA metrics from the data, determine relative importance of design variables and group attributes based on domain knowledge to avoid exploring undesired spaces.
4) Use the above PIM model to represent the objective and/or constraint function in a stochastic algorithm.
5) To explore new design states, use the PCA metrics as probability factor, and pertrubate the variables in-groups.
6) Verify new state satisfies constrain using ML based oracle as a tool.
7) Accept/Reject current design state based on satisfying criteria.

We explain the modification to a familiar meta-heuristics based stochastic processes, i.e Simulated Annealing, below.

*D. Modified Simulated Annealing (**mSA**)*

Simulated annealing (SA) is a general probabilistic local search algorithm generally used to solve difficult optimization problems. The pseudocode [14] for generic SA (gSA) is 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. An acceptable state is a solution to the problem that is being solved.

Algorithm 1: Pseudocode for SA [14]

```
1  initialize(temperature T, random starting point) ;
2  for  $i$ $in$ $T$ do
3      p = select_point_from_neighborhood(i) ■ ;
4      currentCost = compute_currentCost_at(p) ;
5      $\delta$ = currentCost - previousCost ;
6      if $\delta \leq 0$ then
7          accept_neighbor_point(p) ;
8      else
9          accept with probability exp(-$\delta$/T) ♦ ;
10     T = $\beta$ * T ■ ;
```

The SA method has been widely used since the cost function can be easy to put into practice [14]. 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 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), and probabilistically accepting the current state depending on Boltzmann probability factor (line $6\cdots9$). At line 9, if the current solution is accepted, we apply tunneling logic to search for a better local minima. The annealing scheme is defined in line 10. The algorithm stops after reaching a defined cooling temperature (line 2).

Our solution is based on very fast simulated annealing (VFSA) presented by Xu [15], that enhances both the annealing temperature (line 10) and the perturbation model (line 4). Lee [16] and others have discussed VFSA in detail and show the advantages of VFSA over SA. To speed up the convergence rate of SA, VFSA uses Cauchy distribution function as the perturbation [16] which is able to realize a narrower search as the iterative solution approaches an optimum solution, which accelerates the convergence speed [15].

We discuss the supporting functions of gSA and mSA in Table I 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

TABLE I: Very Fast Simulated Annealing Functions

Entity	**gSA**	**mSA**
Annealing temp T_k	$T_0 * exp(-\alpha(k-1)^{1/n})$	
Entropy change δ	$exp(\frac{c_i - c_{i-1}}{T_k})$	$(\frac{c_i - c_{i-1}}{T_k})^3$
Acceptance Probability 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

($0 < \alpha < 1$), μ and U are uniform random variables between 0 and 1, $(B_j - A_j)$ is the range of j^{th} 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 I). 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.

IV. Configuration Modeling Issue

Given the importance of addressing the configuration problem, we applied the above design (i.e. combinatorial optimization based configuration selection) to several publicly available data-sets (See Table. II). As these are published data-sets from various studies, we have no control over the data collected, experiments run, configuration parameters, variability, etc. We explain these data-sets briefly below.

A. Cloud Storage Gateway Data-set(CSG)

CSG[1] [17] is architecturally similar to Edge Computing, IoT Gateways, etc. which are constrained by limited resource capacity and placed between the Edge/IoT/user applications and the Cloud platform. A CSG is usually deployed at a branch office or remote location and has access to a rather limited local compute/storage and is connected to a Cloud data center over the Internet. A CSG essentially uses local storage as a cache for the remote Cloud storage to bridge the gap between the demand for low-latency/high-throughput local access and the reality of high-latency connection to the cloud with unpredictable and usually low throughput.

The observed performance of CSG denoted as ϕ, is influenced by its configuration parameters, ($\vec{x}$ in Eq. 2) and the given workload ($\vec{w}$). The configuration parameters include compute resources (cores, cpu-speed, memory capacity, etc.), IO path (memory bandwidth, disk IO bandwidth, etc), buffer space allocation (cache space, meta-data space), etc. A full description of the CSG system, various configuration parameters influencing the behavior, real-world workloads, etc. is given in our earlier paper [17]. 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 the context of CSG system being studied, we can express Eq. 2 more clearly as: $\vec{x}$= $\{nc, cs, mc, bw, di\}$ and $\vec{w}$ = $\{ar, rs, ms\}$. The research question would then be to find the suitable values for $\vec{x}$ for a given $\vec{w}$ that satisfies the given constraint (Eq. 4) at a minimum cost (Eq. 5).

B. Configuration Modeling for BitBrains Data Center (BB)

The other publicly available data-set used in this research is BitBrains[2] workload trace [18] containing the performance logs of 1,750 VMs from a distributed data center from BitBrains, which are collected over 5000 cores and 5 million CPU hours accumulated over 4 months. This data-set (see Table II) provides specialized interactive services and batch processing workloads for managed hosting and business computation including leading banks, insurance companies, credit card operators, etc.

Workloads for Evaluating BitBrains Authors [18] conduct a comprehensive characterization of both requested and actually used resources, using data corresponding to CPU, memory, disk, and network resources. The initial configuration $\mathcal{C}$ of each VM present in these traces are characterised by the attributes shown in Table II. With limited knowledge into the details of the data-set, we formulate the BitBrains VM configuration as a combination of required compute, storage and network resource - number of cores nc, memory capacity mc, network receive bandwidth $nwrd$, and network transmit bandwidth $nwwr$. We characterise the workload as the load on the storage disks as read request rate $dskrd$ and write request rate $dskrd$ and observed behavior (ϕ) as the CPU usage (%).

We can now represent the configuration problem as selecting the right combination of configuration values (i.e. resources $\vec{x} = \{nc, mc, nwrd, nwwr\}$) for a given workload ($\vec{w} = \{dskrd, dskwr\}$) to satisfy the user defined conditions (Eq. 4 and Eq. 5).

C. Configuration Modeling for Enterprise Data-set (EE)

We group the data-sets of Apache, SQL Lite and Berkeley DB from Ref. [10, 19, 20, 21] as Enterprise Data-sets[3][4] (EE). Apache HTTP Server is a highly popular web server. Xu et al. [1] report that the Apache server has more than 550 parameters and many of these parameters have dependencies and correlations, which further complicates the configuration problem we address here. Reference [10, 21] narrows the CVs

[1][CSG] https://www.kkant.net/config_traces/CHIproject

[2][BB] http://gwa.ewi.tudelft.nl/datasets/gwa-t-12-bitbrains (RND500)

[3][EE] https://github.com/ai-se/Reimplement/tree/cleaned_version/Data

[4][EE] https://goo.gl/689Dve (RawData/PopulationArchives)

TABLE II: Configuration Variables, Workload and Output of the Data-Sets.

Data-Set	**Domain**	**Configuration Variables $\vec{x}$**	**Workload Characteristics $\vec{w}$**	**Output ϕ**
CSG [17]	Cloud Storage	No. of Cores, Core Speed, Memory Capacity, Memory Bandwidth, Disk IO Rate	Request Arrival Rate, Request Size, Metadata Size	Performance
BB [18]	Virtual Machines	No. of Cores, Core Speed, Memory Capacity, Network Data Rcvd., Network Data Transmit	Disk Read Throughput, Disk Write Throughput	CPU Usage (%)
Apache [10]	Web Server	Base, KeepAlive, Handle, HostnameLookups, EnableSendfile, FollowSymLinks, AccessLog, ExtendedStatus, InMemor1	N/A	Performance
SQL Lite server [19]	SQL Server	SetCacheSize, StandardCacheSize, LowerCacheSize, HigherCacheSize, LockingMode, ExclusiveLock, NormalLockingMode, PageSize, StandardPageSize, LowerPageSize, HigherPageSize, HighestPageSize$\cdots$ $\cdots$	N/A	Performance
Berkeley DB C [20]	Embedded database	havecrypto, havehash, havereplicatio0, haveverif1, havesequence, havestatistics, diagnostic, pagesize, ps1k, ps4k, ps8k,ps16k, ps32k, cachesize, cs32mb, cs16mb,cs64mb, cs512mb	N/A	Performance

down to only nine CVs as given in Table. II. Berkeley DB (C) [20] (BDBC) is an embedded key-value-based database library that provides scalable high performance database management services to applications. SQL Lite [19] is the most popular lightweight relational database management system used by several browsers and operating systems as an embedded database. In producing the data-set, the authors [21] stress the application to maximum workload and observe performance data for various configurations. Authors [20, 19] have used 18 CVs for BDBC and 29 CVs for SQL Lite data-sets. This expands the configuration space Ω to a larger degree, and the results show the efficacy of the proposed solution in such a large configuration space.

We refer readers to the detailed literature at Ref. [10, 21, 20, 19] for a full description of the data-set. With our problem at hand, the problem (Eq. 2 and Eq. 3) reduces to finding the best configuration ($\vec{x}$) for a user given performance (p_u) at minimum possible cost.

V. Evaluation

A. Metrics for Evaluation

Using the above data-sets, we evaluate the efficacy of our solution in finding a satisfying solution with two key metrics: (**M1**) the number of calls to the performance function (a.k.a oracle), and (**M2**) the minimum cost of the selected configuration (i.e Eq. 5). Metric M1 is important as it relates to how fast the algorithm can *find an optimal set of parameters* from the vast configuration space Ω. Metric M2 may refer to the monetary cost ($$) of the selected physical configuration, resource consumption of the selected configuration in a virtualized environment, or some other attribute (e.g., energy consumption, provisioning difficulty, etc.). Naturally, metric M2 is generally much more important than M1 (the computation time), but there are two situations that make M1 very important: (a) frequent changes in configuration, which is quite common in current clouds, selection/change happens frequently; for example, Facebook reports thousands of configuration changes per day, and (b) models (oracles) with long running times.

We executed 100s of test cases across all the data-sets, each test case T_i referring to a unique combination of $\vec{w_j}$ and ϕ_k in the data-set. We discuss the evaluation results using M1 and M2 metrics w.r.t the three approaches discussed above, i.e. (a) Generic Simulated Annealing (gSA), (b) Modified Simulated Annealing (mSA), and (c) Modified Simulated Annealing with Tunneling (mSA(T)).

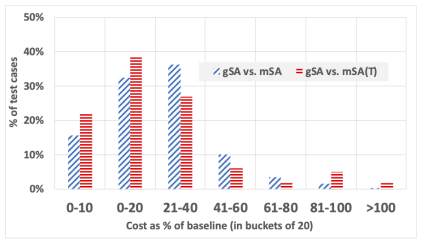
B. Performance Oracle

The efficiency of ML algorithms depends on a variety of factors including the input attributes and hyper-parameters (e.g., regularization parameters, learning rate, etc.) and it is generally not possible to characterize which algorithm works the best in a given situation. Therefore, we tried several models and ultimately settled on Logistic Regression, as it consistently performed well and beat others in most workloads. An extensive analysis of the model ensured that it does not suffer from under-fit or over-fit.

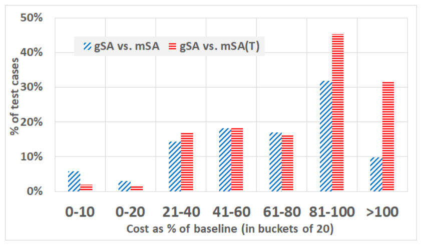
C. Using Domain Knowledge to Group Attributes

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, but the settings across groups can be done independently. In theory, such grouping can be done purely in a data driven manner (e.g., by using clustering techniques), but this is likely to result in spurious groups unless we have a large amount of data covering full ranges of various configuration parameters and the clustering algorithm does not result in anomalies. The value of domain knowledge is to do a suitable grouping either entirely manually, or by coercing the clustering algorithm to prefer certain groupings over others.

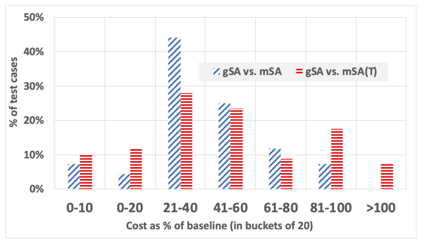
In any configuration context, we are likely to have several generic and usage specific insights into the system. For example, in a computing infrastructure, a faster CPU must be paired with a faster DRAM, else the CPU will simply stall waiting for the memory. A faster disk is also important, but much less so, since the IOs involve a context switch whereas a memory access does not. Similarly, more CPU cores doing



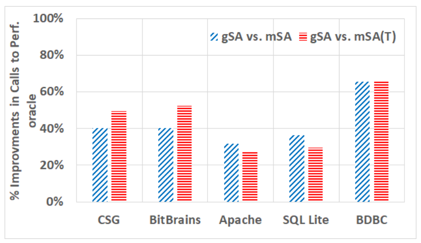
(a) Histogram of Cost for CSG



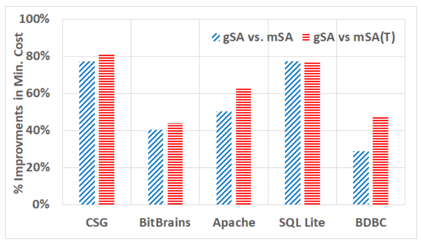
(b) Histogram of Cost for BitBrains



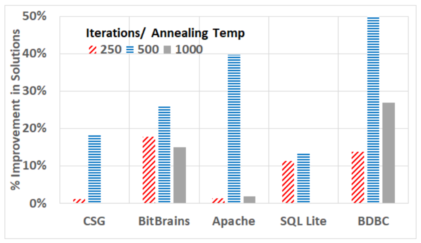
(c) Histogram of Cost for Apache



(d) % Improvement in #calls to Oracle



(e) % Improvement in Solution Cost



(f) % Improvement in % cases that provide a solution (gSA vs. mSA(T))

Fig. 3: Comparing (Savings) gSA, mSA, and mSA(T) for Diff. Data-sets.

TABLE III: Grouping Design Variables for Various Data-Sets

Data-Set	Group	Design Attribute Pairs
CSG	Group G1	Number of Cores, Memory capacity
	Group G2	Core speed, Memory bandwidth
	Independently varied	Data cache, Disk IO rate
BitBrains	Group G1	Number of Cores, Memory capacity
	Independently varied	CPU capacity, Network data transmit, Network data received
Apache BDBC SQL Lite	Independently varied	All CVs

independent work will likely need more memory, and for workloads involving remote IO, both network and IO speeds must increase in tandem. *Grouping* of CVs based on insights avoids exploration of states that are unlikely to be useful and thus is expected to both speed up the convergence and lead to better solutions within a given number of iterations. Table III shows the grouping for different data-sets. It is difficult to do grouping for other data-sets (Apache, BDBC, SQL Lite) since the trace description does not say much about the configuration or the workload.

D. *Efficacy of the algorithms (gSA, mSA and mSA(T))*

We use the gSA algorithm as the baseline, as it presents the naive (or *uninformed*) stochastic method of searching a wide configuration space for a set of suitable CVs.

In our evaluation for metric M2, the minimum cost of the solutions with mSA(T) was much less (hence better/desired) than the solution found by mSA or gSA (shown in Fig 3e). mSA(T) meets to the solution about 81%, 44%, 62%, 77%, and 47% *less* than the gSA solution (for the CSG, BB, Apache, SQLLite, and BDBC data-sets respectively).

Detailed results are shown in Fig 3 for a selected few data-sets where we show the histogram of the improvement in solution cost provided by our two algorithms (mSA and mSA(T)) over the baseline uninformed algorithm gSA. Here the X-axis refers to the cost of mSA and mSA(T) divided by the solution cost of gSA, and expressed as a percentage. That is, the buckets for $\leq 100\%$ represent an improvement over GSA, and $> 100\%$ represent a degradation. The y-axis is the count of test cases whose solution cost falls into that bucket – again, normalized so that it all adds up to 100% of the test cases. These charts are produced by considering 100s of test cases and thus represent an extensive exploration of the configuration space.

Fig. 3a on CSG results shows that in the 1st group (0-10%), mSA achieved the solution with $\leq 10\%$ of the baseline cost for 15% of test cases; and mSA(T) further improved this to 22% of test cases. The maximum improvement observed was in a few cases where the cost of mSA(T) was only 2% of the cost provided by gSA!

Similarly in Fig. 3b in BB results, in the 6th group, (81 to 100%), mSA cost was about 61-80% of the cost of the baseline in about 17% test-cases; and mSA(T) improved this in about 22% of test-cases. The final group (> 100) in all the sub-graphs show cases where gSA cost was better than mSA or mSA(T); however, these cases were small in case of CSG and Apache. With Bitbrains, the evaluation showed that mSA(T) failed to get minimum cost in about 30% of the cases (compared to gSA). Note that because of the inherent randomness in the way the states are explored, no stochastic algorithm can provide a universally better result in all cases.

Our evaluation results show that mSA(T) can find more successful results (i.e. correct values of CVs) compared to gSA. That is, gSA fails to find a result within the allowed limit defined by a hyper-parameter, i.e, annealing temperature/# iterations. On an average, mSA(T) surpasses gSA in finding additional (avg.) **30%** results across all data-sets for 500 iterations. This is shown in Fig. 3f for various data-sets and hyper-parameters (i.e, annealing temperature/# iterations). Additionally mSA(T) converges to the solution about 50%, 52%, 28%, 30%, and 65% faster in CSG, BB, Apache, SQLLite, and BDBC respectively (Fig. 3d).

E. Discussion and Conclusions

In this paper, we presented an efficient methodology to recommend optimal configurations for large scale software systems. The problem involves constrained optimization (achieving minimum cost subject to a performance lower bound) where the relationship between performance and configuration parameters is not explicit and may involve a complex machine learning model. We propose a meta-heuristics based approach enhanced by both the domain knowledge and smart tunneling techniques. We applied the technique to several real-world traces where configuration information was included in the data-set. The results show that the proposed mechanism outperforms a standard naive uninformed approach by 44-81% (depending on the domain and data-set) in terms of the cost of the solution. It also shows evidence of faster convergence by 28-65%, which means that it is possible to set the hyperparameter (# of iterations/ annealing temp) to a lower value without hurting the solution quality.

It is clear that the mSA(T) algorithm outperforms both mSA and gSA in finding a better minimum cost solution. This is because (i) our mSA computes entropy as a quadratic function to give us a wider choice of acceptance and is better able to avoid getting stuck in local minima, and (ii) by intelligently grouping the attributes discussed in section V-C, mSA also avoids exploring undesired search space. By addition of the tunneling logic, mSA(T) avoids *jumping out of local minima too quickly*; instead it explores a few additional states *closer to current local minima* with a goal to find a better minimum cost.

In the future, we will examine how domain knowledge can be extracted automatically or semi-automatically from the best practices specifications.

References

[1] T. Xu and Et al., "Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software," *Proc. Foundations of Software Engineering*, 2015.

[2] Q. Wang and Et. al, "Optimizing n-tier application scalability in the cloud: A study of soft resource allocation," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 2019.

[3] K. Hussain, M. N. M. Salleh, S. Cheng, and Y. Shi, "Metaheuristic research: a comprehensive survey," *AI Review*, 2019.

[4] A. V. Levy and A. Montalvo, "The tunneling algorithm for the global minimization of functions," *SIAM J. Sci. and Stat. Comput.*, vol. 6, no. 1, p. 15–29, 1985.

[5] F. Schoen, "Stochastic techniques for global optimization: A survey of recent advances," *Journal of Global Optimization*, vol. 1, no. 3, pp. 207–228, 1991.

[6] Y. Zhang and K. Xu, "A survey of resource management in cloud and edge computing," in *Network Management in Cloud and Edge Computing*. Springer, 2020.

[7] S. S. Gill and Et al., "Chopper: an intelligent qos-aware autonomic resource management approach for cloud computing," *Cluster Computing*, vol. 21, 2018.

[8] M. Wang and Et al., "Storage device performance prediction with CART models," in MASCOTS 2004.

[9] C.-J. Hsu, R. K. Panta, M.-R. Ra, and V. W. Freeh, "Inside-out: Reliable Performance Prediction for Distributed Storage Systems in the Cloud," SRDS, 2016

[10] N. Siegmund, A. Grebhahn, S. Apel, and C. Kastner, "Performance-influence models for highly configurable systems," in *Proc. Foundations of Software Eng.*, 2015.

[11] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," NSDI 2017

[12] M. Barrette and Et al., "Statistical multi-comparison of evolutionary algorithms," *Bioinspired Optimizaiton Methods and their Applications*, 2008.

[13] Z. Li and Y. Yang, "A modified tunnelling algorithm for global minimization with box constrained," in *2012 Fifth International Joint Conference on Computational Sciences and Optimization*, 2012, pp. 423–427.

[14] K. P. Ferentinos, K. G. Arvanitis, and N. Sigrimis, "Heuristic optimization methods for motion planning of autonomous agricultural vehicles," *J. Global Optimization*, vol. 23, no. 2, pp. 155–170, 2002.

[15] Y. Xu, Q. Ye, and G. Meng, "Hybrid phase retrieval algorithm based on modified very fast simulated annealing," *Intl. J. Microwave and Wireless Technologies*, vol. 10, no. 9, pp. 1072–1080, 2018.

[16] C.-Y. Lee, "Fast simulated annealing with a multivariate cauchy distribution and the configuration's initial temperature," *J. Korean Physical Society*, 2015.

[17] S. Sondur and K. Kant, "Towards automated configuration of cloud storage gateways: A data driven approach," CloudCOm, Springer, 2019, pp. 192–207.

[18] A. Iosup and Et al., "The grid workloads archive," FGCS, vol. 24, 2008.

[19] V. Nair and and Et al., "Finding faster configurations using flash," IEEE TSE, 2018.

[20] V. Nair and and Et al., "Faster discovery of faster system configurations with spectral learning," *Automated Software Engineering*, 2018.

[21] V. Nair and Et al., "Using bad learners to find good configurations," in *Proc. Foundations of Software Engineering*, 2017

[22] S. Sondur and Et al.." Storage on the Edge: Evaluating Cloud Backed Edge Storage in Cyberphysical Systems." in IEEE MASS 2019.