Finding Good Configurations in High-Dimensional Spaces: Doing More with Less

Risi Thonangi, Vamsidhar Thummala, Shivnath Babu
Duke University
{*rvt, vamsi, shivnath*}*@cs.duke.edu*

Abstract

Manually tuning tens to hundreds of configuration parameters in a complex software system like a database or an application server is an arduous task. Recent work has looked into automated approaches for recommending good configuration settings that adaptively search the full space of possible configurations. These approaches are based on conducting experiments *where each experiment runs the system with a selected configuration to observe the resulting performance. Experiments can be time-consuming and expensive, so only a limited number of experiments can be done even in systems with hundreds of configuration parameters. In this paper, we consider the problem of finding good configurations under the two constraints of high dimensionality (i.e., many parameters) and few experiments. We show how certain design decisions made in previous algorithms for finding good configurations make them perform poorly in this setting. We propose a new algorithm called* MOWILE *(*MO*re* WI*th* LE*ss) that addresses these limitations, and outperforms previous algorithms by large margins as the number of parameters increase. Our empirical evaluation gives interesting insights that will benefit system administrators who apply experiment-driven approaches for configuration tuning.*

1. Introduction

Configuring a complex software system for good performance can be a laborious task. For example, picking a good configuration for a database system requires decisions at the level of which indexes and statistics to maintain on the data, which materialized views (cached query results) to create, how to partition the data, how to set configuration parameters like buffer pool sizes and number of I/O daemons, and many others. The difference in performance can be many orders of magnitude between a well-tuned configuration and a bad one.

Configuration parameters are perhaps the most common components of system configuration. These "tuning knobs" give system administrators the capability to tune the system for specific workloads and hardware properties. The power of these knobs seems to have blinded us to the point that systems today have tens to hundreds of configuration parameters. Commercial database systems are good examples of systems having more than a hundred configuration parameters, providing knobs to control everything from selecting indexes, views, and data placement across parallel disks, to thresholds that influence query plan selection and govern the partitioning of memory or multiprogramming level in a multiuser environment [1].

System administrators often complain that default settings of configuration parameters are provided with very simple workloads and resource provisioning in mind [6]. As a result, overall system performance can be increased significantly by tuning these parameters. For instance, IBM DB2 administrators recommend changing the default settings of DB2's configuration parameters whenever the database environment contains one or more of large data sizes, many concurrent connections, unique query or transaction types, or special hardware characteristics [6].

Manually tuning the settings of configuration parameters is an arduous task. Typically, system administrators end up using a mix of rules-of-thumb, trial-and-error, and heuristic methods. However such methods are generally slow, and require a good understanding of the system internals as well as the workload and hardware characteristics. Administrators can have a tough time dealing with the numerous configuration parameters, especially given that there can be interactions among multiple parameters. That is, the optimal setting of one parameter may depend on how one or more other parameters are set. To further complicate things, the new breed of on-demand data centers and grids [3] leave little room for manual intervention in choosing configuration settings.

These factors motivate the need for automated approaches to set configuration parameters. We refer to this problem as the *parameter optimization problem*. There are different ways to address this problem which we will review briefly.

Approach	Requires Experiments	System as a Black-box?
Expert Rules	No	No
Fitting a priori models	Yes	No
Fitting statistical models	Yes	Yes
Adaptive Search	Yes	Yes

Table 1. Approaches for parameter optimization

- **Expert Rules [6]:** In this approach, system experts create a database of parametrized rules that can be evaluated in specific workload and resource settings to recommend a good configuration. The IBM DB2 Configuration Advisor [6] is a prominent example of this approach. The Configuration Advisor asks administrators a series of high-level questions—e.g., does the workload contain short or long transactions, or both?—and recommends configurations based on the answers.
- **Model-fitting [11]:** This approach conducts some number of *experiments* where each experiment runs the system using a chosen configuration e to observe the corresponding performance p. Each experiment gives an $\langle e, p \rangle$ *sample*. A model can be trained from these samples and then used to find good configurations, e.g., using heuristic search. This approach can use both a priori models like queuing networks (e.g., [5]) as well as statistical models like regression trees (e.g., [11]).
- **Adaptive Search [4, 9, 10]:** This approach conducts experiments as part of a search for a good configuration from the space of possible configurations. (Adaptive Search is sometimes referred to as *global optimization*.) Experiments are done in phases where the next set of experiments is determined based on an analysis of the samples generated by the previous set of experiments. There has been a lot of recent activity in applying Adaptive Search to parameter optimization problems that arise in, e.g., tuning application servers [9] and network routing configurations [10].

Table 1 gives a brief comparison among the above approaches. For each approach shown in the first column, the second column shows whether the approach is based on conducting experiments or on predefined information. Both model-fitting and Adaptive Search use samples generated by experiments. The main advantage of an experiment-driven approach is that its recommendations will be based on actual performance observed for the workload and hardware environment for which the tuning is being done. From such observations, it is easy to spot interactions among parameters and to identify parameters whose settings affect performance significantly (or hardly at all).

The third column in Table 1 mentions whether the approach treats the system as a *black-box* or not. A black-box approach does not require knowledge of the domain or the system internals as input; it tries to generate this knowledge automatically from the samples collected through experiments. The widespread applicability of this approach from simple to complex systems has resulted in many recent papers advocating it (e.g., [11, 2]). Given its ease of use, a black-box approach probably has the best chance of generating automated, portable, and ready-to-deploy configuration recommenders across different types of systems.

Model-fitting and Adaptive Search give the benefits of both the experiment-driven and black-box approaches. A significant disadvantage of model-fitting, which we will demonstrate empirically in Section 7, is that it tries to fit one single model to the entire space of possible configurations. Adaptive Search avoids this problem of model fitting, and instead focuses on searching through the space of possible configurations. This search interleaves *global search phases*—where general performance trends in the full configuration space are explored—with *local search phases*—where it drills down into local regions of the full space that are likely to contain many good configurations.

While Adaptive Search looks very promising on paper, it faces a serious challenge, namely, the fact that experiments in real systems can be very expensive. An experiment may have a large startup cost. For example, if an experiment needs a configuration consisting of a specific set of indexes in a database system with gigabytes of data, then a lot of time can be spent in generating these indexes. Furthermore, an experiment may need to be run for some amount of time so that startup effects die down and we can measure a reasonable approximation of performance for that configuration. For example, for each experiment involving an e-commerce service running on a WebSphere application server and a DB2 database, [9] recommends a run-time of at least 15 minutes. 15 minutes per experiment allows at most 96 experiments per day (assuming that the system is devoted entirely to running experiments).

At the same time, WebSphere and DB2 together easily have 100 or more configuration parameters. *96 samples from a 100 dimensional space is an extremely sparse sampling of this space.* Each sample in this setting is very important and could have significant leverage in deciding whether close-to-optimal configurations are found or not. Thus, an Adaptive Search algorithm has to be very careful about which configurations it chooses to sample.

1.1. Our Contributions

- In this paper we introduce a specific, but very practical, version of the parameter optimization problem. We are given a high-dimensional (10-100 parameters) space of configuration parameters. Conducting experiments is expensive, so we can only do a limited number of experiments (20-200) to collect samples from this space. Our goal is to find a good configuration given these two constraints of high dimensionality and limited number of experiments. While there has been

plenty of previous work on parameter optimization, to the best of our knowledge, we are the first to consider the problem with this specific focus.

- We provide a detailed analysis of two state-of-the-art algorithms for parameter optimization from our specific viewpoint. We show that some of the design decisions made in these algorithms make them perform poorly in the settings we consider.
- We present a new Adaptive Search algorithm called *MOWILE* (*MO*re *WI*th *LE*ss) which is designed keeping in mind the limitations of sparse sampling from a high-dimensional space. MOWILE's features such as (i) planning experiments to ensure good coverage of promising subspaces, (ii) considering already-collected samples in a subspace while planning new experiments in that space, and (iii) ensuring sufficient number of restarts of the algorithm within the experimental budget, address the problems with previous parameter optimization algorithms.
- We present an empirical evaluation of different Adaptive Search algorithms on synthetic functions as well as configuration datasets from three deployed systems. The synthetic functions are popular benchmarks used to judge the quality of algorithms for parameter optimization. These functions enable us to consider hundreds of dimensions (parameters) and to validate how close to optimal the configurations produced by different Adaptive Search algorithms get. The performance improvement of MOWILE over previous algorithms gets better and better as the number of dimensions increase. The real configuration datasets have at most seven dimensions because it can take several months (if not years) to generate representative data with 10-15 dimensions. On these low-dimensional datasets, MOWILE is comparable (not necessarily the best always) to the existing algorithms.
- We analyze the performance of each algorithm in depth by breaking it down to its individual components. Our analysis shows, e.g., that it is necessary to strike a good balance between the global and local search phases within the allowed number of experiments. We are also able to provide robust guidelines to achieve this balance.

The rest of the paper is organized as follows. Section 2 gives the problem definition and introduces terminology and notation. Section 3 introduces a general framework for Adaptive Search algorithms. Sections 4–6 discuss how three state-of-the-art algorithms for Adaptive Search, including our own MOWILE algorithm, instantiate the general framework from Section 3. Section 7 gives a detailed empirical evaluation of all three algorithms.

2. Preliminaries

Let the set of n configuration parameters (dimensions) be $X = \{x_1, x_2, \ldots, x_n\}$, and the domain of a parameter x_i be denoted as $Dom(x_i)$. We assume that $Dom(x_i) \subset \mathbb{R}$. We refer to the full space of configurations $Dom(X) = Dom(x_1) \times Dom(x_2) \times \ldots \times Dom(x_n)$ as the *full configuration space*. A *subspace* S of the full space has the form $Dom'(x_1) \times Dom'(x_2) \times \ldots \times Dom'(x_n)$, where each $Dom'(x_i)$ is a single range of values in $Dom(x_i)$.

The *neighborhood* of a configuration is a subspace with the configuration at its center. The full configuration space, subspaces, and neighborhoods of a configuration form hypercubes in the multi-dimensional space of parameters. For example, in a two-dimensional setting with parameters $\{x_1, x_2\}$ having domains $Dom(x_1) = Dom(x_2) = [0, 10]$, the full configuration space would be the square region $[0, 10] \times [0, 10]$ in the two-dimensional grid. The neighborhood of a configuration $\langle 5, 5\rangle$ could be a subspace of this square region, e.g., $[3, 7] \times [3, 7]$, with center $\langle 5, 5\rangle$.

We denote the performance of the system at a configuration e by $P(e)$. To determine the P value at a configuration e, we set the system's configuration parameters to their respective values in e, run the system for the specified workload, and measure the resulting performance. We refer to each such system run as an *experiment*, and the result is a $\langle e, P(e)\rangle$ *sample*. The performance function P is unknown, and the *parameter optimization problem* is to find a configuration that has the maximum (or large enough) P value[1] by running no more than $\mathscr{B}$ experiments. $\mathscr{B}$ represents the experimental budget.

Given the natural variability in systems, we would want to repeat the experiment at configuration e multiple times to capture the variability of performance at e. We do not consider such repetitions in this paper. Reference [4] as well as our own work in [7] present methods to measure performance at a configuration e in a robust fashion by executing multiple experiments at e. Our work in this paper concentrates on an issue that is orthogonal to repetitions, namely, selecting the set of distinct experiments to conduct (i.e., samples to collect). For each configuration e that we choose to sample, the methods from [4] or [7] can be used to find how many samples of $P(e)$ to collect (e.g., in order to meet a given confidence threshold).

3. Overview of Adaptive Search

We begin with a brief overview of Adaptive Search. In general, sampling algorithms falling under the paradigm of Adaptive Search interleave global search phases (*Explorations*) and local search phases (*Exploitations*) [10]. A global search phase is one in which the full configuration

[1] Without loss of generality, we consider the problem of maximization in our discussion. Note that a minimization problem can be converted into a maximization problem by negating the performance metric.

Symbol	Description
n	Number of configuration parameters
$\mathscr{B}$	Total number of experiments
k	Number of samples taken per sampling step
α	Volume shrinkage rate in subspace selection
δ	Volume threshold used to trigger restarts
N_r	Total number of restarts
N_s	Number of subspace selections per restart
c	Weight parameter used in wLHS sampling
β	Size parameter used in k-FF sampling

Table 2. Notation used in the algorithms

space is examined to identify a *promising* subspace; where a subspace S is promising if it is likely that S contains the globally optimum configuration. In the subsequent local search phase, the chosen subspace S is examined in more detail to find the best configuration e in S (hopefully, e has close to the global optimal performance). Adaptive Search interleaves the global and local search phases in order to ensure that the search process does not get stuck in locally optimal regions of the full configuration space [10].

In practice, most Adaptive Search algorithms tend to follow a framework consisting of the following steps:

- **Sampling from a subspace S:** This step involves collecting a set of samples through experiments from the given subspace S with the intention of learning the general trends of the performance function P in S.
- **Selecting a subspace S':** Given a subspace S and a set of samples collected from S in Step 1, Step 2 tries to identify a subspace S' (different from S) that is most likely to contain good configurations, including the optimal configuration in S. S' is usually contained completely within S.
- **Restarts:** A typical run of an Adaptive Search algorithm will alternate between Steps 1 and 2 multiple times. The algorithm starts by choosing samples from the full configuration space, and progressively uses Step 2 to reduce the size of the subspace from which samples are chosen in Step 1. At some point during this execution, a *restart* will be invoked which rewinds the search to start again by sampling from the full configuration space. Restarts enable the algorithm to escape from locally optimal regions.

The next three sections discuss how three algorithms for Adaptive Search (including MOWILE) implement the above steps. Table 2 lists the notation in the algorithms.

4. Smart Hill Climbing (SHC)

Reference [9] proposes an algorithm for Adaptive Search called *Smart Hill Climbing (SHC)*. SHC improves over a previous algorithm called *Recursive Random Search (RRS)* [10] that was based primarily on random sampling. Broadly, SHC runs in an iterative fashion involving the following steps (details are given in Sections 4.1–4.3):

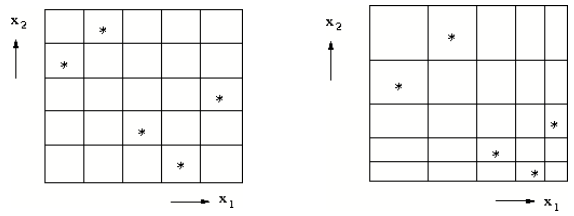
1. In a given subspace S of size ϕ_S, pick a sample of configurations using *weighted Latin Hypercube Sampling (wLHS)*. Run experiments at these configurations.
2. Based on the collected samples, a setting a_i is found for each parameter x_i, and a new configuration $e = \langle x_1 = a_1, \ldots, x_n = a_n \rangle$ is assembled from these settings. An experiment is done at e to determine its performance.
3. If the performance of the newly assembled configuration e is better than the best performance among the samples from wLHS, then repeat from Step 1 taking a neighborhood S' of size ϕ_S centered at e.
4. Otherwise, repeat from Step 1 taking a neighborhood S' of size $\phi_{S'}$ ($< \phi_S$) centered at the best configuration among the samples from wLHS.
5. Restart from the full configuration space when the size of the subspace in Step 4 falls below a given threshold.

4.1. Sampling from a Subspace in SHC

SHC proposes a technique called weighted Latin Hypercube Sampling (wLHS) [9] to collect a set of k samples from a subspace. wLHS is a variant of the popular space-filling sampling technique called *Latin Hypercube Sampling (LHS)*. LHS uses the following steps to generate a sample of k configurations from a subspace S: (1) Let $Dom'(x_i)$ be the domain of configuration parameter x_i that falls in the subspace S. $Dom'(x_i)$ is broken into k *subdomains* of equal size; and (2) k configurations are selected from S such that no two selected configurations have the same subdomain for any configuration parameter. Figure 1(a) shows an example where LHS breaks two dimensions into 5 subdomains each in order to pick 5 samples.

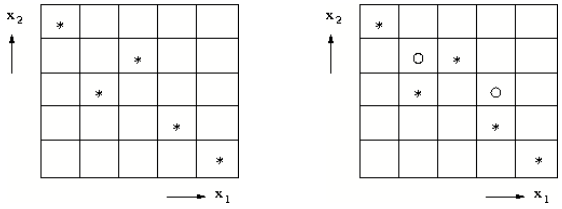
wLHS differs from LHS in that it breaks the domain of a configuration parameter in a subspace S into unequal subdomains that are either increasing or decreasing in size. Intuitively, wLHS aims to collect more samples from regions of S that are more likely to contain good configurations. Hence, the size of a subdomain is inversely proportional to its chances of containing good configurations in S. wLHS estimates the linear correlation coefficient of x_i with respect to observed performance in order to determine the sizes of the subdomains of each parameter, and whether these sizes should be increasing or decreasing. The correlation coefficient is combined with a user-specified *weight parameter* c that specifies how aggressively the correlation coefficient is used to determine the subdomains. When $c = 0$, wLHS is the same as LHS. Once the subdomains are created, wLHS picks a sample set of configurations exactly as LHS does.

Figure 1(b) gives an example of subdomains that wLHS may generate. Note that the domain of the X-axis (Y-axis)



(a) LHS (b) wLHS

Figure 1. Sampling using LHS and wLHS



(a) Lack of good coverage (b) Not accounting for samples that were collected previously

Figure 2. Undesirable sample generation by LHS/wLHS

of the configuration space is broken into subdomains of decreasing (increasing) size. The subdomains in Figure 1(b) facilitate the collection of more samples from the lower right region of the subspace.

4.2. Selecting a Subspace in SHC

SHC implements subspace selection by picking a configuration as the *seed*, and selecting a neighborhood around this seed. The seed is determined as follows. For each configuration parameter x_i, a good setting is determined (independent of other parameters) by first fitting a quadratic curve to the x_i and performance values in the samples collected by wLHS; and then solving for the optimal value of x_i in this curve. The individual optimal values of all configuration parameters are assembled together to form a new configuration, and the performance of this assembled configuration is determined. The seed is assigned the assembled configuration if its performance is better than that of all configurations sampled by wLHS, else the seed is assigned the best configuration sampled by wLHS.

The size of the neighborhood around the seed is determined as follows. If the seed was assigned a configuration sampled by wLHS, then the size of the selected neighborhood is smaller than that of the current subspace by a user-specified *shrinkage rate* α. (See Section 5.2 for more details on how the size reduction is done.) Otherwise, if the seed is the newly assembled configuration, then the size of the selected neighborhood will be the same as that of the current subspace. In the latter case, the subspace has *realigned* since its size is unchanged although its center has changed.

4.3. Restarts in SHC

SHC executes sample selection and subspace selection in an iterative fashion, reducing the size of the subspace progressively. When the subspace size becomes smaller than a fraction δ of the full configuration space, then the iterative procedure is restarted by going back to sample selection from the full configuration space. This restart policy assumes that sampling from subspaces smaller than the threshold size will not lead to any significant improvement. The *volume threshold* δ is user-specified.

5. MOre-WIth-LEss (MOWILE) Adaptive Search

In this section, we present how our new MOWILE algorithm implements the sample selection, subspace selection, and restart steps from Section 3 to find good configurations quickly.

5.1. Sampling from a Subspace in MOWILE

Although LHS and wLHS usually generate samples with better spread than pure random sampling [9], they can still suffer from the problem shown in Figure 2(a). Note the large empty areas in the upper-right and lower-left regions of the sample space. Furthermore, neither LHS nor wLHS considers samples already present in the subspace while determining the positions of the new samples. Figure 2(b) shows an example where LHS/wLHS generates new samples close to already collected samples (denoted by 'o'), thereby decreasing the effectiveness of the new samples.

The above examples motivate two desirable properties of sampling in our setting: (1) generating samples that provide good space coverage, and (2) considering already collected samples present in the subspace to maximize the effectiveness of new samples. MOWILE's implementation of the sampling step incorporates these two desiderata.

MOWILE implements the sampling phase using *k-Furthest First (k-FF)* sampling that works as follows. Suppose we want to sample k configurations from a subspace S. Let the set $I = \{\langle e_i, p_i \rangle\}_i$ represent the configurations and their respective performance that have already been collected from S (i.e., in previous sampling steps). (Here, p_i is short for $P(e_i)$). k-FF will select a set of k new samples (configurations) $N = \{b_j\}_{j=1}^{k}$ from S such that N maximizes the following expression:

$$\sum_{b \in \{b_j\}} \left(\min_{y \in \{e_i \cup b_j\} \setminus b} |b - y| \right) \quad (1)$$

The inner term in the above expression is the minimum distance between each new sample b and *all* other samples in S; by *all* we mean the samples in I (collected from S previously) as well as the newly drawn up samples. Maximizing the above expression involves generating samples from the subspace that are far away from the existing samples as well as far away from each other. The intuition behind this

consideration is that, by collecting samples that are well separated from each other, the sampling scheme explores as much of S as possible with k samples.

k-FF is implemented as follows: (1) to collect a set of samples N of size k from the given subspace S, a candidate set of samples C of size $\beta \cdot k$ is generated randomly from S; (2) the following step is executed k times: the sample in C with the highest value for the expression in Equation 1 is removed from C and included in N. β is a user-specified constant. Our experiments use $\beta = 10$ by default. While this greedy two-step procedure does not generate the optimal set $\{b_j\}$, its output is good enough for our purposes. The complexity of this procedure is $O(k^2 \cdot \beta \cdot (|I| + k))$.

5.2. Selecting a Subspace in MOWILE

k-FF generates a set of samples that has good coverage of the given space S. During subspace selection from S, MOWILE picks a neighborhood S' around the sample (from among the collected samples) with the highest performance. The size of S' is always smaller than that of S by a user-defined shrinkage rate α. Thus, unlike SHC, MOWILE does not perform realignments (recall Section 4.2) that cause S' to have the same size as S.

Both SHC and MOWILE use the volume shrinkage rate α as follows. Let $len(x_i)$ denote the length of the dimension x_i in S. Let $\langle a_1, a_2, \ldots, a_n \rangle$ denote the sample that forms the center of the new subspace S'. For each parameter x_i, a neighborhood of size $\alpha^{1/n} len(x_i)$ is chosen around a_i. The neighborhoods for the n parameters form a hypercube. The part of this hypercube that lies within S will form the new subspace S'.

5.3. Restarts in MOWILE

Recall that restarts in SHC are triggered by a user-specified volume threshold δ. Since SHC may include an arbitrary number of realignments that leave the size of the subspace unchanged, SHC has no control over the total number of restarts done. Our empirical analysis (Section 7) found that the performance of an Adaptive Search algorithm depends critically on the number of restarts. Too many restarts can be equally damaging as too few restarts.

MOWILE currently takes the number of restarts N_r as a user-specified parameter. (Table 2 summarizes the notation used.) Given user-specified values for N_r, k, and δ, both the volume shrinkage rate (α) and the number of subspace selections between restarts (N_s) can be computed automatically for a given experimental budget $\mathscr{B}$. MOWILE performs a restart after every N_s subspace selection steps. Section 7.3.4 provides guidelines for setting N_r, k, and δ.

6. Quick Optimization via Guessing (QOG)

Quick Optimization via Guessing (QOG) [4] is an Adaptive Search algorithm proposed recently for identifying the best among a set of candidate configurations specified by an user. Broadly, QOG runs the following three steps iteratively until it exhausts all candidate configurations: (1) all samples collected so far are used to learn a regression model to estimate performance over the full configuration space, (2) the regression model is used to estimate performance at each configuration that has not been sampled so far, and (3) an experiment is run to determine the actual performance at the configuration with the largest estimated performance.

QOG differs from SHC and MOWILE in two ways. First, a major focus of QOG is on repeating experiments at a configuration e to capture the variability of performance at e. Recall from Section 2 that the focus of this paper is on selecting the set of distinct configurations to sample, which is orthogonal to repetitions. Second, QOG relies on learning a single model that captures the performance trends in the full configuration space. This approach is problematic in high-dimensional spaces when few samples are available, and will be the focus of our comparison of QOG with SHC and MOWILE in Section 7.

7. Results

In this section, we present the results of MOWILE on a suite of synthetic and real datasets. We also analyze the design decisions in MOWILE in comparison with SHC and QOG. Our empirical evaluation shows that MOWILE performs the best among these three algorithms under the practical constraint of a high-dimensional space (10-100 configuration parameters) from which a limited number of experiments (20-200) can be conducted for sample collection. Due to space constraints, we can present only a subset of our results in this paper. The complete set of results is given in the technical report [8].

7.1. Experimental Setting

Framework: To experiment with a variety of Adaptive Search algorithms, we developed a framework where different components can be plugged in to instantiate various algorithms. This framework is modeled on the general nature of Adaptive Search algorithms described in Section 3. For example, an implementation of k-FF, LHS, or wLHS can be plugged into our framework to instantiate the step of sampling from a subspace. Similarly, we can plug in different techniques to implement the selection of a subspace and the restart policy. Our framework also enables the creation of new algorithms, e.g., we developed a variant of SHC by replacing wLHS sampling with k-FF. This framework made it easy for us to evaluate MOWILE, SHC, and QOG in a uniform manner (Section 7.2), as well as to analyze the impact of various design decisions in these algorithms through plug-and-play of alternative implementations of individual steps (Section 7.3).

Datasets: The empirical evaluation was conducted on a suite of datasets comprising both objective functions [10]

Name	Objective function
DeJong	$f(x) = \sum_{i=1}^{n} x_i^2$
Rastrigin	$f(x) = n \cdot A + \sum_{i=1}^{n} x_i^2 - A \cdot cos(2\pi x_i)$
Griewangk	$f(x) = \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} cos(\frac{x_i}{\sqrt{i}}) + 1$
Rosenbrock	$f(x) = \sum_{i=1}^{n-1} 100 \cdot (x_{i+1} - x_i^2)^2 + (1 - x_i)^2$

Table 3. Benchmark objective functions [10]

Dataset	Description
RUBiS	6 parameters varied for an eBay-like auction service developed with JBoss & MySQL
TPC-W	7 parameters varied for a Web e-Commerce benchmark developed with Tomcat & MySQL
Storage	5 parameters varied for an IBM Storage server in (a) well-provisioned & (b) saturated settings

Table 4. Real datasets (see [8] for more details)

as well as datasets from deployed systems. These datasets are summarized in Tables 3 and 4 respectively; more details are given in the technical report [8]. Figures 3 and 4 show sample two-dimensional response surfaces for the datasets.

The four objective functions in Table 3 are popular benchmarks used by researchers to judge the effectiveness of global optimization algorithms [9, 10]. (In these equations, x denotes an n-dimensional vector, and x_i denotes x's i^{th} dimension.) These functions are available for any number of dimensions, so they can be used to evaluate the performance of the algorithms in high-dimensional spaces. For all the functions in Table 3, the objective is minimization, and the global optimum value is 0. The Rastrigin, Griewangk, and Rosenbrock functions are known to be difficult to optimize.

On the other hand, increasing the dimensionality of our real datasets causes exponential increases in the time required to generate the data. For example, for the RUBiS dataset in Table 4, each experiment takes around 10 minutes to run; so we were able to collect at most 5120 samples in about a month. Six parameters (dimensions) were varied in this dataset. The performance metrics collected in all datasets include average response time and throughput.

Defaults: In the following empirical analysis, $n = 100$ and $\mathscr{B}$=100 by default. (Recall the notation in Table 2.) The tuning knobs for SHC are configured as per the guidelines in [9]. The defaults for SHC are: $k = 5$, $\delta = 10\%$ of the full configuration space, $\alpha = 80\%$, and $c = 10^{-5}$. The defaults for MOWILE are: $k = 5$, $\delta = 10\%$ of the full configuration space, $N_r = 2$, and $N_s = 8$. For these settings, α is roughly 75%. The tuning knobs for QOG are configured as per the guidelines in [4]. The sensitivity of the algorithms to the tuning knobs is studied in Section 7.3.4.

All algorithms were implemented in Matlab. All experiments were run on a 3.6 GHz single core Intel Pentium IV processor with 1 GB memory running CentOS 5 Linux.

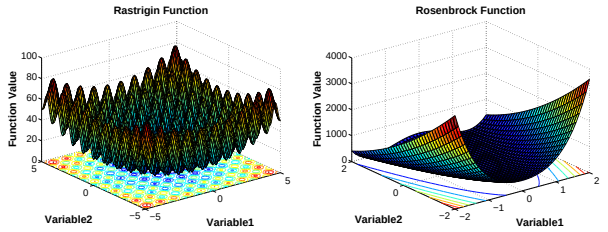


Figure 3. Visualization of the Rastrigin and Rosenbrock functions in two dimensions; Left: Rastrigin function from -5 to 5, optimum at [0,0]. Right: Rosenbrock function from -2 to 2, optimum at [1,1].

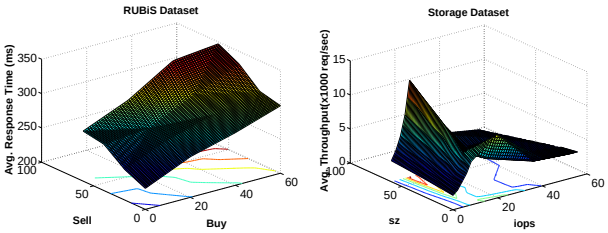


Figure 4. Variation of average response time of RUBiS (left) and average throughput of the storage server (right) for two respective workload-based parameters.

7.2. End-to-End Results

Figure 5 shows the performance of the three Adaptive Search algorithms as the number of dimensions (n) is varied for the objective functions. The Y-axis in Figure 5 shows the best value found by each algorithm after $\mathscr{B}$=100 experiments are done. Since the objective in these functions is minimization, lower Y values are better.

It is clear from Figure 5 that MOWILE outperforms both SHC and QOG as the number of dimensions increases. For the Rastrigin function with 20 dimensions, SHC [9] reports 600 as the best value found on average in 100 experiments. Note that MOWILE finds 200 as the best value on average in 100 experiments, while our implementation of SHC finds a value around 400. Thus, our implementation of SHC is comparable to that in [9]. More importantly, MOWILE finds a configuration that is almost two times better than the one found by SHC. In the case of the Rosenbrock function—which is very sensitive to the input values—MOWILE performs almost four times better than SHC as the number of dimensions approaches 40. Similarly, the results show that MOWILE is consistently better than SHC and QOG for complex functions like Griewangk as well as for simple functions like DeJong.

Figure 6 shows the performance of the algorithms on some of the real datasets as the number of dimensions are varied. Recall that the real datasets have very few dimensions (the maximum is 7 for the TPC-W dataset from Table 4). For so few dimensions, none of the algorithms dominates the others. The same observation can be made from

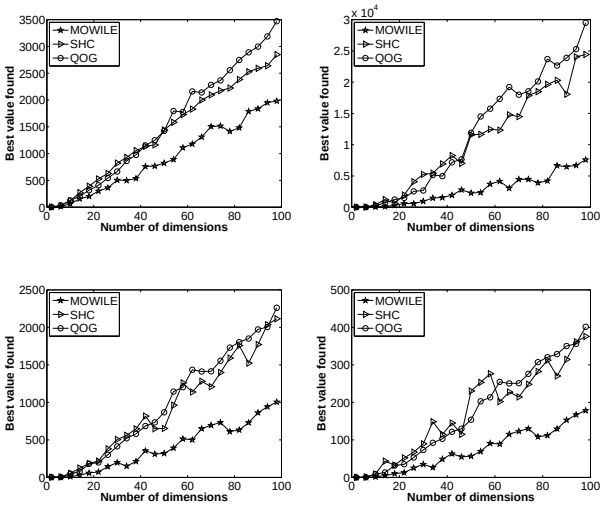


Figure 5. Number of dimensions Vs. Performance for $\mathscr{B}$=100. Top: Rastrigin (left) and Rosenbrock (right) functions, bottom: Griewangk (left) and DeJong (right) functions.

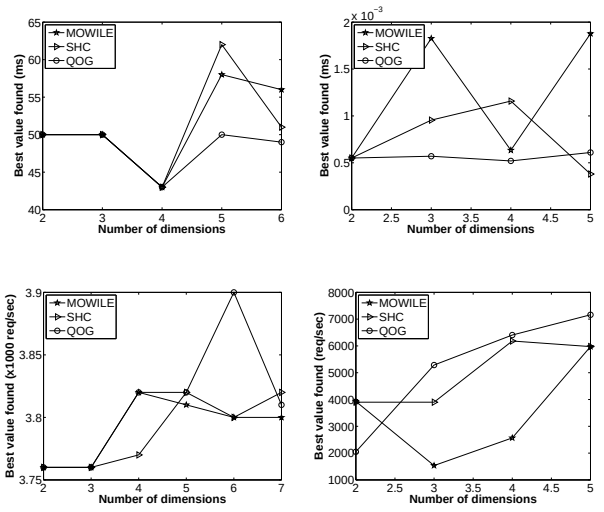


Figure 6. Number of dimensions Vs. Performance for $\mathscr{B}$=100. Top: average response time for RUBiS (left) and Storage (right), bottom: average throughput for TPC-W (left) and Storage (right).

the performance of the algorithms on the objective functions as well: for up to 10 dimensions, the performance of MOWILE is indistinguishable from that of SHC and QOG.

7.3. In-depth Analysis

We now present results from a detailed study to understand from where MOWILE is getting its benefits over SHC and QOG. This study was made easy by our plug-and-play framework for experimenting with alternative implementations of the different steps in Adaptive Search. We identified four candidate factors that can possibly explain why MOWILE outperforms SHC and QOG in high-dimensional spaces when the number of experiments is limited:

1. Impact of restarts
2. Advantages of k-FF over LHS and wLHS
3. Simplification of local search
4. Sensitivity to tuning knobs

We now analyze each of these factors in detail.

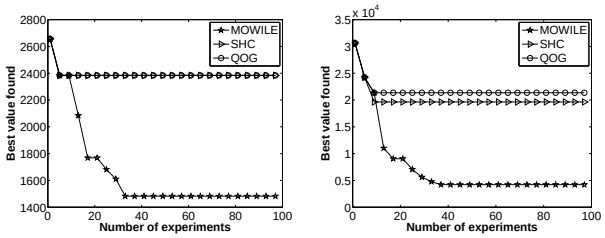


Figure 7. Convergence tests on Rastrigin and Rosenbrock functions for n=100.

7.3.1. Impact of Restarts: We discovered that the number of restarts is a crucial factor in determining the overall performance of an Adaptive Search algorithm. More restarts imply that the algorithm spends more time in global search phases (or exploration) that, intuitively, resemble a random (rather than focused) search in the high-dimensional space. Less number of restarts imply that the algorithm spends more time in local search which has a high chance of getting stuck in local optima. Since it is important to strike a good balance between global and local search, getting the number of restarts right is critical.

Figure 7 compares the convergence of MOWILE, SHC, and QOG as more experiments are done for $n = 100$. The X-axis shows the number of experiments done so far, and the Y-axis shows the best value found so far. MOWILE performs 2 restarts (N_r) overall in this experiment and it conducts 40 experiments in between consecutive restarts (recall that number of experiments between restarts is $k * N_s$); while SHC performs as many as 12 restarts. The aggressive use of restarts help SHC perform better than MOWILE in the beginning (10-15 experiments) as we see in the figure.

However, the downside of an aggressive restart policy is the limited effort that goes into finding local optima that may turn out to be close to the global optima. MOWILE has a better balance between global and local search. Thus, as the number of experimental runs increase beyond 15, MOWILE moves much closer to the true optimal value than SHC. A similar trend is observed with respect to QOG because QOG tries to fit a model to the full configuration space, thereby lacking effective local search.

To further analyze the impact of restarts, we developed a variant of SHC that does the same number of restarts as MOWILE. Recall from Section 4 that the original SHC algorithm has no control on the number of restarts since each

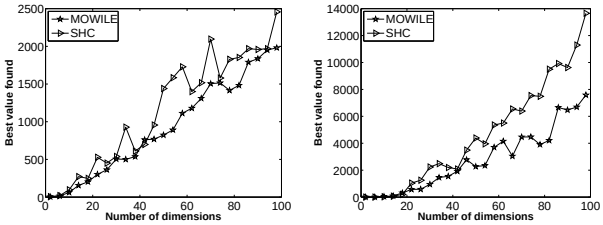


Figure 8. Performance of MOWILE and SHC when the same number of restarts are done on the Rastrigin (left) and Rosenbrock (right) functions.

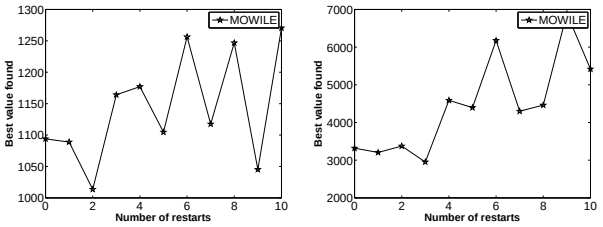


Figure 9. Impact of the number of restarts on MOWILE's performance. Left: 25-dimensional Rastrigin function, right: 50-dimensional Rosenbrock function.

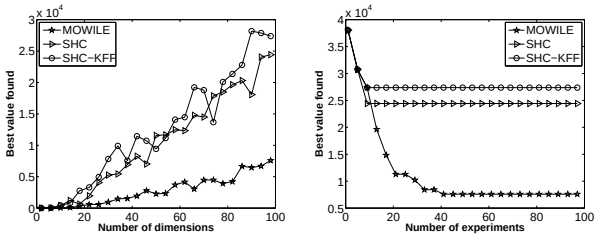


Figure 10. k-FF Vs. LHS sampling for the Rosenbrock function.

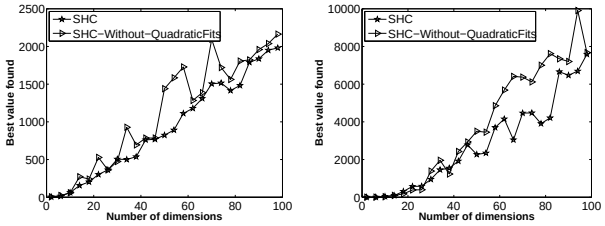


Figure 11. Impact of quadratic fits on SHC's performance for Rastrigin (left) and Rosenbrock (right) functions for n=100.

of SHC's local search phases can do an arbitrary number of realignments. Our variant of SHC controls the number of restarts by constraining the number of selections (N_s) that should be done between consecutive restarts. This variant is similar to the original SHC algorithm in all other respects. Figure 8 shows the performance of MOWILE and SHC when both do the same number of restarts within the given experimental budget. Although MOWILE still performs better on average, the gap between SHC and MOWILE has reduced significantly compared to what was observed in Section 7.2.

We also analyzed the impact of the number of restarts on MOWILE's performance; Figure 9 shows the results. It is clear that MOWILE's performance is best—i.e., there is a balanced tradeoff between local and global search in the given experimental budget—when the number of restarts is a small number like 2 or 3, and it drops sharply as the number of restarts is increased. We will revisit this issue in Section 7.3.4.

7.3.2. Impact of k-FF: Recall from Section 5 that MOWILE uses the k-FF technique for sampling from a subspace for two reasons: (i) k-FF generates samples that provide good space coverage, and (2) k-FF considers already collected samples present in the subspace to maximize the sampling step's effectiveness. On the other hand, SHC ignores samples that were collected before the last restart. Figures 2(a) and 2(b) illustrated the pitfalls that SHC's wLHS sampling can run into in this context.

To analyze the performance impact of k-FF, we replaced the wLHS sampling in SHC with k-FF using our plug-and-play framework. Figure 10 compares the performance of the original SHC algorithm with that of the new SHC-KFF variant. Note that the performance of SHC with k-FF is similar on average to that of the original SHC with wLHS. Thus, there is not much evidence to conclude that wLHS is putting SHC at a disadvantage. Similarly, we observed that the performance of MOWILE does not change significantly if LHS is used instead of k-FF.

7.3.3. Simplification of Local Search: Compared to MOWILE, SHC has a more complex design for the (recursive) step of selecting a subspace for further local search. Recall from Section 4 that SHC uses quadratic fits along each dimension to assemble the center configuration of the selected subspace, and it may realign the current subspace (without reducing the subspace size) if the assembled configuration has better performance than the sampled configurations. In comparison, MOWILE performs neither quadratic fits nor realignments. The reasoning behind MOWILE's design is that while working with very few samples (experiments), decisions that require complex analysis of the samples have a high chance of going wrong.

It is interesting to ask how much benefit SHC is gaining from its complex procedure for local search. Once again, our plug-and-play framework helps answer this question. Figure 11 compares the performance of the original SHC algorithm with that of SHC without quadratic fits. Notice that there is no consistent trend that distinguishes the performance of the two algorithms; showing that SHC's quadratic fits are not having much impact in the settings we consider.

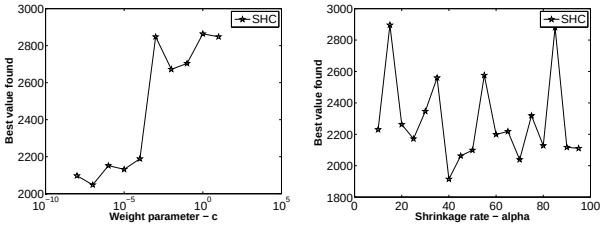


Figure 12. Sensitivity of the weight parameter c (left) and volume shrinkage rate α (right) on SHC's performance on Rastrigin function.

7.3.4. Sensitivity of Tuning Knobs: To our surprise, we discovered that configuring the Adaptive Search algorithms for good performance can be a black art in itself (which can defeat the purpose of black-box approaches for system management). For example, Figure 12 shows the sensitivity of SHC's performance on two of its tuning knobs c and α. Notice that slight changes to these parameters can affect the performance of SHC drastically.

It would seem that MOWILE shares most of the same "sensitive" knobs as SHC. However, our observation is that it is most important to set the number of restarts (N_r) appropriately for the given experimental budget $\mathscr{B}$. Note from Figure 9 that MOWILE's performance is sensitive to the setting of N_r. (Recall that N_r controls the balance between global and local search.) However, our empirical analysis showed the consistent trend that a small nonzero number of restarts (e.g., $N_r \in [1,3]$) gives good performance for budgets in the range of 50-200 experiments. Once the number of restarts is fixed, the settings of other knobs can be found through existing guidelines and back-of-the-envelope calculations.

To sum up, we found the following guidelines to work well in all the settings we worked with:

1. Set the number of restarts $N_r \in [1,3]$ for $\mathscr{B} \in [50,200]$ experiments.
2. Set the number of experiments k per sampling step to 5-10% of $\mathscr{B}$. (This guideline comes from [9].)
3. Set the volume threshold δ to 5-10% of the full configuration space. Instead of depending on δ to trigger restarts, compute $N_s = \frac{\mathscr{B}}{k(N_r+1)}$, and trigger a restart after every N_s subspace selection steps.
4. Set the volume shrinkage rate $\alpha = \delta^{\frac{1}{N_s}}$.

8. Conclusions

In this work, we considered the problem of finding good configurations under the two constraints of high dimensionality and few experiments. We looked at existing solutions for solving this problem and showed how certain design decisions incorporated in them lead to poor performance in our setting. We proposed a new Adaptive Search algorithm called *MOWILE* (*MO*re *WI*th *LE*ss) that addresses these limitations, and outperforms existing algorithms by large margins as the number of parameters increase.

Our current approach is designed for tuning configuration parameters offline, e.g., in a preproduction or test environment where experiments can be run without affecting the production workload. As future work, we plan to consider *online experiments* for tuning configuration parameters to handle the dynamic nature of system workloads. Furthermore, our current work takes a black-box approach. We are interested in developing effective mechanisms that can incorporate into Adaptive Search the knowledge that administrators have about the system.

References

[1] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *VLDB*, pages 1–10, 2000.

[2] I. Cohen et al. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Proc. of Symp. on Operating Systems Design and Implementation*, Dec 2004.

[3] D. Irwin, J. S. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. G. Yocum. Sharing Networked Resources with Brokered Leases. In *Proc. of USENIX Annual Technical Conf.*, Jun 2006.

[4] T. Osogami and S. Kato. Optimizing system configurations quickly by guessing at the performance. In *SIGMETRICS*, pages 145–156, 2007.

[5] P. Padala, X. Zhu, et al. Adaptive control of virtualized resources in utility computing environments. In *Proc. of European Systems Conference*, Mar 2007.

[6] S. Shastry and M. Saraswatipura. DB2 performance tuning using the DB2 Configuration Advisor. *IBM DeveloperWorks*, 2004.

[7] P. Shivam, V. Marupadi, J. Chase, and S. Babu. Cutting Corners: Workbench Automation for Server Benchmarking. In *Proc. of USENIX Annual Technical Conference*, Jun 2008.

[8] R. Thonangi, V. Thummala, and S. Babu. Finding Good Configurations in High-Dimensional Spaces: Doing More with Less. *Technical report, Duke University*, Jun 2008.

[9] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *WWW*, pages 287–296, 2004.

[10] T. Ye. *Large Scale Network Parameter configuration using Online Simulation Framework*. PhD thesis, Rensselaer Polytechnic Institute, March 2003.

[11] L. Yin, S. Uttamchandani, and R. H. Katz. An empirical exploration of black-box performance models for storage systems. In *MASCOTS*, Mar. 2006.