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

Risi Thonangi, Vamsidhar Thummala, Shivnath Babu

Department of Computer Science, Duke University
Durham, NC, USA.
rvt@cs.duke.edu
vamsi@cs.duke.edu
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 (doing MOrE With LEss) 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.

I. INTRODUCTION

Configuring a complex software system for good performance can be a laborious task. For example, picking a good configuration for a commercial database system like IBM DB2 or Oracle 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 [2]. 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 [2].

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 will refer to this problem as the parameter optimization problem. There are a number of different ways in which this problem can be addressed. For example:

1) Expert Rules [2]: 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 [2] is a prominent example of this approach. The Configuration Advisor asks administrators a series of high-level questions—e.g., does the workload consist of short or long transactions, or both?—and recommends configurations based on the answers provided.

2) Model-fitting [4]: This approach conducts some number of experiments where each experiment runs the system using a chosen configuration $C$ to observe the performance $p$ for that configuration. Thus, each experiment gives a sample of the form $(C, p)$. A model can be
trained from these samples which can then estimate performance for any given configuration. The trained model can be 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., [4]).

3) **Adaptive Search** [6], [7], [8]: 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. Simulated annealing is the most common example of Adaptive Search [8]. There has been a lot of recent activity in applying Adaptive Search to parameter optimization, yielding new algorithms like Recursive Random Sampling (RRS) [8], Smart Hill Climbing (SHC) [6], and Quick Optimization via Guessing (QOG) [7].

Table I 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. (Recall that an experiment is a run of the system for a chosen configuration setting.) 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 as well to identify parameters whose settings affect performance significantly (or hardly at all).

The third column in Table I 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., [9], [4]). Given its ease of use, a black-box approach probably has the best chance of generating automated 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 VII, 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. There has been a lot of recent activity in applying Adaptive Search to parameter optimization problems that arise in, e.g., tuning network routing configurations [8] and tuning configuration parameters in application servers [6].

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 some specific indexes in a database system with gigabytes of data, then a lot of time can be spent in generating these indexes. (An index represents a binary configuration parameter which is 0 if the index is absent and 1 otherwise.) 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 setting. For example, for each experiment involving an e-commerce service running on an IBM WebSphere application server and a DB2 database server, [6] 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 configuration settings it chooses to sample.

### A. 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 makes them perform poorly in this setting.
- We present a new Adaptive Search algorithm called

<table>
<thead>
<tr>
<th>Approach</th>
<th>Requires Experiments</th>
<th>Treats System as a Black-box</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expert Rules</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Model-fitting (a priori models)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Model-fitting (statistical models)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Adaptive Search</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**TABLE I**

**COMPARING DIFFERENT APPROACHES FOR PARAMETER OPTIMIZATION**
**MOWILE (MOre With LEss)** 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 in the settings we consider.

- We present an evaluation of different Adaptive Search algorithms on synthetic benchmark functions as well as configuration datasets from three deployed systems. The synthetic benchmark functions we consider are popular functions used to judge the quality of parametric optimization algorithms. These functions enable us to consider hundreds of parameters and to validate how close to optimal the configurations produced by different Adaptive Search algorithms get. We show that the performance improvement of MOWILE over previous algorithms gets better and better as the number of parameters increase. Our real configuration datasets have at most seven dimensions because it can take several months (if not years) to generate representative data with 10-15 parameters. On these low dimensional datasets, we show that MOWILE is comparable (not necessarily the best) to the previous algorithms.

The rest of the paper is organized as follows. In Section II we give the problem definition and introduce the terminology and notation used in the paper. Section III introduces a general framework for Adaptive Search algorithms. Sections IV–VI discuss how three algorithms for Adaptive Search, including our own MOWILE algorithm, instantiate the general framework introduced in Section III. Section VII gives a detailed experimental analysis of all algorithms we consider.

## II. PRELIMINARIES

Let the set of configuration parameters be \( X = \{x_1, x_2, \ldots, x_n\} \), and the domain of a parameter \( x_i \) be denoted as \( Dom(x_i) \). We refer to the full space of configuration settings \( Dom(X) = Dom(x_1) \times Dom(x_2) \times \ldots \times Dom(x_n) \) as the **full configuration space**. Some configurations in the full space may be invalid. For example, if \( x_1 \) and \( x_2 \) are two buffer pool sizes in the database server, then it could be that any configuration with \( x_1 + x_2 > M \) may be invalid; where \( M \) is the total memory available for buffer pools.

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 subset of \( Dom(x_i) \). The **neighbourhood** 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 multidimensional space of parameters. For example, in a two dimensional setting with parameters \( \{x_1, x_2\} \) that have domains \( Dom(x_1) = Dom(x_2) = [0, 10] \), the full configuration space would be the square region \([0, 10] \times [0, 10]\) on the two-dimensional grid. The neighbourhood of a configuration \((5, 5)\) could be a subspace of this square region, e.g., \([3, 7] \times [3, 7]\), which has \((5, 5)\) at its center.

We denote the performance of the system at a configuration setting (or just setting) \( c \) by \( f(c) \). The performance function \( f \) is unknown, and the **parameter optimization problem** is to find a configuration setting that has the maximum (or large enough) \( f \) value\(^1\). To determine the \( f \) value at a configuration setting \( c \), we set the system’s configuration parameters to their respective values in \( c \), 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 \((c, f(c))\) sample.

Given the natural variability in systems, we would want to repeat the experiment at \( c \) multiple times to capture the variability of performance at \( c \). We do not consider such repetitions in this paper. Reference [7] as well as our own work in [10] present methods to measure performance at a configuration in a robust fashion by executing multiple experiments. Our work in this paper concentrates on issues orthogonal to repetitions like selecting the set of distinct experiments (samples) and the Adaptive Search framework. The methods from [10] and [7] can be incorporated into our algorithms in a straightforward manner to include aspects of robustness during performance measurement.

## III. 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). A global search phase is one during which the full configuration 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. During the subsequent local search phase, the chosen subspace \( S \) is examined in more detail to find a good configuration that has close to the optimal performance. Adaptive Search interleaves the global and local search phases in order to ensure that the search process does not get stuck in a locally optimum region \([8]\).

Most Adaptive Search algorithms tend to follow a general framework that consists of the following steps:

1. **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 \( f \) in \( S \).
2. **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 (but not always) completely contained within \( S \).

\(^1\)Without loss of generality, we consider the problem of maximization in our discussion. However, our experimental section includes results for both maximization as well as minimization problems.
3) **Restarts**: A typical run of an Adaptive Search algorithm will involve multiple iterations of Step 1 followed by Step 2, progressively reducing 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 reverts the search to start again by sampling from the full configuration space. Restarts enable the algorithm to escape from a locally optimal region.

The next three sections discuss how three algorithms for Adaptive Search, including our own MOWILE algorithm, instantiate the above three steps.

### IV. SMART HILL CLIMBING (SHC)

Reference [6] proposes an algorithm for Adaptive Search called **Smart Hill Climbing (SHC)**. The authors applied this algorithm to find good configuration settings for application servers like IBM’s WebSphere server. SHC improves over a previous algorithm called **Recursive Random Search (RRS)** [8] that was based primarily on random sampling. This section describes how SHC instantiates the three steps described in Section III.

Broadly, SHC runs in an iterative fashion involving the following steps (details will be provided momentarily): (1) In a given neighborhood, collect a $k$ size sample of configuration settings using the **weighted Latin Hypercube Sampling (wLHS)** scheme; (2) Based on samples collected until now, a configuration value is found for each parameter independently of other parameters, and a full configuration setting is assembled from these values; (3) if the performance at the assembled setting better the optimal performance found in the wLHS step, set the assembled configuration setting as the seed and repeat from Step (1), else (4) set the optimal configuration found during wLHS as the seed, and repeat from Step (1) by considering a reduced neighborhood around the seed. The procedure restarts when the size of the neighborhood becomes smaller than a threshold value.

**A. Sampling from a Subspace in SHC**

To collect a set of samples of size $k$ from a subspace, SHC proposes an approach called **weighted Latin Hypercube Sampling (wLHS)**. The wLHS approach is a variant of the well known **space filling Latin Hypercube Sampling (LHS)** scheme. LHS does the following to generate a set of $k$ configuration settings from a subspace: (1) the domain of each configuration parameter that falls in the subspace is broken into $k$ equal sub-domains, and (2) $k$ configuration settings are generated from the neighborhood such that no two configuration settings fall in the same sub-domain for a configuration parameter. Figure 1(a) shows a set of 5 samples generated from a subspace by LHS in two dimensions.

The wLHS scheme differs from LHS in that it examines already collected samples in the full configuration space to divide the domain of a configuration parameter into unequal sub-domains. The size of a sub-domain is relatively smaller if already collected samples indicate that the performance values in the sub-domain are relatively higher. The intuition here is that wLHS wants more samples from regions that are more likely to contain good configurations. Figure 1(b) shows a set of 5 samples generated by wLHS in a two dimensional subspace. Note that the parameter $x_1$ is divided into sub-domains of decreasing lengths, while $x_2$ is divided into sub-domains of increasing lengths. Once the sub-domains are created, wLHS picks a sample set of settings exactly as LHS does. The sub-domains in Figure 1(b) facilitate generation of more number of samples from the lower-right region of the subspace.

**B. Selecting a Subspace in SHC**

SHC implements subspace selection by picking a configuration setting from the subspace as a seed, and selecting a neighborhood around this seed. The seed is determined in the following way. For each configuration parameter $x_i$, a good configuration value is determined (independently from other parameters) by first fitting a quadratic curve to the performance values and $x_i$ values in already collected samples, and then solving for the optimal value of $x_i$ in this curve. The seed is assigned the assembled setting if its performance is better than the samples collected in the wLHS phase, else the seed is assigned the best sample collected during the wLHS phase.

The size of the neighborhood around the seed is determined in the following way: if the seed was assigned a configuration setting collected during the wLHS phase, the size of the selected neighborhood will be smaller than the current subspace by a shrinkage factor $\alpha$. Otherwise, in the case that the seed is the assembled setting, the size of the selected neighborhood will remain the same as that of the current subspace. In this case, since the size of the subspace is unchanged although the seed has been changed, we say that the subspace has been realigned.

**C. Restarts in SHC**

SHC executes the above two steps, sample selection and subspace selection, in an iterative fashion until the size of the neighborhood reaches a threshold value. When the neighborhood size becomes smaller than the threshold, the iterative procedure is **restarted** from the sample selection step by starting from the full configuration space.

The neighborhood size based bail-out policy assumes neighborhoods smaller than the threshold size are not fit for further sampling and don’t lead to any significant improvements. The intuition behind this cutoff based scheme is that very small.
changes in system configuration parameters may not lead to practical and significant improvements with high probability as the system’s operating characteristics change only a little.

V. MORE-WITH-LESS (MOWILE) ADAPTIVE SEARCH

In this section, we present our new MOWILE sampling approach for finding good configuration settings quickly. MOWILE falls under the adaptive search paradigm. Below, we discuss how MOWILE implements the sample selection, subspace selection, and restart steps.

A. Sampling from a Subspace in MOWILE

![Image of sample generation by LHS]

(a) A bad case of sample generation by LHS

(b) A bad case of sample generation by LHS when it does not consider already existing samples

Fig. 2. Bad Sample generation by LHS

SHC uses a variant of LHS for sampling subspaces. Although LHS and wLHS generate samples with better spread than pure random sampling [6], it can generate sets of samples close to each other. Figure 2(a) shows an example scenario where LHS generated samples close to each other, leaving out empty areas in the upper-right and lower-left regions of the sample space. Furthermore, neither LHS nor wLHS consider samples already present in the space while determining the positions of the new samples. Figure 2(b) shows an example scenario where LHS may generate samples around an already collected sample (denoted by ‘o’), thus decreasing the effectiveness of sample generation during space exploration.

The desirable property during sparse sample generation is to get as much space coverage as possible. Generating future samples by considering already collected samples in the subspace ensures that old and new samples don’t fall beside each other, thus saving a lot of sampling potential. MOWILE’s implementation of the sampling subspace step has two distinct advantages over SHC’s implementation: (1) samples are generated for good space coverage, and (2) samples already present in the subspace are considered to maximize sampling’ effectiveness.

MOWILE implements the sampling phase using k-FF (k-Furthest First) sampling in the following way: given a set of \( \{(x_i, p_i)\}_i \) pairs in a neighborhood \( N \), select a set of \( k \) settings \( \{x_j^*\}_{j=1}^k \) in the neighborhood such that they maximize the following expression:

\[
\sum_{x_j^* \in \{x_j^*\}} \min_{y \in \{x_i \cup \{x_j^*\}\} \setminus x_j^*} |x_j^* - y|
\]

(1)

The inner term in the above expression is the minimum distance between the newly collected sample \( x_j^* \) and all the other samples in the neighborhood; by \( all \) we mean the old samples in the subspace whose performance values are known, as well as the newly drawn up settings. Maximizing the above expression involves generating samples from the space which are far away from existing samples as well as far away from each other. The intuition behind this consideration is, by collecting samples far away from each other, the sampling scheme explores as much space as possible.

The above sampling scheme is implemented as follows: (1) to collect a set of samples \( S \) of size \( k \) from the given subspace, a candidate set of samples \( C \) of size \( \beta \cdot k \) is generated randomly (\( \beta \) is a pre-determined constant), (2) the following step is iteratively executed \( k \) times: the sample in \( C \) with the highest value for the expression in Equation 1 is included in \( S \) and removed from \( C \). Note that the above greedy two-step procedure does not generate the optimal set \( \{x_j^*\} \), but it outputs a good enough set for our purposes.

Let \( I = \{x_i, p_i\} \) be the set of samples already present in the subspace. The complexity of the above algorithm is \( O(k^2 \cdot \beta \cdot (|I| + k)) \). This is so because, the second step is executed \( k \) times, and each time all samples from \( S \) of size \( \beta \cdot k \) are checked to find the best sample that optimized equation 1.

B. Selecting a Subspace in MOWILE

The goal of the subspace selection step is to choose a neighborhood from the given space which is rich in good configuration settings and potentially contains the optimal setting. The outcome of the selection step influences which regions are sampled in future, and hence plays an important role in determining how well the overall sampling scheme runs.

The sample generation step presented in the previous section generates a set of samples which has good coverage of the given space. To select a neighborhood in the space, we pick the sample from this neighborhood from among the collected samples with the highest performance as the seed. The neighborhood around it which is a fraction \( \alpha \) of the given space is selected as the subspace.

Let \( L(x) \) denote the length of the dimension \( x_i \) in the given space. The selection phase first picks a seed \( ([a_1, a_2, \ldots, a_n], f) \) from the given space with maximum performance. Next for each dimension \( x_i \), a neighborhood around \( a_i \) of size \( \frac{\alpha \cdot L(x)}{n} \), where \( n \) is the number of configuration parameters, is chosen. The hypercube formed by the neighborhoods for all the dimensions is chosen as the subspace in the selection phase.

The implementations of subspace selection in MOWILE and SHC differ in two ways: selecting the seed, and selecting a neighborhood around it. MOWILE does not consider quadratic fits, and always picks a shrunk neighborhood. Thus, realignments are not present in MOWILE.

C. Restarts in MOWILE

MOWILE iteratively runs the two steps sample selection and subspace selection one after the other until the size of the
subspace under consideration drops below a threshold. When the size of the subspace drops below a threshold, the execution is restarted from the sample selection step by running it on the complete configuration space.

We observe that the number of restarts play an important role on the effectiveness of the adaptive search. It is important to get a minimum number of restarts during the execution of the complete algorithm so as to facilitate search in multiple regions of the total configuration space.

SHC execution may include subspace realignment steps, which do not decrease the size of the subspace. This can lead to prolonged execution of the search algorithm, without a restart. Such an execution may fall in locally optimum regions and sacrifice on global exploration. Since realignments are not considered in MOWILE, the number of restarts can be engineered without significant effort by changing subspace size threshold and the parameter \(k\) in the sample generation step.

VI. Quick Optimization via Guessing (QOG)
Quick Optimization via Guessing (QOG) [7] is another approach for server configuration. Broadly, QOG runs in an iterative fashion involving the following steps: (1) consider all samples collected so far and build a regression model over the full configuration space, (2) solve the regression model for a guess of the optimal configuration setting, and evaluate the performance of the system at this guess.

QOG differs from MOWILE and SHC in two major ways. First, it uses a regression-model-based approach to guess a good configuration setting. It differs from the approach in SHC which uses an assembly method that involves independently determining the best configuration value for each parameter. Also, QOG collects one sample at a time, unlike MOWILE and SHC. A second major difference is that, to account for variability in determining performance at a configuration setting, QOG repeat system runs at a configuration setting until it is sure of the performance value to a high degree of confidence. Recall from Section II that repetitions are orthogonal to the problem of determining which distinct samples to collect.

VII. Results
In this section, we present the results of MOWILE on suite of benchmark objective functions. Further, we analyze the design motivations of MOWILE in comparison to Smart Hill Climbing (SHC) and demonstrate that MOWILE is a right choice when the number of parameters are high and experimental budget is low. Because of space constraints, only a subset of our results are presented in this paper. For the complete set of results, see the on-line technical report[11].

A. Experimental Setup
Our experiments were conducted with the following system configuration: 3.6 GHz Intel Pentium IV, 1GB RAM, Linux CentOS v5.0, Matlab v7.5

B. Benchmark Objective functions
The following suite of benchmark objective functions are used to test the performance of MOWILE. In the equations, \(x\) denotes a n-dimensional vector and \(x_i\) its \(i\)-th element. While the DeJong function is considered to be simple benchmark function to predict, Rastrigin, Griewangk and Rosenbrock’s Saddle are considered to be difficult benchmark functions. The surface of the functions for 2 dimensions is illustrated in Figures 3 and 4.

- **DeJong function**: Continuous, convex and unimodal. The global minima is 0 at \(x_i = 0\).
  \[ f(x) = \sum_{i=1}^{n} x_i^2 \]

- **Rastrigin function**: Multi-modal function with location of the local minima evenly distributed across the surface.
The global minima is 0 at \( x_i = 0 \).

\[
f(x) = n \cdot A + \sum_{i=1}^{n} x_i^2 - A \cdot \cos(2\pi x_i)
\]

- **Griewangk function**[8]: Multi-modal function with widespread local minima. The global minima is 0 at \( x_i = 0 \).

\[
f(x) = \sum_{i=1}^{n} x_i^2/4000 - \prod_{i=1}^{n} \cos(x_i/\sqrt{i}) + 1
\]

- **Rosenbrock’s Saddle**[8]: The global optimum for Rosenbrock’s Saddle is inside a long, narrow, parabolic shaped flat valley. There are strong integrations between variables. The global minima is 0 at \( x_i = 1 \).

\[
f(x) = \sum_{i=1}^{n-1} 100 \cdot (x_{i+1} - x_i^2)^2 + (1 - x_i)^2
\]

\[
\text{Rastrigin function} \quad \text{Number of experiments are: 200}
\]

\[
\text{Griewangk function} \quad \text{Number of experiments are: 200}
\]

\[
\text{Rosenbrock function} \quad \text{Number of experiments are: 200}
\]

\[
\text{DeJong function} \quad \text{Number of experiments are: 200}
\]

**C. Experimental Considerations**

The motivation of MOWILE is to quickly find the good configuration and scale to high number of dimensions. Finding the good configuration quickly is important since conducting experiments is expensive and time consuming. For a database system like DB2, each experiment typically takes more than 20 minutes. This makes the number of experiments per day less than 72 and so the search algorithm should be able to find the good configuration with limited number of experiments. With this constraint, we set the number of permissible experiments to be 200.

Many modern services have hundreds of configurable parameters. For e.g. the database system DB2 has more than 100 configurable parameters, out of which 35-40 are considered to be important. We show the the performance plots for MOWILE up to 100 dimensions.

**D. Experimental Results**

Figure 5, 6, 7 and 8 shows the performance of the algorithms on the suite of objective functions with varying number of dimensions and Figure 9 and 10, shows the performance with varying number of experiments.

The parameter settings for SHC are configured as per the guidelines in [6]. The initial number of LHS samples as well as the samples after restart are set to 5, the restart threshold is set to 10% and volume shrinkage after local search is set to 80% of the total search space. The weight parameter \( c \) for LHS is set to 0.00001. For MOWILE, we set the initial number of random samples to be 5. MOWILE selects 8 samples at each selection step and repeats the selection step 5 times before it restarts. This makes MOWILE to collect 40 samples before it
Fig. 9. Convergence tests on Rastrigin and Griewangk function for 30 and 100 dimensions as the number of experiments approaches 200.

Fig. 10. Convergence tests on Rosenbrock and DeJong function for 30 and 100 dimensions as the number of experiments approaches 200.

Fig. 11. SHC - Quadratic fits on Rastrigin and Rosenbrock functions for 50 dimensions. The performance impact due to quadratic fits is negligible.

Fig. 12. MOWILE and SHC performance on Rastrigin and Rosenbrock functions each performing the same number of restarts for global search.

Fig. 13. Convergence plots for MOWILE and SHC performance on Rastrigin and Rosenbrock functions each performing the same number of restarts for global search.

Fig. 14. Reats vs Performance
restarts. The idea here is balance the local and global search phases given the constraint on total number of permissible experiments. If the algorithm spends too much time in the local search phase, it may get stuck with a local optima and may exhaust the permissible experiments without exploring the complete space. So the algorithm has to trade-off between global and local search smartly. Based on our experiment analysis, we recommend the algorithm to perform at least 3 restarts to get close to a good configuration. The number of selections at a local phase can be set accordingly.

First, we compare the performance of SHC and MOWILE with increasing dimensions. Figures 5, 6, 7, 8 show as the number of dimensions increase, MOWILE clearly outperforms SHC. For Rastrigin function with 20 dimensions, SHC[6] reports 600 as the mean of the optimization result. Figure 5 shows 400 and 200 as the mean of the optimization result for SHC and MOWILE respectively. This shows MOWILE performing 2 times better than SHC. In the case of Rosenbrock function Figure 7, which is more sensitive to the input setting, MOWILE performs 4 times better than SHC as the number of variables approaches 40. Similarly, MOWILE is performing consistently better for the complex functions like Griewangk as well as for the simple functions like De Jong. This demonstrates MOWILE is performing better than SHC with increasing number of dimensions.

Second, we compare the convergence of SHC and MOWILE while varying the number of experiments. Figures 9 and 10 show the SHC performs better in the beginning (20-30 experiments) compared to MOWILE. This is due to the fact that SHC’s volume threshold setting is 10%, which results in SHC restarting more often compared to MOWILE in higher dimensions; and thereby SHC spends less time in local search. Upon restarts, SHC hits a local optima and after first 20-30 experiments, there after restarts may not help in finding global optima. Since before the local search converges, SHC tends to restart and so SHC performance improvement remains more or less constant. On the other contrary, MOWILE can constrain the number of restarts and make the search more balanced. This makes MOWILE converge quicker than SHC and our results demonstrate this phenomenon.

E. Experimental Analysis

The efficiency of MOWILE compared to SHC is due to the improvements in global and local search phases. We analyzed the performances of each algorithm by plugging different components in each phase. We found the success of MOWILE is mainly due the following five factors: i) flexibility to select the number of restarts based on the number of permissible experiments ii) use of k-FF instead of wLHS for global search iii) considering the previous experimental samples while doing the global search for better space exploration iv) simplifying the local search with out re-aligning v) sensitivity of the algorithm to it’s own parameter configuration. This section discusses in detail how these factors influence each algorithm.

1) Performance vs Restarts: The number of restarts should be chosen carefully as our experiments show that the number of restarts an algorithm performs is the major factor contributing to performance. With our settings, MOWILE is performing on the average 4 restarts where as SHC is performing 18 restarts for 100 dimensions and 200 experimental samples. To analyze the effect, we first restricted the number of restarts that SHC is allowed to do and made SHC to do the same as the number of restarts as MOWILE. Figure 12 and Figure 13 shows the performance of MOWILE and SHC with the same number of restarts. Although MOWILE still performs better, the gap between SHC and MOWILE is narrower and SHC performs much better when compared to previous case with 18 restarts. Second, we analyzed the effect of restarts on MOWILE’s performance for 200 samples. Figure 14 shows
as the performance is best when the number of restarts are 4 and it worse with either less or more number of restarts. More restarts imply the algorithm spends more time in global search and the performance depends on how well the random sampling gets closer to the optimum. Less restarts imply the algorithm will spend more time in local search and it has high chance of getting stuck in the local optima. So, the search algorithm should make a good trade-off between global search and local search for quickly finding the global optimum. For e.g. for 80-100 experiments, we observed 3 restarts to give a balanced trade-off between global and local search. With SHC, it is difficult to estimate the number of restarts based on the volume threshold and so we have chosen the approach of explicitly constraining the restarts through number of selections in the local phase.

2) k-FF vs LHS for Global Search: Using k-FF in global search gives a better representation of space compared to LHS. As seen in Figure 1, one of the possible ways of selecting the samples in the space using LHS can be selecting all the diagonal samples. This completes the upper and lower diagonal space completely left unexplored. On the other hand, using k-FF will avoid such scenarios since k-FF tries to select the samples as distant as possible from the previous seen samples. To analyze the performance of k-FF, we ran SHC replacing the global search phase with k-FF instead of LHS. Figure 15 shows SHC with k-FF performs better than SHC with LHS. This makes k-FF a better choice of space filling compared to LHS.

3) Importance of previous samples during restart: MOWILE takes to account the previous experiment samples while selecting the samples for the current run in the global search phase. This makes the new samples to be as distant as possible from the previous samples and there by reduces the number of redundant experiments. On the other hand, SHC ignores the previous samples after restart and there by increasing the possible of selecting the same samples again. This is important to reduce the redundant experiments or experiments which might result in the same performance value given the constraint on the total experiments.

4) Simplifying the local search with out re-aligning: In local search phase, SHC uses quadratic fits across each dimension and re-aligning to other sub-space in search of getting closer to local optima even if the initial samples miss it. SHC does not give the flexibility to impose the constraint on the number of experiments in the re-aligning step. Rather, a restart happens when the local optima is found after several re-aligns and volume shrinks below some threshold. If SHC finds a sample near local optima, SHC tries to re-align to get closer to the local optima if the volume threshold is low and there by spend most of the budget in conducting experiments around the locally optimum area. Coming up with a right volume threshold might be difficult as we see in next section. The quadratic fits in the local search have negligible effect on the performance as we see in Figure11. This may be due to the quadratic fits are done independently across each dimension and they do not consider the interactions between parameters.

In comparison, the local search phase of MOWILE simply picks the local optima without re-alignment and goes to global search for exploration.

5) Sensitivity of algorithm parameters: Configuring the search algorithms for optimization is a black art in itself. SHC has number of parameters like weight factor c, shrinkage factor α, number of samples to be selected at restarts, initial number of samples, threshold volume, which has to be configured appropriately for better search result. The configuration which work for one workload might not work for other workload. We particularly observed this for the factors c and α (Figure 16 and 17) where a slight changes to these parameters will affect the performance drastically. On the contrary, the parameters for MOWILE are less and exhibits more stable behavior. This can be seen in the results which exhibit consistent convergence for MOWILE.

VIII. 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 different solutions for solving this problem and showed how certain design decisions incorporated in them lead to poor performance in this setting. We proposed a new search algorithm called MOWILE (doing MOre With LEss) that addressed these limitations, and outperformed previous algorithms by large margins as the number of parameters increase.

REFERENCES