A Smart Hill-Climbing Algorithm for Application Server Configuration

Bowei Xi
Department of Statistics
University of Michigan
Ann Arbor, MI
xbw@umich.edu

Zhen Liu
IBM T.J. Watson Research
Center
Hawthorne, NY 10532
zhenl@us.ibm.com

Mukund Raghavachari
IBM T.J. Watson Research
Center
Hawthorne, NY 10532
raghavac@us.ibm.com

Cathy H. Xia
IBM T.J. Watson Research
Center
Hawthorne, NY 10532
cathyx@us.ibm.com

Li Zhang
IBM T.J. Watson Research
Center
Hawthorne, NY 10532
zhangli@us.ibm.com

ABSTRACT

The overwhelming success of the Web as a mechanism for facilitating information retrieval and for conducting business transactions has led to an increase in the deployment of complex enterprise applications. These applications typically run on Web Application Servers, which assume the burden of managing many tasks, such as concurrency, memory management, database access, etc., required by these applications. The performance of an Application Server depends heavily on appropriate configuration. Configuration is a difficult and error-prone task due to the large number of configuration parameters and complex interactions between them. We formulate the problem of finding an optimal configuration for a given application as a black-box optimization problem. We propose a Smart Hill-Climbing algorithm using ideas of importance sampling and Latin Hypercube Sampling (LHS). The algorithm is efficient in both searching and random sampling. It consists of estimating a local function, and then, hill-climbing in the steepest descent direction. The algorithm also learns from past searches and restarts in a smart and selective fashion using the idea of importance sampling. We have carried out extensive experiments with an online brokerage application running in a WebSphere environment. Empirical results demonstrate that our algorithm is more efficient than and superior to traditional heuristic methods.

Categories and Subject Descriptors

Performance of Systems [**Optimization**]

General Terms

Performance,Algorithm, Management

Keywords

System Configuration, Automatic Tuning, Importance Sampling, Simulated Annealing, Gradient Method

1. INTRODUCTION

The Web has emerged as the central mechanism for the interchange of information and for the transaction of commerce. To streamline interactions with businesses, consumers, and employees, organizations have undertaken the development of web-based applications for many of their core business processes. Such applications range from online stores and brokerage to human resource applications to supply-chain management applications. The performance and availability of these applications, especially in situations of heavy demand, are essential to the operation of an organization.

An enterprise application can be difficult to develop and deploy. It requires the integration of complex business logic with a presentation layer that provides an intuitive web-based front to the application, and with legacy systems and databases. To facilitate the rapid deployment of such applications, which is a central concern of businesses, frameworks such as Java 2 Enterprise Edition (J2EE) and .NET have been developed. In these frameworks, application developers are presented with high-level abstractions that simplify the development of enterprise applications. Programmers are shielded from handling issues such as transactions, database interactions, concurrency, memory, etc., explicitly. These issues are handled by the *application server*, a complex software system on which enterprise applications developed in these frameworks are deployed and executed.

The performance of an application server depends heavily on appropriate configuration. An application developer must configure an application server so that it can manage, for example, the concurrency of an enterprise application appropriately. In general, configuration is a difficult and error-prone task [12] due to the large number of configuration parameters and complex interactions between them — an application server may have more than a hundred parameters that can be modified. Examples of the parameters include the configuration of multiple thread pools, queues, cache size, timeouts and retry values, and of memory.

Consider Figure 1 which shows an instance of a J2EE-based application server and the components with which interacts. Typically, HTTP and Web Service requests flow through a Web (or HTTP) Server to an application server. An application server can be thought of consisting of three components: a Web Container, the component corresponding to the presentation layer, where JSPs,

static HTML pages, and servlets execute, an EJB Container, the component corresponding to the business logic layer, where Enterprise Java Beans (EJBs) execute, and the Data Source layer, an abstraction of a database or other back-end, where transactions and interactions to persistent data stores are handled. Requests generally flow from Web Containers to EJB containers to Data Sources (and from there to a database), but other access patterns are possible as well.
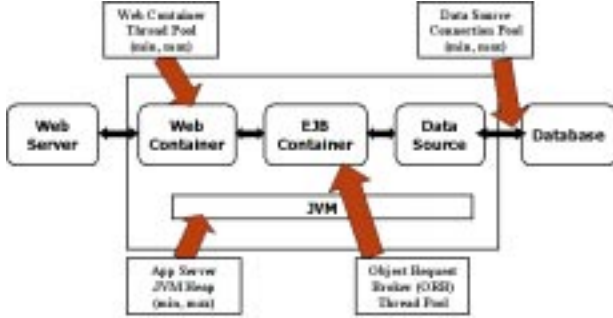

Figure 1: System Model

Each of these components has configurable parameters that can have a significant effect on performance. For example, the Web container maintains a thread pool to process inbound HTTP requests for servlet and Web services, which controls how many requests are active at any time. Enterprise beans are generally invoked by servlets and their execution is managed by the Object Request Broker (ORB) inside the EJB container. The ORB thread pool size can also be customized. The size of the Data Source Connection Pool can affect the amount of concurrent access to the database. In addition to the parameters that affect the behavior of the components, one can also configure the Java Virtual Machine (JVM) on which the application server executes, for example, to set the size of the heap available to the application server.

Not all configuration parameters will be relevant for any given application, but determining the relevant parameters and setting them appropriately can have a significant effect on the performance of an application (and sometimes, determines whether the application executes at all). The system performance often depends on these parameters in a non-linear non-convex way, which further complicates reasoning about them. Furthermore, once these parameters are set, it is often impossible to change them without bringing down the system or the application services, generally an unacceptable option.

Despite the importance of configuration, the setting of these parameters is a black art in practice today. Developers either use rules-of-thumb, heuristics, along with best practice guidelines provided by software vendors to derive these settings. The configuration of an application server, however, depends heavily on the particular application being deployed and its expected workload. Furthermore, the configuration of an application server may depend on that of the systems, such as databases, with which it interacts, and the overall infrastructure in which the application server executes. Trial-and-error efforts may assist with the determination of appropriate configuration settings, but this testing process requires a significant amount of expert knowledge about the system and the applications being tested. The trial-and-error process may also require a considerable amount of time. A typical test run under a given setting can take 20 to 30 minutes. Therefore, efficient systematic methods for finding close to optimal system configurations automatically are useful in practice. In this paper, we investigate

several sampling and search algorithms for finding the best configuration setting with a small number of test runs.

There has been little previous work on the optimal configuration for Web application servers. In the context of HTTP servers, which have less complex interactions than Web Application Servers, [11] describes an Apache implementation that manages web server resources based on maximizing revenue. This approach requires substantial modifications to the Apache resource management schemes. [8] uses layered-queueing modeling to model business process applications. The method basically requires thorough knowledge about the software architecture and can be expensive and time-consuming. [16] describes an approach that combines queueing theory and control theory for response time regulation, the approach can be used to handle a limited and small number of parameters. Using feedback control systems, [2, 9] studied the problem of regulating system performance within specified QoS value. The approach works well for a small number of tuning parameters with some linear dependency assumptions.

We formulate the problem of finding an optimal configuration of an application server for a given application as a black-box optimization problem. We then propose a Smart Hill-Climbing algorithm based on the ideas of importance sampling and Latin Hypercube Sampling (LHS). The algorithm is efficient in both searching and random sampling. Note that a similar black-box approach has been used in [19] for large-scale network parameter configuration using on-line simulation, where a random recursive algorithm was proposed to search for a reasonably good solution. Our experimental results demonstrate that our algorithm is significantly more efficient and superior.

The rest of the paper is organized as follows. Section 2 presents the black-box optimization formulation and reviews existing methods solving such a problem. Section 3 presents our Smart Hill-Climbing algorithm in detail. Section 4 demonstrates the efficiency of our algorithm using test functions. Section 5 provides experimental results on a WebSphere environment. Finally we present concluding remarks in Section 6.

2. BLACK-BOX OPTIMIZATION

It is natural to consider the system performance y as a function g of a given number of tunable parameters $x_1, \cdots, x_N$ and some other fixed environment settings and the load condition. That is, $y = g(\mathbf{x})$, where $\mathbf{x} = (x_1, \cdots, x_N)$. We assume that the performance can be measured through a single-dimension metric. Such a metric can be throughput, response time, system utilization, or a combination of them. For example, if one needs to optimize both response time and throughput, an artificial formula could be introduced to tie the two together as a single metric. Let I_i denote the parameter range for parameter x_i, and $\mathbf{I} = I_1 \times \cdots \times I_N$. The parameter tuning problem is to find the parameter setting $\mathbf{x}^*$ that achieves the the best performance, i.e.:

$$\mathbf{x}^* = \operatorname*{argmin}_{\mathbf{x} \in \mathbf{I}} g(\mathbf{x}). \tag{2.1}$$

We assume that the parameter space $\mathbf{I}$ is a compact set in $\mathbb{R}^N$, and the function g is continuous on $\mathbf{I}$. Hence, the existence of optima $\mathbf{x}^*$ in the searching space $\mathbf{I}$ is guaranteed.

In practice, the performance function $g(\mathbf{x})$ is often unknown or does not have a closed-form. Certain function evaluation can be obtained through experiments or simulation. We therefore have to model the above problem as a black-box optimization problem where the objective function is a black-box with limited function evaluation.

Black-box optimization problems, also known as *global opti-*

mization [17] problems, arise commonly in many areas such VLSI design, performance engineering, inventory management, and medical treatment planning. The challenge is to obtain the global optimal solution, since the objective function is usually high-dimension, highly non-linear, non-convex and multi-modal, where a local optimum is typically not the global optimal solution.

When **I** is large, finding x^* is generally hard. In addition, function evaluation for each individual parameter setting x can be expensive and time consuming, requiring effort in setting up the experiments and data collection. One simply cannot afford to carry out too many of these experiments. Therefore, *efficiency* is one of the greatest concerns in solving the problem. What is required is an efficient search algorithm that obtains a *good* solution with a *minimum number* of experiments.

Many heuristic search algorithms are in the above spirit, attempting to find near-optimal or *best-possible* solutions instead of global optima. These include simulated annealing [14], random recursive search [18], genetic algorithms [6], Tabu search [5], and hill-climbing [15, 1]. The first three are generally applicable as they require little *a priori* knowledge of the problem. When the objective function has an explicit form, Hill-climbing could quickly reach an optimal point by following the local gradients of the function. We now review some of these algorithms which are more relevant to our problem setting and to the new algorithm that we propose later.

Simulated annealing is a search heuristic commonly used to solve global optimization problems, especially in the presence of many false minima. It was motivated by the annealing process for a material to reach the thermal equilibrium [7]. A simulated annealing optimization starts with a Metropolis Monte Carlo simulation at a high temperature. This means that a relatively large percentage of the random steps that result in an increase in the energy will be accepted. After a sufficient number of Monte Carlo steps, or attempts, the temperature is decreased. The Metropolis Monte Carlo simulation is then continued. This process is repeated until the final temperature is reached. The way in which the temperature is decreased is known as the cooling schedule. When the cooling schedule is controlled appropriately, the algorithm is guaranteed to achieve a global optimum [4]. Despite its many successful applications, using simulated annealing efficiently is a bit of an art — convergence can be slow.

Recursive random search [18] utilizes pure random sampling. The algorithm uses initial random sampling to identify promising areas, and then, starts recursive random sampling processes in these areas which shrink gradually to local optima. The algorithm then *restarts* random sampling, trying to find a more promising area to repeat the local recursive search. The algorithm in general produces a local optimum and has no guarantee to be optimal or near-optimal. Moreover, since the algorithm uses naive random sampling, it may waste effort on restarts.

In the next section, we propose a *Smart Hill-Climbing* algorithm based on the ideas of importance sampling and Latin Hypercube Sampling (LHS). The algorithm is efficient in both searching and random sampling. It consists of estimating a local function and then in Hill-Climbing in the steepest descent direction. The algorithm also learns from past searches and restarts in a smart and selective fashion using importance sampling. Empirical results demonstrate that our algorithm is efficient and superior than traditional heuristic methods.

3. SMART HILL-CLIMBING

The Smart Hill-Climbing algorithm introduced in this paper has the same basic structure as the recursive random search algorithm.

The algorithm has two main phases, a global search phase and a local search phase. The goal of the global search phase is to cover the search space as broadly as possible in order to identify a good start for the local search phase. The local search phase then starts from the starting point selected in the global search and applies a gradient-based sampling method to search around its neighborhood for a better solution. The size of the neighborhood becomes smaller as the local search progresses. The algorithm's performance depends on the efficiency of both the global and the local search algorithms. For the global search phase, we employ Latin Hypercube Sampling (LHS) which generally provides high quality sampling coverage. We have further extended LHS with importance sampling. This extension takes advantage of correlation factors to ensure that the algorithm samples more frequently from the region that is likely to provide better results. For the local search phase, gradient-based searching algorithms usually converge quickly to local optimal solutions. We apply a gradient algorithm based on constructing locally fitted quadratic functions, which leads to better convergence for the overall algorithm. We now provide a detailed description of our Smart Hill-Climbing algorithm.

3.1 Latin Hypercube Sampling

One of the important components in our Smart Hill-Climbing algorithm is the sampling strategy. As pointed out in [18], "the disadvantage of random sampling is its apparent lack of efficiency". Since the dimension of the problem we are dealing with is usually high, naive sample methods can become very expensive. We therefore rely on the Latin Hypercube Sampling (LHS) scheme [10].

LHS is considered to be an extremely efficient space-filling sampling strategy for handling high dimensions. It is considered more powerful than pure random Monte Carlo sampling. The basic idea of LHS is to divide probability distributions into intervals of equal probabilities and take one sample from each interval. Specifically, the general LHS algorithm for generating K random vectors (or configurations) of dimension N can be summarized as follows:

1. Generate N random permutations of $\{1, ..., K\}$, denoted by $\vec{P}^1, ..., \vec{P}^N$, where $\vec{P}^i = (P_1^i, ..., P_K^i)'$.

2. For the i-th dimension ($i = 1, \cdots, N$), divide the parameter range I_i into K non-overlapping intervals of equal probabilities.

3. The kth sampled point is an N dimensional vector, with the value for dimension i uniformly drawn from the P_k^i-th interval of I_i.

Figure 2 illustrates two sets of such LHS samples, one denoted by dots and the other by triangles, both generated 5 random samples of dimension 2. Note that a set of LHS sample with K vectors will have exactly one point in every interval on each dimension. That is, LHS attempts to provide a coverage of the experimental space as evenly as possible. Compared to pure random Monte Carlo sampling, LHS provides a better coverage of the parameter space and allows a significant reduction in the sample size to achieve a given level of confidence without compromising the overall quality of the analysis [3].

Given the above advantages of LHS, we use LHS in our Smart Hill-Climbing algorithm whenever there is a need to do random sampling. In the following sections, we shall show that by using LHS instead of purely random sampling, one can achieve better efficiency gains even in existing search algorithms such as simulated annealing or random recursive search.
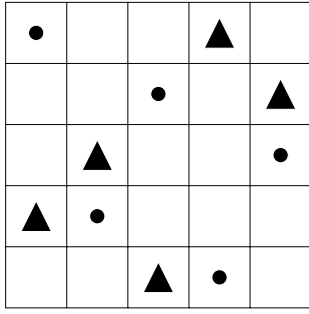
Figure 2: LHS Samples from 2 Variables

3.2 Weighted Latin Hypercube Sampling

We have extended the standard Latin Hypercube Sampling algorithm to take into account knowledge about the correlations between parameters and system performance. System performance is often correlated with certain parameters. This kind of correlation corresponds to a linear approximation of the objective function, which can be estimated using the sampled points. The correlation information can be combined with the Latin Hypercube Sampling to generate skewed random search samples that are likely to lead to more efficient searches.

The weighted LHS incorporates the correlation between the parameters and the performance function to generate the intervals for each dimension and to sample points from a given range. Assume at a given point in time, we have carried out S experiments, (note that S increases as the algorithm proceeds), where $\mathbf{x}^s = (x_1^s, \ldots, x_N^s)$ is the vector of parameter setting at experiment s, $s = 1, \ldots, S$, and y^s is the corresponding function value of the black-box objective function. For example, y^s could be the client response time. We then have for the unknown function g,

$$y^s = g(\mathbf{x}^s), \quad s = 1, \ldots, S.$$

We first analyze the correlation structure based on these measurements. For simplicity, we study the correlation between the performance and each individual tuning parameter. Let

$$Y = (y^1, ..., y^S), \text{ and } \mathbf{X}_i = (x_i^1, ..., x_i^S), \quad i = 1, \ldots, N.$$

Here $\mathbf{X}_i$ collects all used values for parameter i in the past S experiments.

We perform linear regression to analyze the relationship between the performance and parameter i. That is, we obtain estimates for a_i and b_i, $i = 1, \ldots, N$ based on the past S measurements: $Y = a_i \mathbf{X}_i + b_i$. Notice that

$$a_i = \rho_i \, \text{std}(Y) \, / \, \text{std}(\mathbf{X}_i), \tag{3.1}$$

where ρ_i is the correlation coefficient between the performance and parameter i.

One of the key ideas in designing the Smart Hill-Climbing algorithm is that new samples should follow the correlation pattern exhibited by the past measured points. If the past measurements show that smaller values of parameter i tend to make the performance better (i.e., a strong positive correlation), then smaller values are more *important* than larger ones. Hence we should sample more on the smaller values for parameter i. We call this sampling strategy as *importance sampling*.

To realize the above importance sampling idea, we use an truncated exponential density function for generating the samples. For each dimension i, $i = 1 \ldots N$, we assume a truncated exponential

density function of the form:

$$f(c, d, x) = de^{-acx}, \tag{3.2}$$

on sampling range $x \in [A, B]$, where a is determined by (3.1) and represents the correlation between performance and a parameter through all past observations. Parameter c is used to reflect how aggressive the user wants the importance sampling to be. Parameter d is the normalizing factor so that $f(c, d, x)$ is a density function, thus

$$d = \frac{ac}{e^{-acA} - e^{-acB}}.$$

We would like to divide the interval $[A, B]$ into K intervals with equal probability $1/K$. Let z_j be the jth dividing point, with $j = 1, \ldots, K$, and $z_0 = A$ and $z_K = B$. Then,

$$\frac{j}{K} = \int_{z_0}^{z_j} f(c, d, x) dx.$$

This leads to the solution for z_j:

$$z_j = -\frac{\log\left(e^{-acA} - \frac{acj}{dK}\right)}{ac}. \tag{3.3}$$

We now need to draw one point ξ_j from given interval $[z_j, z_{j+1}]$ which follows the conditional probability $f(c, d, x)/h$, where h is the scaling constant. It is easy to solve for h from the normalizing equation,

$$1 = \int_{z_j}^{z_{j+1}} \frac{f(c, d, x)}{h} dx.$$

Therefore, the expression for h is,

$$h = \frac{d(e^{-acz_j} - e^{-acz_{j+1}})}{ac}.$$

We first draw a random number u, from the uniform distribution in $[0, 1]$. We need to draw the point ξ_j such that,

$$u = \int_{z_j}^{\xi_j} \frac{f(c, d, x)}{h} dx$$

After standard algebraic manipulations, the expression for ξ_j becomes:

$$\xi_j = -\frac{\log\left(e^{-acz_j} - u\left(e^{-acz_j} - e^{-acz_{j+1}}\right)\right)}{ac} \tag{3.4}$$
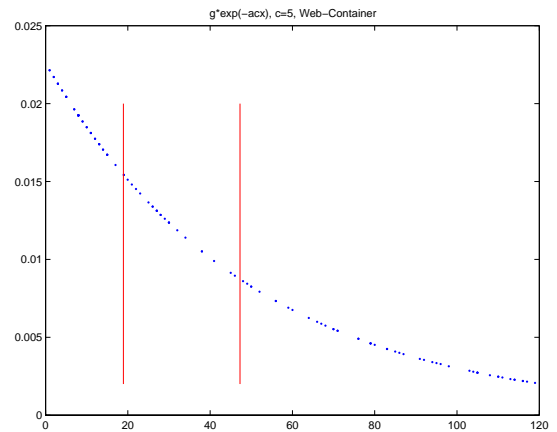

Figure 3: Importance Sampling using the Correlation Structure

Figure 3 illustrates that for a particular parameter with a positive correlation with $\rho \sim 0.6$, the importance sampling strategy using the truncated exponential density function (3.2), would therefore divide the sampling space $[1, 120]$ into 3 equi-probability intervals. Clearly smaller values are stressed more under importance sampling. The constant c in (3.2) can be determined through some preliminary studies. Larger c would naturally result in more aggressive importance sampling.

REMARK 1. *We emphasize that the concept of important sampling so as to take advantage of past observations is applicable in a much more general context than the one described above. The importance can be based on other metrics (rather than correlation), also one can choose other density functions rather than exponential to realize similar concept of assigning different weights on different sampling regions.*

The weighted LHS algorithm for generating K random vectors (or configurations) of dimension N can be summarized as follows:

1. Generate N permutations of $\{1, \ldots, K\}$, $\vec{P}^1, \ldots, \vec{P}^N$, where $\vec{P}^i = (P_1^i, \ldots, P_K^i)'$.

2. For the ith dimension ($i = 1, \ldots, N$), divide the parameter range into K non-overlapping intervals with equal probability $1/K$. The dividing points are given by equation (3.3).

3. Within the kth interval (for the ith dimension), generate a random sample, ξ_k^i, following the truncated exponential distribution, according to equation (3.4).

4. The jth sampled point is an N dimensional vector, with the value for dimension i equal to $\xi_{P_j^i}^i$, $i = 1, \ldots, N$.

3.3 Algorithm: Smart Hill-Climbing

The Smart Hill-Climbing is similar to the recursive random search procedure in that it draws random global samples at the beginning and the restarting phase. The global sampling can be performed using either LHS, or for more efficiency, weighted LHS. From the candidate points selected in the global sampling phase, further local searches are performed. The local search phase also takes advantage of knowledge of sampled points to guide local searches. Quadratic approximations based on sampled points guide the local search procedure so that it converges quickly to an optimal solution. After generating a predefined number of samples within a neighborhood, a quadratic fitting curve is constructed based on the existing local samples. The optimal point on the quadratic curve is the next candidate for the best solution. This fitting procedure is repeated a number of times until no better solutions can be found, after which the size of the sampling neighborhood is reduced. After the size of the sampling neighborhood is reduced to be within a threshold, the algorithm restarts the global sampling procedure from the beginning.

We are now ready to describe the Smart Hill-Climbing algorithm in detail. The algorithm initially samples m points and picks the point with the best performance. Let $\vec{X}_{min}$ be the best performance point. We then sample a set of n points using the weighted Latin Hypercube Sampling around a pre-specified neighborhood of $\vec{X}_{min}$. Along each dimension, we find the best quadratic curve to fit all the points in the neighborhood. We obtain the minimum point for the fitted quadratic curve within the neighborhood range. Combining the minimum points along all the dimensions we obtain the next candidate sampling point. If this new point is better than the existing points, we then shift the center of the neighborhood

to be around this new point and repeat the local sampling. If this new point is worse than the existing best point, we then shrink the size of the searching neighborhood, and repeat the local sampling. The local sampling procedures stops when the searching neighborhood is smaller than a predefined threshold. This means that it is unlikely that there are any better points in the local range. The procedure then restarts. During the restart phase, the algorithm follows the local neighborhood search from a newly sampled point only if this point is better than a certain fraction, say 70%, of the existing points.

The algorithm is described as follows:

[Smart Hill-Climbing Algorithm]:

1. Initialize sampling parameter m, n, and l. Set the size for the local search neighborhood, threshold for neighborhood size and shrink factor α.

2. Take an initial sample of size m using weighted LHS. Find the point with best performance and set it to be the center of neighborhood for local search.

3. Generate n samples from the local neighborhood using the weighted LHS. Update the best configuration information.

4. For each dimension,
   - Collect the points within the local neighborhood and obtain the best fit quadratic curve.
   - Generate the minimal point according to the quadratic curve.
   - Combine the minimal points for all the dimensions to form the next candidate.

   (a) If the candidate point is better than all other points, update the center of the local search neighborhood. Go to step 3.

   (b) Else repeat fitting including the new sampled point, and generate an updated candidate sampling point.
      i. If the candidate point is better than all other points, update the center of the local search neighborhood. Go to step 3.
      ii. Else shrink the size for the local neighborhood.
         A. If size of local neighborhood larger than a threshold, go to step 3.
         B. Else go to step 5 to restart.

5. Restart: Take a set of weighted Latin Hypercube Sample of size l.

   (a) If the best point in the sample is better than a specified fraction of existing points, go to step 3.

   (b) Else repeat step 5.

The parameters in the initialization step should be chosen according to the budget on the total number of samples on can afford. In real applications, one can typically perform a couple of experiments per hour, which leads to a total of only tens or a couple of hundred experiments at most. Over the total experiments, one would like the algorithm to drill down the local neighborhood to a local optima a number of times. One could use around 5% of the total number of expected runs as the initial sample size m. The restart sample size l for the LHS can be the same or smaller. The neighborhood sampling size n can be a little smaller than m. The neighborhood for the local search can usually start from half of the original searching space, and then shrink at a rate around 75-85%. Once

the neighborhood size is smaller than 10% of the original searching range, then the algorithm stops the neighborhood search and restarts. This corresponds to $7 \sim 8$ times of consecutive shrinkage for a shrink factor $\alpha = 80\%$. These are very crude guidelines from our numerical experiences. In practice, one can perform a couple of experiments to get a initial feeling and refine these parameters for later experiments.

One of the major ingredients of the algorithm is that it takes advantage of the global trend and correlation information when generating samples from the global level. This is consistent with the intuition that the system performance may depend on certain parameters in a rough monotonic fashion on a global level. Another key advantage of the algorithm is that it estimates the local gradient properties and uses these estimates to guide the local search procedure. These two key properties allow our algorithm to quickly find the high quality solutions. This is demonstrated by the experimental results in Section 5.

4. NUMERICAL EXPERIMENTS

In this section, we carry out a series of numerical experiments to demonstrate the efficiency of the proposed Smart Hill-Climbing algorithm. We further compare the algorithm with other existing algorithms such as simulated annealing [14], and the random recursive search algorithm introduced in [18]. To better evaluate and understand the performance of different algorithms, we assume the black-box function has an explicit form. We use one of the standard benchmark functions, the Rastrigin function [13], which is defined as:

$$f(x) = N \cdot \beta + \sum_{i=1}^{N} (x_i^2 - \beta \cdot cos(2\pi x_i)). \qquad (4.1)$$

Figure 4 provides a 3D view of the Rastrigin function with $N = 2$ and $\beta = 0.8$ on the range $[-1, 1] \times [-1, 1]$. Note that this benchmark function has many local minima which makes it suitable for testing the performance of the algorithms. The global minima is at the origin $(0, 0)$.
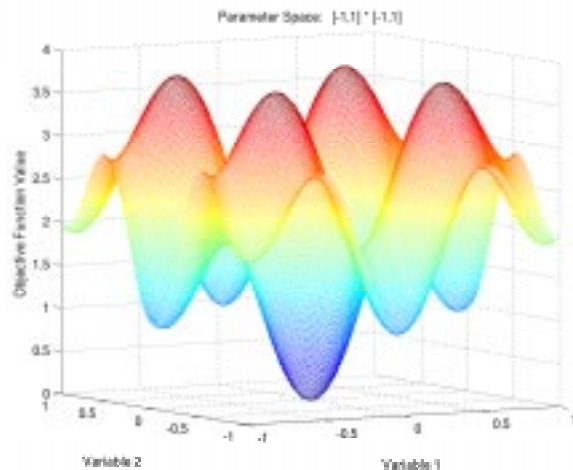

Figure 4: Two-dimensional Rastrigin Function

Figures 5, 6, and 7 provide the search results on the above Rastrigin surface for simulated annealing, random recursive search and our proposed Smart Hill-Climbing algorithms. Each algorithm carried out $1,000$ search steps. The parameter settings for simulated annealing and random recursive search are based on the guide-

lines provided in [14] and [18], respectively. For the Smart Hill-Climbing algorithm, we set $m = 4$, $n = 6$, $l = 4$, the neighborhood shrink factor is $\alpha = 5/6$. the stopping threshold on neighborhood search is 10% of the original searching space,
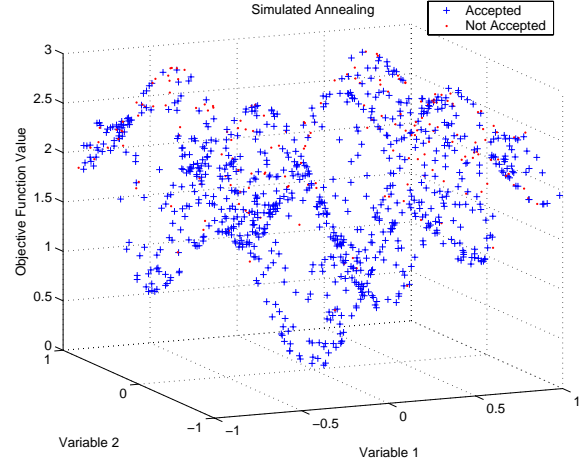

Figure 5: Simulated Annealing

Figure 5 shows that the simulated annealing algorithm explored the whole space quite evenly and exhaustively. The sampled points do have specific concentrations. This behavior is consistent with the random sampling nature of the simulated annealing algorithm. The random recursive search algorithm, as plotted in Figure 6, shows some improvement over the simulated annealing algorithm. It spends a number of the searches on the good neighborhood. However, the most of its searches are still exhaustive and uniformly across the whole space.
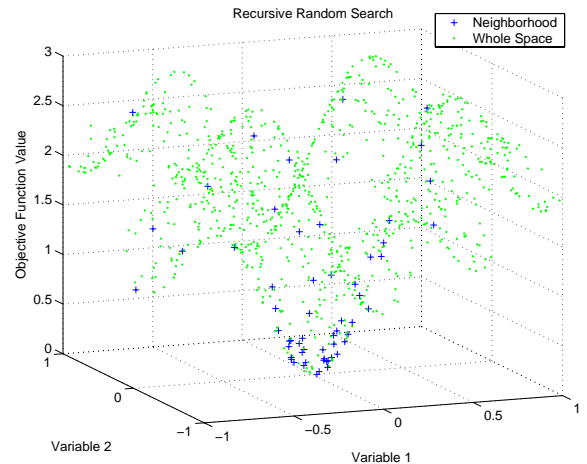

Figure 6: Random Recursive Search

The searches in Figure 7 under the Smart Hill-Climbing algorithm are much more guided. The weighted Latin Hyper Sampling strategy guides much more samples from the neighborhood close to the local or global minima. With a relative small set of initial samples, the algorithm soon learns the correlation structure, adjusts its importance sampling strategy. As shown in Figure 7, the algorithm quickly directs the samples to the first valley (local minimum). The sampling in the restart process also has focused slightly more on

important regions. Once the restart process reveals that there are better parameter regions (close to the origin), the algorithm quickly climbed (slipped) to the new valley (global minimum), as shown by the triangles in Figure 7. The improved performance is from the gradient following and correlation estimation features of the Smart Hill-Climbing algorithm.
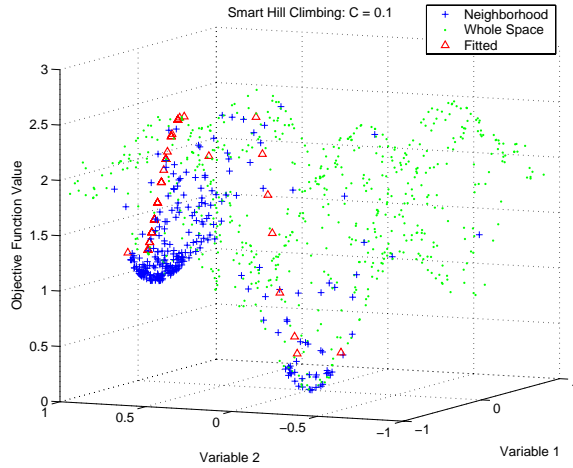

Figure 7: Smart Hill-Climbing with weighted LHS

The advantage of the Smart Hill-Climbing algorithm becomes even clearer when dealing with higher dimensional problems. Figure 8 shows the testing results on a 20-dimensional Rastrigin function with $N = 20$, $\beta = 88$, and the range $[-6, 6]$ for each dimension. Here each algorithm carried out $1,000$ search steps. The parameter settings for the Smart Hill-Climbing algorithm are: $m = 6$, $n = 8$, $l = 6$, the first local search starts with a neighborhood that is $1/2$ of the total searching space, and then shrinks at factor $\alpha = 5/6$, the threshold on neighborhood size is 10% of the original searching space. To take into account the stochastic nature of the search algorithms, tests are repeated for 50 times for each algorithm, and the average of the results are presented. Observe from Figure 8 that the Smart Hill-Climbing algorithm consistently outperforms the other two. The Smart Hill-Climbing algorithm quickly generates high quality samples in its initial set of searches, thus it has the potential to be highly efficient in achieving good performance in limited time frame. This makes the algorithm very promising in practice, especially when experiments are extremely expensive. which is demonstrated in the case study in the next section.

5. A CASE STUDY: OPTIMAL TUNING FOR WEBSPHERE

In this section, we examine the application of the Smart Hill-Climbing algorithm to the configuration of the IBM WebSphere Application Server [1]. We also compare the efficacy of the Smart Hill-Climbing Algorithms with previous search algorithms, such as simulated annealing and random recursive search.

5.1 Experiment Setup: System Environments

Our experimental testbed mimics a production system for an online brokerage application. The setup consists of client machines,
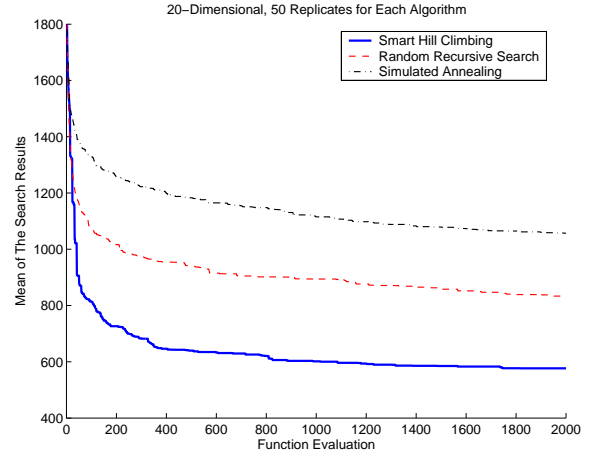
[1] http://www.ibm.com/software/info1/websphere

Figure 8: With 20 parameters

application server machines and database machines. Since we focus on application server configuration, we do not use an HTTP server in our experiments, but access the application server directly. All the servers involved are 1.8GHz Pentium III machines running Linux version 7.3, with 1GB of memory. The application server is IBM WebSphere Application Server (WAS) version 5.0.2. The database server is DB2 version 7.1. The machines are connected on a same local area network with 100Mbps Ethernet.

We use the Trade (version 3) application as a benchmarking application. Trade is an end-to-end web application modeled after an online brokerage. It leverages J2EE components such as servlets, JSPs, EJB, and JDBC to provide a set of user services such as login/logout, stock quotes, buy, sell, account details, etc., through standards-based HTTP and Web Service protocols. The Trade application accesses a database that contains stock and user account information. Users perform transactions on the database through servlets running on the application servers. The client machines generate a number of users who repeatedly generate random requests and transactions. There is a think-time distribution that each user uses to determine how long to wait between requests.

We conduct test runs and collect throughput and response time data for each system configuration. Each test typically lasts around fifteen minutes. Test runs have consistently shown that the system performance stabilizes fairly quickly. Fifteen minutes is sufficient to obtain good estimates for the system performance measures, such as throughput and response time.

As discussed earlier, our focus is on system parameters for the WebSphere Application Server. As illustrated in Figure 1, each component of an application server has configuration parameters that can affect its performance. We will focus, in our experiments, on four of these parameters:

- WebMax: The maximum number of threads in the Web Container thread pool. This parameter limits the concurrency of the Web Container.

- OrbThreadMax: The maximum number of threads in the EJB Container thread pool. This parameter limits the concurrency of the EJB Container.

- DSMax: The maximum number of connections to the database in the Data Source Connection Pool.

- HeapMax: The maximum size of the Java Virtual Machine (JVM) on which WebSphere runs (in Megabytes).

The goal of the experiments is to tune each parameter in coordination automatically so as to achieve minimize the average response time of users, while maintaining overall system stability. The parameter ranges used for tuning each parameter are listed in Table 1.

HeapMax	WebMax	OrbMax	DsMax
[256,768]	[1,120]	[1,120]	[1,120]

Table 1: Parameter Ranges for Configuration Parameters used in Experiments

5.2 Experiment Results

For the Trade benchmark, the objective function can be evaluated for a given parameter setting by executing test runs for 20 to 30 minutes. Given that it is expensive and time consuming for each function evaluation, in total we can only a couple of hundred experiments. Because of the large number of variables involved, our Smart Hill-Climbing algorithm can be readily applied. We have also implemented two other optimization algorithms, simulated annealing and recursive random search. Here we set the budget on the total number of search steps to be 120 for each algorithm. The parameter settings for simulated annealing and random recursive search are based on the guidelines provided in [14] and [18], respectively. For the Smart Hill-Climbing algorithm, we set $m = 3, n = 4, l = 3$. The neighborhood for the local search starts from half of the original searching space, and then shrinks at rate $\alpha = 2/3$. The stopping threshold on neighborhood search is 10% of the original searching space.

We compare the results of Smart Hill-Climbing with that of the other algorithms. Our experimental results show that by exploring the local structure of the objective function, Smart Hill-Climbing is the most efficient algorithm among the three black-box optimization algorithms.
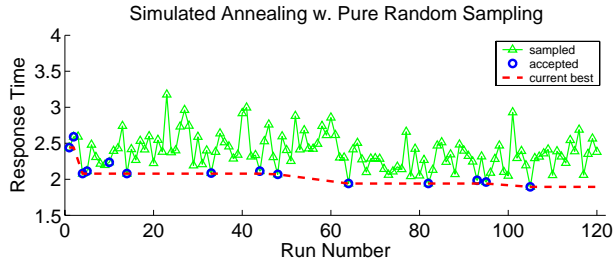

Figure 9: Simulated Annealing under Simple Random Sampling
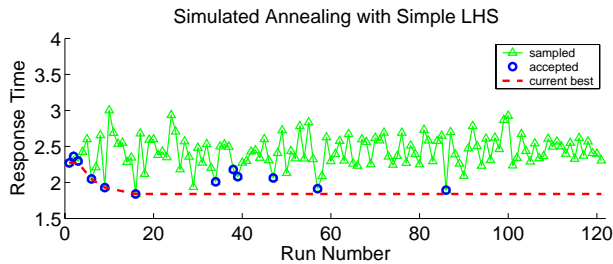

Figure 10: Simulated Annealing Combined with LHS

We present several sets of results. The first set illustrates the power for the Latin Hypercube Sampling procedure. We implement the three black-box algorithms with and without the Latin Hypercube Sampling. Figures 9 and 10 plot the simulated annealing algorithm with and without the LHS (both versions use the same temperature cooling schedule). We observe that the simulated annealing algorithm with LHS reached high quality solutions within only twenty samples. The simple simulated annealing algorithm, however, requires a significantly longer time and does not reach a comparable performance level even after one hundred samples. Although neither of the two simulated annealing experiments converge (over 120 samples), the LHS version of the algorithm generates more high quality samples. A more detailed look at the actual configuration parameter values reveal that it is easier to understand the interactions between configuration parameters and system performance with LHS sampling. In particular, the ordering structure in the parameter setting is revealed clearly by LHS.
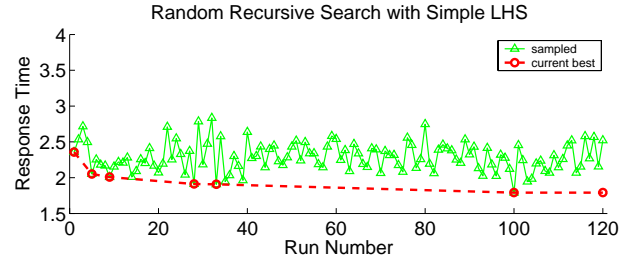

Figure 11: Recursive Random Search Combined with LHS

Figure 11 shows the response time results for the recursive random search combined with LHS. The algorithm took about 100 runs to reach a high quality sample. Although the performance is slightly better than that of simulated annealing with LHS sampling, it took much longer to achieve the result. With one sample path for each algorithm and no significant difference in performance, it is not conclusive whether the simulated annealing or the recursive random search is more efficient.
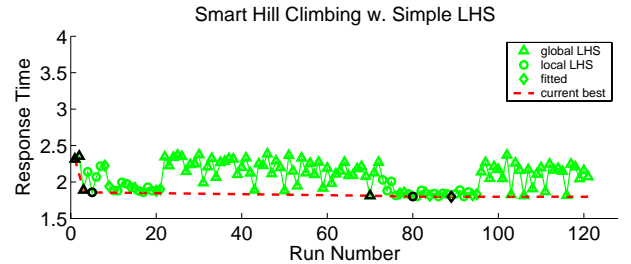

Figure 12: Smart Hill-Climbing with Simple LHS

For the Smart Hill-Climbing algorithm, Figure 12 depicts the sampling sequence for Smart Hill-Climbing with LHS sampling. The plot show the global sampling (or restart) phase and the local sampling phase during the whole process. We observe that the algorithm quickly generates high quality samples in the local sampling phases. The samples in the local searching phases are of consistently high quality. Furthermore, it took the algorithm less than 10 samples to obtain a near best sample. The sample path shows that there were 4 improvements in the experiment and one of them is the result of the fitting process. This confirms the power of our refined local search procedure.

To compare the performance of Smart Hill-Climbing using the Weighted LHS for sampling, we have performed an experiment that applies weighted LHS in the restart step. The samples selected by the algorithm are shown in Figure 13. Although the initial global sampling is slightly worse than that of Smart Hill-Climbing with LHS, there are 9 improvements during the whole run; the restart global searching process generates more high quality samples. Furthermore, the final result of the search is better than that of Smart Hill-Climbing with LHS. This experiment validates the effectiveness of the weighted LHS combined with the Smart Hill-Climbing.
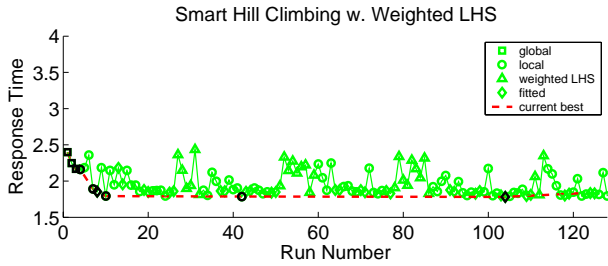

Figure 13: Smart Hill-Climbing with weighted LHS

We have also plotted the response time vs. WebMax in Figure 14. This plot shows a smooth curve which describes the general correlation between the response time and the WebMax parameter. The response times can vary for the same WebMax value because the values of other parameters could be different. This plot not only shows a global trend for the WebMax parameter, the relative smoothness of the curve provides strong support for using the quadratic approximations. Clearly, the Smart Hill-Climbing quickly discovered the best parameter region for WebMax regardless of other parameters. This proves the power of the weighted LHS procedure to adapt quickly to correlation properties.
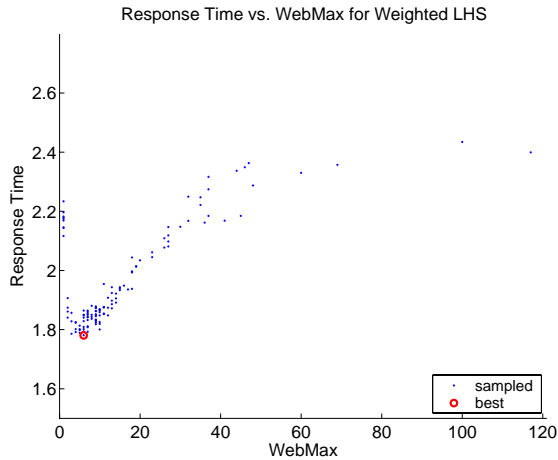

Figure 14: Response Time vs. WebMax in Smart Hill-Climbing with Weighted LHS

5.3 Further Insights

Based on our experiments, we perform further statistical analysis to understand the underlying properties of a good configuration for Trade on WebSphere. Overall, the following rules of thumb can be used as guidelines:

Correlation Structure: Table 2 shows the correlation with response times for each of the four tuning parameters. *WebMax* is the one that has most impact on performance. Besides WebMax, the other three parameters have relatively mild correlation with response times. The negative correlation of HeapMax with response times indicate that larger Heap Size is preferred in order to have better performance.

HeapMax	WebMax	OrbMax	DsMax
-0.13	0.60	0.18	0.15

Table 2: Correlation with Response Time

WebMax should be small: Given that WebMax has strong positive correlation with response times, clearly WebMax should be small. However setting WebMax (the maximum number of threads for WebContainer) too small may not be beneficial, since time-sharing and parallel processing do have performance advantages. In fact, the optimal value of WebMax is around 8-10. Observe from Figure 14 that smaller WebMax values are sampled more frequently than high values, indicating the algorithm has learned from the correlation structure and directed its importance sampling strategy accordingly.

Order matters: In the current experiment setting, most trading transactions go through the complete sequence as specified in Figure 1. That is, most transactions visit the Web container, EJB container, Data source, and Database sequentially. Note that a thread at the Web container is only released until the required information returns from the EJB container, and the same is true, for threads in the EJB container. Therefore, in order to avoid blocking, the number of threads at any downstream station should be always larger than its upper stream. In other words, the three parameters controlling number of threads at each layer should be in the increasing order as follows:

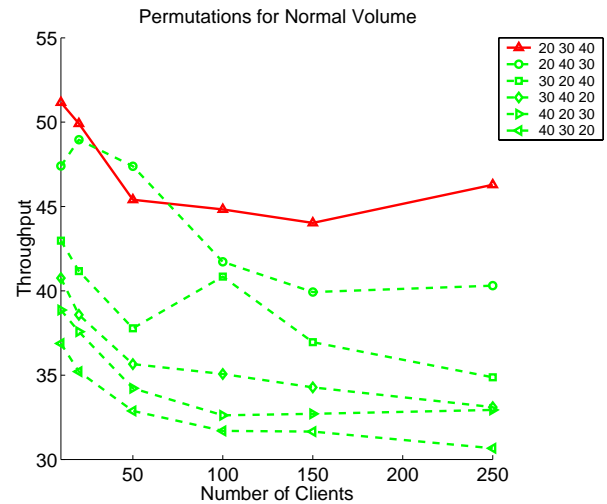$$\text{WebMax} < \text{OrbMax} < \text{DsMax}.$$

Figure 15: Permutations of WebMax OrbMax DsMax under Normal Volume Trade

This observation has been further confirmed by many of our experiment results. Figure 15 gives one such example where the system throughput is plotted as a function of the number of concurrent

clients, six different permutations of $(20, 30, 40)$ for configuring the above three parameters (WebMax, OrbMax, DsMax) are experimented. Observe that the increasing order outperforms almost always outperforms the other permutations.

6. CONCLUSION

We have studied the problem of optimal system configuration for Web application servers. We formulate the problem of finding an optimal configuration for a given set of applications as a black-box optimization problem. We then proposed a Smart Hill-Climbing algorithm using ideas of importance sampling and Latin Hypercube Sampling. The algorithm is efficient in both searching and random sampling. It consists of estimating a local function, and then, Hill-Climbing in the steepest descent direction. The algorithm also learns from past searches and restarts in a smart and selective fashion using the idea of importance sampling. We have carried out extensive experiments with an on-line brokerage application running in a WebSphere environment. Empirical results demonstrate that our algorithm is efficient and superior than traditional heuristic methods. Further insights and rules of thumb for the optimal configuration are also discussed.

Further research directions include ways of improving the proposed method through the use of analytical models. The idea here is to consider various components of the system and build parametric/analytic models of such components. We can then derive structural/qualitative properties of these components in order for the higher level optimization algorithm to reduce the search space. Another direction of research would be the development of stochastic optimization techniques for such complex systems, for example, Markov decision processes. The challenge here is to deal with the size and dimension of the problem. All these issues are the subjects of our on-going work.

7. REFERENCES

[1] Boyan, J. and Moore, A. (2000). Learning evaluation functions to improve optimization by local search. *Journal of Machine Learning Research*, 1:77-112.

[2] Diao, Y., Gandhi, N., Hellerstein, J. L., Parekh, S. and Tilbury, D. (2002). Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server. *Proceedings of the Network Operations and Management Symposium 2002*.

[3] Helton, J. C. (1993). Uncertainty and sensitivity analysis techniques for use in performance assessment for radioactive waste disposal. *Reliability Engineering and System Safety*, Vol. 42, no. 2-3, 327-367.

[4] Geman, S. and Geman, D. (1984). Stochastic relaxation, Gibbs distributions and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:721–742.

[5] Glover, F. and Laguna, M. (1993). Tabu search. *Modern Heuristic Techniques for Combinatorial Problems*. Scientific Publications, Oxford.

[6] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, Mass.

[7] Kirkpatrick, S., Gelatt, C. D. and Vecchi, M. P. (1983). Optimization by Simulated Annealing, *Science*, 220, 671-680

[8] Liu, T. K., Behroozi, A. and Kumaran, S. (2003). A Performance Model for a Business Process Integration Middleware. *IEEE International Conference on E-Commerce (CEC 2003)*.

[9] Liu, X., Sha, L., Diao, Y., Froehlich, S., Hellerstein, J. L. and Parekh, S. (2003). Online Response Time Optimization of Apache Web Server. IWQoS 2003: 461-478

[10] McKay, M. D., Conover, W. J. and Beckman, R. J. (1979). A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code, *Technometrics*, 21, 239-245

[11] Menascé, D. A., Almeida, V. A. F., Fonseca, R. and Mendes, M. A. Business-oriented resource management policies for e-commerce servers. *Performance Evaluation*, 42:223–239, 2000.

[12] Raghavachari, M., Reimer, D. and Johnson, R. D. The Deployer's Problem: Configuring Application Servers for Performance and Reliability, *ICSE 2003*, Portland, OR.

[13] Muhlenbein, H.,Schomisch, M. and Born, J. (1991). The parallel genetic algorithm as function optimizer. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth Intl. Conf. on Genetic Algorithms*, pages 271-278, Morgan-Kaufman.

[14] Romeijn, H. E. and Smith, R. L. (1994). Simulated Annealing and Adaptive Search in Global Optimization, *Probability in the Engineering and Informational Sciences*, 8, 571-590.

[15] Russell, S., and Norvig, P. (1995) *Artificial Intelligence: A Modern Approach*. Prentice Hall.

[16] Sha, L., Liu, X., Lu, Y. and Abdelzaher, T. (2002). Queueing Model Based Network Server Performance Control. *IEEE Real-Time Systems Symposium 2002*, 81-90.

[17] Törn, A. and Žilinskas, A. (1989). *Global Optimization*, vol. 350 of *Lecture Notes on Computer Science*. Springer-Verlag.

[18] Ye, T., and Kalyanaraman, S. (2003). A Recursive Random Search Algorithm for Large-Scale Network Parameter Configuration, *SIGMETRICS 2003*, San Diego, California.

[19] Ye, T., Kaur, H. T., and Kalyanaraman, S. (2002). Large-scale network parameter configuration using on-line simulation. Technical report, ECSE Department, Rensselaer Polytechnic Institute, 2002.

[20] Zabinsky, Z. B., Smith, R. L., McDonald, J. F., Romeijn, H. E., and Kaufman, D. E. (1993). Improving Hit and Run for Global Optimization, *Journal of Global Optimization*, 3:171-192.