Autotuning Configurations in Distributed Systems for Performance Improvements using Evolutionary Strategies

Anooshiravan Saboori
Department of Electrical and Computer Engineering
University of Illinois at Urbana Champaign, IL 61801
saboori2@uiuc.edu

Guofei Jiang and Haifeng Chen
NEC Laboratories America, Princeton, NJ 08540
{gfj,haifeng}@nec-labs.com

Abstract

Distributed systems usually have many configurable parameters such as those included in common configuration files. Performance of distributed systems is partially dependent on these system configurations. While operators may choose default settings or manually tune parameters based on their experience and intuition, the resulted settings may not be the optimal one for specific services running on the distributed system. In this paper, we formulate the problem of autotuning configurations as a black-box optimization problem. This problem becomes quite challenging since the joint parameter search space is huge and also no explicit relationship between performance and configurations exists. We propose to use a well known evolutionary algorithm called Covariance Matrix Adaptation (CMA) to automatically tune system parameters. We compare CMA algorithm to another existing techniques called Smart Hill Climbing (SHC) and demonstrate that CMA algorithm outperforms SHC algorithm both on synthetic data and in a real system.

1 Introduction

With the growth of Internet services, large information systems are built to support millions of user transactions every day. Most of such systems are implemented with a distributed architecture where an application software runs on multiple machines. A three-tier web architecture is a typical example of such distributed systems that consists of a web server which acts as an interface to present data to the client's browser, an application server which supports bulk of business logics and finally a database server which are for persistent data storage. The performance of such three-tier web applications is usually characterized by metrics such as throughput and response time. System throughput measures the number of completed requests per time unit while response time is the time taken to complete a request. If a system is not saturated with user requests, its throughput increases as the volume of requests grows. After certain volume of user requests (per time unit), the system becomes saturated and its throughput could not increase anymore. We denote the range where throughput increases linearly with the number of user requests by *linear zone* and the range where throughput is not increasing proportionally by *non-linear zone*. For example, the range $[A, B]$ shown in Figure 1 is the linear zone while the point C is located in the non-linear zone. We refer to the point at which the linear zone switches to the nonlinear zone as the *saturation* point. In Figure 1, the point B is the saturation point of the system performance. The non-linear zone is not desirable since a system operating in this zone usually produces poor or undesirable performance. Cox et.al. [2] refers to this zone-switching phenomena as *"falling of the cliff"* because the downside of this behavior might be so serious that results in a partial or complete system collapse. There are various reasons for such non-linear behavior: exhaustion of resources such as CPU, memory and network connections and etc. [2].

Distributed systems usually have many configurable parameters and performance of such systems is partially dependent on these configurations. In this paper, by changing system configurations, we attempt to improve the throughput without increasing the response time. This amounts to a shift of the saturation point of the system, e.g., the point B can be shifted toward the top-right corner in Figure 1. In this way, we can extend the linear zone without adding hardware and software resources. This is very attractive to

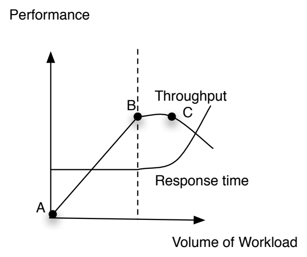

Figure 1. System performance at saturation.

service providers because they can use the same infrastructure to generate more revenues by actively tuning system configurations.

To this end, we propose a heuristic method to automatically search for configurations that maximize the throughput while keeping the response time below a threshold in Service Level Agreements (SLA). To illustrate how system performance changes with configurations, we depict the response time and throughput at the saturation point of a three-tier web system in Figure 2. When two parameters in the application server changes over a small range, in fact the system performance is very sensitive to such changes. For example, system performance has a strong dependency on the parameter of $KeepAliveTimeOut$ from the application server. As this timeout parameter increases, TCP sessions take longer time to expire. Therefore, connected users can reuse the existing sessions to reduce response time. Conversely, if $KeepAliveTimeOut$ decreases, more users can get connected to the server and then throughput increases. But due to the overhead of making new TCP connections, the response time will increase accordingly. To capture this dependency, we use *multivariate functions* $t(x)$ and $r(x)$ to represent throughput and response time at the saturation point respectively. Here x is a vector that represents the joint setting of configurable parameter space. Figure 2 demonstrates the projection of t and r on the space spanned by two parameters from an application server, where $x = [KeepAliveTimeOut, MaximumPoolSize]$.

As discussed earlier, we attempt to search system configuration x that can maximize $t(x)$ while keeping $r(x) < \tau^0$. τ^0 is the response time threshold specified in SLA. There are many challenges in this optimization problem: (i) the aforementioned functions that describe the relationship between configuration parameters and the performance metrics is unknown. From our experimental results and Figure 2, we notice that these functions are non-convex and multi-modal. Moreover, they are not separable, i.e., optimizing the function along each variable space separately does not imply optimizing the function in the whole search space; (ii) the combinational search space of parameters is huge. For example, the main configuration file of Apache web server has more than 240 uncommented lines of configurable parameters [9] and each parameter has some acceptable range; (iii) a typical trial run under a given setting can take as many as 25 minutes, which include the time to reboot software and generate sufficient workload volume to saturate the system. Therefore, given a single x, it is already quite time-consuming to evaluate its $t(x)$ and $r(x)$.

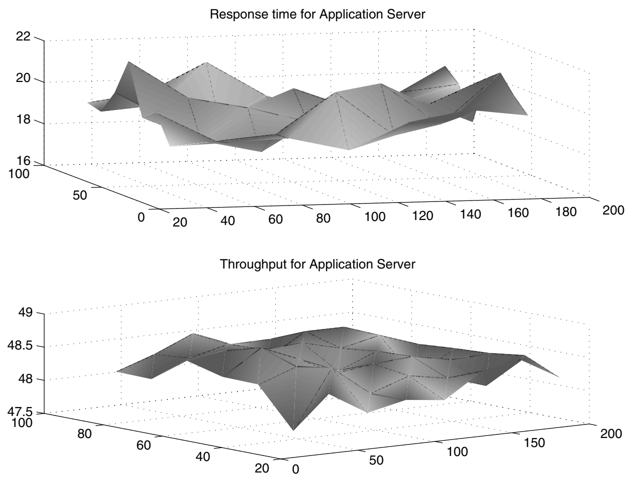

Figure 2. Response time and throughput at saturation point as a function of configuration parameters $KeepAliveTime$ and $MaximumPoolSize$ in Application server.

As a result of these challenges, we formulate the autotuning of configuration parameters as a black box optimization problem. Since each trial of function evaluation is expensive, we impose an upper limit on the allowed number of trials (e.g., 100 in our experiments). Now given a small number of trials and the complexity of the functions $t(x)$ and $r(x)$, we believe that it is not practical to learn these functions from trials and then optimize these approximated functions. This opts out many function approximation algorithms such as neural network but suggests heuristic search methods, which perform guided search across the parameter space in an iterative manner. At the first iteration, a set of candidate solutions is generated based on a probability distribution. After that, the heuristic algorithm calculates the objective function (the function to be optimized) of those candidates. Good solutions in the current generation are used to sample the next generation and this process continues until the termination criterion is met. The performance of such algorithms are affected by the number of candidate solution in each generation (population size) and the number of iterations. The multiplication of these two factors equals to the total number of trials, which determines the time of autotuning process.

Due to limited number of possible function evaluations in configuration auto-tuning problem, we have to reduce either the population size and/or the number of iterations. If we choose a big population size with a small number of iterations, heuristic search algorithms could not observe enough patterns to guide the following search and start behaving randomly. As a result, in our auto-tuning method, we decide to have more iterations as opposed to bigger population size. However, with a small population size, population diversity can vanish fast and lead to unreliable estimation of search areas. In order to keep the diversity of populations, we use the information obtained from the whole search history rather than one generation to evaluate search areas. Meantime, in this paper, the dependency of parameters is always tracked and statistically summarized across explored search space and along search process.

One class of heuristic methods, evolutionary strategies [4], use the historical statistics of the search space and also track dependencies among parameters in their search process. Covariance Matrix Adaptation algorithm (CMA algorithm) [4] is a sub-category of evolutionary strategies which uses a multivariate Gaussian distribution to summarize the statistics of good solutions and only the parameters of Gaussian distribution are updated at each step. In a recent comparison [6], it is shown that CMA algorithm performs better than other algorithms in this class. Therefore we propose to use CMA algorithm in our black box optimization problem.

To emphasize the importance of tracking the dependency of configurable parameters, we compare CMA algorithm with another heuristic algorithm named Smart Hill Climbing (SHC) [8]. SHC algorithm is the only heuristic algorithm that is proposed in literature for configuration tuning. This algorithm is based on hill-climbing which exploits the statistics of search space but does not consider the dependency among parameters. CMA algorithm outperforms SHC algorithm on both synthetic data and in a real system. This result confirms our intuition that any algorithm that performs well in this black-box optimization should exploit the statistics of search history and also track dependency among parameters simultaneously.

2 Related Work

We have not noticed much previous work on auto-tuning configurations to improve distributed system performance. In model-based approaches, a system model is assumed (e.g., a second order system or a queuing network) and mainly feedback control is used to achieve the desired performance [3, 7]. Since auto-tuning system configuration is a black box optimization problem, here we do not discuss these model-based approaches.

Zheng et.al. [9] proposes a search procedure which is accompanied with a parameter dependency graph derived through some experiments. They follow the dependency graph and use simplex algorithm to search best parameters. Xi et.al. [8] suggest a Smart Hill-Climbing algorithm for auto-tuning configurations in an application server which learns from past searches and uses correlation factor among performance changes to summarize dependency information. Chung et.al. [1] automatically tune system configurations in real time to adapt to changing workloads in a three-tier web system. For different type of workloads, they propose a simplex algorithm to tune system parameters. They also consider partitioning of parameter search space to improve the speed of simplex algorithm.

Our work is quite different from all of the above work. Our performance metric is the throughput and response time measured at the saturation point of distributed systems because our motivation is to extend the linear zone of system operations. Other methods use averaged response time and throughput as performance metrics to tune system parameters. We believe that our formulation is more reasonable and close to SLA management practice, i.e., we intend to increase system throughput while keeping response time below SLA threshold. In terms of the algorithms used for solving this black-box optimization problem, we take the first effort to introduce evolutionary strategies into auto-tuning system configurations. As discussed earlier, evolutionary strategies can efficiently summarize search space as well as the dependency among system parameters. Therefore, given the complexity of functions like $t(x)$ and $r(x)$ and the limited number of possible trials, such approaches are more practical for real problems.

3 Covariance Matrix Adaptation Algorithm

In this paper, we are interested in autotuning configurations of a three-tier web system to increase its linear zone, i.e., the area between point A and point B in Figure 1. We can formally formulate this problem as the following optimization problem:

$$\max_{s.t.\ r(x)<\tau^0} t(x) \tag{1}$$

where $t(x)$ and $r(x)$ are the throughput and response time evaluated at the saturation point. τ^0 denotes the threshold of response time specified in SLA and we denote the default setting by x^0. Meantime, we assume that we can always generate enough workloads (user requests) to saturate the system.

As mentioned in Section 1, both $t(x)$ and $r(x)$ are multimodal, non-convex and non-separable functions and it also takes time to evaluate these functions. Therefore heuristic algorithms should utilize the information from previous search results to infer the relationship between the variables and the location of potential optima. Evolutionary strategy algorithms use a probability distribution to represent this relationship and guide the search in the following steps:

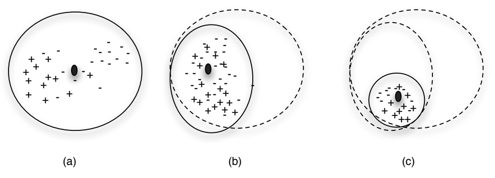

Figure 3. Two iterations of an evolutionary strategy algorithm.

(1) Initialize distribution parameters θ^0.
(2) Repeat for generations $g = 0, 1, \dots$ until termination criterion is met:
(i) Sample λ points from distribution $P(x|\theta^g) \rightarrow x_1, \dots, x_\lambda$.
(ii) Evaluate and select $\mu(\le \lambda)$ of the samples according to their objective function f.
(iii) Use the selected samples to update the parameters $\theta^{g+1} = update(\theta^g, x_1, f(x_1), \dots, x_\mu, f(x_\mu))$.

Figure 3 illustrates the search process of an evolutionary strategy algorithm. Among evolutionary strategy algorithms, CMA algorithm has been proved to perform well on non-separable functions even with small population size [6]. Before introducing CMA algorithm formally, we need to define $P(x|\theta^g)$, θ as well as the $update$ function shown in the above algorithm. In CMA algorithm, a multivariate Gaussian distribution is used to represent the probability distribution $P(x|\theta^g)$ in sampling. The choice of Gaussian distribution leads to two benefits: i) for a given mean and covariance, this distribution generates data points with highest entropy, i.e., data points sampled from this distribution cover more part of the search space compared to other distributions, and (ii) the dependencies among parameters are well profiled in the covariance matrix. A multivariate Gaussian distribution can be precisely described by its mean vector m and covariance matrix C. Therefore the parameter updated in CMA algorithm is $\theta = (m, C)$ where the mean vector and covariance matrix describe the centroid and distribution of candidate population respectively.

The function $update$ updates the distribution parameter $\theta = (m, C)$ according to the search history. It is challenging to update C since reliable covariance matrix estimation requires large number of samples. To achieve satisfactory performance, the update function $update$ keeps tracking consecutive steps and exploits the hidden correlation between them. The changes of the centroid of population is tracked with an evolution path along the sequence of successive generations [4]. Meantime, two evolution paths p_σ and p_C are used in updating the covariance matrix C. Step size evolution path p_σ is used to control the step size σ (to be described shortly). Note that the covariance matrix C is a quadratic function of the centroid changes so that it cannot capture the sign of the centroid changes; hence, covariance evolution path p_C summarizes the *sign* of centroid changes between consecutive steps when updating the covariance matrix C. Both vectors p_C and p_σ are n-dimensional vectors with distribution $\mathcal{N}(0, C)$ and are updated with an exponential smoothing technique so that the importance of previous search information decays exponentially as time goes on.

Step size control is very important for heuristic algorithms that can guide search across rugged and smoothed spaces. A large step size implies that data points are sampled at a coarse granularity. Given a fixed number of trials, a larger step size could enable the algorithm to explore a bigger part of search space but each area will not be carefully examined. Conversely, a small step size results in sampling search space at a fine granularity but it will take many trials to cover the search space. Therefore, with step size control, we can balance the amount of efforts taken in exploration and exploitation. As discussed earlier, the step size evolution path p_σ is used to control the step size σ. If p_σ is short, single search steps are moving against each other, which indicates rugged area and more exploitation is required. As a result, the step size σ is decreased. Conversely, if p_σ is long, search steps are correlated and moving on a smooth surface so that we can take longer steps with increased σ. The value of $\|p_\sigma\|$ is compared with a random selection of $E\|\mathcal{N}(0, I)\|$ to determine whether p_σ is short or long. Here $\|.\|$ denotes the Euclidean norm.

We describe each of these updates in the following algorithm, where the objective function is denoted by f.

$(\mu, \lambda)-$**CMA algorithm**:
(1) Initialize $g = 0$ and $C^0 = I$; Initialize m^0 and σ^0 based on the problem at hand.
(2) Choose λ, μ, w_i, c_σ, c_C, c_{cov}, μ_{cov} and d_σ (see their definition in Table 1).
(3) Generate the g^{th} generation $\{x_1^g, \dots, x_\lambda^g\}$ as λ candidates are sampled from the distribution $\mathcal{N}(m^g, (\sigma^g)^2 C^g)$.
(4) Evaluate the objective function f on these sampled points and rank them accordingly. Denote the k^{th} best candidate by z_k^g.
(5) Update the mean vector according to $m^{g+1} = \sum_{k=1}^{\mu} w_i z_k^g$ where $\sum_{i=1}^{\mu} w_i = 1, w_i > 0$. Hence, the new mean is a weighted average of μ selected points.
(6) Update the covariance evolution path according to $p_C^{g+1} = (1 - c_C)p_C^g + \sqrt{(\mu_{eff})c_C(2 - c_C)}\left(\frac{m^{g+1} - m^g}{\sigma^g}\right)$, with $p_C^0 = 0_{n\times 1}$. The first term amounts to an exponential smoothing of the historical information which decays according to c_C. The second term is the normalized value of the covariance of the current generation, assuming that its mean is same with its previous generation's mean.
(7) Update the step size evolution path according to $p_\sigma^{g+1} = (1 - c_\sigma)p_\sigma^g + \sqrt{(\mu_{eff})c_\sigma(2 - c_\sigma)}\left(C^{(g)}\right)^{-\frac{1}{2}}\left(\frac{m^{g+1} - m^g}{\sigma^g}\right)$ with $p_\sigma^0 = 0_{n\times 1}$.

Table 1. Description of tunable parameters in CMA algorithm.

Parameter	Description
λ	Population size of each generation
μ	Number of selected candidates from each generation
w_i	Weights used in evaluating the mean of the current generation
c_σ	Tunes the effect of previous step size evolution path p_σ^g on current step size evolution path p_σ^{g+1}
c_C	Tunes the effect of previous covariance evolution path p_C^g on current covariance evolution path p_C^{g+1}
c_{cov}	Tunes the effect of previous covariance matrix C^g on current covariance matrix C^{g+1}
μ_{cov}	Tunes the effect of current generation statistics on current covariance matrix
d_σ	Tunes the effect of current covariance evolution path p_σ^{g+1} on the step size σ^{g+1}

(8) Update the covariance matrix according to $C^{g+1} = (1 - c_{cov})C^g + \frac{c_{cov}}{\mu_{cov}}\ (p_C^{g+1}(p_C^{g+1})^T) + c_{cov}(1 - \frac{1}{\mu_{cov}}) \times \sum_{i=1}^{\mu} w_i(\frac{z_i^g - m^g}{\sigma^g})(\frac{z_i^g - m^g}{\sigma^g})^T$. Covariance matrix update uses three pieces of information: i) information from the previous generation (the first term) which decays exponentially according to c_{cov}, ii) information from the covariance evolution path (the second term), iii) information from the current generation (the third term).

(9) Update the step size according to $\sigma^{g+1} = \sigma^g exp(\frac{c_\sigma}{d_\sigma}(\frac{\|p_\sigma^{g+1}\|}{E\|\mathcal{N}(0,I)\|} - 1))$.

In this paper, we compare the performance of CMA algorithm with SHC algorithm, which also uses the statistics from candidates to guide its space sampling [8]. The main difference between them is the distribution used for space sampling. SHC algorithm uses an (univariate) exponential distribution for random sampling in the range of each parameter. Note that exponential distribution is characterized via a *rate* parameter. Formally, in SHC algorithm, for each parameter dimension i, $i \in \{1, \dots, n\}$, there is a truncated exponential density function of the form $\rho_i(x_i) = \frac{a_i}{e^{-a_i x_i^I} - e^{-a_i x_i^F}} e^{-a_i x_i}$ on the sampling range $x_i \in [x_i^I, x_i^F]$ where the rate parameter a_i is defined as $a_i = c_i \frac{std(f)}{std(x_i)}$. Here c_i denotes the correlation between the objective function f and the single parameter x_i while $std(.)$ represents standard deviation. The joint distribution of sampling space at each generation is summarized as $\rho(x_1, \dots, x_n) = \prod_{i=1}^{i=n} \rho_i(x_i)$. This means that all parameters are assumed to be independent so that SHC algorithm is only suitable for separable functions. At each iteration, the rate parameters a_i are updated using $c_i \frac{std(f)}{std(x_i)}$. Note that $\frac{std(f)}{std(x_i)}$ describes the performance improvement rate along the single parameter space of x_i and a larger value always leads to a bigger a_i. Therefore, the areas with better performance (i.e., with larger $\frac{std(f)}{std(x_i)}$) are sampled with higher probability due to the larger value of a_i, which resembles the basic idea of *importance sampling*.

The initialization phase of this algorithm consists of a global sampling using Latin Hypercube Sampling (LHS) method [5]. In order to sample m points in an n dimensional search space, each dimension is divided to m equal probable intervals and no two samples are chosen from the same interval of any dimensions. After the global sampling, a local search is conducted by first fitting a quadratic function into the sampled points and then hill-climbing in the steepest direction. At the minimum point of the fitted quadratic function, a new random global sampling with a shrunk window size is carried out. This window shrinking and curve fitting process continues until no performance improvement is achieved or the size of window falls below a pre-specified threshold. At this point, SHC algorithm restarts another round of global search. The new sampled points are only accepted if they achieve at least 80% of the best performance obtained so far. The algorithm terminates after the maximum number of function evaluations is reached.

Compared to CMA algorithm, an implicit assumption of SHC algorithm is that the function optimization is separable along different parameter spaces. This assumption does not hold for many optimization problems including the performance tuning problem defined in this paper. In next sections, with experiments on synthetic data and in a real system, we show that CMA algorithm outperforms SHC algorithm where the separability of objective functions does not hold.

4 Experiments with Synthetic Data

In this section, we conduct a series of numerical experiments to demonstrate the efficiency of CMA algorithm on both separable and non-separable functions. These numerical experiments are also used to highlight the difference between CMA and SHC algorithms. We use both separable Rastrigin function and non-separable Rosenbrock function as the objective functions in our experiments. These functions are often used as benchmark problems to eval-

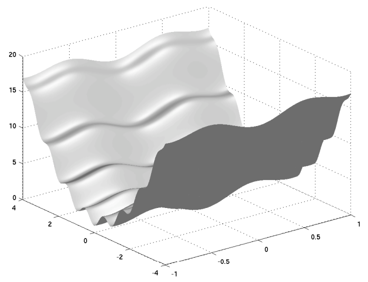

Figure 4. Rastrigin function.

Table 2. SHC algorithm vs. CMA algorithm on Rastrigin function.

Algorithm	The best f(x)	Average f(x)	σ^0
SHC algorithm	26.5832	63.6109	6
CMA algorithm	17.1745	33.6176	6
SHC algorithm	26.5832	63.6109	12
CMA algorithm	23.6825	105.3330	12

uate various heuristic optimization algorithms and Rastrigin function was also used in testing SHC algorithm. At first, a separable Rastrigin function is defined as $f(x) = 20A + \sum_{i=1}^{20}\big(x_i^2 - Acos(2\pi x_i)\big)$. Here $x_i, i \in \{1,\dots,20\}$, varies between -6 and 6, and A is a constant chosen to be 0.8. Both SHC and CMA algorithms try to minimize this objective function. The global minimum is at $x_i = 0$, $i \in \{1,\dots,20\}$ with $f(x) = 0$. Figure 4 depicts this separable function.

A $(4,9)-$CMA algorithm (i.e., $\mu = 4$ and $\lambda = 9$) is initiated with $m^0 = 3(0.5 - rand(20,1))$ and $\sigma^0 = 6$. In this way, the initial data points are sampled uniformly from half of the search space. SHC algorithm does not require initialization since it uses LHS to sample the whole space. Due to the randomness embedded in these algorithms, we try 1000 runs of each algorithm and record the best point for each run. Further we find the minimum and average of these 1000 points as two indicators to compare the performance of two algorithms. Table 2 shows the results for the separable Rastrigin function. It is straightforward to see that CMA algorithm performs better than SHC algorithm.

Now we investigate how the performance of CMA and SHC algorithms is affected by the uncertainty of initial settings. For this, we increase the initial uncertainty about the potential optimum point from half of the search space ($\sigma^0 = 6$) to the whole search space ($\sigma^0 = 12$). This implies that there is no hint about the location of the optimum. As shown in Table 2, SHC algorithm results in better average performance for this case. This is due to the fact that CMA algorithm does not have the same global coverage guarantee as the LHS method of SHC algorithm does.

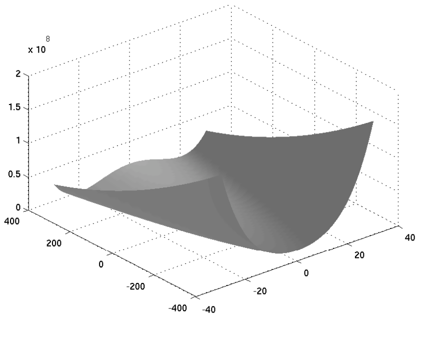

Figure 5. Rosenbrock function.

Table 3. SHC algorithm vs. CMA algorithm on Rosenbrock function.

Algorithm	The best f(x)	Average f(x)
CMA algorithm	647.1207	6.4795e+03
SHC algorithm	3.3636e+03	3.4406e+04

Next we compare the performance of CMA and SHC algorithms on the non-separable Rosenbrock function. This function is defined as $f(x) = \sum_{i=1}^{19}\big(100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2\big)$, where x_i, $i \in \{1,\dots,20\}$, varies between -6 and 6. The global minimum is located at $x_i = 1$, $i \in \{1,\dots,20\}$ with $f(x) = 0$. Figure 5 depicts this non-separable Rosenbrock function. The experiment has the similar setup as the previous one with $\sigma^0 = 6$. As illustrated in Table 3, the result from CMA algorithm is 5 times better than that of SHC algorithm. This is because SHC algorithm assumes the separability of the objective function and Rosenbrock function is not separable.

In summary, CMA algorithm performs much better than SHC for objective functions that are not separable. On the other hand, CMA algorithm does not have the gradient based search mechanism as well as the global sampling mechanism of SHC algorithm. However, given a rough guidance on initial points, CMA algorithm is able to outperform SHC algorithm on separable functions. This is appealing to us since the default system setting is already a good starting point. Also the objective functions in our configuration auto-tuning problem is not separable; hence, we propose that CMA algorithm is a better solution for this problem and prove this in the next section, by testing these two algorithms on a real system.

5 Experiments with Real Systems

In this section, we utilize CMA algorithm to autotune a three-tier web system, which includes an Apache web server, a JBoss application server and a MySQL database server. Figure 6 illustrates the architecture of our experimental system as well as its components. The application software running on this system is Pet store which is an application written by Sun Microsystems to demonstrate how to use J2EE platform in developing e-commerce applications. Just like other web services, users can visit the Pet store website to buy a variety of pets. We develop a client emulator to generate a large class of different user scenarios and workload patterns. Certain randomness of user behaviors is also considered in the emulated workloads. For example, a user action is randomly selected from all possible user scenarios that could follow the previous user action.

In our experiments, we randomly choose two parameters from each tier as $x =$

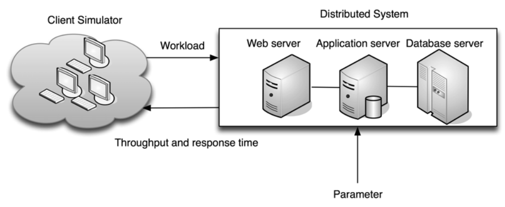

Figure 6. The experimental system.

$(KeepAliveTimeout, MaxClients, KeepAliveTime, MaxPoolSize, MaxConnection, KeyBufferSize)^T$. We generate sufficient amount of workloads with as many as 105 users to saturate the three-tier system. With the default setting $x^0 = [15, 50, 10000, 8, 60, 8388600]^T$, we observe that system throughput and response time at the saturation point are $t(x^0) = 257.5 \quad (req/s)$ and $\tau^0 = r(x^0) = 180.1(ms)$.

Parameter space $[10, 255] \times [10, 200] \times [5000, 30000] \times [5, 50] \times [30, 200] \times [3M, 300]$ is linearly quantized into the interval [0,144]. Then a configuration solution is a vector of six elements with their values from [0,144]. We initialize a $(4, 9)-$CMA algorithm with $\sigma^0 = 32$ and m^0 equal to the quantized value of the default setting x^0. A super linear function is used for calculating weights, i.e., $w_i = \frac{log(\mu+1)-log(i)}{\sum_{i=1}^{4}(log(\mu+1)-log(i))}$, where $i \in \{1, \ldots, 4\}$ because we choose a $(4, 9)-$CMA algorithm. The rest of the tunable parameters shown in Table 1 are set to their default values [4]. To evaluate the objective function, all quantized values of the configuration vector are mapped back to their real values. If CMA algorithm wants to evaluate a new setting, the parameters are reconfigured with their new setting and related software is rebooted to activate this setting. After this, our client emulator follows a sinusoidal pattern to create enough workloads that can saturate the system. We have developed software tools to analyze the web log files so as to calculate the system throughput and response time at the saturation point.

CMA algorithm is proposed for unconstrained optimization problems. As defined in Equation 1, our configuration autotuning problem includes a constraint on response time when maximizing system throughput. Therefore, we introduce an extra penalty item into the following objective function so as to covert a constrained problem to an equivalent unconstrained problem. After system throughput and response time are calculated from log files, the value of the following objective function rather than $-t(x)$ is passed to CMA algorithm for next iterations

$$\begin{cases} -t(x) + \frac{r(x)-r(x^0)}{r(x^0)}t(x) & \text{if } r(x) > \tau^0 \\ -t(x) & \text{if } r(x) < \tau^0 \end{cases}$$

Note that CMA algorithm is designed for minimizing objective functions. Since we want to maximize the throughput, the negative value of throughput is used as the objective function. The item $\frac{r(x)-r(x^0)}{r(x^0)}t(x)$ acts a penalty because it is positive when the response time constraint is violated. We set CMA algorithm to perform 100 iterations and each iteration takes around 25 minutes. Therefore it takes two days to complete 100 experiments. The result of our experiments is shown in Figure 7. In summary, CMA algorithm found the following suboptimal point: $t^*(x^*) = 277.14 \quad (req/s)$ and $r^*(x^*) = 122.08(ms)$, where $x^* = \arg\max_{r(x)<\tau^0} t(x)$ =(25.83, 53.94, 11327, 13.5, 49.0, 19100925.3)^T.

Compared to the default setting, this new setting can lead to 7.5% increase in throughput with 47.5% decrease in response time at the saturation point. For comparison, we also repeat the same experiments using SHC algorithm. The optimal solution found by SHC algorithm is: $t^*(x^*) = 272.3 \quad (req/s)$ and $r^*(x^*) = 64.9 \quad (ms)$, with $x^* = (75, 21, 14801, 28, 62, 137394449)^T$ (Figure 8). Note that SHC algorithm finds this optimum during its initial global sampling phase while CMA algorithm discovers its optimum after 5 generations. Therefore, the optimum from CMA algorithm is not discovered by random chance.

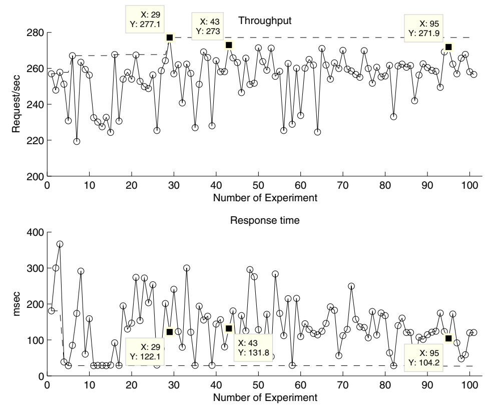

Figure 7. Performance of CMA algorithm.

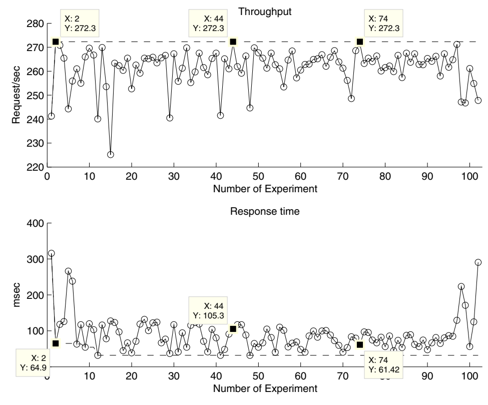

Figure 8. Performance of SHC algorithm.

CMA algorithm discovers the throughput equal to the optimal throughput from SHC algorithm twice along its search path. Eventually it moves on to find its optimal throughput, which has a 3% improvement compared to that from SHC algorithm. Though the response time of the optimum from SHC algorithm is smaller than that from CMA algorithm, this does not violate our conclusion on that CMA algorithm outperforms SHC algorithm because the optimization problem defined in Equation 1 is to maximize throughput while keeping response time below a threshold. In our experiments, this threshold is the response time resulted from the default setting. We assume that users are always satisfied as long as their response time is below the threshold.

6 Scalability

We only used six parameters in the above experiments with real systems. This is considerably smaller than the total number of tunable parameters from real distributed systems. Since CMA algorithm uses a joint distribution to track dependencies among all parameters, it can not scale very well to high dimensions. In this section, we propose a concurrent version of CMA algorithm to decouple search space. Our intuition is that we can use system knowledge or dependency testing to split parameters into multiple subsets and run one CMA algorithm on one subset separately. These subsets are weakly-dependent while the parameters from the same subset are highly-dependent. The concurrent version of CMA algorithm is illustrated as follows:

(1) Split the set of parameters into multiple subset $X = \bigcup X_i$ such that the parameters in one set X_i have shown strong dependency based on system knowledge or some high-level dependency testing. Initialize all of these parameters to their default values.

(2) For each subset X_i, use CMA algorithm to find optimal configuration while assuming parameters from other subsets are the best ones from previous steps. Replace all the parameters in X_i with their best values and set $i = i+1$. Go back to step 2.

We compare the performance of single CMA algorithm and concurrent CMA algorithm on the Rosenbrock function shown in Figure 5. The concurrent CMA is implemented by splitting the search space into $X_1 = \{x_1, x_2, \ldots, x_{10}\}$ and $X_2 = \{x_{11}, x_{12}, \ldots, x_{20}\}$ and iterating twice between these two sets. This results in 400 objective function evaluations and we also run the single CMA with similar number of function evaluations. The minimum $f(x)$ from the single CMA algorithm over 1000 runs is 584.19 while that from the concurrent CMA algorithm is 201.33. This verifies that the concurrent CMA algorithm could perform better when the dimension of search space is high and can be decoupled into weakly-dependent subsets.

Acknowledgement

The authors are grateful to Professor Bowei Xi for providing his software code of SHC algorithm. This work was performed when the first author worked as a summer intern at NEC Labs.

References

[1] I. Chung and J.K. Hollingsworth. Automated cluster-based web service performance tuning. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 36–44, 2004.

[2] Y. Coady, R. Cox, J. DeTreville, P. Druschel, J. Hellerstein, A. Hume, K. Keeton, T. Nguyen, C. Small, L. Stein, and A. Warfield. Falling off the cliff: when systems go nonlinear. In *HOTOS'05: Proceedings of the 10th Conference on Hot Topics in Operating Systems*, volume 10, pages 25–25, 2005.

[3] Y. Diao, J.L. Hellerstein, S. Parekh, and J.P. Bigus. Managing web server performance with autotune agents. *IBM Syst. J.*, 42(1):136–149, 2003.

[4] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

[5] R.L. Iman, J.C. Helton, and J.E. Campbell. An approach to sensitivity analysis of computer models, part 1. introduction, input variable selection and preliminary variable assessment. *Journal of Quality Technology*, 13(3):174–183, 1981.

[6] S. Kern, S.D. Müller, N. Hansen, D. Büche, J. Ocenasek, and P. Koumoutsakos. Learning probability distributions in continuous evolutionary algorithms- a comparative review. *Natural Computing: an International Journal*, 3(1):77–112, 2004.

[7] T. Osogami and S. Kato. Optimizing system configurations quickly by guessing at the performance. *SIGMETRICS Perform. Eval. Rev.*, 35(1):145–156, 2007.

[8] B. Xi, Z. Liu, M. Raghavachari, H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *WWW'04: Proceedings of the 3rd International Conference on World Wide Web*, pages 287 – 296, New York, NY, 2004.

[9] W. Zheng, R. Bianchini, and T.D. Nguyen. Automatic configuration of internet services. *SIGOPS Oper. Syst. Rev.*, 41(3):219–229, 2007.