Fast Autotuning Configurations of Parameters in Distributed Computing Systems Using Ordinal Optimization

Fan Zhang[1], Junwei Cao[2,3,*], Lianchen Liu[1,3] and Cheng Wu[1,3]
[1]*National CIMS Engineering and Research Center, Department of Automation*
Tsinghua University, Beijing 100084, P. R. China
[2]*Research Institute of Information Technology, Tsinghua University, Beijing 100084, P. R. China*
[3]*Tsinghua National Laboratory for Information Science and Technology, Beijing 100084, P. R. China*
[*]*Corresponding email: jcao@tsinghua.edu.cn*

Abstract

Conventional autotuning configuration of parameters in distributed computing systems using evolutionary strategies increases integrated performance notably, though at the expense of consuming too much measurement time. An ordinal optimization (OO) based strategy is proposed in this work, combined with neural networks to improve system performance and reduce measurement time, which is fast enough to autotune configurations for distributed computing applications. The method is compared with a well known evolutionary algorithm called Covariance Matrix Algorithm (CMA). Experiments are carried out using high dimensional rastrigin functions, which show that OO can reduce one to two orders of magnitude of simulation time while at the cost of an acceptable scope of optimization performance. We also carried out experiments using a real application system with three-tier web servers. Experimental results show that OO can reduce 40% testing time on average at a reasonable cost of optimization performance.

1. Introduction

1.1 Research Background & Motivation

Various advanced computing technologies, e.g. cluster computing [1], grid computing [2], cyber-infrastructure [3] and cloud computing [4], requires improvement of system performance, such as achieving high throughput and reducing response time. These mainstream distributed computing technologies are implemented in many different computational burden fields, such as real time on-line e-commerce transactions, transportation communication system simulation and large scale scientific workflow analysis. How to improve the performance of these systems is one hot research topic lots of vendors and scientists care about. One feature in common is that the performances of those complex distributed computing systems are affected by combinational configuration parameters, such as session time[1], maximum clients[2] and cache pool size[3], etc. Default settings are not necessarily suitable for any scenario. Nowadays many people conduct parameter tuning by experience, which is not scientific and not easily applicable into different applications.

In terms of session time, if it is set too short, although it is safe for potential malicious attack, clients have to connect the server frequently, which leads to extra overhead to web systems and decrease client satisfaction. Maximizing the maximum client number and cache pool size tends to increase system throughput, which cannot guarantee a satisfied response time since the server has to deal with more client data, conservational states and temporary information. How to find a set of optimal parameters to run applications on web servers and/or server farms to improve system performance is still an open issue.

The dynamicity and randomness of those complex systems, whose performance is affected by these combined factors, are difficult to be predicted by precise mathematical models and formulas, thus limited the efficiency of traditional heuristic optimization methods. Many researchers proposed to use black box model [5, 6] to optimize performance. Then how to get a proper combinatorial configuration of parameter values to meet this requirement is one challenge.

On the other hand, finding a proper configuration of parameter values is always time consuming and error prone in the black box search model. Generally it took 25 minutes or more to find an optimal value [5, 6], which is not acceptable in many scenarios. Then how to find a configuration quickly becomes another challenge. This work is mainly focused on fast autotuning configurations of parameters in dynamic distributed computing systems to deal with this issue.

[1] Session time specify how long the conversational state with one client across multiple requests can be maintained.
[2] Maximum clients specify how many concurrent clients can be connected to the server at a same time.
[3] Cache pool is used in database system to store temporary variable, tables, results, etc.

1.2 Related Work

There is a significant amount of works focusing on autotuning configurations to improve system performance for distributed systems. Y. Diao, et, al. [7] proposed an agent-based optimization solution, which not only automated ongoing system tuning but also automatically designed an appropriate tuning mechanism for target systems. B. Xi, et, al. [5] formulated the problem of finding an optimal configuration as a black box optimization problem and proposed a Smart Hill-Climbing (SHC) algorithm to show that it is more effective than simulated annealing and random recursive search both in synthetic rastrigin functions and real web server systems. A. Saboori, et, al [6] implemented the Covariance Matrix Algorithm (CMA) in both synthetic functions and real web systems to show it outperformed SHC by 3% or more in performance improvement. Many other works on autotuning real application systems include [8-11]. The common issue of these optimization problems is that measurement time to find an optimal configuration is too long to be suitable for real applications.

Some recent works not only dealt with the black box optimization problem, but also included consideration of the measurement time issue, which had a similar motivation of our work. Osogami and Itoko [12] proposed to use checkpoints to save system states and find probably better system configurations quickly. However, with the increase of complexity of setting checkpoints, it does not scale well in real world systems. Osogami and Kato [13] then proposed another strategy, quick optimization via guessing (QOG), which reduced measurement time by selecting best strategies. While this work is model-based, we still tend to take the issue as black box autotuning optimization since different parameters lead to different mathematical models and precise evaluation is time consuming and error prone.

Ordinal Optimization (OO) [14] is proposed to solve large scale searching problems to shorten simulation time, which is applicable in this work since the combinatorial search space is quite large. Also multi-layered neural networks [15] are applied in quickly finding a rough model to get the problem type of black box optimization, since no precise mathematical model can be used to describe application and scenario dependant problems. As far as we know, no previous work using OO for autotuning parameter configurations have been found. Our work focuses on reducing simulation time in order to gain performance improvement and reducing measurement time for real-time applications, which differentiate our work from those previous efforts.

The rest of this paper is organized as follows. Concepts of OO and large space problems are introduced in Section 2. Section 3 describes methodologies and algorithms we develop in our performance strategies for distributed applications. A comparison of Covariance Matrix Algorithm (CMA) and OO are included in Section 4 to show their optimization efficiency in multi-dimensional rastrigin functions in terms of optimal values and simulation time. We also compare the two methods in a real-world web application system to show our fast autotuning performance in Section 5. And we conclude the paper and propose some future work in Section 6.

2. OO Preliminaries

2.1 OO and its Applications

Ordinal Optimization (OO) [14] was first proposed by Professor Yo-Chi Ho of Harvard University in 1992 to deal with simulation based optimization problems, which is verified to be effective and efficient in problems of very complex subject functions with ultra large search space. This method has been widely put into practical use and won very much popularity in industrial, communicational and transportation systems.

The main tenet of optimization is based on two practical true life experiences.

- Order is much better to judge than value. You can very easily figure out one hamburger is heavier than the other by your intuition from the size and other visible characteristics - an “ordinal” problem. When comes to how heavier the one than the other, - a “cardinal” problem, things will be much difficult.
- Nothing but the best is very costly. Actually it is very intuitive for us to know this idea by throwing a dart. If we focus on the center, it’s a hard time. Things will be much easier if we relax the goal and satisfy by the “good enough” results, which are the whole board.

Generally speaking, OO is designed to deal with single-objective optimization problems. Here are two functions, throughput *t(x)* and response time *r(x)*, are used to quantify system performance. In our system testing in Section 5, the response time is divided into two parameters. One is the average http response per second, which is used to describe response situation of servers given a certain throughput. The second is average transaction response time, which is used to describe response time of different transactions of a certain throughput. We consider three objectives separately. Actually in some scenarios these objectives are application dependent and even interdependent. Several notations and symbols of OO are listed in Table 1.

Table 1. Summary of notations

Notation	Descriptions
$\theta(N)$	The whole search space (# of the set)
θ_N	Randomly selected space to represent θ
$G(g)$	Good enough set (# of the set)

G_θ	Good enough set of space , same as G
G_N^r	Reduced good enough set
$S(s)$	Selected set (# of the set)
S_k	Truly top k designs in S
k	Required alignment level
k'	Required alignment level of reduced set
α	Universal alignment probabilities (UAP)
J_i	Objective functions
$\lvert\bullet\rvert$	# of $\bullet$

We apply OO in complex optimization problems as follows:

(1) Uniformly and randomly sample *N* designs from the whole search space *θ*.
(2) Use a crude and computationally fast model to estimate the two performance criteria of these *N* designs.
(3) Estimate the Ordered Performance Curve (OPC) class and the noise level based on the problem type. The user specifies the size of good enough set, *g*, and the requirement alignment level, *k*.
(4) Use regressed values for UAP to calculate $s = Z$ (α, g, k/OPC class, noise level).
(5) Choose the observed first *s* results as the selected set *S*.
(6) The theory ensures that *S* contains at least *k* truly good enough designs with probability no less than *α*.

2.2 OO with Large Search Space

The most prominent characteristic of autotuning configuration problems is its high dimension (many parameters combination), which leads to large search space. Based on Lin and Ho's work of large space selection [16], there are some basic mathematical works to show that OO is applicable in such large search space problems. If we deal with the issue of very large search space (θ), one difficulty is that how much we should choose in the search space. Suppose G_N^r as the reduced good enough set, say top m% designs of θ_N, so $G_N^r \subseteq G_\theta$ is obvious with very large probability. Then the alignment level is *k'* instead of *k*. Then we try to find a new alignment probability of $\mathrm{P}\left[\left|G_\theta \cap S\right| \geq k'\right]$ instead of the previous alignment probability $\mathrm{P}\left[\left|G \cap S\right| \geq k'\right]$. The relationship of all the notations is shown in Figure 1.

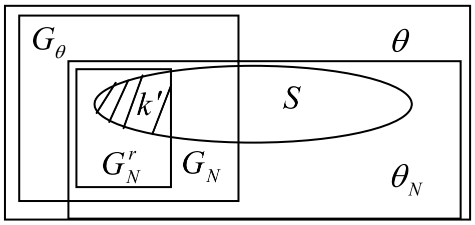

Figure 1. Relationships of notations

Here we just provide a concise proof to show the representative of θ_N to θ, and detailed proof can be referred to [16].

$$
\begin{aligned}
& P\left[\left|G_\theta \cap S\right| \geq k'\right] \\
& \geq P\left[\left|G_N \cap S\right| \geq k, \left|G_N^r \cap S_k\right| \geq k', G_N^r \subseteq G_\theta\right] \\
& = P\left[\left|G_N \cap S\right| \geq k\right] \\
& * P\left[\left|G_N^r \cap S_k\right| \geq k', G_N^r \subseteq G_\theta \middle| \ \left|G_N \cap S\right| \geq k\right] \\
& = P\left[\left|G_N \cap S\right| \geq k\right] \\
& * P\left[G_N^r \subseteq G_\theta\right] * P\left[\left|G_N^r \cap S_k\right| \geq k' \middle| \ \left|G_N \cap S\right| \geq k\right]
\end{aligned}
$$

Mathematically, the proof can show that:

$$P\left[G_N^r \subseteq G_\theta\right] = 1 - \sum_{\mathrm{i}=0}^{\left|\mathrm{G_N^r}\right|-1} P[\left|G_\Theta \cap \Theta_N\right| = i] \,.$$

$P[\left|G_\Theta \cap \Theta_N\right| = i]$ follows the Bernoulli distribution with $B(N, {}^{G_\Theta}\!/\!{}_{\Theta})$, so $P\left[G_N^r \subseteq G_\theta\right]$ nearly equals to 1. On the other hand, lots of experimental results show that $P\left[\left|G_N^r \cap S_k\right| \geq k' \| G_N \cap S\right| \geq k\right]$ is also near 1. As a result, $P\left[\left|G_\theta \cap S\right| \geq k'\right]$ and $P\left[\left|G_N \cap S\right| \geq k\right]$ is almost equal.

The proof shows that θ_N is a proper and sufficient representative of large search space when *N* is very large, say 10^{10}. Actually in our autotuning configuration parameter problem, the search space is much less than *N*. Our abundant experiments show that 200 designs randomly and uniformly sampled from the whole space is more than enough to represent the whole parameters' space.

3. OO-based Configuration Autotuning

In this section we introduce the main challenge in optimization of autotuning configuration parameters in distributed computing systems and the corresponding algorithms. The major difficulty is that we have to balance the rough model that is computationally fast with precision of system representation. Computationally fast model requires rough and simple representation, which cannot make it proper enough to show true performance of the system. We tend to utilize a three layer neural network, a heuristic method, to represent system performance as the rough model.

Another essential issue during the implementation is to minimize search space to reduce measurement time and guarantee a satisfied performance. Problems arise that how much search space we should choose to guarantee a good enough solution since we have to deal with a large number of parameters, always defined in different regions and leading to combinational space to search. Large search space leads to inefficiency of autotuning performance and long measurement time. From the large space searching proof provided in Section 2, it is enough to choose 200 parameter values to represent the whole search space.

After we find the much smaller substitute space of the whole very large search space, we should then use its OPC type and the noise level based model to "see" how the structure look like, which is decisive to how much selected set S to choose for precise evaluation. We can see from here why the OO based method saves so much computational budget in that it digs out the structural information to see where good enough results are more likely to be, based on the OPC type, noise level and neural network model. This prior knowledge is of course very helpful in our searching.

Parameters of the mainstream JSP/Servlet system are [*MaxClients*, *MaxConnections*, *SessionTime*, *KeyBufferSize*, *MaxPoolSize*, …], which are typically in between [10, 1000], [50, 40000], [30s, 200s], [3M, 500M], [10, 150], respectively. Firstly we should linearly normalize these value spaces into a uniformed space, e.g. [0, 100], in our applications. Secondly these values have to be linearly re-quantized back to their values in the process of performing system behaviors.

Suppose we have n parameters $\{p_1, p_2, \cdots, p_n\}$ and the range of parameter p_i is $\left[\underline{p}_i, \overline{p}_i\right]$, the algorithm of applying OO is as follows.

(1) For i = 1 : n, linearly quantize $\left[\underline{p}_i, \overline{p}_i\right]$ into [0, 100];

(2) Uniformly and randomly choose 200 groups of configuration parameters $\{v_{1,1}, v_{2,1}, \cdots, v_{n,1}\}$, $\{v_{12}, v_{22}, \cdots, v_{n2}\}$,…, $\{v_{1,200}, v_{2,200}, \cdots, v_{n,200}\}$ in [0,100] and quantize their values back to $\left[\underline{p}_i, \overline{p}_i\right]$.

(3) Use the three-tier neural network model to evaluate some typical inputs $\left[\underline{p}_i, \overline{p}_i\right]$ pairs with related output response time. This model is the computational fast and rough model and is shown in Figure 2.

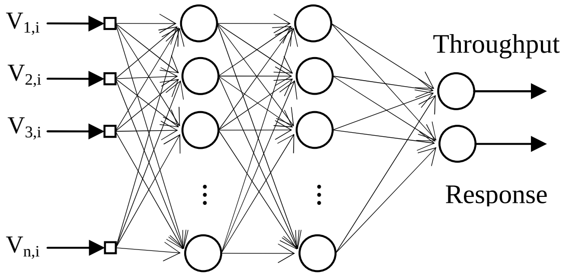

Figure 2. The neural network rough model for autotuning configurations of parameters

The rough model can be applied to other parameter groups and thus this part of work has to be done only once. This step is only included in real system applications since there is no exact mathematical model to evaluate. Using synthetic functions such as rastrigin introduced in the next section, we need not establish the neural network model since the rough model is easily derived from our observation to true mathematical functions.

(4) Use the model to roughly evaluate the OPC type (Flat, U-Shaped, Neutral, Bell or Steep). As the simulation goes, the majority of applications (both in synthetic functions and real applications) in our experiments shows their OPC to be Bell shape, which implicitly tells us that the good enough data distribution is neutral (neither too good nor too bad) for us to search.

(5) Generally the noise level can be derived from two methods. The first one is calculating the maximum margin of rough model *r(t)* and precise simulation time *p(t)* as the noise, which is $\left|\max\left(p(t) - r(t)\right)\right|$. Then the noise level of this problem can be calculated as $\left|\max\left(p(t) - r(t)\right)\right| \big/ \max\left(p(t)\right) - \min\left(p(t)\right)$.
If still we cannot find such noise level in reference tables, we can then use the other method to estimate. Namely use linear limiting and the known noise level to estimate other circumstances. In most cases such as in synthetic functions and real systems in next sections, the first method is enough.

(6) Look up in the table [17] to calculate the candidates of the selection set S with the specific good enough set g (set it be 20), required alignment probability α (set it be 98%) and the required alignment level k (set it be 10). OO theory can guarantee that S contains at least k good enough parameter values with a probability no less than α.

4. Performance Evaluation Using Synthetic Functions

This section introduces the performance improvement and measurement time reduction using ordinal optimization in the high dimension rastrigin function, which is a typical example of explicitly formulated and non-linear multimodal functions that contains many local minima. We use this synthetic function not only because of its intrinsic high dimensional similarity with autotuning configuration parameter problems as mentioned in Section 2.2, but also because it is highly efficient in testing these heuristic methods based on many literature reviews [5, 6].

The function is defined as:

$$f(x) = An + \sum_{i=1}^{n}\left(x_i^2 - A\cos\left(2\pi x_i\right)\right),$$

where n is the dimension of the function and the global minima is $x_i = 0$ and $f(x) = 0$.

Two methods, Covariance Matrix Algorithm (CMA) and OO, are compared to show the performance of OO, which can guarantee to find a desirable and good enough result, without lost of too much performance.

Different from real system environments, the rastrigin function is a definite function without stochastic factors, which simplify the analysis of the problem. Due to the strong random of the simulation of CMA, several factors should be included. Iteration times should be fairly considered to represent the true average time used in simulation. The sorting algorithms, which consume simulation time significantly, should be also legitimately in part excluded since many are not a component of the CMA or OO. Many values used in final comparisons are from the same experiments which take advantage of the average and repeated simulation. The steps of the application of OO methods in rastrigin functions can be described as follows.

(1) Choose $\{x(1), x(2), \cdots, x(1000)\}$ of the search space uniformly and randomly, which are sampled from multidimensional Gaussian distribution.

(2) The crude and computationally fast model of rastrigin is used in the following form, which is much easier to compute.

$$r(x(j)) = n * A + \sum_{i=1}^{n} x^2(j),\ j = 1 \cdots 1000 .$$

(3) Use the chosen $\{x_1, x_2, \cdots, x_{1000}\}$ to roughly calculate the *r(x)* in order to get the OPC class. The simulation is based on 1000 designs in that it is differentiable in the shape of the final curve. From the OPC curve, it is a typical Bell shape problem, as shown in Figure 3.

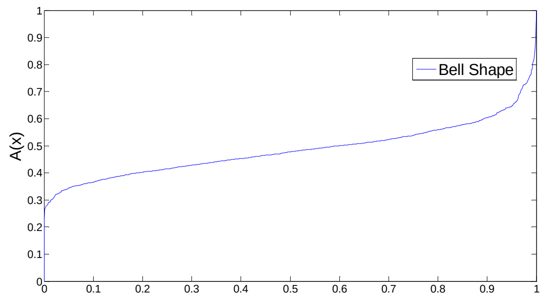

Figure 3. The OPC type of the rastrigin function

(4) Estimate the noise level. The difference between the precise and rough model is $A\sum_{i=1}^{n}\cos(2\pi x_i)$, thus we choose to use its maximum value as the noise and compare the value with $\max(f(x)) - \min(f(x))$, which described the noise level of the crude model.

(5) Specify the good enough set g and the required alignment level k, the alignment probability α should also be reasonably included.

(6) Look up the table in [17] to find out $s = (g, k, \alpha / Bell, small\ \ noise)$ to get the number of the selected set *S*.

(7) Select the top s candidate x from the 1000 set by the rough model *r(x)*. This function enables us to use the precise model in that it is not difficult to calculate the true value.

The theory of OO ensures that the selected set *S* contains at least *k* truly good enough *f(x)* with probability no less than *α*.

The experiment is carried out using two computers separately. One is Windows XP with Intel Core 2 Duo T5870, 2G RAM and the other same experiment is done under the Windows XP with Intel Pentium D 2.8GHz, 2G RAM. The final results come from the analysis of both results and their average.

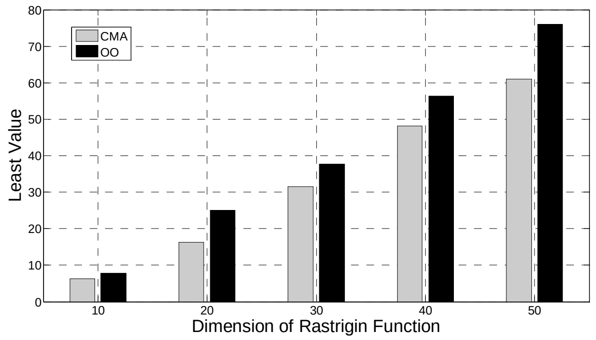

Figure 4. Comparison of least values using CMA and OO against different dimensions ranging from 10 to 50

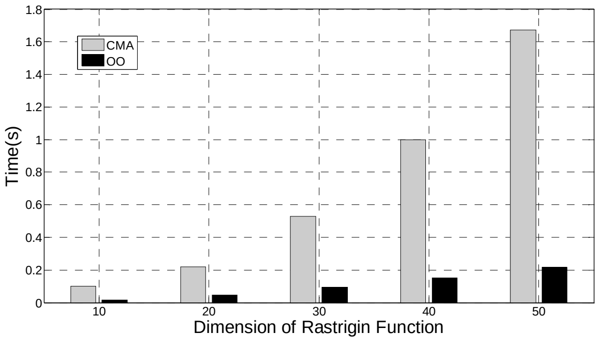

Figure 5. Comparison of simulation time using CMA and OO against different dimensions ranging from 10 to 50

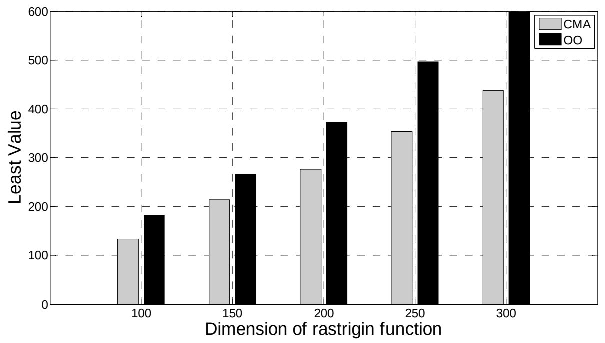

Figure 6. Comparison of least values using CMA and OO against different dimensions ranging from 100 to 300

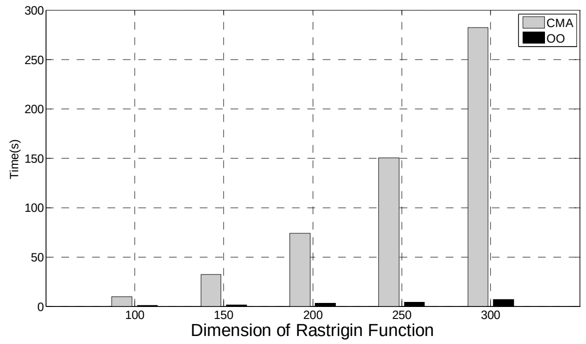

Figure 7. Comparison of simulation time using CMA and OO against different dimensions ranging from 100 to 300

From the results comparison, we can easily see that OO reduced one to two orders of magnitude of simulation time at the cost of only a small portion of optimization performance lost. This advantage is even more noticeable in higher dimensions.

5. Performance Evaluation Using Real Web-based Systems

Experiments using real running system are carried out in a typical mainstream three-tier system of JSP/Servlet, which takes advantage of MVC framework. This system is used to provide a platform to organize different roles in a company to corporate with each other in the workflow of task related events. The first tier is a web (HTTP/HTML) server to handle requests from remote clients, using Tomcat of Apache servers. The application server, also Tomcat, has a controller component to figure out the request type and target person to handle the request, then the request is transferred to the related servlet component to invoke its model and find its execution source. During the process of invoking, a database system, MySQL, is also used to manage relational data tables, the content of files and documents. The system structure and its corresponding workflow are illustrated in Figure 8.

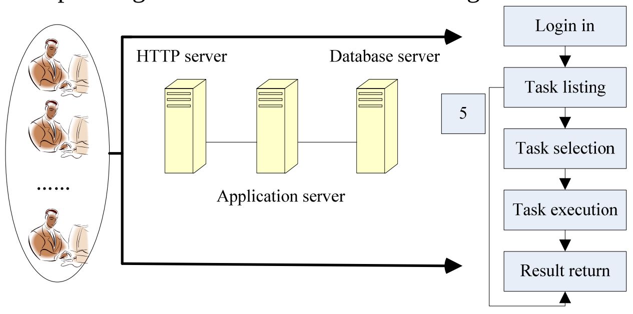

Figure 8. Experimental system structure and corresponding execution workflow

There are totally 5 categories of actions. Each user has to login only once, and lists tasks he/she has to deal with. Some tasks are interrelated and thus their order of selection, which is the following step, is one concern. After tasks are selected and executed, and the system returns the result, the user goes back to the task list and continues the procedure. To make the workflow more reasonable, we set every user repeat the process five times before logout.

We consider seven parameters here to represent system performance, [*MaxKeepAliveRequests*, *KeepAliveTimeOut*, *ThreadCacheSize*, *MaxInactiveInterval*, *MaxConnections*, *KeyBufferSize*, *SortBufferSize*]. All these parameters have impact on system performance when the system is saturated. The default values are [100, 5s, 8, 2s, 400, 20M, 256K]. Corresponding parameter spaces are [10, 200]×[10s, 200s]×[5, 100]×[5s, 50s]×[100, 500]×[8M, 256M]×[128K, 1024K], respectively. We chose parameter spaces according to our specific application and experiment type thus some default settings in software packages maybe not necessarily in between the region. Some of the setting spaces are dependent on application types, such as *ThreadCacheSize* and *MaxInactiveInterval*. Some of the settings are from the intrinsic characteristics of the server type, such as the *SortBufferSize* which is allocated when MySQL needs to rebuild the index, and it is allocated per thread so large settings may lead to low system performance. To some extend these parameter spaces are chosen by our empirical experience and configuration characteristics of servers.

Neural networks are used based on several existing groups of input/response pairs to generate the rough model. Given this particular problem, we use the method described in Section 3 to estimate the noise level, which is quite high due to many unpredictable factors of the dynamic system. Then we combined with OPC type (Bell), the given good enough set g to be 100 (actually, much more than this figure because of the quite large search space), required alignment level k to be 5 and the alignment probability α as 98%. We predefine those parameters before the experiments and get the size of the selected set S to be 20.

It is again formulated as $P\left[\left|G_\theta \cap S\right| \geq 5\right] \geq 98\%$.

We should bring in a concept of cost performance c as the throughput (t) vs. response time (r) ratio to describe the combinational performance of each configuration of parameters. It is defined as $c=t/r$. Different from [6], we also want to quantify the value of response time as our analytical criterion rather than making it only as a threshold since we model it as a multi-objective programming problem.

Our experiment shows the impact of the seven groups of parameters on some predefined performance metrics, e.g. throughput (average amount of requests that can be processed per second), responses time (average time to finish a servlet execution) and cost performance as defined above. We loaded 410 virtual users to saturate the three-tier system. To make the system work more practically applicable in real system scenarios, we do not generate the workload simultaneously rather than make the workload gradually increase and run for a certain period of time, generally 60 seconds, after all users are initiated. Performance results are illustrated in Figures 9, 10, and 11. In particular, performance results with default parameters are as follows.

- Default throughput $t(X^0)$ (requests/sec): 149.21
- Default responses time $r(X^0)$ (msec): 501.8
- Default cost performance $c(X^0)$: 0.2695

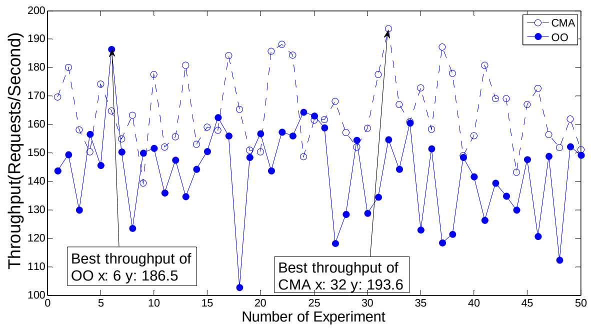

Figure 9. Comparison of throughput

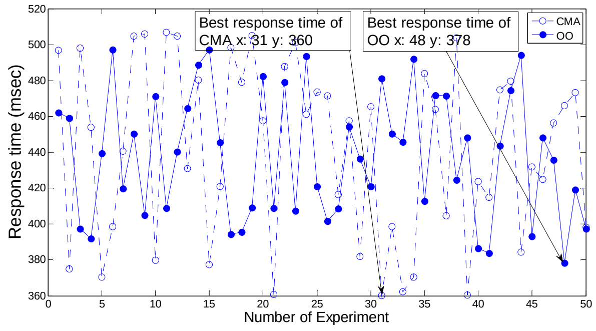

Figure 10. Comparison of response time

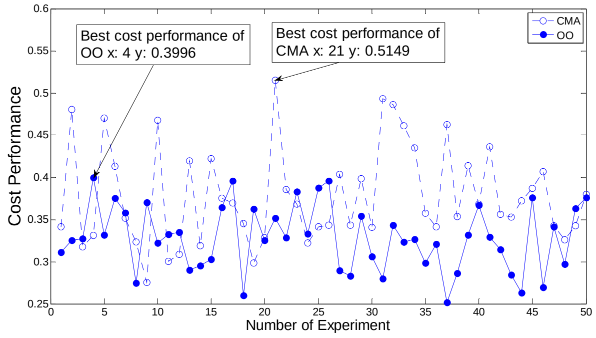

Figure 11. Comparison of cost performance

In experiment No. 21, we use CMA to find out the best cost performance. Our parameters are X^{CMA} = [126, 28.19, 16, 29, 89, 136, 323.51]^T, which results in the throughput, $t(X^{CMA})$ = 185.73 (requests/sec) and cost performance $c(X^{CMA})$ = 0.5149, 24.5% and 91.1% more than the default setting respectively. The response time, $r(X^{CMA})$ = 361 (msec), which decreased default setting by 28.1%. OO optimal value can be seen from experiment No. 4, when the configuration parameter value is X^{OO} = [194, 46.28, 46, 35, 101, 188, 445.95]^T. The throughput and cost performance are $t(X^{OO})$ = 156.6 (requests/sec) and $c(X^{OO})$ = 0.3996, leading to 5.0% and 48.3% increase, respectively, compared with the default setting. The response time is decreased by 21.9%, which is $r(X^{OO})$ = 392 (msec).

As for the comparison of CMA and OO, from the three simulations we can easily see that CMA is better than OO by 15.7% and 8.0% in throughput and response time, respectively. The reason is that OO tends to explore more estimated best value prone space (by rough model set by our neural network), the performance of which is strongly decided by the accuracy of the rough neural network model. On the contrary CMA tries to do both exploration and exploitation, which is easier to pick out better configuration settings in search space. The tradeoff then is that CMA gets a slightly more optimal result at a cost of long simulation time; OO achieves reasonable optimal results in a dramatically faster way. Comparison on experimental time is illustrated in Figure 12.

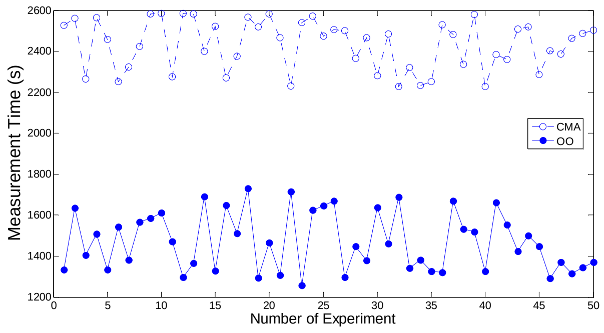

Figure 12. Comparison of experimental time

As for the comparison of measurement time, we use the same (4,9)-CMA algorithms as [6] to perform the search and on average it requires five generations to arrive at the optimal value in each experiment, which is also reasonable for evaluation of simulation time. The time consuming of OO is dramatically reduced compared with CMA. Generally the measurement of this method is composed of the several parts, the establishing of neural network rough model, defining OPC type and noise level, looking up the table to get selection set *S*. Actually these parts are ignorable since they should be done only once during the whole experiment. Then the measurement time depends only on the selected set *S* that we choose.

The average measurement time with CMA is $t(X^{CMA})$ = 2432.25 (s), and dramatically reduces to $t(X^{OO})$ = 1470.43 (s) with OO. Averagely OO reduces experimental time by about 40%, at a cost of less 20% loss of throughput and response time as mentioned above, compared with CMA. This verifies the advantages of OO in performance improvement and measurement time reducing of autotuning configurations of parameters.

6. Conclusions and Future Work

In this paper, ordinal optimization based strategy is proposed for distributed computing systems to improve performance and reduce measurement time of autotuning configurations of parameters. Compared with traditional evolutionary optimization strategy, this method reduce experiment time dramatically both in synthetic functions and real distributed systems with acceptable optimization performance. Future work can be categorized into three directions.

(1) No free lunch theorem suggests us to search intrinsic characteristics of autotuning configuration problems and dig out more information to support decision-making. Searching strategies of CMA and simulation time saving method of OO can be combined to make a complementary solution.

(2) Web servers have too many parameters to be exhausted in our experiments. Some of them are application-dependent thus limit traditional optimization methods. Suggestive configuration parameters according to the application type can be concluded according to specific scenarios.

(3) Large-scale distributed computing systems may contain many web, application and database servers. Each server has its own search space interrelated with others. The method proposed in this work will be applied in larger distributed systems in future.

Acknowledgement

This work is supported by National Science Foundation of China (grant No. 60803017), Ministry of Science and Technology of China under the national 863 high-tech R&D program (grants No. 2006AA10Z237, No. 2007AA01Z179 and No. 2008AA01Z118), Ministry of Education of China under the program for New Century Excellent Talents in University and the Scientific Research Foundation for the Returned Overseas Chinese Scholars, and the FIT foundation of Tsinghua University.

References

[1] K. Hwang and Z. Xu, *Scalable Parallel Computing*, Chapters 9 and 10, McGraw-Hill, 1998.

[2] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan-Kaufmann, 1998.

[3] D. E. Atkins, K. K. Droegemeier, S. I. Feldman, et. al., Revolutionizing Science and Engineering through Cyberinfrastructure, *National Science Foundation Blue - Ribbon Advisory Panel on Cyberinfrastructure*, 2003.

[4] G. Boss, P. Malladi, D. Quan, L. Legregni, and H. Hall, Cloud Computing, *IBM High Performance On Demand Solutions (HiPODS)*, Oct. 2007.

[5] B. Xi, Z. Liu, and M. Raghavachari, A Smart Hill-Climbing Algorithm for Application Server Configuration, in *Proceedings of the 3rd International Conference on World Wide Web (WWW'04)*, pp. 287-296, New York, NY, 2004.

[6] A. Saboori, G. Jiang, and H. Chen, Autotuning Configurations in Distributed Systems for Performance Improvements using Evolutionary Strategies, in *Proceedings of the 28th International Conference on Distributed Computing Systems (ICDCS'08)),* pp. 765-772, Beijing, China, 2008.

[7] Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus, Managing Web Server Performance with Autotune Agents, *IBM System Journal*, 42(1):136-149, 2003.

[8] I. Chung and J. K. Hollingsworth, Automated Cluster-based Web Service Performance Tuning, in *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)*, pp. 36-44, June 2004.

[9] X. Liu, L. Sha, S. Froehlich, J. L. Hellerstein, and S. Parekh, Online Response Time Optimization of Apache Web Server, in *Proceedings of the 11th International Workshop on Quality of Service (IWQoS'03)*, pp. 461-478, June 2003.

[10] M. Raghavachari, D. Reimer, and R. D. Johnson, The Deployer's Problem: Configuring Application Servers for Performance and Reliability, in *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pp. 484-489, 2003.

[11] Y. Zhang, W. Qu, and A. Liu, "Automatic Performance Tuning for J2EE Application Server Systems", in *Proceedings of the 6th International Conference on Web Information Systems Engineering (WISE'05)*, pp. 520-527, November 2005.

[12] T. Osogami and T. Itoko, Finding Probably Better System Configurations Quickly, in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE 2006)*, pp. 264-275, June 2006.

[13] T. Osogami and S. Kato, Optimizing System Configurations Quickly by Guessing at the Performance, in *Proceedings the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'07)*, pp. 145-156, June 2007.

[14] Y. C. Ho, An Explanation of Ordinal Optimization: Soft Computing for Hard Problems, *Information Science*, 113, 1999.

[15] S. Haykin, *Neural Networks: a Comprehensive Foundation (2nd Edition)*, Prentice Hall, 1998.

[16] S. Y. Lin and Y. C. Ho, Universal Alignment Probability Revisited, *Journal of Optimization Theory and Applications*, 113(2): 399-407, 2002.

[17] T. W. E. Lua and Y. C. Ho, Universal Alignment Probabilties and Subset Selection of Ordinal Optimization, *Joumal of Optimization Theory and applications*, 93(3): 455-489, 1997