Performance Improvement of Distributed Systems by Autotuning of the Configuration Parameters[1]

ZHANG Fan(张帆)[1] , CAO Junwei(曹军威)[2,3,*], LIU Lianchen(刘连臣)[1,3] and WU Cheng(吴澄)[1,3]

1.National CIMS Engineering and Research Center, Tsinghua University, Beijing, 100084, China;

2.Research Institute of Information Technology, Tsinghua University, Beijing, 100084, China;

3.Tsinghua National Laboratory for Information Science and Technology, Beijing, 100084, China.

Abstract

Performance of distributed computing systems is partially dependent on configuration parameters recorded in configuration files. Evolutionary strategies, with their ability having a global view of the structural information, have been shown to effectively improve performance. However, most of these methods consume too much measurement time. This paper introduces an Ordinal Optimization (OO) based strategy combined with a back propagation neural network for autotuning of the configuration parameters. The strategy was first proposed in the automation community for complex manufacturing system optimization and is customized here for improving distributed systems performance. The method is compared with the Covariance Matrix Algorithm. Tests using a real distributed system with three-tier servers show that the strategy reduces the testing time by 40% on average at a reasonable performance cost.

[*] **Received: 2011-01-12**

This work was supported by the National Science Foundation of China (grant No. 60803017) and the Ministry of Science and Technology of China under the National 973 Basic Research Grant Nos. 2011CB302505 and No. 2011CB302805.

[*]**To whom correspondence should be addressed. Tel: 01062772260; E-mail: jcao@tsinghua.edu.cn**

Key Words: Distributed Systems; Performance Evaluation; Autotune Configurations Parameters; Ordinal Optimization; Covariance Matrix Algorithm

Introduction

Performance improvements, such as high throughput and reducing end-to-end response times, are critical in distributed computing systems [1] such as cluster computing [2], grid computing [3], cyber-infrastructure [4] and cloud computing [5]. These mainstream distributed computing technologies are implemented in various computationally intensive areas to provide online services, such as real time on-line e-commerce transactions, collaborative games and large scale scientific workflow analyze. One major difficulty in improving performance of distributed computing systems is that the services are deployed on different servers, which requires system performance improvement at a global level.

Distributed systems usually have some common configuration file, and many configurable parameters, such as the session time, maximum number of connected clients and cache pool size. Their values are recorded in a plain text file. The session time specifies how long the conversational state with one client is maintained across multiple requests. Maximum number of clients specifies how many concurrent clients can be connected to the server at one time. The cache pool is used in database systems to store temporary variables, tables and results.

The performance of distributed systems is partially dependent on these system configuration parameters, so there is an escalating interest in how to make best use of these configuration parameters for system performance improvements for the following three reasons:

(1) Parameter tuning is easy since their values are recorded as plain text or as XML text, which is commonly used in mainstream servers such as those byApache, JBoss and MySQL.

(2) Parameter tuning requires little extra cost compared with hardware investments.

(3) Parameter values significantly impact system performance. Tests show that optimal parameters can increase system throughput by 24.5% and reduce average response time by 28.1% compared with the default settings.

Many companies tune these parameters based on their experience. However, this study provides a

scientific approach applicable to different application scenarios in this work.

Although configuration parameter autotuning can significantly improve system performance, there are many difficulties.

For example, if the session time is set too short, even this reduces the potential for malicious attacks, clients have to more frequently connect to the server, which leads to connection overhead. Increasing the number of maximum clients and cache pool size increases system throughput, but cannot guarantee satisfactory response time since the server has to deal with more client data, conservational states and temporary information. Thus the optimal configuration parameters to run applications on web servers and/or server farms to improve system performance is still an open issue.

Dynamic and random events in these complex systems, whose performance is affected by a combination of many factors, are difficult to predict using precise mathematical models, so the effect of traditional heuristic optimization methods is limited. Many researchers have proposed use of the black box model [6][7] to optimize performance, but proper combination of configuration parameters to maximize system throughput as well as minimize average response time is still a challenge.

In another way, determination of the proper configuration parameters is quite time consuming in the black box search model. Searches takes 25 minutes or more to find an optimal value [6][7], which is not acceptable in many real-time scenarios. Thus quickly finding a configuration value is another challenge.

This work focuses on fast autotuning of configuration parameters in dynamic distributed computing systems to address these challenges.

1. Preliminaries

Several concepts used in this work are defined here.

Def. 1. A Policy, denoted by θ, is one vector of configuration parameters. For example,

θ = [200, 300, 50 sec, 200 MB, 20 MB] is one policy for the configuration parameters: [MaxClients, MaxConnections, SessionTime, KeyBufferSize, MaxPoolSize].

Def. 2. Performance, defined as a cost performance, is Ω = throughput/response time. The throughput is the average number of requests processed per second. The response time is the end-to-end time from initiating a request to receiving the result.

Def. 3. The Ideal performance, $J(\theta)$ is used. There are too many uncertainties affecting the performance autotuning. Thus tests should be replicated N (generally a large number) times for each policy with the arithmetic mean used to get Ω for each policy. Thus, the exact relationship J is $\Omega = J(\theta)$.

Def. 4.The Measured performance $\hat{J}(\theta)$ is a relatively simple but computationally fast model. $\hat{J}(\theta)$ is used approximate the ideal performance. ω type and noise level analytical methods defined in (7) and (8) are used to bridge the gap between the ideal performance and the measured performance as much as possible. This simple model used here is a neural network.

This model significantly reduce the computational cost to get a good enough policy, but not necessary the best one. The measured performance can be seen as the ideal performance plus some noise:

$$\hat{J}(\theta) = J(\theta) + noise \quad (1)$$

Def. 5. The Good enough set, *G* is best g policies of the ideal performance are defined as the good-enough set G. Number g is preset by the users.

Def. 6. Selected set, *S* are the best s policies of the *measured performance*.

Instead of finding the optimal policy θ^* in the whole search space, the OO method searches for a small set S, which contains k good-enough policies in G. The success probability of such a search was set as α (e.g. 98%). The good-enough policies are the top g $(g \geq k)$ in the search space Θ. The numbers k and g are preset by the users. They follow the condition in Eq. (2) and illustrated in Fig. 1.

$$P\left[\left|G \cap S\right| \geq k\right] \geq \alpha \tag{2}$$

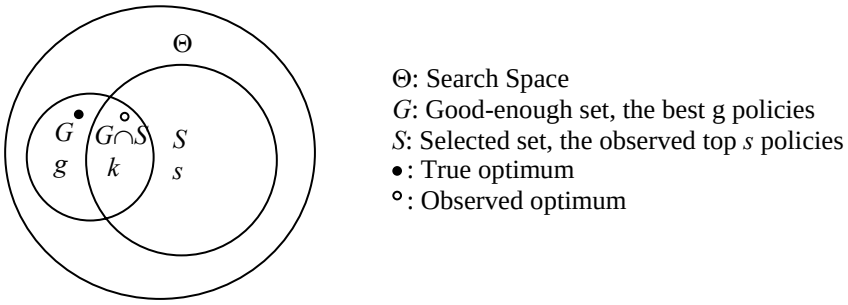


Figure 1. Graphical illustration of the notation describing the key concepts used in ordinal optimization. *G* and *S* represents the good-enough and selected set with sizes *g* and *s*. *S* should have at least *k* overlapped policies in *G*. This narrows the search space from a very large |Θ| to a much smaller *s*.

The simple illustration in Fig. 2 show how how the OO method works. Suppose the problem is

$$\min_{\theta \in \Theta} J(\theta) = \theta \qquad \Theta = \{1, ..., 9\} \tag{3}$$

The measured performance can be seen as Eq. (1) for the ideal performance plus noise.

The noise is assumed to be a random variable uniformly distributed in [0, 4].

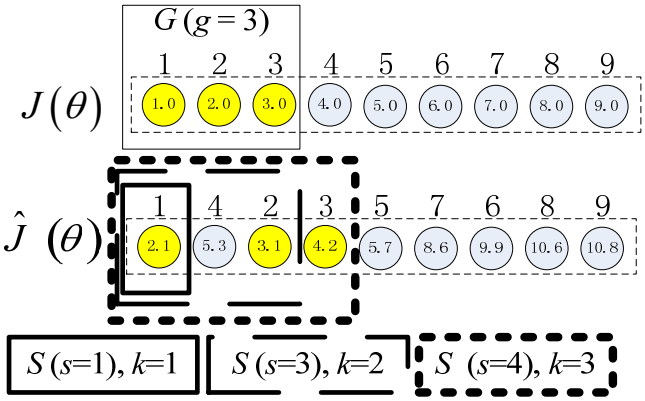


Figure 2. Example to illustrate how ordinal optimization works. The search space consists of 9 polices in ascending order. The good-enough set *G* is shown by the left (best) 3 (*g* = 3) yellow shaded policies. The first, second and third policy in the *measured performance* set (*s* = 1, 3, or 4) are selected to get at least 1, 2, or 3 policies (*k* = 1, 2, or 3) in *G*. Different sizes of *s* will give different *k*. The search is narrowed from 9

to 1/3/4 depending on the number of good-enough policies.

After generating selection set *S*, the ideal performance model is used to simulate the *s* policies to get the best policy. In this way, the simulation set is reduced from the original |Θ| to *s*, which reduces much computational time .

The size of *s* is determined by the size of good-enough set, *g*, the overlap number *k*, and success probability α. Two other concepts are also important.

Def. 7. ω type is used to define the distribution of all the policies. There are generally five ω types as shown in Fig. 3. Suppose $\{\theta_1, \theta_2, \cdots, \theta_n\}$ is the policy set Θ with their measured performance being $\left\{\hat{J}(\theta_1), \hat{J}(\theta_2), \cdots, \hat{J}(\theta_n)\right\}$. Their performances are then listed in an ascending order $\left\{\hat{J}\left(\theta_{[1]}\right), \hat{J}\left(\theta_{[2]}\right), \cdots, \hat{J}\left(\theta_{[n]}\right)\right\}$, with *x* is [0, *n*-1], and *y* is as:

$$\left[\hat{J}\left(\theta_{[i]}\right) - \hat{J}\left(\theta_{[1]}\right)\right] \Big/ \left[\hat{J}\left(\theta_{[n]}\right) - \hat{J}\left(\theta_{[1]}\right)\right] \quad (4)$$

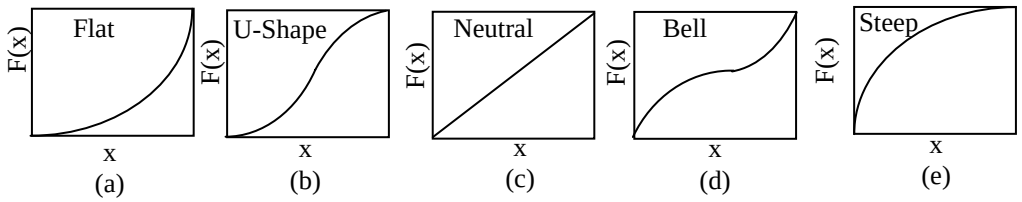


Figure 3. Five ω types based on the measured performance of each policy. X represent the x*th* smallest value of the measured performance while F(x) is the normalized value from Eq. (4)

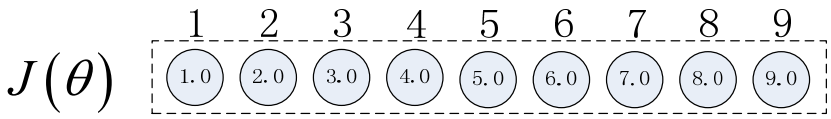
For a minimization problem, ω is like (a) in Fig. 3, then many policies are good since they are very small. The selection set *S* could then be relatively small and still guarantee many good enough results. Similar rules can be applied to the other figures to get relationships between the problem type and the selection set *S*.

Def. 7. **The Noise level (*NL*)** is another factor that affects the size of selection set *S*. *NL* describes how "rough" the rough model $\hat{J}(\theta)$ is compared with the true performance model $J(\theta)$.

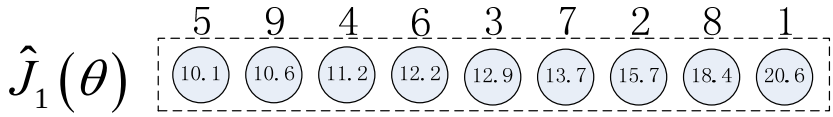
Again using Fig. 2 as an example, the standard deviation of the noise is 3. Thus *NL* = 3/(9-0) = 0.33, which

is a small noise level. The noise level can be calculated by normalizing the *measured performance* $\left\{\hat{J}(\theta_1),\hat{J}(\theta_2),\cdots,\hat{J}(\theta_n)\right\}$ into [-1, 1], and finding the maximum standard deviation, σ. These can be categorized into: small noise levels ($0<\sigma\leq 0.5$), medium noise levels ($0.5<\sigma\leq 1$), large noise levels ($1<\sigma\leq 2.5$) and very large noise levels ($\sigma>2.5$).

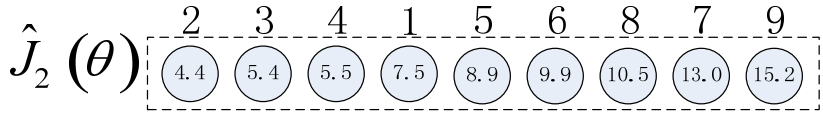
A small noise example is given in Fig. 2, Three other noise levels are illustrated in Fig. 4with the noise on a continuous interval [-3, 3].



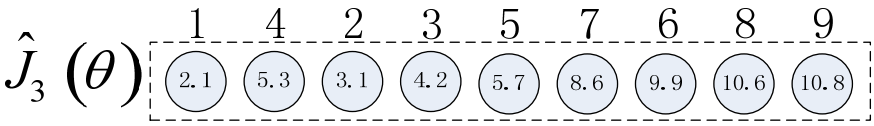
(a) Ideal performance



(b) Large noise, noise is i.i.d [0,20] and the noise level is 2.5



(c) Medium noise, noise is i.i.d [0,8] and the noise level is 1



(d) Small noise, noise is i.i.d [0,4] and the noise level is 0.5

Figure 4. Noise level can affect on the measured performance. (a) ideal performance (values in the circles) and their order (figures above the dashed rectangular box). (b) measured performance with large noise with uniformly distributed random noise between [0,20]and a noise level of (20-0)/(9.0-1.0)=2.5. The number above values show are original order. In this example, the best policy (1.0 in (a)) becomes the worst (20.6 in (b)) after the large noise

evaluation. Similar rules are applied in (b) and (c). The disorder in (d) is intuitively less than that in (b) and (c).

The characteristics of the autotuning configuration problem perfectly match the OO characteristics. Each configuration parameter θ would require a time consuming simulation to get the performance, which is not practical with a large search space. Finding the best θ^* in the good-enough set G, perhaps the top 2%, would be much easier. The OO method guarantees finding at least k policies in the good-enough set with a probability larger than α (e.g. 98%).

2. OO-based Configuration Parameter Autotuning

This section first introduces when and why OO is effective. Then, the neural network training samples is defined. Finally, a detailed OO procedure of is given to better illustrate how this method is used.

2.1 When and why OO works

The configuration parameter must be specified for the distributed system, for example, Apache + MySql servers or Tomcat + JBoss + Oracle servers. The parameters of the mainstream JSP/Servlet system are [MaxClients, MaxConnections, SessionTime, KeyBufferSize, MaxPoolSize], which typically have values between [10, 1000], [100, 500], [30 s, 200 s], [3 M, 500 M] and [10, 150]. Different scenarios may have different intervals and configuration parameters. The objective is to determine which configuration parameters are used and their value space for a specific scenario. The OO method can then be used to identify the proper configuration parameters to make the application run smoothly and efficiently.

The method can effectively to find good-enough configuration parameters because it digs out structural information to find the ranges for the good enough results, based on the ω type, noise level and neural network models. This rough model fast records the structural information to reduce the number of options.

The OO method greatly reduces computational cost because the neural network model is only rough, which precision loss to reduce the overhead, to further reduce the number of searches.

Rather than taking much time to search for the optimal policy like some heuristic method does, the OO method searches for a proper balance between performance and simulation time.

2.2 Number of training samples

The OO method reduces the very large search space, $|\Theta|$ to a much smaller set s. Since some configuration parameters, are continuous, such as SessionTime in [30s, 200s], a uniform and random sample in Θ is sufficient for the optimization

The problem is to set the training sample size, T, to enable the neural network model to represent the performance of the true model. Thus, the system needs to balance the neural network model effectiveness and the test time.

One extreme would do 1000 tests in a brute-force search of the entire space, Θ_N to get the best one, but this is quite time consuming. The other extreme would do only 1 real test, but resulting rough model is too rough to be use, even the time cost is small. Traditional evolutionary strategies, such as SHC, CMA are close to the brute-force end of the spectrum, while OO will towards the fast end. The test will illustrate how to decide how many tests are needed.

The good enough policy with OO method is very difficult to determine. This depends on how much test time is available. Neural network model rules, such as avoiding underfitting and overfitting, can be used to determine the number of training samples. In our experiment, The number was 40 initially with better values found in tradeoffs. The tests show what happens to the system performance if this number is changed.

2.3 Configuration autotuning procedure

Given n parameters $\{p_1, p_2, \cdots, p_n\}$ with value parameters p_i range of $[\underline{p}_i, \overline{p}_i]$, the OO method is then:

(1) For i = 1 : n, linearly quantize $\left[\underline{p}_i, \overline{p}_i\right]$ into [0, 144];

Map $p_i \in \left[\underline{p}_i, \overline{p}_i\right]$ to $\overline{p}_i + \frac{144}{\overline{p}_i - \underline{p}_i} p_i$.

(2) Randomly choose 1000 groups uniform *n* dimensional Gaussian distribution of configuration parameters, $\{\theta_1, \theta_2, \ldots, \theta_{1000}\}$, in $[0,144]^n$ and quantize p_i back to its original space $\left[\underline{p}_i, \overline{p}_i\right]$. Here θ_j represents the *n* dimensional vector $\left\{v_{1j}, v_{2j}, \cdots, v_{ij}, \cdots, v_{nj}\right\}$, $(j \in [1,1000])$.

Map $v_{ij} \in [0,144]$ to $\frac{\overline{p}_i - \underline{p}_i}{144} v_{ij}$.

(3) Use $\{\theta_1, \theta_2, \ldots, \theta_{40}\}$ as training samples. Test the real performance for each θ_t ($t \in [1,40]$)and record the cost performance as $\{c(\theta_1), c(\theta_2), \ldots, c(\theta_{40})\}$.

(4) Use the three-tier BP neural network model [8] shown in Fig. 5 to evaluate the *T* inputs, θ_t, with their related outputs $c(\theta_t)$. This rough model is computationally fast.

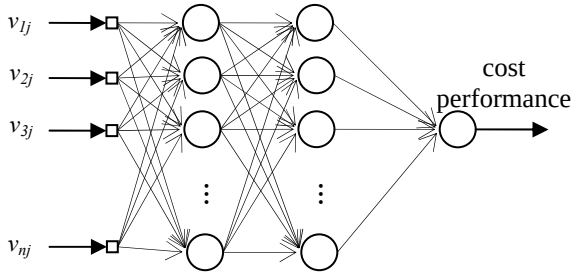


Figure 5. Three-tier neural network (rough model) for autotuning configuration parameters with *n* dimensional input vectors $\theta_j = \{v_{1j}, \ldots v_{nj}\}$ (j = 1…1000) for the *n* configuration parameters and the output as the cost performance. 40 training samples from real experiments results are used to train the neural network.

The BP neural network roughly learns the relationship between each configuration parameter, θ_t and its cost performance $c(\theta_t)$ by defining weight on each edge. The cost performances of the remaining samples (θ_{41} to θ_{1000}) are derived from the neural network output, so this is called rough model. These are denoted as $\left\{\hat{c}(\theta_l), l = 41,42,\ldots,1000\right\}$.

(5) Use the model to roughly evaluate the ω type (Flat, U-Shaped, Neutral, Bell or Steep). Simulation suggests that the majority of applications have a Bell shape, which implies that the good enough data distribution is neutral (neither too good nor too bad) for us to search. The effect of the Bell shape on the final selection set *S* is described in Ho, et al. [9].

(6) Specify the noise level. Normalize the observed performance in $\{\hat{c}(\theta_l), l = 41, 42, \ldots, 1000\}$ into [-1,1], and find the maximum standard deviation σ. The noise level is then found based on σ.

(7) Specify the size, *g*, of the good enough set *G*, the alignment level size, *k,* and the alignment probability α, as shown in Fig. 1.

(8) Use a look-up table [9] to calculate the size, *s*, of the selection set *S*. OO theory guarantees that *S* contains at least *k* good enough configuration parameter vectors with a probability no less than α.

(9) Find the best *s* θ_ls based on the rough model with the least cost performance based on the neural network model output.

(10) Use tests to evaluate the *s* θ_l to get the best vector for configuration parameters.

These steps greatly reduce the computational cost from Θ to Θ_N and then to *s*.

3. Performance Evaluations of Actual Systems

3.1 System testbed

Tests of an actual running system used a typical mainstream three-tier JSP/Servlet system, using the MVC framework. This system is used to organize different workflow tasks in a company to corporate with each other in task related events. The first tier is a web (HTTP/HTML) server to handle requests from remote clients, using Apache servers. The application server, Tomcat, has a controller component to identify the request type and target the 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 invoking process, a MySQL database system is also

used to manage the relational data tables, the contents file and documents. The system structure and workflow are illustrated in Fig. 9.

The system has 5 actions categories. Each user has to login only once, with all of the use is listed. After that, the tasks are selected and executed and the system returns the result. The user then returns to the task list and continues the procedure. The workflow is simplified by having each user repeat the process five times before logging out.

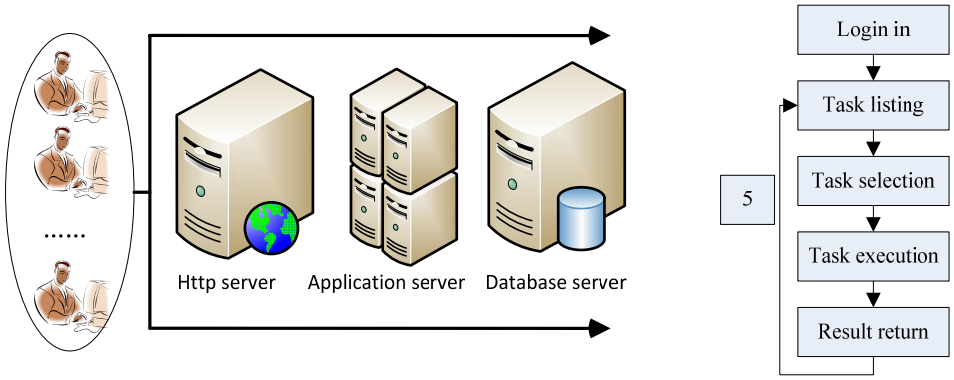


Figure 9. Test system structure and workflow

Seven parameters are used to represent the system performance, [MaxKeepAliveRequests, KeepAliveTimeOut, ThreadCacheSize, MaxInactiveInterval, MaxConnections, KeyBufferSize, SortBufferSize]. All these parameters impact system performance when the system is saturated. The default values are X^0 = [100, 5 s, 8, 2 s, 400, 20 M, 256 K]. Parameter spaces are [10, 200]×[10 s, 200 s]×[5, 100]×[5 s, 50 s]×[100, 500]×[8 M, 256 M] and [128 K, 1024 K]. The parameter spaces were chosen for this specific application with some default settings not be necessarily in between the interval in the software. Some of the settings depend on the application type, such as ThreadCacheSize and MaxInactiveInterval. Some of the settings depend on the intrinsic characteristics of the server, such as the SortBufferSize which is dependent on MySQL. All these configuration parameters are inter-related so the optimal set is difficult to find.

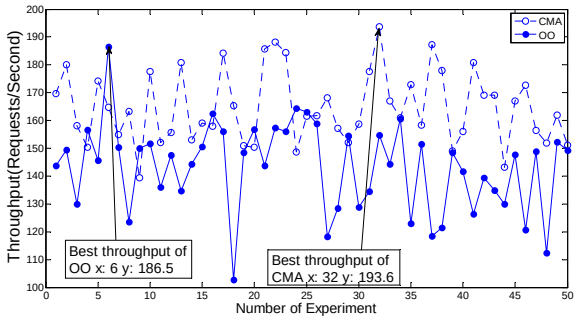
3.2 Test results and analysis

Neural networks was used with 40 groups of input/response pairs to generate the rough model. The method described in Section 3 was used to estimate the noise level, which is quite high (the large noise level) due to many unpredictable factors in the dynamic system behavior. The Bell curve ω type was used to get a good-enough set g with 100 vectors (actually, with many more because of the large search space), with the required alignment level k equal to be 5 and the alignment probability α as 98%. Those parameters were set before the tests to give a selected set S size of 20.

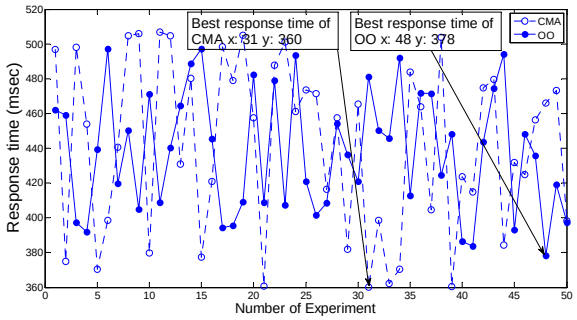
Probability is $P\left[\left|G_\theta \cap S\right| \geq 5\right] \geq 98\%$.

The tests show the impact on three performance metrics, throughput (average number of requests processed per second), response time (average time to finish a servlet execution) and cost performance. 410 virtual users were used to saturate the three-tier system. To make the test more applicable to a real scenario, the workload was not added simultaneously. Instead, the workload was gradually increased and lasted for a specific period of time (60 seconds) after each user was initiated. The performance results are illustrated in Fig. 10. with the default parameters.

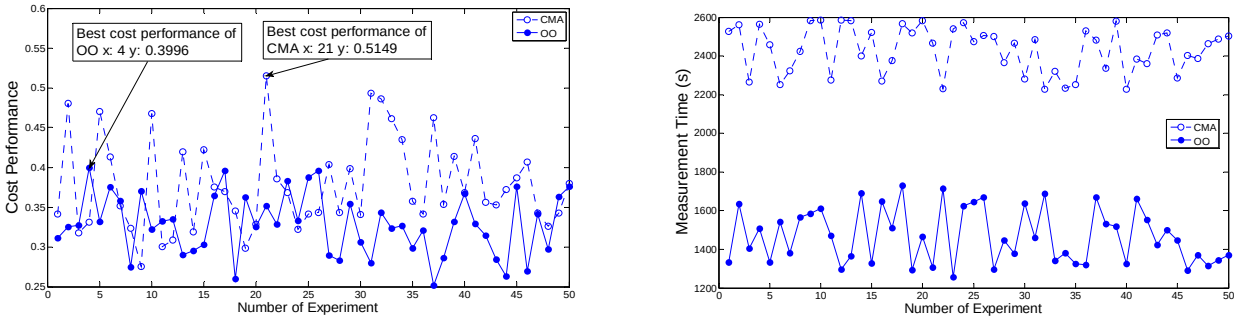
- Default throughput $t(X^0)$ (requests/sec): 149.21
- Default response time $r(X^0)$ (msec): 501.8
- Default cost performance $c(X^0)$: 0.2695



(a)



(b)



(c) (d)

Figure 10. Performance with the OO and CMA systems in (a) throughput, (b) response time, (c) cost performance and (d) measurement time.

The CMA method had the best cost performance in the 21st test. For X^{CMA} = [126, 28.19, 16, 29, 89, 136, 323.51]^T, which resulted in a throughput, $t(X^{CMA})$ = 185.73 (requests/sec) and cost performance $c(X^{CMA})$ = 0.5149, 24.5% and 91.1% better than the default setting. The response time, $r(X^{CMA})$ = 361 (msec), which decreased default setting by 28.1%. OO optimal result was in test 4th for X^{OO} = [194, 46.28, 46, 35, 101, 188, 445.95]^T. The throughput was $t(X^{OO})$ = 156.6 (requests/sec) and the cost performance was $c(X^{OO})$ = 0.3996, a 5.0% and 48.3% increase compared with the default setting. The response time was reduced by 21.9% to $r(X^{OO})$ = 392 (msec).

A comparison of CMA and OO method shows the CMA throughput is 15.7% better, response time is 8.0% better. The OO method tends to explore more of the estimated best value space with the BP neural network model used here, the performance is strongly determined by the accuracy of the rough neural network model. On the contrary, the CMA method tries both exploration and exploitation, so it more easily picks out better configuration settings in the search space. The tradeoff then is that CMA gives a slightly more optimal result at a cost of a longer simulation time, while the OO method gets a reasonably optimal results in a dramatically short time. The test times are illustrated in Figure 10(d).

The measurement time comparison used the (4,9)-CMA algorithm, which on average required five generations to arrive at the optimal result, which is also reasonable for evaluating the simulation time. The time

consumed of the OO method is much less than CMA. The OO testing time includes establishing the neural network rough model, defining the ω type and noise level and looking up the table to get the selection set *S*. Actually these parts can be neglected since they are done only once before the tests. Thus time depends only on the selected set, *S*.

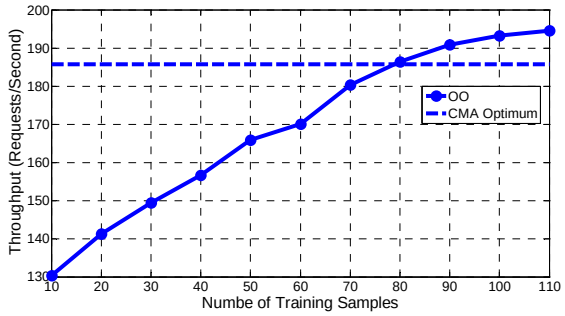
The average measurement time with CMA was $t(X^{CMA}) =$ 2432.25 (s) compared to only $t(X^{OO}) =$ 1470.43 (s) with OO. The average OO time was 60% less than CMA. Thus, the OO method gives a significant much reduced measurement time for autotuning the configuration parameters.

3.3 Test results with different number of training samples

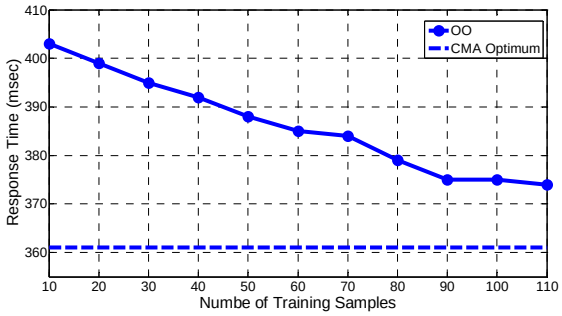
The number of training samples for the neural network rough model is quite important. If this number is too large, the system will find more information about good policies at the cost of more simulation time.

The BP neural network in the previous tests was 40 training samples. This number will affect the OO performance. Thus, different numbers of training samples *s* were used to show their impacts on the throughput, response time, cost performance and most importantly, the test time. The results are also compared with the CMA method.

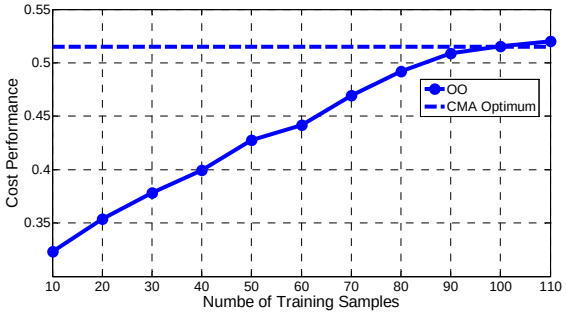
110 tests were made using the OO method in the way as the previous section. Additional tests did not lead to noticeable performance improvements. More tests would require more measurement time, which reduces the advantage of the OO method. All the tests used the same configuration parameters set and values set $\{\theta_1, \theta_2, \ldots, \theta_{1000}\}$. The test time was reduced by uniformly sampling a number of training samples (10, 20, etc).



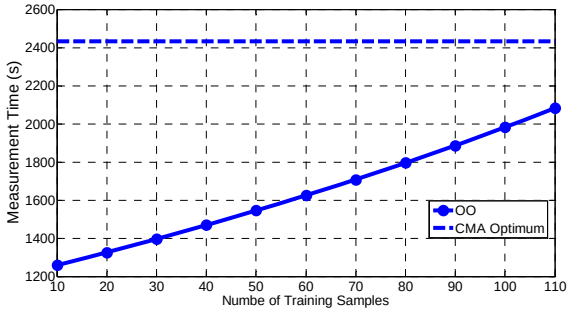
(a)



(b)



(c)



(d)

Figure 11. Impact of neural network training samples on (a) throughput, (b) response time, (c) cost performance and (d)measurement time. With increase numbers of training samples, the OO performance is better at the cost of more measurement time. The CMA optimum is the best result.

The results of Fig. 11 show the use of more samples to train the BP neural network finds a better configuration parameter set because the method goes deeper into the structural information, which compensates for the drawback of the OO method. However, the measurement time increase is almost linear with the number of samples. So the number of training samples depends on measurement time is available. At same number of training samples, 110 in these tests, the throughput, response time and cost performance didn't improve more.

4. Conclusions and Future Work

This paper describes an ordinal optimization based strategy to improve the performance with less measurement time for autotuning configuration parameters. This method significantly reduces the measurement in real distributed systems with a slight optimization performance decrease compared with traditional evolutionary optimization strategy. Future work can be categorized into four directions.

The "no free lunch theorem" suggests that the algorithm needs to carefully search the intrinsic characteristics of the autotuning configuration problem to dig out more information to support the decision-making. The iterative rough model described is an important extension of the OO algorithm.

Web servers have too many parameters to be thoroughly searched. Some of these are application dependent

thus limiting traditional optimization methods. The configuration parameters can be selected according to the application type for specific scenarios.

There have been many extensions of the OO method introduced in related work. Some, such as Optimal Computing Budget Allocation (OCBA) [10, 11, 12, 13] and Breadth & Depth (B&D) [14], can be used together to further reduce the test time. Vector Ordinal Optimization (VOO) [15] is also effective in dealing with multiple objective problems and other performance metrics.

The number of training samples used here was specified based on experience and some rules of thumb for neural networks, such as to avoid over fitting or under fitting. These could be further improved for the characteristics of distributed computing systems to further reduce the measurement time.

Acknowledgement

Junwei Cao thanks Professor Erik Katsavounidis of the MIT LIGO Laboratory for his collaboration support on the LIGO gravitational-wave research. Fan Zhang was supported by 2010-2011 and 2011-2012 IBM Ph.D. Fellowship.

References

[1] Hwang K, Xu Z. Scalable Parallel Computing. chubandi:McGraw-Hill, 1998.

[2] Jia Q, Zhao Q. A SVM —— based method for engine maintenance strategy optimization. In: Proceedings of the 2006 IEEE International Confrence on Robotics and Automation. Orlando, FL, USA. 2006:1066-1071.

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

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

[5] Boss G, Malladi P, Quan D. IBM high performance on demand solutions, danwei. Oct. 2007.

[6] Xi B, Liu Z, Raghavachari M. A Smart hill-climbing algorithm for application server configuration. In: Proceedings of the 3rd International Conference on World Wide Web. New York, NY, USA. 2004: 287-296.

[7] Saboori A, Jiang G, Chen H, Autotuning configurations in distributed systems for performance improvements using evolutionary strategies, in Proceedings of the 28th International Conference on Distributed Computing Systems, Beijing, China, 2008:765-772.

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

[9] Ho Y C, Zhao Q C, and Jia Q S. Ordinal Optimization, Soft Optimization for Hard problems, Springer, 2007.

[10] Teng S Y, Lee L, and Chew P, Integration of Indifference-zone with Multi-objective Computing Budget Allocation, European Journal of Operational Research, 2010, 203(2): 419-429.

[11] He D H, Lee L, Chen C H, Fu M, and Wasserkrug S, Simulation Optimization Using the Cross-Entropy Method with Optimal Computing Budget Allocation, ACM Transactions on Modeling and Computer Simulation, 2010, 20(1):1-22.

[12] Hsieh B, Chen C H, Chang S H, Efficient Simulation-based Composition of Dispatching Policies by Integrating Ordinal Optimization with Design of Experiment, IEEE Transactions on Automation Science and Engineering, 2007, 4(4):553-568.

[13] Chen J and Lee L , A multi-objective selection procedure of determining a Pareto set, Computers and Operations Research, 2009, 36(6):1872-1879.

[14] Chen C H, Lin Ji W, Yücesan E, Chick S, Simulation Budget Allocation for Further Enhancing the Efficiency of Ordinal Optimization" Journal of Discrete Event Dynamic Systems: Theory and Applications, 2000, 10:251-270.

[15] Li D, Lee H, Ho Y C, Vector ordinal optimization – a new heuristic approach and its application to computer networks routing design problems, International Journal of Operations and Quantitative Management, 1999,5:211-230.