Constructing a Non-Linear Model with Neural Networks for Workload Characterization

Richard M. Yoo[1], Han Lee[2], Kingsum Chow[2], and Hsien-Hsin S. Lee[1]
[1]School of Electrical and Computer Engineering
Georgia Institute of Technology, Atlanta, GA 30332
[2]Middleware Products Division, Software and Solutions Group
Intel Corp., Hillsboro, OR 97123
{yoo, leehs}@ece.gatech.edu, {han.lee, kingsum.chow}@intel.com

Abstract

Workload characterization involves the understanding of the relationship between workload configurations and performance characteristics. To better assess the complexity of workload behavior, a model based approach is needed. Nevertheless, several configuration parameters and performance characteristics exhibit non-linear relationships that prohibit the development of an accurate application behavior model. In this paper, we propose a non-linear model based on an artificial neural network to explore such complex relationship. We achieved high accuracy and good predictability between configurations and performance characteristics when applying such a model to a 3-tier setup with response time restrictions. As shown by our work, a non-linear model and neural networks can increase the understandings of complex multi-tiered workloads, which further provide useful insights for performance engineers to tune their workloads for improving performance.

1 Introduction

Workload characterization [1, 8, 9] is a process of identifying and characterizing the intrinsic properties of an application in terms of quantifiable measures. It shows how a workload behavior responds as its execution environment such as hardware, operating system, libraries, etc. changes over time. With detailed understandings of their intricate relationship, one can use such information to guide performance optimizations. Nevertheless, as the complexity of a computer system and its workloads increases, quantitatively analyzing the performance behavior of workloads has become very difficult, if not entirely impossible. The introduction of managed runtimes like Java and C# further aggravated the situation. Due to its layered structure of runtime stack, the analysis of these workloads is inherently complicated.

To manage such insurmountable analysis tasks in a more effective manner, a model-based approach is indeed needed in order to simplify and systemize the analysis work. In the abstract level, a model is a multivariate relation between the controllable parameters and the performance indicators. Constructing this model then amounts to approximating this relationship from the collected performance data. With an accurate, working model, we can then analyze the workload in a continuous fashion, being able to predict how the performance metrics will change as the input parameters change.

Difficulties in approximating this relation mostly come from the presence of nonlinearity in program behavior. The performance of an application does not necessarily improve or degrade in a linear fashion in response to a linear adjustment in its input parameters.

Due to their simplicity, prior research works usually relied on linear models to approximate program behavior [2, 20,21]. Some linear models do demonstrate a remarkable accuracy. For example, in [2], Chow *et al.* introduced a linear model to describe the performance behavior of a seemingly complex web application and also validated the model with regression statistics and average prediction error. To successfully approximate a non-linear behavior with a linear model, however, may not always be possible.

To address the shortcomings, in this paper, a new methodology based on *artificial neural network* for constructing a non-linear model is proposed. Via the use of neural networks, our performance model can derive intrinsic relationship during workload characterization. Since neural networks do not make any assumptions for the functions we try to approximate and in fact that they can approximate any non-linear function, it can lead to a more general framework that is applicable to various types of workloads.

As a case study, we applied the construction of an artificial neural network to build a performance model for a 3-tier web service workload. This model investigates the relationship amongst programs configurations and workload performance characteristics, providing a substantial assistance to the performance tuning efforts. The model is shown to be very accurate and reliable. We will detail the methodology, limitations and advantages later in this paper.

In summary, the contributions of this paper are the following.

- We propose and demonstrate a neural network based non-linear model approach for characterizing and analyzing workloads.
- Out method provides performance tuning guidance by exploiting model prediction.

The rest of the paper is organized as follows. Section 2 gives background knowledge that underlies in this study. Section 3 details our methodology. Our experiments are based on the settings in Section 4, and the results are presented in Section 5. A review of related workload characterization works is given in Section 6. Finally, we conclude in Section 7.

2 Background

2.1 Artificial Neural Networks

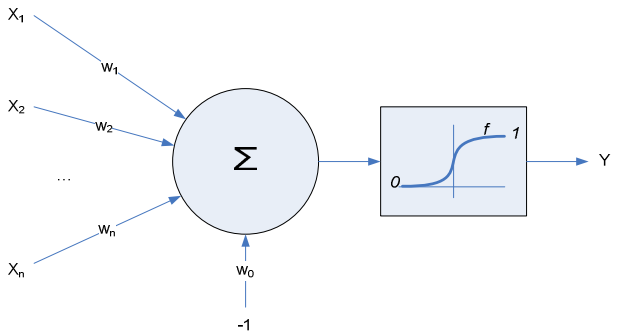

Figure 1: A Perceptron

An artificial neural network (ANN) [3, 4, 5] consists of a large number of small computational elements, resembling to biological neural networks. The building block inside an ANN is called a perceptron (or neuron.) A typical perceptron is illustrated in Figure 1. The perceptron computes a weighted sum of its (n + 1) input signals, and then passes the result through a nonlinear activation function. Mathematically, the output of a perceptron can be represented as

$$y = f(\sum_{i=1}^{n} w_i x_i - w_0)$$

where f is a non-linear activation function, w_i is the weight associated with the i^{th} input, and w_0 is a constant threshold (or bias) value.

The activation function is also referred to as a squashing function; It limits the amplitude of the output of a perceptron to a closed interval, e.g. [0, 1]. The operation of this function can be interpreted as a mapping from a linear space to a non-linear space. Note that the non-linear characteristic of an ANN comes from this non-linearity of the activation function. By far the most common form of an activation function is a *sigmoid* function, which is defined as a strictly increasing function that exhibits smoothness and asymptotic properties. One example of a sigmoid function — logistic function, is defined as follows:

$$f(x) = \frac{1}{1 + \exp(ax)}$$

where *a* is the slope parameter and is used to determine the fuzziness of the decision boundary.

The behavior of this function is illustrated in Figure 2. The function approaches a hard limiter as the absolute value of the slope parameter increases.

Basically, a perceptron forms a hyperplane to bisect the sample space; those that have a weighted sum of less than the threshold produce an output 0 and the others produce an output 1. The weights define the orientation of this hyperplane and the bias determines the offset of the plane from the origin.

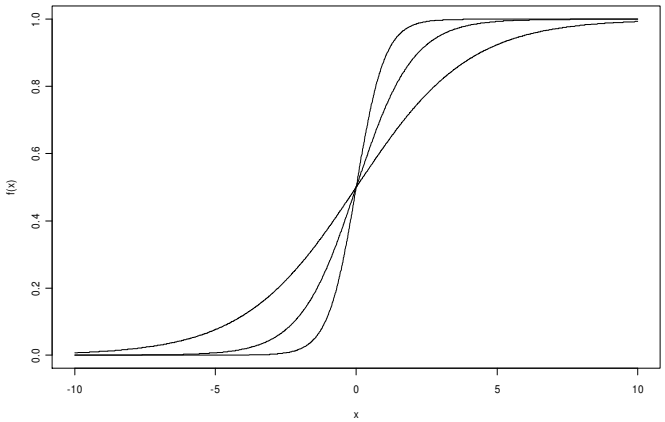

Figure 2: A Sigmoid Function

ANNs can be classified into a variety of models depending on the topology of network, the activation function used, and the training algorithm. Among them, each individual model can be designed for a particular purpose with its own application where it can excel. In the function approximation area, single or multilayer perceptrons and Radial Bases Function (RBF) networks are used.

2.2 Multilayer Perceptrons

Among those that are used for function approximation, multilayer perceptrons (MLPs)[1] are the most popular types. Figure 3 shows the typical topology of an MLP mapping an n dimensional space into an m dimensional space.

An MLP consists of one input layer and one output layer, but can have multiple hidden layers in-between. The MLP shown in Figure 3 contains two hidden layers and is a three layer perceptron for the input layer usually is not considered a layer. Note that each circle in these 3 layers represents a perceptron including a summation unit and an activation function.

Each perceptron in the first hidden layer creates bisectors in the sample space. Then the second hidden layer is tuned to perform logical AND operations to the sections created by the input layer. For example, if all the weights from n nodes in the first hidden layer to a node in the second hidden layer are 1, then setting the threshold in the second hidden layer node to n – ε where $0 < \varepsilon < 1$ corresponds to an AND operation since

[1] In this paper MLP always stands for Multi-layer Perceptrons, not Memory Level Parallelism.

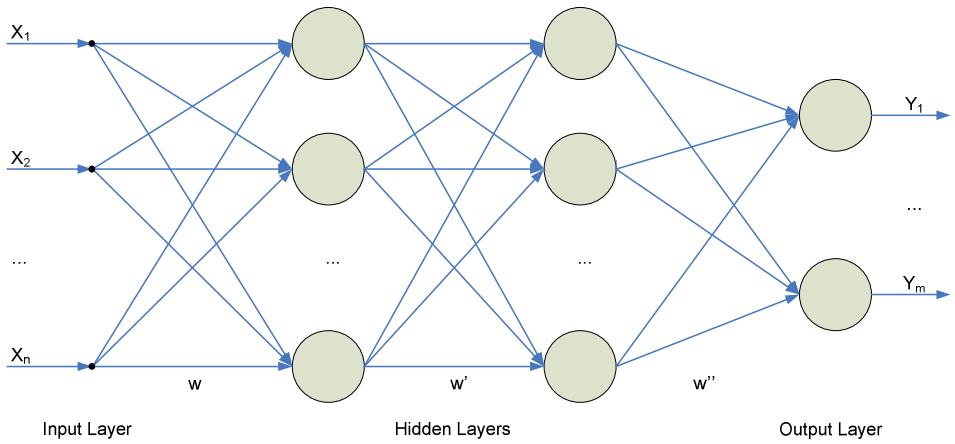

Figure 3: Multilayer Perceptron

the output will be 1 only when all the outputs from the first hidden layer are 1. Usually 2n perceptrons are needed to create a confinement in an n dimensional space. The output layer then performs OR operations to those confinements produced from the second hidden layer. Setting the threshold in the output layer node as 0.5 will suffice if all the weights from the second hidden layer are 1. Since any finite volume could be approximated as a sum of many confinements, MLPs can carve the sample space in an arbitrary manner with 3 layers.

It has also been shown that these networks can approximate any continuous function to a desired degree of accuracy [7]. Nevertheless, real world performance can be limited from the inordinate number of node count, which in turn requires large amounts of sample data and training time. Due to its widespread use and availability in software, we chose MLP to construct our non-linear model.

MLPs are trained with samples. In our case a sample is represented as a tuple

$$(X,Y) = (x_1, x_2, ..., x_n, y_1, y_2, ..., y_m)$$

where $X = (x_1, x_2, ..., x_n)$ stand for n parameters in the configuration of a given workload, and $Y = (y_1, y_2, ..., y_m)$ stand for m performance indicators collected by running an application under the configuration X. The configuration parameters should be those we choose to measure their effect on the overall application performance; e.g., JVM heap size, thread pool size, injection rate, etc. A performance indicator can be any performance metric, e.g., response time, throughput, etc.

A set of training samples are collected by running the identical application under various configurations; each sample amounts to one specific configuration and the performance of the application under the configuration. Each time a training sample is presented, an MLP predicts the performance indicator value $\tilde{Y}$ by examining the configuration setting X. Training algorithms then tune the weights and biases with a goal to minimize the error between the predicted value ($\tilde{Y}$) and the actual value (Y), i.e. $\|\tilde{Y} - Y\|$. This process is repeated over all the training samples until a desired error threshold is met. Thus learned knowledge is kept in MLPs by memorizing their weights and biases. When an unseen workload configuration $\tilde{X}$ is presented, MLP can predict the performance value based on the prior knowledge.

Among various training methods, a gradient descent based back-propagation method is by far the most popular [5]. Details of the method are outside the scope of this paper. Interested readers are referred to the literature [5].

3 Constructing a Neural Network Model

In this section we discuss the details concerning the construction of the neural network model. Sample pre-processing, discussed in Section 3.1, is a data manipulation process required for maximizing the model accuracy. This process should be applied to sample collection before they are used to build the model. Section 3.2 then explains the model parameters and settings that are required during the training process. Section 3.3 describes how to maintain the model flexibility for unseen samples, and its correlation with model validity.

3.1 Pre-Processing the Samples

As described in Section 2.2, a neural network is trained with a set of samples. Constructing a sample collection is a necessity. One set of samples should be prepared for each application to characterize. After the samples are collected, they need to be processed in order to bring about the best accuracy of the constructed model.

For configuration parameters $X = (x_1, x_2, ..., x_n)$, each parameter must be standardized. By standardization we mean the process of subtracting the mean and then dividing it by the standard deviation of a feature. This results in a feature of which the mean is 0 and the standard deviation is 1. This process is crucial to avoid the possibility of MLPs ending up in a local minimum. Since the back-propagation method is based on a gradient descent approach, without proper caution it is highly likely to stop at a local minimum, missing the global minimum and leading to a model that fails to fit the samples in a global manner.

More specifically, this attributes to the fact that the weights and biases of the network are initialized with random values when the training process begins. Each perceptron then shoots a hyperplane against a cloud of samples to cut it into half. However, when the input values to perceptrons are with their usual magnitude, the initial distance of the hyperplane from the origin may be too small to cut through the cloud; the plane then misses all the samples. This tends to lead the system to a local minimum.

Standardizing the performance indicators $Y = (y_1, y_2, ..., y_m)$ depends on the approximation task. If we only approximate one performance indicator, there is no need to standardize. On the other hand, when approximating multiple performance indicators at the same time, we might as well standardize those performance indicators. This is also due to the fact that we are using a gradient based training method. When one of the performance indicators has a higher magnitude compared to the others, MLPs will spend most of the time to fit that indicator since it will produce more gradient. This has the effect of ignoring small variations from the other indicators.

3.2 Choosing Model Parameters

In choosing model parameters, the first question will be how many MLPs should be used. Note that an n configuration parameter to m performance indicator approximation can be done by approximating m instances of n configuration parameter to 1 performance indicator relation. However, although the prediction accuracy will suffer to a small extent, we opt to approximate each workload with 1 instance of n-to-m relation in the belief that it will model the synthetic behavior of the application more accurately. The next question will concern the node count in the hidden layers of MLP. When it comes to this question there seems to be no definite answer. In [6], the authors mention that the node count depends on the following:

- The numbers of configuration parameters and performance indicators
- The number of training samples
- The amount of noise in the collected samples
- The complexity of the workload to be learned
- The structure of the MLP
- The type of the activation function
- The training algorithm

It depends on the complexity of the distribution of the samples which in turn determines the complexity of the function. Each node in an MLP can be thought of as a point that pins down the function in a virtual space. This notion gives us a rough order of nodes that are needed.

3.3 Maintaining Model Flexibility and Validity

Theoretically, MLPs can approximate a function to any precision level. However, in a modeling problem, if we approximate a function too much, the model will lose its flexibility to adapt to unseen data. This phenomenon is called *overfitting*. It is better to loosely fit to the training sample to maintain the flexibility of a model. A threshold value is needed to indicate when to stop training.

Flexibility is also highly correlated with the validity of the model. The more flexible the model is, the lower the prediction error gets for unseen samples; which also mean that the model is valid over a wider range of samples. Prediction error over unseen samples can be used to quantify the validity of the model. Toward this we used *k-fold cross validation* [22]. In k-fold cross validation, a training set is divided into k sets of equal size. Then the model is trained for k times. For each trial, one set is excluded from those k sets; k - 1 sets, called training set, are used to train the model, and the excluded set, termed validation set, is used to calculate the error metric for the model. Thus collected error values are then averaged over k trials. For error metric, harmonic mean of (absolute error) / (actual value) is used.

4 Experiment Settings

To demonstrate the validity and usefulness of our methodology, we applied the method as an aid to application performance tuning.

The workload used is a 3-tier web service modeling the transactions among a manufacturing company, its clients and suppliers. The workload is composed of a driver to inject the load to the system, a middle-tier server running a leading commercial Java application server[2] and a backend database system to provide data storage. Both the driver and the database server are not CPU-bound. The performance characteristics of our study focus on the middle-tier application server. Inside the application server, different *thread counts* can be assigned to three different queues modeling the work flow including an *mfg queue* that models the manufacturing domain, a *web queue* for modeling the web front end, and a *default queue* which handles the rest. These are considered to be our input parameters. The selection of these thread pool sizes was found to be difficult and sometimes not quite intuitive to the experienced performance engineers. The performance measurements demonstrate non-linearity when we varied the thread pool size. Another input parameter, denoted as *injection rate*, is the rate of requests injected to the software system. Overall, there are four input parameters in this application including the thread counts assigned to the mfg queue, web queue, default queue, and the injection rate.

In this model, there are five performance indicators. The first four indicators are designated by the workload itself that specifies four response time constraints. They include *manufacturing response time, dealer purchase response time, dealer manage response time*, and the *dealer browse autos response time*. In addition, the throughput, i.e., *effective transactions per second*, is also included as one performance indicator. In summary, our model amounts to a 4-input, 5-output relation.

[2] The name of the application cannot be disclosed due to commercial confidentiality.

The system hardware used in our experiment is listed in Table 1. As the workload has a steady state behavior, the averages of collected counter values are used to reduce the effect of sampling error.

With the constructed model, *5-fold cross validation* was performed. The MLP node count and the termination threshold were manually tuned for the first trial; then the next four trials were generated automatically with the same node count and the same threshold value.

Table 1: Experiment Hardware Settings

CPU	4 Intel® Xeon® dual core 3.4 GHz with Hyper-Threading enabled
L2 Cache	1 MB per core
Memory	16 GB

After validating the model, we drew 3D diagrams of the model to demonstrate and analyze the workload behavior. Note that this model pertains to the specific combination of our experimental systems and the workload.

5 Experimental Results

With data samples collected from the experiments described previously, we modeled the mapping from application configuration to macroscopic application performance indicators to provide aids to the application performance tuning process.

Even though our approaches are quite straightforward, our model is shown to be very accurate. For the workloads, we use a training set and a validation set for our evaluation. The training set was used to train our neural network model while the validation set was used to evaluate the effectiveness based on the training outcome. Using the training set, Figure 5 shows the predicted values of one of the 5 trials based on the 5-fold cross validation. On the other hand, Figure 6 shows the values predicted for the validation set after our model was trained by the training set. In these figures, each "**o**" denotes the actual value while each "**x**" denotes the predicted value; each sample index corresponds to a specific application configuration. For both figures, all 5 plots were generated from the same trial. The first 4 plots show the four response times respectively (shown in the titles of Y-axis) while the last plot at the bottom shows the effective number of transactions (throughput). As can be seen from Figure 5, the MLP is loosely fit to the training set on purpose to avoid overfitting. As we explained earlier, such fitting leaves some certain flexibility to the model and guarantees smaller prediction errors.

The average prediction errors for the validation set are given in Table 2. The errors for the performance indicators are small, ranged from 0.2% to 10%, achieving an overall average prediction accuracy of 95%. Note that only the first trial was hand tuned, and the rest is generated automatically. If we hand tuned each trial, the error could have been reduced even more.

In application tuning process, it is important to find the best configuration that delivers the best application performance. The next step will be to minimize the test cases to reduce the amount of heuristic effort. Now we will show how we perform an in-depth analysis using 3D diagrams of performance indicators predicted by our model. Based on the characteristics of the 3D diagrams of the workload, we classify their variation behavior into three categories: parallel slope, valleys, and hills and discuss them subsequently. These typical behaviors appeared repetitively across diverse configuration settings.

Note that each 4-tuple on the top of Figure 4, Figure 7, and Figure 8 stands for a tuple comprised of (*injection rate, default queue, mfg queue, web queue*). In this particular analysis we focus on (560, x, 16, y) case, which means that the figures were obtained by fixing the injection rate and the mfg queue thread count at 560 and 16, respectively, then adjusting the default queue and the web queue thread count. The Z-axis indicates the predicted performance indicator values. Moreover, those dots indicate the location of the actual data. They spread over (or under) the surface with the same accuracy described in Table 2. Lastly, each figure has been rotated to increase readability which resulted in different orientations of x and y axis amongst the figures.

5.1 Parallel Slopes

Figure 4 shows a particular example belonging to this type. In this case one of the configuration parameters does not affect the performance indicator value much once the values of the other parameters are fixed. For example, once the web queue is set to 18, the manufacturing response time maintains at value 4 regardless of the default queue sizes. From the tuning perspective, it means that it will be of no use if one attempts to tune the default queue to achieve a better manufacturing response time.

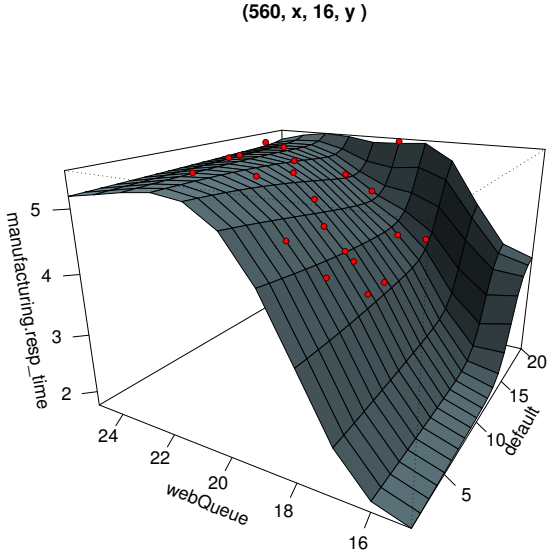

Figure 4: Case of Parallel Slopes

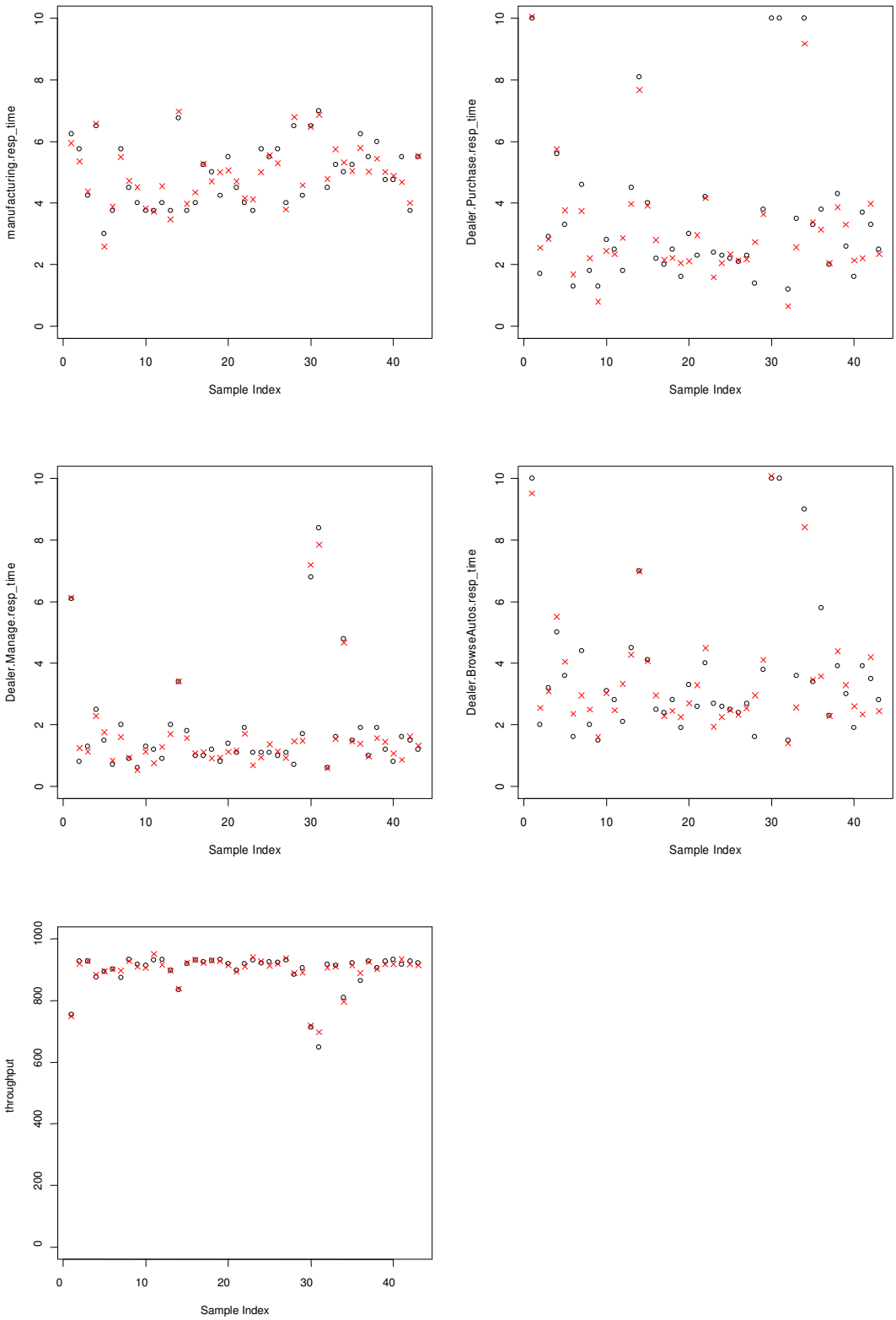

Figure 5: Actual (o) and Predicted (x) Values for the Training Set

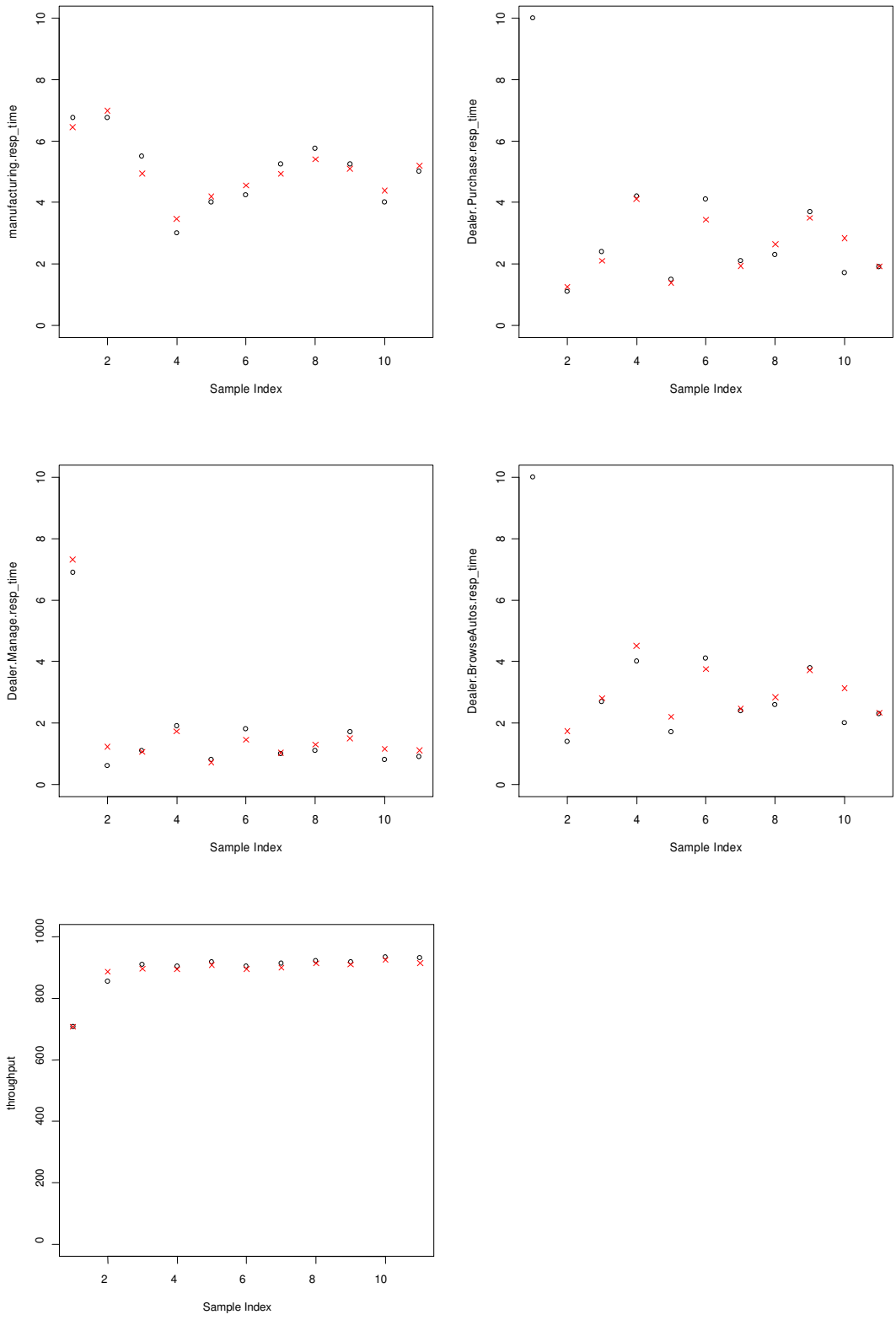

Figure 6: Actual (o) and Predicted (x) Values for the Validation Set

Table 2: Average Prediction Error for the Validation Set

Trial	Manufacturing Response Time	Dealer Purchase Response Time	Dealer Manage Response Time	Dealer Browse Autos Response Time	Effective Transactions per second
1	3.3 %	10.1 %	5.7 %	9.5 %	0.1 %
2	1.5 %	7.3 %	2.7 %	4.2 %	0.3 %
3	4.5 %	8.9 %	3.3 %	5.0 %	0.2 %
4	4.0 %	12.6 %	12.6 %	11.3 %	0.1 %
5	1.4 %	11.3 %	10.7 %	6.4 %	0.2 %
Average	3.0 %	10.0 %	7.0 %	7.3 %	0.2 %

5.2 Valleys

As shown in Figure 7, the Valley type of trends may be the cases that show the most dramatic benefit of using our model. Note that the meaning of the valley changes depending on the performance indicator category. For the response times it is better to have smaller values. Figure 7 shows this case. Note the valley formed from (default queue, web queue) = (0, 18) to (20, 20). In this figure the minimum dealer purchase response time could be obtained when we adjust two configuration parameters concurrently to stay in the valley. However, for throughput measures it is better to stay out of the valley.

Two other response time measures, i.e., dealer manage response time and dealer browse autos response time, also showed similar distribution.

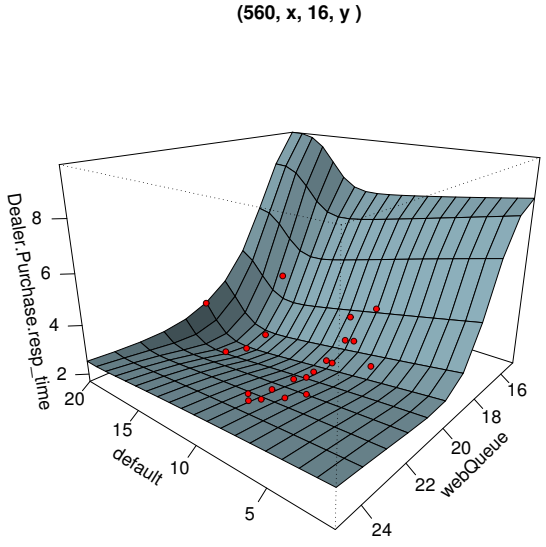

Figure 7: Case of Valleys

5.3 Hills

This type of plot characteristics is the opposite of valleys. In Figure 8, it is clear that the best throughput can be obtained only when the (web queue, default) = (20, 10). If performance engineers try to tune the throughput by varying the web queue while setting the value for default at 7, it is highly likely that they miss the local maximum regardless of how many experiments they perform. Without such modeling, a huge optimization effort will be futile.

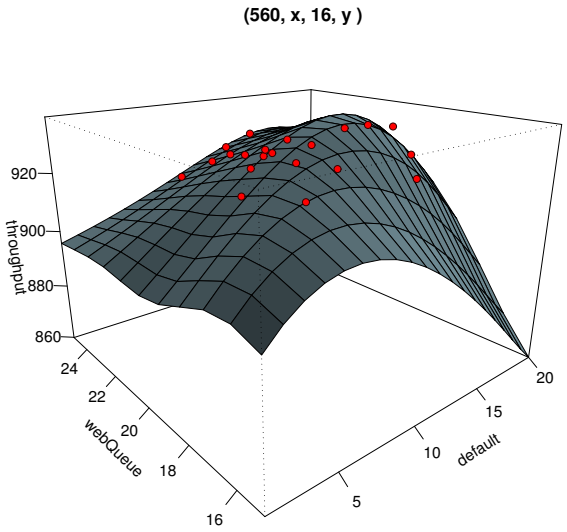

Figure 8: Case of Hills

As can be readily seen from these cases, our model can effectively narrow down the configuration combinations which we should concentrate, thus radically reducing ineffectual experiments. On the other hand, as we demonstrated, our model also provides a much larger scope for the overall performance trend, unveiling many otherwise lost opportunities for performance tuning. In addition, we can further build a system that recommends the best configuration according to a scoring function.

Regardless, it is hard to perform a quantitative analysis for a complete understanding of the individual contribution of a particular feature to the output. This could be attributed to the fact that neural network gives no assumption on the function it is approximating. By removing the assumption on the

underlying function, we are trading off the analytical power of the model for generality.

Moreover, neural network models cannot be used for extrapolation. That is, it cannot be used to predict the performance for the configuration that is far apart from the training data. The prediction accuracy of MLPs drop rapidly outside the range of training data. This is a known limitation of MLP, and researchers in neural network area proposed variants of MLPs [23] to overcome this limitation.

6 Related Work

Several prior works to characterize Java and object oriented application behavior are well described in [16,17,18]. Li *et al.* [16] utilize a full system simulator to incorporate operating system kernel behavior into workload characterization. Hauswirth *et al.* attempt to characterize a workload by collecting samples over vertical layers of the system execution stack [17,18]. Different from our work that highlights the workload performance variation across multiple executions with different configurations, these prior efforts were targeted to understand the chronological application behavior change in a single execution.

Previous thrusts on approximating a multi-tier workload with linear models can be found in [2,20,21]. These works attempted to train the model in the Design of Experiments (DOE) approach. First, a fixed order linear model is assumed, and the coefficients are then determined by a carefully designed set of experiments. Compared to their approach, our methodology is more general — (1) it does not make any assumption on the approximating function, and (2) it can readily construct a model from a rough mixture of data points.

This paper is in line with the researches applying advanced statistical methods to characterize computer workloads. Principal Components Analysis has been extensively used for Java workload characterization [10,11] and benchmark analysis and subsetting [12,13,14,19].

7 Conclusion and Future Work

In this paper we apply artificial neural networks to construct a non-linear program behavior model to characterize workload performance behavior. The main contribution includes (1) introducing the use of neural network models to derive intrinsic relationships in workload characterization, and (2) providing performance tuning guidance by exploiting our model prediction. Through case studies using real commercial application workloads, we show that our model can approximate non-linear workload behavior with an average prediction accuracy of 95%.

Now that we have better understandings in non-linear behavior of the workload, we can try to approximate it with other non-linear functions such as polynomial and logarithmic functions. As pointed out, neural network compromises the analytical power of the model for generality. By analyzing the behavior of the workload with prototype model constructed with neural networks, we will establish an analytic non-linear model that can fit the specific workload.

Acknowledgements

This research was conducted under the Software and Solutions Group Research Intern Program at Intel Corp. Intel Corp. generously supplied all the data and experimental equipments used in this work.

References

[1] L. K. John, P. Vasudevan, and J. Sabarinathan, "Workload characterization: Motivation, goals and methodology," in Workload Characterization: Methodology and Case Studies, pp. 3-14, November 1998.

[2] K. Chow, M. Bhat, and J. A. Davidson, "Minimizing performance test cases for multi-tiered software systems," in Proceedings of the Pacific Northwest Software Quality Conference, October 2002.

[3] A. K. Jain, J. Mao, and K. M. Mohiuddin, "Artificial neural networks: A tutorial," IEEE Computer, vol. 29, no. 3, pp. 31–44, 1996.

[4] R. P. Lippmann, "An introduction to computing with neural nets," IEEE ASSP Magazine, vol. 4, no. 2, pp. 4–22, April 1987.

[5] S. Haykin, Neural Networks: A Comprehensive Foundation. Macmillan College Publishing Company, Inc., 1994.

[6] "AI FAQ/Neural Nets." http://www.faqs.org/faqs/ai-faq/neural-nets.

[7] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," Neural Networks, vol. 2, no. 5, pp. 359–366, 1989.

[8] T. M. Conte and W. W. Hwu, "Benchmark characterization," IEEE Computer, vol. 24, no. 1, pp. 48–56, 1991.

[9] P. Bose, "Workload characterization: A key aspect of microarchitecture design," IEEE Micro, vol. 26, no. 2, pp. 5–6, 2006.

[10] K. Chow, A. Wright, and K. Lai, "Characterization of Java workloads by principal components analysis and indirect branches," in Proceedings of the Workshop on Workload Characterization, pp. 11–19, November 1998.

[11] L. Eeckhout, A. Georges, and K. D. Bosschere, "How Java programs interact with virtual machines at the microarchitectural level," in Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 169–186, October 2003.

[12] H. Vandierendonck and K. D. Bosschere, "Many benchmarks stress the same bottlenecks," in the 4th Workshop on Computer Architecture Evaluation using Commercial Workloads, January 2001.

[13] H. Vandierendonck and K. D. Bosschere, "Eccentric and fragile benchmarks," in Proceedings of the 2004 IEEE International Symposium on Performance

Analysis of Systems and Software, pp. 2–11, March 2004.

[14] L. Eeckhout, J. Sampson, and B. Calder, “Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation,” in Proceedings of the 2005 IEEE International Symposium on Workload Characterization, pp. 2–12, October 2005.

[15] L. Eeckhout, R. Sundareswara, J. J. Yi, D. J. Lilja, and P. Schrater, “Accurate statistical approaches for generating representative workload compositions,” in Proceedings of the 2005 IEEE International Symposium on Workload Characterization, pp. 56–66, October 2005.

[16] T. Li, L. K. John, V. Narayanan, A. Sivasubramaniam, J. Sabarinathan, and A. Murthy, “Using complete system simulation to characterize SPECjvm98 benchmarks,” in Proceedings of the 14th International Conference on Supercomputing, pp. 22–33, 2000.

[17] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind, “Vertical profiling: Understanding the behavior of object-oriented applications,” in Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 2004.

[18] M. Hauswirth, A. Diwan, P. F. Sweeney, and M. C. Mozer, “Automating vertical profiling,” in Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 2005.

[19] H. Vandierendonck and K. D. Bosschere, “Experiments with subsetting benchmark suites,” in Proceedings of the IEEE 7th Annual Workshop on Workload Characterization, pp. 55–62, October 2004.

[20] K. Chow, “Methods for e-commerce web performance prediction and capacity planning,” in Proceedings of the Intel Quality and Reliability Conference, October 2000.

[21] K. Chow and M. Bhat, “Methods for web performance prediction and capacity planning,” in BEA eWorld 2002.

[22] R. O. Duda, P. E. Hart, and D. G. Stork, Pattern Classification. 2nd ed., Wiley Interscience, 2000.

[23] J. W. Hines, "A logarithmic neural network architecture for unbounded non-linear function approximation," IEEE International Conference on Neural Networks, vol. 2, pp. 1245-1250 vol. 2, June 1996.