Using Computational Intelligence to Identify Performance Bottlenecks In a Computer System

Faraz Ahmed, Farrukh Shahzad and Muddassar Farooq

Next Generation Intelligent Network Research Center (nexGIN RC)
National University of Computer & Emerging Sciences (FAST-NUCES)
Islamabad, Pakistan
{faraz.ahmed,farrukh.shahzad,muddassar.farooq}@nexginrc.org

Abstract. System administrators have to analyze a number of system parameters to identify performance bottlenecks in a system. The major contribution of this paper is a utility – EvoPerf – which has the ability to autonomously monitor different system-wide parameters, requiring no user intervention, to accurately identify performance based anomalies (or bottlenecks). EvoPerf uses Windows `Perfmon` utility to collect a number of performance counters from the kernel of Windows OS. Subsequently, we show that artificial intelligence based techniques – using performance counters – can be used successfully to design an accurate and efficient performance monitoring utility. We evaluate feasibility of six classifiers – UCS, GAssist-ADI, GAssist-Int, NN-MLP, NN-RBF and J48 – and conclude that all classifiers provide more than 99% classification accuracy with less than 1% false positives. However, the processing overhead of J48 and neural networks based classifiers is significantly smaller compared with evolutionary classifiers.

1 Introduction

The pervasive penetration of Internet and associated next generation intelligent networks has resulted in great demand for e-commerce, gaming and e-health applications that must provide ubiquitous and instant access to its potential customers in a reliable and efficient manner. Currently, in most of the cases, system administrators themselves analyze and correlate a number of parameters – CPU usage, memory usage, network utilization etc. – to identify bottlenecks in a computer system [8]. A performance bottleneck can seriously compromise or undermine the functionality of a given business: unavailability of service due to a denial of service attack or crashing of a server process on account of low memory.

System administrators need diagnostic tools that can automatically identify bottlenecks on different server machines; as a result, they can efficiently invoke countermeasure strategies to gradually remove the bottleneck in the system. Therefore, in this paper, we propose a tool that can automatically monitor the system-wide performance of a computer and raise an alarm if the system is ex-

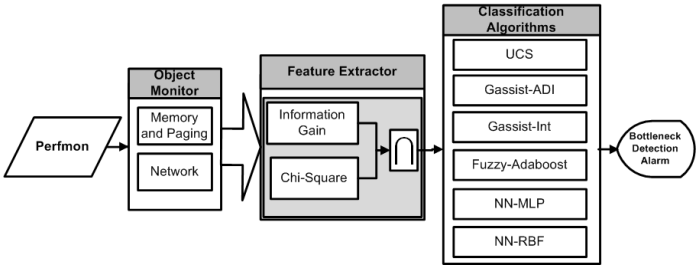


Fig. 1. Architecture of EvoPerf

periencing a bottleneck. [1] Our monitoring system consists of three sub-modules: (1) object monitor, (2) feature selector, and (3) classifier. An object monitor uses Windows `Perfmon` utility to collect a number of performance counters from the kernel of Windows OS. The task of feature selector is to use feature selection techniques to reduce the dimensionality of input space. Finally, the reduced features' set is given as an input to a number of classifiers that raise the final alarm.

2 Related Work

Monitoring the performance of a system in an automatic fashion is an active area of research. In [8], `SysProf` is presented that can monitor performance parameters of network applications at different granularity of time period. Moreover `SysProf`, also requires active user feedback to tune its different parameters and identify network related bottlenecks. Other tools like `Paradyn` [11] exist but they are only capable of analyzing the performance of application level programs. In comparison, our system is capable of identifying bottlenecks in a relatively large number of memory and network performance counters.

In [4] S. Duan et. al have presented a comparative study in which machine learning techniques (clustering, classification and regression trees and Bayesian networks) are empirically compared for identification of different system states, states comparison and short-listing the attributes for system failures. The authors propose some important challenges in the identification of system failure when performing classification: use of high dimensional dataset without compromising accuracy is one of the main issues. Our proposed scheme is composed of a step-wise identification methodology which efficiently eradicates all the three problems. We resolve high dimensionality of our dataset by monitoring and classifying performance parameters of the system independently.

[1] It is important to note that we collect the logs of selected performance parameters because maintaining logs of each process significantly increases the processing overhead of the logging process.

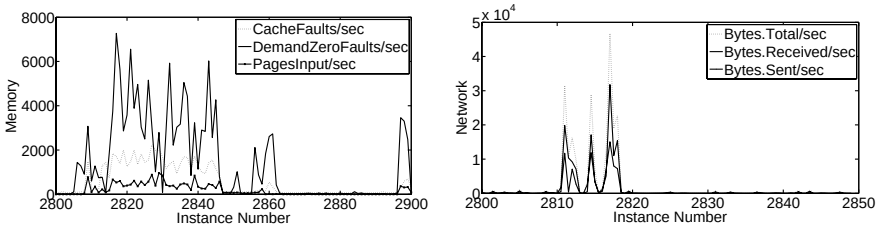
3 EvoPerf: Architecture and Functionality

We present the architecture of our `EvoPerf` utility in Figure 1 that consists of three sub-modules: (1) object monitor, (2) feature extractor, and (3) classifier. We now describe the functionality of each module.

3.1 Object Monitor

As mentioned before, this module captures logs of different *objects* of the operating system. Each object, provides one or more *counters* that represent a particular performance indicator of a given computer system. The values of the counters are updated after periodic intervals. One can select any `object monitor` and log its associated counters with the help of `Perfmon` utility. In EvoPerf, we use two types of objects: (1) Memory and paging, and (2) Network.

Memory and Paging The memory and paging objects depict the behavior of physical and virtual memory of a computer system. We know that physical memory is fast random access memory (RAM) on a computer; while virtual memory consists of RAM and secondary storage on the hard disk. A number of counters in the paging object monitor the information transfer – in the unit of fixed size memory chunks (pages) – between RAM and virtual memory. Thrashing is a special scenario in which a processor spends all its time in moving pages from main memory to virtual memory and vice versa. In this scenario, the response of a system might become significantly degraded that might eventually result in a denial of service [2]. We used 33 counters associated with the memory and paging objects. Some of these counter are described in Table 1. Just to substantiate the thesis that counters of memory objects can be used, we show a time series plot of three counters: CacheFaults/sec, DemandZeroFaults/sec and PagesInput/sec, in Figure 2(a) and it is obvious that the value of these counters are perturbed during the bottleneck period in between instance number 2810 and 2850.



(a) `Memory` (b) `Network`

Fig. 2. Performance bottleneck plot

Table 1. Memory Counters

Pool.Nonpaged.Allocs:	is the number of calls to allocate space in the nonpaged pool.
Pool.Paged.Allocs:	is the number of calls to allocate space in the paged pool.
Pages.Input/sec:	is the rate at which pages are read from disk to resolve hard page faults.
Pages/sec:	is the rate at which pages are read from or written to disk to resolve hard page faults.
Committed.Bytes:	is the amount of committed virtual memory, in bytes.
Committed.Bytes.In.Use:	is the ratio of Committed Bytes to the Commit Limit.
Cache.Faults/sec:	is the rate at which faults occur when a page sought in the file system cache is not found.
Page.Reads/sec:	is the rate at which the disk was read to resolve hard page faults.
System Cache Resident Bytes:	gives the size of pageable operating system code present in the cache of file system.
System Code Resident Bytes:	the size of operating system code in memory available to be written to physical disk.
System Driver Resident Bytes:	gives the size of the pageable physical memory being used by device drivers.
Pool Paged Resident Bytes:	is the sampled size of paged pool in bytes. Paged pool describes the area of physical memory under use of operating system for writing available objects to the disk.
pagefile.sys	The amount of the Page File instance in use in percent
System.Driver.Total.Bytes	bytes of the pageable virtual memory currently being used by device drivers.
Free.System.Page.Table.Entries	is the number of page table entries not currently in used by the system.
Cache.Bytes.Peak	gives the maximum number of bytes used by the file system cache since the last system restart.
Pool.Nonpaged.Bytes	is the size, in bytes, of the nonpaged pool.
Cache.Bytes	is the sum of the System Cache Resident Bytes, System Driver Resident Bytes System Code Resident Bytes and Pool Paged Resident Bytes counters.
Available.MBytes	is the physical memory available to processes running on the computer, in Megabytes
Available.Bytes	is the physical memory, in bytes, available to processes running on the computer.
Available.KBytes	is the physical memory available to processes running on the computer, in Kilobytes

Network Network activity is a key element in identifying bottlenecks in computers that are connected on the network. A computer system typically consists of multiple wired and wireless interfaces. The counters of network interface object mostly consist of volumetric traffic statistics and connection errors. The majority of network activity consists of TCP or UDP traffic (in case of TCP/IP network); therefore, we log counters of TCP and UDP objects together with the network interface objects[2]. TCP activity mostly results because of internet browsing. We use 48 counters which are associated with the network interface, TCP and UDP objects. The description of selected used Network counters is in Table 2. Figure 2(b) shows the plots of three important network counters – BytesTotal/sec, BytesReceived/sec and BytesSent/sec. We can easily see the values of these counters change significantly from instance number 2820 to 2840 an interval in which the bottleneck was created. We can see that bottleneck activity occur at the same instances in both memory and network objects. This is because heavy network activity has a direct effect on the memory of the system, so a bottleneck at the network interface causes a bottleneck on the memory.

3.2 Feature Extractor

One can appreciate the fact that our initial list of features' set consists of 33 memory and 48 network counters. It means that we need to keep track of 81 counters in our system which will not only increase the logging overhead but also increase the dimensionality space of our feature set. Therefore, it becomes relevant to use well know feature selection techniques to reduce the number of counters in our features' set.

We utilize two well known schemes for feature selection: (1) information gain [19], and (2) chi-square method [20]. We provide our raw features' set – obtained from the object module – to these feature ranking schemes. Both schemes rank

Table 2. Network Counters

TCP Segments Sent/sec:	gives the rate at which TCP segments are sent.
TCP Segments Received/sec:	gives the rate at which TCP segments are received, this includes segments received in error.
Packets Received/sec:	gives the rate at which packets are received on the network interface.
Packets Received Unicast/sec:	gives the rate of (subnet) unicast packet delivery to a higher-layer protocol.
Bytes Total/sec:	gives the rate of sending/receiving bytes over the network adapter.
Bytes Received/sec:	is the rate at which bytes are received over each network adapter.
Packets/sec:	is the rate at which packets are sent and received on the network interface.
TCP Segments/sec:	is the rate at which TCP segments are sent or received using the TCP protocol.
OutputQueueLength	is the length of the output packet queue (in packets).
TCPConnectionsEstablished	gives the number of TCP connections whose current states are either ESTABLISHED or CLOSE-WAIT.
TCPConnections.Active	is the number of TCP connection transition from the CLOSED state to the SYN-SENT state.
TCPConnections.Reset	is the number of direct TCP connection transition to the CLOSED state.
TCPConnections.Passive	is the number of direct TCP connection transition to the SYN-RCVD state from the LISTEN state.
Packets.Outbound.Errors	is the number of outbound packets that were not transmitted because of errors.
UDPDatagrams.Received.Errors	is the number of received UDP datagrams that were not delivered due to reasons excluding failure of application at the destination port.
TCPConnection.Failures	is the number of times TCP connections have made a direct transition to the CLOSED state from the SYN-SENT state or the SYN-RCVD state, plus the number of times TCP connections have made a direct transition to the LISTEN state from the SYN-RCVD state.
Bytes.Sent/sec	is the rate at which bytes are sent over each each network adapter.
Packets.Sent.Unicast/sec	is the rate at which packets are requested to be transmitted to subnet-unicast addresses by higher-level protocols.
Packets.Sent/sec	is the rate at which packets are sent on the network interface.

features separately on the basis of a feature's ability or role in enhancing the classification accuracy. After applying both schemes, the number of features – indicated in Table 3 – reduces from 81 to 40 (15 for memory and 25 for network). We have selected only those top ranked features which are common in both schemes. We now provide a brief description of each scheme to make the paper self contained.

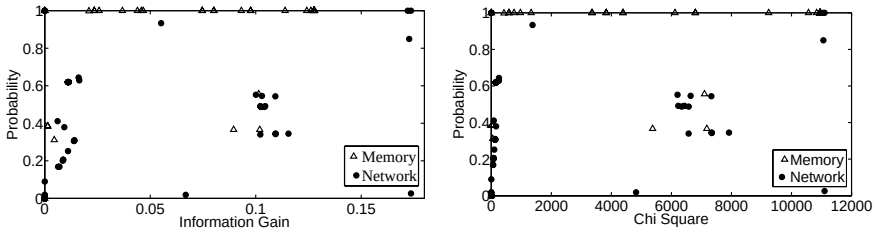
Table 3. Feature Selection

Objects	Memory	Network
Threshold IG	0.089	0.067
Feature IG	15	25
Threshold CS	5374.5	4823.9
Feature CS	15	25

Information Gain Information gain is an entropy based information theoretic measure. A feature with higher information gain will have higher classification power and vice versa. For a given attribute X and a class attribute Y, the uncertainty is given by their respective entropies $H(X)$ and $H(Y)$. Then the information gain of X with respect to Y is given by $I(Y;X)$, where

$$I(Y;X) = H(Y) - H(Y|X)$$

Table 3 shows the threshold values of information gain for memory and network objects. Figure 3(a) shows the normal probability distribution plot of information gain of all features [19].



(a) `Information Gain` (b) `Chi-squared`

Fig. 3. Normal Probability Plot

Chi-Squared Statistics The χ^2 method performs its feature selection by the use of chi-squared statistics of each feature with respect to its class. Initially χ^2 value of all features is calculated. The χ^2 is calculated as:

$$\chi^2 = \sum_{i=1}^{n} \sum_{j=1}^{k} \frac{(O_{ij} - E_{ij})}{E_{ij}} \tag{1}$$

where n is the number of intervals, k is the number of classes, O is the number of samples and E is the expected frequency. Table 3 shows the threshold χ^2 values. The features with χ^2 values greater then the threshold are eventually selected. Figure 3(b) plots the normal probability distribution of Chi-Square method of all features.

A closer look at Figure 3(a) and Figure 3(b) reveals that the memory and network objects follow approximately the same distribution pattern with minor differences. Therefore, a correlation of the top ranked features only, removes any discrepancy in the selected features' set. This analysis shows that `Available Bytes` – a counter of memory object – is an important feature even though it is a middle ranking feature. The reason is that a reduction in the available memory counter depicts a serious bottleneck because it could lead to an eventual denial of service. So the integrity of the selected feature set is an important issue [20].

3.3 Classifier

We have selected a number of well known classifiers in order to evaluate their feasibility for our EvoPerf. The choice of six classifiers is as following:(1) UCS is a state-of-the-art Michigan-style classifier [13], (2) two state-of-the-art Pittsburgh-style classifiers – GAssist-ADI [16] and GAssist-Intervalar [15], (3) two state-of-the-art neural network based classifiers – MLP [21] and RBF [22] and (4) a decision tree J48 [23]. The purpose of using classifiers from diverse learning paradigms is to select a classifier that provides the best accuracy with minimum processing overheads. (Remember in our case real time deployment of the system is very important.) We have used implementations of evolutionary classifiers

Table 4. Results of Experiments on Raw Features' Set

	Features		UCS	Gassist-Int	Gassist-ADI	RBF	MLP	J48	Average
Memory	33	Acc	0.994	0.999	0.999	0.999	0.999	0.995	0.998
		TP	140	194	195	195	195	195	
		TN	10805	10811	10812	10812	10812	10812	
		FP	56	2	1	1	1	1	
		FN	7	1	0	0	0	0	
Network	48	Acc	0.999	1	1	1	0.997	1	0.999
		TP	272	288	288	288	160	288	
		TN	10813	10813	10813	10813	10796	10813	
		FP	16	0	0	0	17	0	
		FN	0	0	0	0	16	0	

– UCS [13], GAssist-Int [15], GAssist-ADI [16]– provided in toolkit `KEEL` [12]; while for neural networks – RBF and MLP – we have used `WEKA` [5]. We used implementation of J48 in `WEKA`. We empirically determined the best configurations for different parameters of classifiers.[2]

4 Experiments

We have collected the performance logs on a computer system in our networks lab. The hardware specifications of the system are: Intel(R) Core2Duo 1.8 GHz CPU, 2GB of RAM and 160GB of physical drive. We utilized Windows `Perfmon` utility for monitoring performance counters. We have collected two sets of performance counter logs: normal and artificially created bottleneck. We have selected a sampling rate of 12 samples per minute. We have monitored user activity on the system for more than 15 hours over a period of 3 days to get a better idea about the normal usage behavior. Later, we have created a number of performance bottlenecks by maximizing network and memory usage of the system for a period of 15 minutes. The results of our experiments show that used six classifiers – UCS, GAssist-ADI, GAssist-Int, NN-MLP, NN-RBF and J48 – provide approximately the same accuracy. However, Neural Networks (NN) based classifiers have significantly smaller processing overhead, but when compared to machine learning algorithms, J48 outperforms the rest of the classifiers. This makes machine learning algorithms suitable for real world deployment.

We now report the results of our experiments. We follow a 10-fold cross validation strategy in all experiments. The dataset of each object is divided into 10 folds, 9 folds are used for training and the rest is used for testing. The process is reported for all folds and we report average value of all folds.

The results are tabulated in Table 4. It is obvious from the results of our evaluation that all classifiers provide approximately the same accuracy. Therefore, we now need to focus on our next objective: to reduce the processing overhead of the classifiers. Towards this end, we evaluate the impact of feature selection module.

[2] Parameters values of RBF: clustering seed = 1, minStdDev = 0.1, numClusters = 2, ridge = 1.0E-8

Table 5. Results of Experiments with Selected Features' Set

	Features		UCS	GAssist-Int	GAssist-ADI	MLP	RBF	J48	Average
Memory	15	Acc	0.998	0.999	0.999	0.999	0.999	0.995	**0.998**
		TP	183	192	195	195	195	195	
		TN	10811	10812	10811	10812	10811	10812	
		FP	1	0	1	1	1	1	
		FN	13	4	1	0	1	0	
Network	25	Acc	0.993	1	1	0.997	1	1	0.998
		TP	207	288	288	288	288	288	
		TN	10813	10813	10813	10813	10813	10813	
		FP	0	0	0	0	0	0	
		FN	81	0	0	0	0	0	

Table 6. Testing and Training times (seconds) of classifiers using all features

Parameter	**Memory**		**Network**	
	Training Time	**Testing Time**	**Training Time**	**Testing Time**
UCS	608.1766	31.0547	523.4749	42.2907
GAssist-ADI	2307.7	114.34	2187.12	81.6
GAssist-Int	2254.98	107	2096.54	78
NN-MLP	977.47	0.16	860.35	0.13
NN-RBF	6.33	0.04	6.35	0.08
J48	1.52	0.07	1.47	0.1

5 Results and Discussions

In this section, we discuss the results obtained once we apply features' selection techniques (see Table 5). Once we compare the results reported in Table 4 with those of Table 5, it is clear that the accuracy of the classifiers remain (almost) unaffected even with a reduced features' set. Now we analyze the impact of features' reduction on training and testing times of different classifiers.

5.1 Timing Analysis

We run two sets of experiments: (1) measure training and testing times once classifiers are using raw features' set, and (2) repeat the same experiments as in (1) but with reduced features' set. The obtained results for the first case are tabulated in Table 6. It is obvious from the table that J48 has the smallest training times while other classifiers take significantly large amount of time – GAssist-ADI is the worst – making them infeasible for real time deployment on a computer system. Similarly J48 takes almost the same time for testing, as the neural networks based classifiers.

Table 7. Testing and Training times (seconds) of classifiers using selected features

Parameter	**Memory**		**Network**	
	Training Time	**Testing Time**	**Training Time**	**Testing Time**
UCS	261.9124	15.2766	236.2954	14.7342
GAssist-ADI	1199.85	62.4	1018	27.4
GAssist-Int	1050.6	53.5	996.52	24
NN-MLP	137.95	0.02	113.21	0.02
NN-RBF	4.51	0.03	2.2	0.03
J48	0.57	0.04	0.66	0.04

Table 7, shows the training and testing times of different classifiers once we have reduced our features' set. The results prove our thesis: the training and testing time of all classifiers are reduced more than 50% due to features' selection. Again, J48 is having the smallest training and testing time without compromising on the detection accuracy.

6 Conclusions

The major contribution of the paper is an online real time autonomous performance monitoring system that has the capability to detect bottlenecks without user intervention. In a large network of interconnected systems, the proposed system can significantly reduce the workload of system administrator by relieving him of man-in-the-loop analysis; as a result, he can focus his attention on countermeasure strategies. Our research shows that using performance counters of memory and network objects, classifiers can identify bottlenecks with high accuracy. Our future work involves using other objects and analyzing the robustness of the system to evasion strategies.

7 Acknowledgments

The research presented in this paper is supported by the grant # ICTRDF/TR&D /2007/13 for the project titled ISKMCD, by the National ICT R&D Fund, Ministry of Information Technology, Government of Pakistan. The information, data, comments, and views detailed herein may not necessarily reflect the endorsements of views of the National ICT R&D fund.

References

1. Perfmon, Microsoft Technet, Microsoft Corporation, `http://technet.microsoft.com/en-us/library/bb490957.aspx`.
2. A. Silberschatz, P.B. Galvin, G. Gagne, "Operating System Concepts", John Wiley & Sons, 7th Edition, Inc, 2004.
3. Z. Ji and D. Dasgupta, "Revisiting Negative Selection Algorithms", Evolutionary Computation Journal, Vol. 15, No. 2, pp. 223-251, MIT Press, 2007.
4. Duan, S. and Babu, S., "Empirical Comparison of Techniques for Automated Failure Diagnosis", USENIX, 2009.
5. I.H. Witten, E. Frank, "Data mining: Practical machine learning tools and techniques", Morgan Kaufmann, 2nd edition, USA, 2005.
6. A.K. Jain, M.N. Murty, P.J. Flynn, "Data clustering: a review", ACM Computer Survey, Vol. 31, No. 3, pp. 264-323, ACM Press, 1999.
7. D. Pelleg, A.W. Moore, "X-means: Extending k-means with efficient estimation of the number of clusters", International Conference on Machine Learning (ICML), pp. 727-734, Morgan Kaufmann, USA, 2000.
8. Sandip Agarwala and Karsten Schwan. SysProf: Online Distributed Behavior Diagnosis through Fine-grain System Monitoring. Proceedings of the 26th IEEE International Conference on Distributed Computing Systems, Page 8, 2006.

9. S. Agarwala, C. Poellabauer, J. Kong, K. Schwan, and M. Wolf. "Resource-Aware Stream Management with the Customizable dproc Distributed Monitoring Mechanisms" In Proc. of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12), Seattle, WA, Jun 2003
10. M. Massie, B. Chun, and D. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. 2003.
11. Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jekrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tool. IEEE Computer, 28(11):37-46, 1995. G. Williams, K.
12. J. Alcala-Fdez, S. Garcia, F. J. Berlanga, A. Fernandez, L. Sanchez, M. J. del Jesus, F. Herrera, KEEL: A data mining software tool integrating genetic fuzzy systems, Genetic and Evolving Systems (GEFS), 3rd International Workshop, Vol. 4 , Issue 7, pp. 83-88, 2008.
13. E.B. Mansilla, J.M.G. Guiu, "Accuracy-Based Learning Classifier Systems: Models, Analysis and Applications to Classification Tasks Ester", Evolutionary Computation, 11(3), pp. 209-238, MIT Press, 2006.
14. R. Rivest, "Learning Decision Trees", Machine Learning, Vol. 2, pp. 229-246, 1987.
15. J. Bacardit, J.M. Garrell, "Bloat control and generalization pressure using the minimum description length principle for a Pittsburgh approach Learning Classifier System", International Workshop on Learning Classifier Systems (IWLCS), Volume 4399 of Lecture Notes in Artificial Intelligence, pp. 59-79, Springer, UK, 2007.
16. J. Bacardit, J.M. Garrell, "Evolving Multiple Discretizations with Adaptive Intervals for a Pittsburgh Rule-Based Learning Classifier System", Genetic and Evolutionary Computation Conference (GECCO), Volume 2724 of Lecture Notes in Computer Science, pp. 1818-1831, Springer, USA, 2003.
17. Z. Zheng, Z. Lan, B-H. Park, and A. Geist, "System Log Pre-processing to Improve Failure Prediction", Proc. of DSN'09, 2009.
18. F. Salfner and S. Tschirpke, "Error Log Processing for Accurate Failure Prediction," Proc. of WASL'08, in conjunction with OSDI, 2008.
19. Liu,H., Li,J. and Wong,L. (2002) A comparative study on feature selection and classification methods using gene expression profiles and proteomic patterns. Genome Inform., 13, 51-60.
20. Liu, H. and Setiono, R., Chi2: Feature selection and discretization of numeric attributes, Proc. IEEE 7th International Conference on Tools with Artificial Intelligence, 338-391, 1995.
21. F. Moller. A scaled conjugate gradient algorithm for fast supervised learning. Neural Networks, 525-533, (1990) .
22. D.S. Broomhead, D. Lowe. Multivariable Functional Interpolation and Adaptative Networks, Complex Systems, 321-355, (1988).
23. J.R. Quinlan, C4. 5: programs for machine learning, Morgan Kaufmann, 1993.