Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 83 (2016) 621 – 627

Procedia Computer Science

The 7th International Conference on Ambient Systems, Networks and Technologies (ANT 2016)

Towards Identifying Performance Anomalies

Haroon Malik[a]*, Elhadi M. Shakshuki[b]

[a]*Weisberg Division of Computer Science, Marshall University, WV, USA*
[b]*Jodery School of Compueter Science, Acadia, Univeristy, NS, Canada*

Abstract

Large-scale-software systems (LSSs) are composed of hundreds of subsystems that interact with each other in an unforeseen and complex ways. The operators of these LSSs strictly monitor thousands of metrics (performance counters) to quickly identify performance anomalies before a catastrophe. The existing monitoring tools and methodologies have not kept in pace with the rapid growth and inherit complexity of these LSSs; hence are ineffective in assisting practitioners to effectively pinpoint performance anomalies. We propose a methodology that uses entropy analysis to assist practitioners/operators of LSSs in quickly detecting underlying anomalies in the system. Our performance tests conducted on an open source benchmark system reveal that the proposed methodology is robust in pinpointing anomalies, do not require any domain knowledge to operate, and avoid information overload on practitioners.

Keywords: Performance counter; Large scale systems; Datacenter; Performance

1. Introduction

Today's large scale systems (LSSs) such as Facebook, Google, Amazon and many other datacenters comprise hundreds or thousands of machines running complex software applications that require high availability and responsiveness. They provide composite services, support a large user base and handle complex business demands.

* Corresponding author. Tel.: +1-304-696-5655.
E-mail address: malikh@marshall.edu

doi:10.1016/j.procs.2016.04.140

In line with Lehman's laws of continuing change and increasing complexity[1], the periodic monitoring of such LSS has become more critical and challenging than before since processing is spread across hundreds of subsystems and millions of hardware nodes (and users). These LSSs must be carefully monitored for performance anomalies before a serious harm is done[2-4]. A performance anomaly is an unexpected situation that causes system to deviate from abiding its Server Level Agreements (SLAs)[5-8]. Its symptoms include, but not limited to delayed response time, increases latency, decreased throughput and in cases, system hanging, freezing and crashing under heavy workload; usually introduced into the system by operator errors, hardware software failures, resource over-/under-provisioning, and unexpected interaction between geographically distributed system components[6].

LSSs are usually service oriented systems and generate revenue by providing composite services to a large user base. Any discrepancy in their performance can cause huge monetary losses. For example, an hour-long PayPal outage due to periodic maintenance may have prevented up to $7.2 million in customer transactions[9]. Similarly, an overloading of Google Server resulted into thousands of accounts being inaccessible for several days, worst many contents of many of the many of the clients were lost. Therefore, the operator of these LSSs continuously monitor their system to identify performance anomalies so a fix can be made quickly[10-13].

2. Current State of Practice

In LSSs, the current practice of discovering performance anomalies is centered on three major approaches:

2.1. Reactive Approach

Reactive techniques are used to set thresholds for observed performance counters (e.g., CPU utilization, disk I/O, memory consumption and network traffic) and raise alarms when these thresholds are violated. In LSSs, such as data centers and cloud providers, hosting multitenant application, the workload volume can be un-predictive. Using static thresholds, may lead to false alarms, thereby wasting analyst's time. Moreover, reactive approach is inadequate for understanding the performance changes resulting from application updates.

2.2. Proactive Approach

This category includes techniques for continuous evaluation of a system behavior by comparing it against baselines or statistical models. LSSs are continuously evolving and baseline rarely exist. Furthermore, there is an overhead involved in keeping the performance models up-to-date since continuous training of models on the performance data is required to keep them abreast with the dynamic and evolving behavior of LSSs.

2.3. Rule of Thumbs

In this category, analysts use a few of the important performance counters known to them from past practice and domain gurus, among thousands that are collected, during the performance monitoring process. They usually perform manual ad-hoc checks such as conducting simple correlation tests and producing plots for visual inspections. For example, up-ward trend for the memory usage, throughout, is a good indicator of a memory leak.

3. Proposed Methodology

We believe the current practice of identifying performance anomalies in not effective since it can take hours of manual analysis and still analyst may miss performance anomalies that are not associated with 'rule of thumbs'. Towards this end, we proposed a methodology based on Shannon Entropy; which intuitively provides a measure of the uncertainty remaining in the system after an observation has been made[2]. The entropy of a continuous random variable *X* with probability density function p(x), is given by:

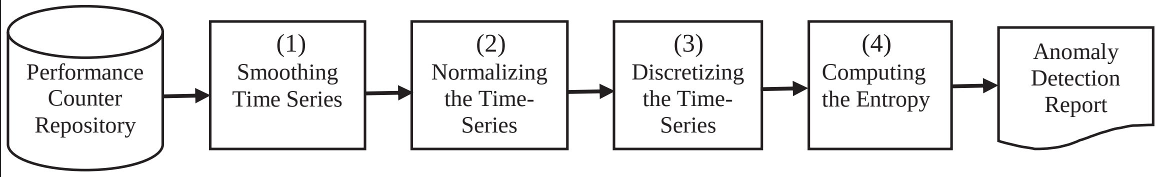

Performance Counter Repository
(1) Smoothing Time Series
(2) Normalizing the Time-Series
(3) Discretizing the Time-Series
(4) Computing the Entropy
Anomaly Detection Report

Fig. 1 Steps Involved in Our Proposed Methodology

$$H(X) = \int p(x) log_2\, p(x) dx$$

Though a valuable statistical measure, the entropy of a string (i.e., a sequence) is a poor predictor of anomalies in the system generating that string, because it focuses on randomness. As a consequence, a performance counter variable reading pure white noise would have the highest entropy and a performance counter variable reading a flat line would have zero entropy.

Unfortunately, most interesting phenomena occur somewhere between these two extremes, meaning that neither a rise nor a drop in entropy necessarily mean that a performance anomaly is detected. Furthermore, entropy only focuses on the frequency of the observed performance counter values, not their ordering.

As such a periodic binary string and a random binary string containing the same number of ones and zeros have the same entropy. To overcome some of these limitations of entropy, in our proposed methodology, we examine all the data (values) of performance counter variables for a particular time-stamp at once to generate a 'signature' value. Examining only a signature value rather that the data of thousands of performance counters makes the job of discovering performance anomalies easier for analysts. Before computing signature values for the entire system, (i.e., system wide entropy), we perform following series of pre-processing operations as shown in Fig 1:

3.1. Smoothing the Time-Series

For the first step, we smoothen the time series data for each of the performance counter variable. This is done by averaging time-series values together in groups of twenty. Smoothing is required due to the fact that components of LSSs are usually deployed across different machines, each of which is responsible for gathering performance data. The clock on each machine can be out-of-sync. Thus the performance counter data recorded by different machines may have slight difference of the timestamps. Moreover counters can be recorded at different rates. For example, the CPU utilization can be recorded after 10 seconds while disk I/O is recorded every half minute.

3.2. Normalizing the Time-Series

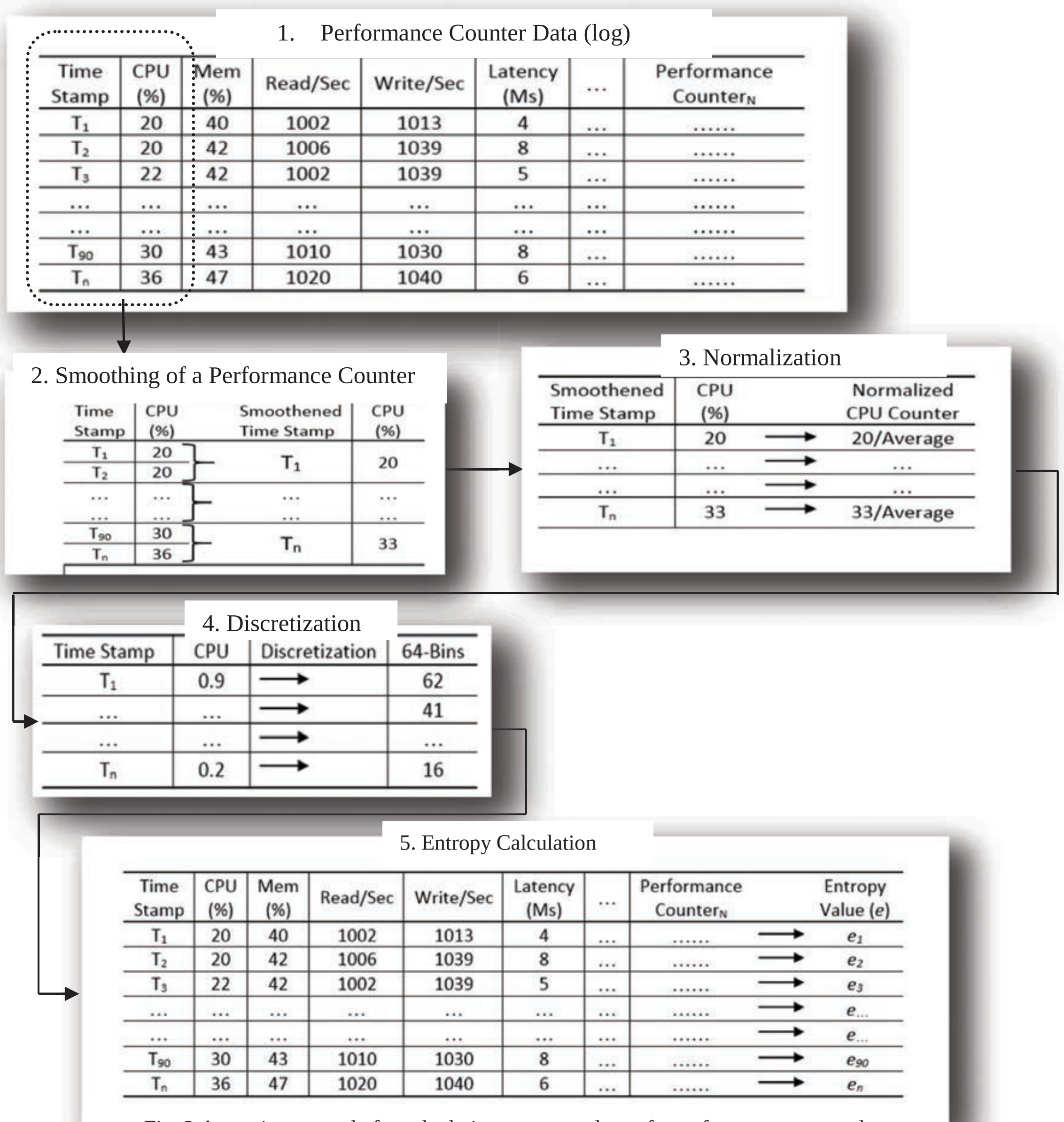
We normalized the performance counter time series by dividing each counter value by the average value in the series.

3.3. Discretizing the Time-Series

We discretized the time series values into uniformly sized K bins. Through the experimentation, we found that sixty-four bins are sufficient to capture most of the variability in performance counter data. Nevertheless, the value of K is a tunable parameter. Some experimentation is required to find an optimal value of K for a system in a given domain.

3.4. Computing the Entropy

Once each performance counter time-series has been preprocessed, the Shannon Entropy is computed for each


1. Performance Counter Data (log)
2. Smoothing of a Performance Counter
3. Normalization
4. Discretization
5. Entropy Calculation

Time Stamp	CPU (%)	Mem (%)	Read/Sec	Write/Sec	Latency (Ms)	…	Performance Counter$_N$
T_1	20	40	1002	1013	4	…	……
T_2	20	42	1006	1039	8	…	……
T_3	22	42	1002	1039	5	…	……
…	…	…	…	…	…	…	……
…	…	…	…	…	…	…	……
T_{90}	30	43	1010	1030	8	…	……
T_n	36	47	1020	1040	6	…	……

Time Stamp	CPU (%)	Smoothened Time Stamp	CPU (%)
T_1	20	T_1	20
T_2	20		
…	…	…	…
…	…	…	…
T_{90}	30	T_n	33
T_n	36		

Smoothened Time Stamp	CPU (%)		Normalized CPU Counter
T_1	20	$\longrightarrow$	20/Average
…	…	$\longrightarrow$	…
…	…	$\longrightarrow$	…
T_n	33	$\longrightarrow$	33/Average

Time Stamp	CPU	Discretization	64-Bins
T_1	0.9	$\longrightarrow$	62
…	…	$\longrightarrow$	41
…	…	$\longrightarrow$	…
T_n	0.2	$\longrightarrow$	16

Time Stamp	CPU (%)	Mem (%)	Read/Sec	Write/Sec	Latency (Ms)	…	Performance Counter$_N$		Entropy Value (e)
T_1	20	40	1002	1013	4	…	……	$\longrightarrow$	e_1
T_2	20	42	1006	1039	8	…	……	$\longrightarrow$	e_2
T_3	22	42	1002	1039	5	…	……	$\longrightarrow$	e_3
…	…	…	…	…	…	…	……	$\longrightarrow$	$e_{…}$
…	…	…	…	…	…	…	……	$\longrightarrow$	$e_{…}$
T_{90}	30	43	1010	1030	8	…	……	$\longrightarrow$	e_{90}
T_n	36	47	1020	1040	6	…	……	$\longrightarrow$	e_n

Fig. 2 A running example for calculating entropy values of a performance counter log

time-stamp across all the performance counter variables. This is done by first forming a vector S of size n, containing all the pre-processed performance counters values for a time-stamp. The Shannon Entropy for the discrete string S of length n is given by:

$$H(S)\sum_{i=1}^{n}\frac{n_i}{n}log_2\frac{n_i}{n}$$

Where n_i is the count of number of times value i appears in S. When most of the elements in S are equal to a small subset of the total possible values, the entropy is low.

This often happens when most of the performance counters are in highly correlated groups, since the normalized values in these groups will tend to be the same. When the elements in S tend to take a wide range of possible values, the entropy is high. This often happens when many counter variables become uncorrelated, or there is lots of noise induced in the system. Note that entropy is insensitive to noise coming from a small subset of performance counters, since each term in the summation is weighted by $\frac{n_i}{n}$. Fig 2 shows a running example of our proposed methodology.

If a performance counter erroneously takes on a common value, then n_i is large and the single performances counter value has little impact on the term. Also, if a performance counter erroneously takes on an uncommon value, then $\frac{n_i}{n}$ is small and the entire term has little impact on the entropy.

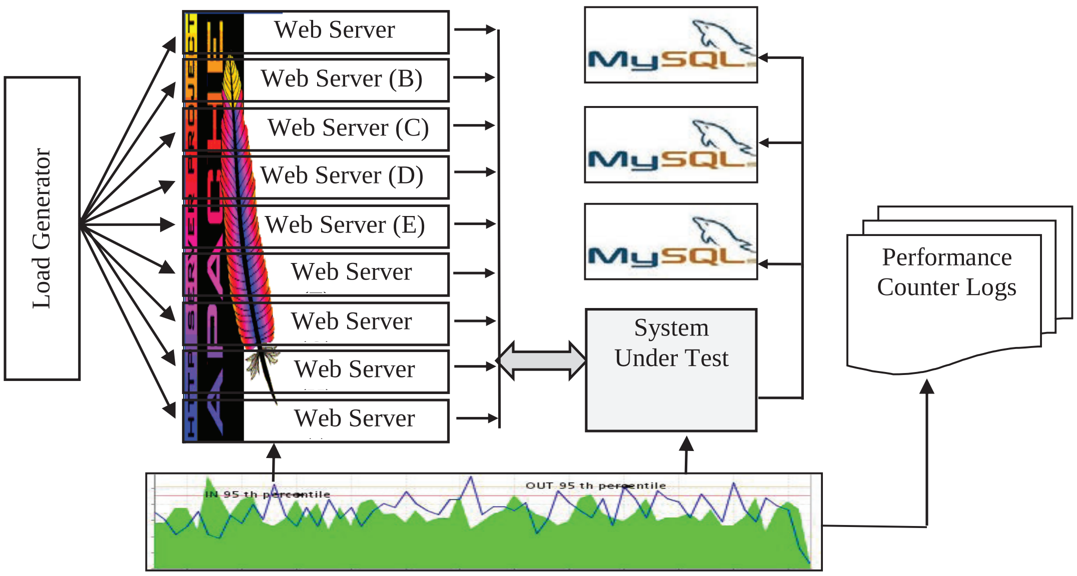
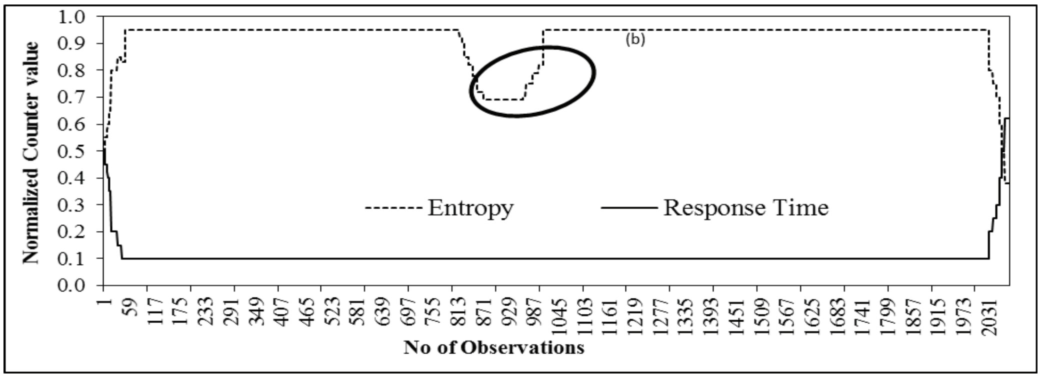


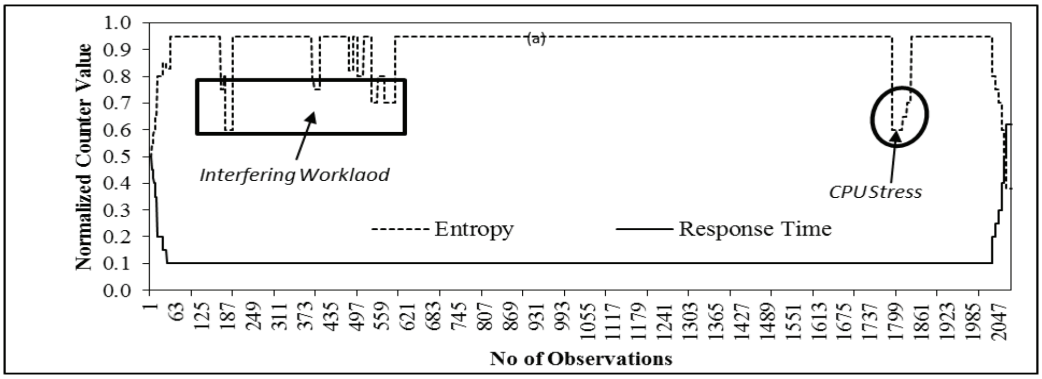
Fig. 3 Enviornment for the performance test experiments

4. Results/Discussion

To evaluate the effectiveness of our anomaly detection approach, we conducted performance tests using Dell DVD store (DS)[14]; an open source e-commerce system used by Dell to benchmark the performance of their systems. It is designed for benchmarking Dell hardware. It includes basic e-commerce functionalities such as user registrations, user login, product search and purchase. DS2 consists of a back-end database component, a web application component, and a driver programs (load generator).

We used the framework of Thakkar[15] to automate the load tests and to ensure that the environment remains constant throughout the experiments. We used Thakkar framework due to its simplicity and previous success in practical performance testing[15]. DS2 has multiple distributions to support different languages such as PHP, JSP, or ASP and databases such as MySQL, Microsoft SQL server, and Oracle. In this case study, we use the JSP distribution and a MySQL database(s). The JSP code runs in a Tomcat container. Our load consists of a mix of transactions, including user registration, product search and purchases. We created a similar environment as in Fig 3 for running performance test experiments on the open source system.





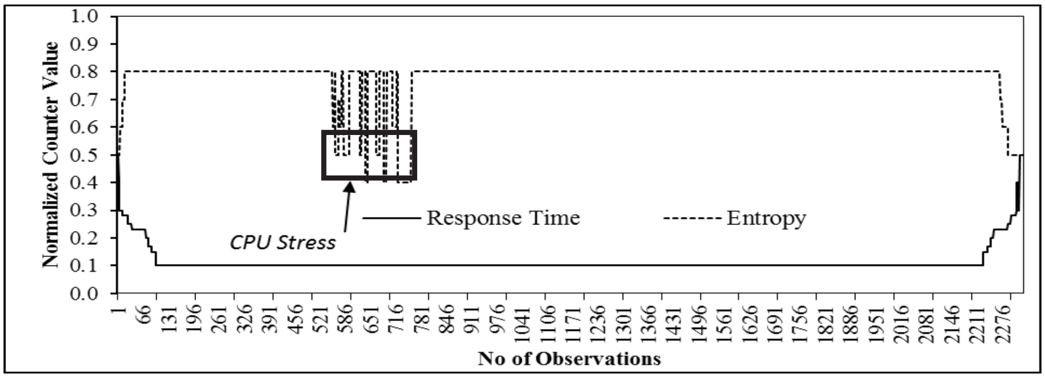


Fig. 4 Detection of Injected Anomalies in Performance Tests

4.1. Performance Tests

To evaluate the effectiveness of our anomaly detection approach, we conducted performance tests using Dell DVD store (DS)[14]; an open source e-commerce system used by Dell to benchmark the performance of their systems. The system-under-test comprises of fifteen Dell PowerEdge 860 severs; fourteen to set up the DS and one server as load generator. For all the performance test, we kept the workload constant to maintain a stable response time of the DS system. We injected the following anomalies into our performance tests and made note of times.

- **CPU Stress:** We slowed down the CPU of the one of DS web server using a CPU stress tool, known as win Throttle.
- **Memory Stress:** We injected a memory bug into the webserver using a customized open-source memory stress tool called EatMem. The tool allocates a random amount of available memory rapidly and at recurring intervals to mimic a transient memory spike.
- **Interfering Workload:** This performance test aims to trigger an interfering workload anomaly, mostly due to procedural errors. Such as, planning a security scan at the time when peak workload is expected, or due to unconstrained activities such as RAID construction, self-cleanup activities of mail stores and storage replications. We created an interfering background workload anomaly mimicking a situation where the administrator schedules an antivirus scan that conflict with the timing of the performance test. We scanned one of the web server machines with an antivirus for 50 minutes to disrupt the normal workload.

The duration of each test was set for eight hours and a total of 52 million counter values, for each test, from twelve hundred different performance counters across the fifteen severs were logged. We applied our methodology on the logged performance counter data for each performance test and found the findings to be consistent across all the tests. Fig 4 shows the result of three of the performance tests. The injected anomalies are clearly visible in the entropy trajectory of all the plots, i.e., the three performance test. The magnitude of injected anomaly was not large enough to disturb the response time equilibrium of the system. Hence, only observing the response time counter, i.e., 'rule-of-thumb', may lead to many anomalies being undetected.

5. Limitations of the Proposed Methodology

The proposed methodology only pinpoints to occurrences of performance anomalies. It does not distinguish between the different types of anomalies as well as overlapping anomalies, i.e., anomalies occurring at the same time. Nevertheless, the analyst has to manually investigate root-cause of detected anomalies. As a future work, we plan on extending our methodology to automatically recommend the performance counters to analysts that are likely associated with anomalous resources/components.

References

1. M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE,* vol. 68, pp. 1060-1076, 1980.
2. M. A. Munawar and P. Ward, "Adaptive monitoring in enterprise software systems," *SysML, June,* 2006.
3. Miao Jiang, M. A. Munawar, T. Reidemeister and P. A. S. Ward, "Automatic fault detection and diagnosis in complex software systems by information-theoretic monitoring," in *Proceedings of International Conference on Dependable Systems & Networks,* pp. 285-294, 2009.
4. M. Jiang, M. A. Munawar, T. Reidemeister and P. A. S. Ward, "Information-theoretic modeling for tracking the health of complex software systems," in *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds,* Toronto, Canada, pp. 236-247, 2009.
5. F. Mattosinho, "Pip: Detecting the Unexpected in Distributed Systems," in *Proceedings of of 3rd Symp on Networked Systems Design and Implementation (NSDI),* 2009.
6. E. J. Weyuker and F. I. Vokolos, "Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study," *IEEE Trans.Softw.Eng.,* vol. 26, pp. 1147-1156, 2000.
7. E. Boschi, S. Denazis and T. Zseby, "A measurement framework for inter-domain SLA validation," *Comput. Commun.,* vol. 29, pp. 703-716, 3/31, 2006.
8. A. Di Stefano, G. Morana and D. Zito, "A P2P strategy for QoS discovery and SLA negotiation in Grid environment," *Future Generation Comput. Syst.,* vol. 25, pp. 862-875, 9, 2009.
9. S. Stephen, "PayPal hit by global outage," in 2009.
10. D. Xikun, W. Huiqiang and L. Hongwu, "A comprehensive monitor model for self-healing systems," in *Proceedings of Multimedia Information Networking and Security (MINES), 2010 International Conference On,* 61-2010, pp. 751-756.
11. R. Voicu, I. C. Legrand and C. Dobre, "A monitoring framework for large scale networks," in *Intelligent Computer Communication and Processing (ICCP), 2011 IEEE International Conference On,* 2011, pp. 429-432.
12. M. Acharya and V. Kommineni, "Mining health models for performance monitoring of services," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering,* 33-2009, pp. 409-420.
13. C. Rathfelder, S. Becker, K. Krogmann and R. Reussner, "Workload-aware system monitoring using performance predictions applied to a large-scale e-mail system," in *Proceedings of Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference On,* 2012, pp. 31-40.
14. Dave. J. Available: http://linux.dell.com/dvdstore/.
15. D. Thakkar, A. E. Hassan, G. Hamann and P. Flora, "A framework for measurement based performance modeling," in *Proceedings of the 7th International Workshop on Software and Performance,* Princeton, NJ, USA, 2008, pp. 55-66.