Fast Extraction of Adaptive Change Point Based Patterns for Problem Resolution in Enterprise Systems

Manoj K. Agarwal, Narendran Sachindran, Manish Gupta, and Vijay Mann

IBM India Research Labs, Block 1, IIT campus,
Hauz Khas, New Delhi - 110016, India
{manojkag, nsachind, gmanish, vijamann}@in.ibm.com

Abstract. Enterprise middleware systems typically consist of a large cluster of machines with stringent performance requirements. Hence, when a performance problem occurs in such environments, it is critical that the health monitoring software identifies the root cause with *minimal* delay. A technique commonly used for isolating root causes is rule definition, which involves specifying combinations of events that cause particular problems. However, such predefined rules (or problem signatures) tend to be inflexible, and crucially depend on domain experts for their definition. We present in this paper a method that automatically generates change point based problem signatures using administrator feedback, thereby removing the dependence on domain experts. The problem signatures generated by our method are flexible, in that they do not require exact matches for triggering, and adapt as more information becomes available. Unlike traditional data mining techniques, where one requires a large number of problem instances to extract meaningful patterns, our method requires few fault instances to learn problem signatures. We demonstrate the efficacy of our approach by learning problem signatures for five common problems that occur in enterprise systems and reliably recognizing these problems with a small number of learning instances.

Keywords: fault localization, patterns, problem signatures, change point detection, adaptive learning.

1 Introduction

Modern enterprise systems are often required to provide services based on service level agreement (SLA) specifications at minimum cost. SLA breaches typically result in a significant penalty. Performance problems in these systems usually manifest themselves as high response times, low throughput, or a high rejection rate of requests. However, the root cause of these problems may be due to subtle reasons hidden in the complex stack of this execution environment. For example, badly written application code may cause an application to hang. Network problems like non availability of a connection between an application server and a database server can cause critical transactions to fail. Backup processes on a machine could cause performance degradation of servers running on that machine. Further, various components in such systems could have inter-dependencies which may be temporal or

R. State et al. (Eds.): DSOM 2006, LNCS 4269, pp. 161 – 172, 2006.
IFIP International Federation for Information Processing 2006

non-deterministic as they may change with changes in topology, application or workload. This further complicates root cause localization.

A commonly used event correlation technique for localizing the root cause of performance problems is rule definition [4]. In rule definition, all possible root causes are represented by rules specified as condition-action pairs. Conditions are typically specified as logical combinations of events, and are defined by domain experts. A rule is satisfied when a combination of events raised by the management system exactly matches the rule condition. Rule based systems while popular, suffer from two major drawbacks. First, they need domain experts to define rules. Second, rules are inflexible - they require exact matches and do not adapt as the environment changes.

Automatic learning of rules has been studied earlier by Hellerstein et al. [1]. They discover patterns using association rule mining based techniques [14]. They observe that when a fault occurs, it is usually accompanied by a burst of events. Additionally, each fault is usually associated with an event pattern. To corroborate these findings, we performed experiments on a multi-tier application running in a cluster. We employed change point based monitoring of performance metrics to generate alarms. The experiments consisted of several repetitions of different faults and resulted in the following observations:

- Certain alarms always occur when a fault occurs, resulting in a pattern that is very indicative of the underlying fault. This core set of alarms is repeated for every occurrence of a particular fault under different operating conditions.
- A few alarms occur repeatedly. These alarms represent innocuous events that occur during normal operation, and will probably not help in root cause analysis.

In this paper we present a method that exploits these properties to automatically associate patterns of change point based alarms with a given fault. Unlike earlier approaches [1], we can learn the problem signature for a fault with a very small number of fault instances. Our method also adaptively updates problem signatures as new information becomes available. Additionally, our method does not assume any prior domain-expert knowledge, and it learns effective problem signatures based only on feedback from the system administrator. Further, the problem signatures learned by our method are flexible and do not require exact matches to locate a root cause.

The layout of this paper is as follows. Section 2 presents related work. Section 3 describes our learning method. Section 4 describes our system design. Section 5 presents experimental results. Section 6 discusses future work and conclusions.

2 Related Work

The most common approaches to fault localization include AI techniques [3] such as rule-based techniques, model-based techniques, neural networks, decision trees, model traversing techniques such as dependency graphs [5][11] and fault propagation techniques [9] such as Bayesian networks and causality graphs.

As discussed in Section 1, automatic learning of rules has been studied earlier by Hellerstein et al. [1]. They discover patterns using association rule mining based techniques [14]. Additionally, each fault is usually associated with a specific pattern of events. Association rule based techniques require a large number of sample

instances before discovering "*k-itemset*" [16] in a large number of events. The method presented in this paper overcomes this limitation and is able to discover patterns with very few fault instances. Another drawback of their technique is their reliance on pattern periodicity. Our method does not make any such assumption.

In another closely related work [9], the authors describe an event driven fault diagnosis technique that employs incremental learning. The authors propose techniques to rank a fault according to a "goodness" measure that allows multiple simultaneous faults to be identified. Fault diagnosis is incrementally improved as more symptoms become available. Although this technique is promising, it makes an assumption about the presence of a symptom-fault map as an input. Such a map may not be available in an enterprise environment. Our method makes no such assumption.

Several earlier approaches have used dependency analysis for fault localization. In [5][11] the authors assume that the mechanism to generate events is already in place and the root cause analysis algorithm analyzes these events in a systematic way using certain properties of the executing environment such as a dependency tree. Alarms relying on static dependencies between system components may be analyzed for problem determination [7]. Katker et al. [12] also shows how the dependency graph may be used to perform systematic analysis of a problem and identify the root cause in the network fault management domain. In both these approaches, the authors assume the presence of a dependency tree. These approaches may not work in dynamic enterprise systems where dependencies are ephemeral.

Other related work [10][6] has focused on studying the behavior of the various components and structural changes in the system and looking for anomalies in them. These approaches usually isolate the problem to one system component. Thus, they fall short of localizing the actual root cause and can only detect bottlenecks in the path of transactions. In [10], the incoming requests are traced and the list of the components used by several succeeded or failed requests are clustered to statistically identify the set of failed components. In [6], an optimized set of synthetic transactions is used to probe the system for possible problems. This technique puts additional load on the system which may not be acceptable to customers in a production environment. Further, constructing an optimized set of probes is an N-P hard problem.

In [8], a combination of probing (using fault injection) and dependency analysis is used for fault localization. Dependency information is generated by Active Dependency Discovery (ADD). ADD builds the system dependency graph by individually perturbing the system components during a testing phase, while fault injection is used at run-time. This technique suffers from similar disadvantages as [6].

Rule based systems such as [4] are used to define rules, and events are generated based on satisfaction of these rules. In classical rule based systems, rules are specified manually and they are static in nature i.e. they do not evolve automatically.

3 Learning Methodology

In this section we describe our method for learning patterns (or *problem signatures*[1]) corresponding to faults that occur in enterprise environments. We assume that no two

[1] We use the terms patterns, signatures and problem signatures interchangeably in this paper.

faults occur simultaneously. The learning method operates on the premise that when a fault occurs in a system, it is usually associated with a specific pattern of events. In our system, these events correspond to abrupt changes in performance metrics.

The input to our learning method comprises of:

a. A sequence of time-stamped events representing change point based alarms that arise from each application server in a clustered system;
b. Times of occurrence of faults at a given application server;
c. Input from a system administrator who correctly labels a fault when it occurs for the first time, or when the method fails to detect it altogether;
d. Feedback from a system administrator to verify the correctness of our output.

The mechanism to provide the first two inputs is described in Section 4. We first define two scores computed by our learner - *co-occurrence* score and *relevance* score, and then describe our learning and matching algorithm.

3.1 Co-occurrence Score

Our learning method computes a co-occurrence score, or *c-score,* for every alarm that is ever raised within a fixed time window around the occurrence of a fault. For a fault *F*, the *c-score* measures the probability of an alarm *A* being triggered when *F* occurs. The *c-score* is computed as follows

$$c = \frac{\#(A \,\&\, F)}{\# F}$$

Here $\#(A \,\&\, F)$ is the number of times *A* is raised when *F* occurs and $\# F$ is the total number of occurrences of *F*. The *c-score* for an alarm-fault pair ranges from 0 to 1. A high *c-score* indicates a high probability of *A* occurring when *F* occurs.

3.2 Relevance Score

Our learning method computes a relevance score, or *r-score,* for every single alarm that it ever encounters. The *r-score* for an alarm is a measure of the importance of the alarm as a fault indicator. An alarm has high relevance if it usually occurs only when a fault occurs. The *r-score* for an alarm A is computed as follows

$$r = \frac{\#(A \,\&\, Fault)}{\# A}$$

where $\#(A \,\&\, Fault)$ is the number of times *A* is raised when *any* fault occurs in the system, and $\# A$ is the total number of times *A* has been raised so far. The *r-score* for an alarm ranges from 0 to 1. Note that the *r-score* is a global value for an alarm i.e. there is just one *r-score* for an alarm unlike the *c-score* which is per alarm-fault pair.

An assumption here is that the system runs in normal mode more often than it does in faulty mode. When this is true, alarms raised regularly during normal operation have low *r-scores*, while alarms raised only when faults occur have high *r-scores*.

3.3 Learning and Matching Algorithm

We present here our method for learning and matching fault patterns. The method uses a *pattern repository* to store patterns that it learns. It starts with an empty repository and adds patterns based on administrator feedback. If a fault occurs when the repository is empty, our method just notifies the administrator that a fault has occurred. After locating the root cause, the administrator provides a new fault label[2]. Our method then records the alarm pattern observed around the fault, along with the fault label, as a new signature. Each alarm in this signature is assigned a *c-score* of 1.

For every subsequent fault occurrence, our method uses the following procedure in order to attempt a match with fault patterns that exist in the repository. Assume that S_F is the set of all the faults currently recorded in the repository. For each fault $F \in S_F$, let S_{AF} represent the set of all the alarms A that form the problem signature for F. Let each alarm $A \in S_{AF}$ have a *c-score* $C_{A|F}$, when associated with a fault F. Also, assume that the set of alarms associated with the currently observed fault in the system is S_C. For each fault $F \in S_F$, the learner computes two values, a *degree of match* and a *mismatch penalty*. The *degree of match* rewards F for every alarm in S_C that also occurs in S_{AF}. The *mismatch penalty* penalizes F for every alarm in S_C that does not occur in S_{AF}.

To compute the degree of match for a fault $F \in S_F$, the learning method first obtains an intersection set S_{CF} - a set of alarms common to S_{AF} and S_C

$$S_{CF} = S_{AF} \bigcap S_C .$$

It then computes the degree of match D_F as follows

$$D_F = \frac{\sum C_{A|F} \forall A \in S_{CF}}{\sum C_{A|F} \forall A \in S_{AF}}$$

The numerator in the above formula is the sum of the *c-scores* of alarms in the intersection set S_{CF}, and the denominator is the sum of the *c-scores* of alarms in S_{AF}. The ratio is thus a measure of how well S_C matches with S_{AF}. When a majority of alarms (that have a high *c-score)* in S_{AF} occur in S_C, D_F is high.

To compute the *mismatch penalty* for a fault $F \in S_F$, the learning method first obtains a difference set S_{MF} - a set of alarms that are in S_C but not in S_{AF}

$$S_{MF} = S_C - S_{AF}$$

It then computes the mismatch penalty as follows

$$M_F = 1 - \frac{\sum R_A \forall A \in S_{MF}}{\sum R_A \forall A \in S_C}$$

[2] Fault labels have a one to one correspondence with problem signatures in the repository.

The numerator in the second term for the M_F formula is the sum of the *r-scores* of alarms in S_{MF}, and the denominator is the sum of the *r-scores* of alarms in S_C. By definition, the *r-score* is high for relevant alarms and low for irrelevant alarms. Hence, if there are mostly irrelevant alarms in S_{MF}, the ratio in the second term would be very low and M_F would have a high value.

Using D_F and M_F we compute a final ranking weight W_F for a fault *F* as,

$$W_F = D_F * M_F$$

Once our method computes ranking weights for all faults in the repository, it presents to the administrator a sorted list of faults with weights above a threshold. If no fault in the repository has a weight above the threshold, it reports that there is no match.

The administrator uses this list to locate the fault causing the current performance problem. If the actual fault is found on the list, the administrator accepts the fault. This feedback is used by the learning method to update the *c-scores* for all alarms in S_C for that particular fault. If list does not contain the actual fault, the administrator rejects the list and assigns a new label to the fault. The learner then creates a new entry in the pattern repository, containing the alarms in S_C, each with a *c-score* of 1.

3.4 Matching Algorithm Example

We present here an example that explains the functioning of our method. Assume that S_F is the set of faults currently in the fault repository and $S_F = \{F_1, F_2, F_3\}$. These faults have the following signatures stored as sets of alarm and *c-score* pairs.

$S_{AF1} = \{(A_1,1.0),(A_2,1.0),(A_3,0.35)\}$, $S_{AF2} = \{(A_2,0.75),(A_4,1.0),(A_5,0.75)\}$

$S_{AF3} = \{(A_5,0.6),(A_6,1.0),(A_7,0.9)\}$

Suppose we now observe a fault with a set of alarms $S_C = \{A_1, A_2, A_4, A_6\}$. Assume that *r-scores* of these alarms are $R_{A1} = 0.4$, $R_{A2} = 1.0$, $R_{A4} = 0.9$, $R_{A6} = 0.45$.

The intersection of the alarms in S_C with S_{AF1}, S_{AF2} and S_{AF3} yields the sets $S_{CF1} = \{A_1, A_2\}$, $S_{CF2} = \{A_2, A_4\}$ and $S_{CF3} = \{A_6\}$

The degree of match for each problem signature is computed as

$$D_{F1} = \frac{1.0 + 1.0}{1.0 + 1.0 + 0.35} = 0.85, \; D_{F2} = 0.7 \text{ and } D_{F3} = 0.4$$

For mismatch penalties, we compute the difference of set S_C from S_{AF1}, S_{AF2}, S_{AF3} to obtain $S_{MF1} = \{A_4, A_6\}$, $S_{MF2} = \{A_1, A_6\}$ and $S_{MF3} = \{A_1, A_2, A_4\}$.

The mismatch penalties are

$$M_{F1} = 1 - \frac{0.9 + 0.45}{0.4 + 1.0 + 0.9 + 0.45} = 0.51, \; M_{F2} = 0.69 \text{ and } M_{F3} = 0.16$$

The ranking weights are $W_{F1} = 0.85 * 0.51 = 0.43$, $W_{F2} = 0.48$, $W_{F3} = 0.06$. With a weight threshold of 0.4, the output list is F_2, F_1. Note that even though F_1 has a higher degree of match than F_2, F_1 is second on the list due to a higher mismatch penalty.

4 System Design

We describe here our system design for providing inputs to the learning method. The first input required by our method is a sequence of time-stamped alarms for each server in the cluster. For this, we monitor and sample runtime performance metrics at each server and use change point detection techniques such as difference of means [2] to generate alarms. A learning component is implemented on each server, and a pattern repository is shared amongst all learning components. The trigger for the method comes from an SLA breach predictor (*SBP)* operating at each server.

The *SBP* triggers the learning method when it detects an abrupt change in response time or throughput in the absence of any significant change in the input load on a server. Once the learning component gets a trigger from the *SBP*, it fetches all the alarms in a fixed time window around the current trigger. These alarms are then fed to the learning method and it operates on them as described in Section 3. The output from the learning method is a list of faults sorted in order of relevance. This list of faults is sent to a central controller which takes one of the following actions:

a. If only one server reports a list of faults during a given time interval, a single list is displayed to the administrator along with the name of the affected server.
b. If all running servers report a list of faults during a given time interval and the most relevant fault is the same for all servers, it is assumed that the fault is at a resource shared by all the servers, typically a database system. The controller chooses the most relevant fault and displays that fault to the administrator.
c. If a subset of running servers report a list of faults during a given time interval, this could either be caused by multiple independent faults or by a fault that occurred on one server and has affected the runtime metrics of other servers due to an “interference effect”. In our current design, the controller treats the two cases in the same manner and displays the lists for all affected servers.

5 Evaluation

We describe in this section, our test-bed, three-tier application and workload generator, system implementation, and our experimental results.

5.1 Test-Bed, Application and Workload

Our test-bed consists of eight machines: one machine hosting two load generators, two request router machines, three application server machines, a relational database server machine, and a machine that hosts the cluster management server. The back end servers form a cluster, and the workload arriving at the routers is distributed to these servers based on a dynamic routing weight assigned to each server. The

machines running the back end servers have identical configurations. They have a single 2.66GHz Pentium4 CPU and 1GB RAM. The machine running the workload generators is identical except that it has 2GB RAM. Each of the routers have one 1.7GHz Intel Xeon CPU and 1GB RAM. The database machine has one 2.8GHz Intel Xeon CPU and 2GB RAM. All machines run RedHat Linux Enterprise Edition 3, kernel version 2.4.21-27.0.1.EL. The router and back end servers run the IBM WebSphere middleware platform, and the database server runs DB2 8.1.

For our experiments, we ran Trade 6 [17] on each of the servers. Trade 6 is an end-to-end benchmark that models a brokerage application. It provides an application mix of servlets, JSPs, enterprise beans, message-driven beans, JDBC and JMS data access. It supports operations provided by a typical stock brokerage application.

We used IBM WebSphere Workload Simulator [18] to drive our experiments. The workload consists of multiple clients concurrently performing a series of operations on their accounts over multiple sessions. Each of the clients has a think time of 1 second. The actions performed by each client and the corresponding probabilities of their invocation are: *register new user* (2%), *view account home page* (20%), *view account details* (10%), *update account* (4%), *view portfolio* (12%), *browse stock quotes* (40%), *stock buy* (4%), *stock sell* (4%), and *logoff* (4%). These values correspond to the typical usage pattern of a trading application.

5.2 Experimental Runs

In order to perform a detailed evaluation of our learning method over a number of parameters and fault instances, we generated traces containing the inputs required by our method and performed an offline analysis. The only difference from an online version is that the administrator feedback was provided as part of the experimentation.

We implemented the breach predictor as a component that resides within one of the routers in our test-bed. It subscribed to router statistics and logged response time information per server at a 5 second interval. Each server in the cluster also monitored and logged performance metric information. We ran a total of 60 experiments, each of duration one hour (45 minutes of normal operation followed by a fault). The five faults that we randomly inserted in our system were:

- CPU hogging process at a node hosting an application server
- Application server hang (created by causing requests to sleep)
- Application server to database network failure (simulated using Linux *iptables*)
- Database shutdown
- Database performance problem (created either by a CPU hog or an index drop).

We maintained a constant client load during individual experiments, and varied the load between 30 and 400 clients across experiments. After obtaining the traces for 60 experiments, the learning and matching phase involved feeding these traces to our method sequentially. This phase presents a specific sequence of alarms to the learning method. In order to avoid any bias towards a particular sequence of alarms, we repeated this phase a 100 times, providing a different random ordering of the traces each time. For all our experiments we used a *c-score* threshold of 0.5.

5.3 False Positives and Negatives

We first explore the performance of our learning method in terms of false positives and negatives. We compute the false negative count as the number of times our method does not recognize a fault. However, when our method sees a fault for the first time, it does not count as a false negative. After completing all 100 runs, we compute the average number of false negatives generated by our method.

False positives occur when a newly introduced fault is recognized as an existing fault. We use the following methodology to estimate false positives. We randomly choose a fault *F* and remove all traces containing *F* from the learning phase. We then feed traces containing *F* to our method and calculate the number of times it is recognized as an already observed fault. We repeat this procedure for each fault and compute the average number of false positives.

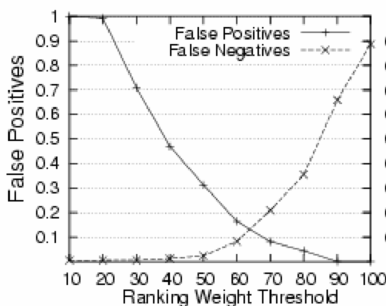

Fig. 1. False positives and negatives

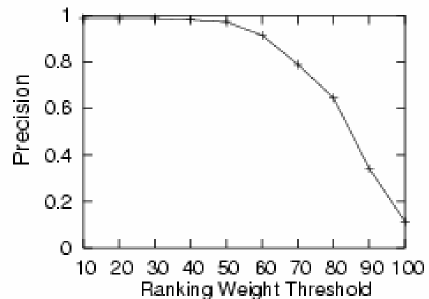

Fig. 2. Precision

Figure 1 shows the average percent of false positives and false negatives generated by our method as we vary the ranking weight threshold between 10 and 100. Recall that the ranking weight is our estimate of the confidence that a new fault pattern matches with a pattern in our repository. Only pattern matches resulting in a ranking weight above the threshold are displayed to the administrator.

As one would expect, when the threshold is low (20% or lower) we generate a large number of false positives. This is because at low thresholds even irrelevant faults are likely to generate a match. As we increase the threshold beyond 20%, the number of false positives drops steadily, and it is close to zero at high thresholds (80% or higher). Note that false positives are generated only when a new fault occurs in the system. Since new faults can be considered to have relatively low occurrence over a long run of a system, a false positive percent of 20-30% may also be acceptable after an initial learning period. Our method generates few false negatives for thresholds under 50%. For thresholds in the 50-70% range, false negatives range from 3-21%. Thresholds over 70% generate a high percent of false negatives.

Hence, there is a trade off between the number of false positives and negatives. The curves for the two measures intersect when the ranking weight threshold is about 65%, and the percent of false positives and negatives is each about 13%. A good region of operation for our method is between a weight threshold of 50-65%, with more false positives at the lower end, and more false negatives at the higher end. An

approach that we can use to obtain good overall performance is to start our method using a threshold close to 65%. During this initial phase, it is likely that a fault occurring in the system will be new, and the high threshold will help in generating few false positives. As our method learns patterns, and new faults become relatively rare, the threshold can be lowered to 50% in order to reduce false negatives.

5.4 Precision

If a fault is always detected but usually ends up at the bottom of the list of potential root causes, the analysis is likely to be of little use. In order to measure how effectively our method matches new instances of known faults, we define a *precision* measure. Each time our method detects a fault, we compute a precision score using the formula $\frac{(\#F - i - 1)}{\#F}$, where *#F* is the number of faults in the repository, and *i* is the position of the actual fault in the output list. A false negative is assigned a precision of 0, and our method is not penalized for new faults not present in the repository. We perform 100 iterations over the traces using the random orderings described above, and compute the average precision.

Figure 2 shows average precision values for ranking weight thresholds ranging from 10-100. We can see that our precision score is high for thresholds ranging from 10-60%. For thresholds ranging from 10-30%, the average precision is 98.7%. At a threshold of 50% the precision is 97%, and at a threshold of 70% the precision is 79%. These numbers correspond well with the false negative numbers presented in the previous section, and they indicate that when the method detects a fault, it usually places the correct fault at the top of the list of potential faults.

5.5 Rate of Learning

We demonstrate in this section a key property of our learning method – it can learn a relevant pattern for a fault given very few instances of the fault. To evaluate the rate at which our method learns patterns, we perform the following experiments. We first set a *learning threshold* which is the maximum number of instances of a fault that our method can use in order to learn. Any fault instances over the learning threshold are only used to evaluate the precision of our method, and cannot be used by the learner to update its scores. We then run our method over each fault using several values of the learning threshold, and obtain an average precision score for each threshold value.

Figure 3 shows precision scores for three values of the learning threshold, 1, 2, and 4. The precision values are shown for ranking weight thresholds ranging from 10-100. We can see that when our method is provided with only a single instance of a fault, it has precision values of about 90% when the ranking weight is 50%. This is only about 8% worse than the best possible precision score. At a ranking weight threshold of 70%, the precision is about 14% lower than the best possible precision.

This data clearly shows that our method learns patterns rapidly, with as few as two instances of each fault required to obtain high precision. This is largely due to two reasons. First, we use change point detection techniques to generate events and we have found that they reliably generate unique patterns for different faults. Second, the *c-score* and the *r-score* used by our method help us filter out spurious events.

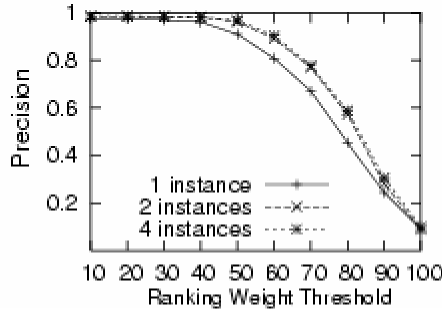

Fig. 3. Rate of Learning

6 Conclusions and Future Work

In this paper, we presented a novel technique for discovering change point based adaptive patterns for problem resolution in enterprise systems. We demonstrated the efficacy of our technique by learning the problem signatures for five common faults that occur in enterprise systems and reliably recognizing these problems with high precision. One of the main contributions of this paper is that we discover these patterns quickly, with few fault instances. This is a significant improvement over traditional data mining techniques which require a large number of fault instances to discover patterns. The patterns generated by our method are flexible, in that they do not require exact matches for triggering. Another significant contribution of our work is that our technique can discover adaptive patterns i.e. if a fault pattern changes over time due to reasons such as changes in topology, workload, application version, our method automatically updates the pattern repository with the new pattern over time.

There are a few future directions to the work presented in this paper. One of the issues that we intend to tackle is the absence of certain alarms during the problematic phase. The absence of a particular alarm during the problematic phase may be as indicative of a fault as the presence of other alarms. Our method currently does not handle cases where significantly different patterns are generated for a single fault. An extension to our method would associate more than one pattern to a fault if there is a significant mismatch between the patterns. Another improvement to our technique is the use of negative feedback from the administrator. In our future research, we also intend to include events generated by sources other than the currently monitored performance metrics, such as event logs.

References

1. Hellerstein J. L., Ma S., Perng C.: Discovering Actionable Patterns in Event Data. IBM Systems Journal, Vol 41, No 3, 2002.
2. Agarwal M., Gupta M., Mann V., Sachindran N., Anerousis N., Mummert L.: Problem Determination in Enterprise Middleware Systems using Change Point Correlation of Time Series Data. 9thIEEE/IFIP Network Operations and Management Symposium (NOMS), Vancouver, Canada, May 2006.

3. Steinder M., Sethi A.:The present and future of event correlation: A need for end-to-end service fault localization. SCI-2001, 5th World Multiconference on Systemics, Cybernetics, and Informatics, Orlando, FL (July 2001), pp. 124-129
4. Appleby K., Goldszmidt G., Steinder M.: Yemanja A Layered Fault Localization System for Multi-domain Computing Utilities. IM 2001
5. Gruschke B.: Integrated Event Management: Event Correlation Using Dependency Graphs. DSOM 1998.
6. Brodie M., Rish I., Ma S., Odintsova N.: Active Probing Strategies for Problem Diagnosis in Distributed Systems. IJCAI 2003
7. Gao J., Kar G., Kermani P.: Approaches to Building Self Healing Systems using Dependency Analysis. IEEE/IFIP Network Operations and Management Symposium (NOMS), April, 2004.
8. Brown A., Kar G., Keller A.: An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment. IM 2001.
9. Steinder M., Sethi A.: Non-deterministic Event-driven Fault Diagnosis through Incremental Hypothesis Updating, In Integrated Network Management, VIII} (G. Goldszmidt and J. Schonwalder (eds.)), pp. 635-648, Boston, MA: Kluwer Academic Publishers, 2003
10. M. Y. Chen, E. Kıcıman, E. Fratkin, A. Fox, E. Brewer: Pinpoint: PD in Large, Dynamic Internet Services, International Conference on Dependable Systems and Networks (DSN'02), 2002.
11. Choi J., Choi M., Lee S.: An Alarm Correlation and Fault Identification Scheme Based on OSI Managed Object Classes. IEEE International Conference on Communications, Vancouver, BC, Canada, 1999, pp. 1547–51.
12. Katker S., Paterok M.: Fault Isolation and Event Correlation for Integrated Fault Management. Integrated Network Management V, Chapman and Hall, May 1997.
13. Aguilera M. et.al.: Performance Debugging for Distributed Systems of Black Boxes. 19th ACM Symposium on Operating Systems Principles, October 2003.
14. Agarwal R., Imielinski T., and Swami A.: Mining association rules between sets of items in large databases. ACM SIGMOD Conference on Management of Data, pp. 207-216, May 1993.
15. Agarwal M., Appleby K., Faik J., Kar G., Neogi A., Sailer A.: Threshold management for Problem Determination in Transaction Oriented e-Commerce Systems., 9th IFIP/IEEE International Symposium on Integrated Network Management (IM 2005), May 2005.
16. Fu A., Kwong R., Tang J., "Mining N most interesting Itemsets" 12th International Symposium on Methodologies for Intelligent Systems (ISMIS), Springer-Verlag, LNCS, Charlotte, North Carolina, USA, Oct 11-14, 2000
17. IBM Trade Performance Benchmark Sample, http://www-306.ibm.com/software/ webservers/ appserv/was/performance.html
18. IBM Websphere Studio Workload Simulator, http://www-306.ibm.com/software/awdtools/ studioworkloadsimulator/