IPSJ Transactions on Advanced Computing Systems Vol.5 No.1 1–12 (Jan. 2012)

Regular Paper

Clustering Performance Anomalies Based on Similarity in Processing Time Changes

Satoshi Iwata[1,a)] Kenji Kono[1,2]

Received: May 11, 2011, Accepted: August 30, 2011

Abstract: Performance anomalies in web applications are becoming a huge problem and the increasing complexity of modern web applications has made it much more difficult to identify their root causes. The first step toward hunting for root causes is to narrow down suspicious components that cause performance anomalies. However, even this is difficult when several performance anomalies simultaneously occur in a web application; we have to determine if their root causes are the same or not. We propose a novel method that helps us narrow down suspicious components called *performance anomaly clustering*, which clusters anomalies based on their root causes. If two anomalies are clustered together, they are affected by the same root cause. Otherwise, they are affected by different root causes. The key insight behind our method is that anomaly measurements that are negatively affected by the same root cause deviate similarly from standard measurements. We compute the similarity in deviations from the non-anomalous distribution of measurements, and cluster anomalies based on this similarity. The results from case studies, which were conducted using RUBiS, which is an auction prototype modeled after eBay.com, are encouraging. Our clustering method output clusters crucial in the search for root causes. Guided by the clustering results, we searched for components exclusively used by each cluster and successfully determined suspicious components, such as the Apache web server, Enterprise Beans, and methods in Enterprise Beans. The root causes we found were shortages in network connections, inadequate indices in the database, and incorrect issues with SQLs, and so on.

Keywords: performance anomaly, diagnosis, web application

1. Introduction

Performance anomalies are a serious problem in commercial web applications. They are unexpected, adverse degradations in performance such as sudden increases in response times or unusual decreases in server throughput. They lead to violations of service level agreements (SLAs) and loss of potential customers due to lowered quality of services [3].

The increasing complexity of web applications makes it much more difficult to identify the root causes of performance anomalies and methods of analyzing root causes are required to help web administrators diagnose the anomalies. If a performance anomaly is detected during test operations, a method that is good at analyzing the root cause shortens the time for diagnosis and services can be started without delays. If a performance anomaly is detected during real operations, the shortened diagnosis time avoids or moderates the damage caused by the anomaly.

Although many methods of analyzing root causes have been proposed, there is one problem with these. When more than one anomaly is detected simultaneously, we have to determine if their root causes are the same or not. Correct identification of this facilitates the analysis of root causes. As illustrated in **Fig. 1**, if we detect an anomalous request, we can narrow down suspicious components to those that are passed through by the request. If request R_1 in this figure is detected as being anomalous, components A and C are suspicious. However, when two requests R_1 and R_2 simultaneously exhibit anomalous performance, we can take two cases into consideration. The first is where component C is anomalous; thus, both requests R_1 and R_2 are anomalous. The second is where components A and B are simultaneously anomalous; thus, both requests R_1 and R_2 become anomalous.

We propose applying a method to cluster performance anomalies based on root causes that helps us determine which situation is occurring inside the above problem; it clusters anomalous requests based on root causes. We call this method *performance anomaly clustering*. In the above example, if requests R_1 and R_2 are clustered together, they are probably caused by the same root cause and component C is thus considered to be suspicious. If requests R_1 and R_2 are clustered differently, they are probably caused by different root causes and components A and B are thus considered to be suspicious.

Clustering results guarantee that the requests clustered together are affected by the same set of problems and those clustered differently are affected by different sets of problems. A formal method that systematically narrow down root causes is out of the scope of this paper since this paper focuses on the clustering. As shown in the case studies, the clustering results can be utilized in various ways. At least, searching for the components exclusively used by each cluster would be helpful and shorten diagnosis time.

Some readers may notice that there are three more cases we can take into consideration, when two requests R_1 and R_2 simul-

[1] Keio University, Yokohama, Kanagawa 223–8522, Japan
[2] CREST, Japan Science and Technology Agency, Chiyoda, Tokyo 102–0075, Japan
[a)] satoshi@sslab.ics.keio.ac.jp

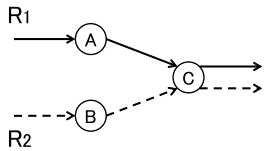


Fig. 1 Root causes are often ambiguous. Request R_1 passes through components A and C. Request R_2 passes through components B and C. Components may be machines, server processes, or Enterprise Beans.

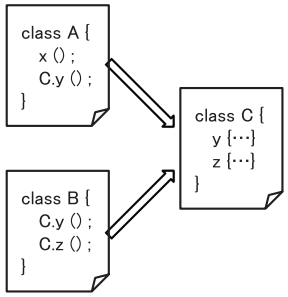


Fig. 2 Performance anomaly clustering is helpful even with finer grained monitoring. Method x is defined in Java class A, while methods y and z are defined in class C. Methods x and y are invoked in class A, while methods y and z are invoked in class B.

taneously exhibit anomalous performance. 1) Both A and C are anomalous. 2) Both B and C are anomalous. 3) All of A, B, and C are anomalous. We set aside the explanation of them and mention later in Section 3.1 to simplify the explanation.

Monitoring each component A, B, and C appears to avoid the obstacle with ambiguous root causes, as shown in Fig. 1. We can identify anomalous components since alarms are raised according to the granularity of components. Even if this is the case, our method is useful for identifying suspicious components with granularities finer than that of monitoring. When all three components are detected as being anomalous, we can take several cases into consideration. We can illustrate this situation using **Fig. 2**. In one case, anomalous method y in C is shared by A and B; thus, a performance anomaly is also detected in A and B. In another case, x in A and z in C are anomalous and B is not anomalous by itself but is considered to be anomalous due to anomalous method z in C. Our clustering method is useful for distinguishing these cases. In the former, all three components (A, B, and C) are clustered together. In the latter, B and C are clustered together but A is clustered differently. Monitoring regions need to be divided with the finest granularity to circumvent our clustering method so that the regions never share any root causes. However, the granularity of monitoring is limited due to the performance overhead and cost of implementation.

We illustrated two ideal examples in which our clustering method was very helpful, using Figs. 1 and 2, e.g., there would be cases in which anomalous monitoring regions did not share any components and thus, clustering output was less valuable. However, even if this is the case, we believe clustering results will remain helpful to some extent and will not hinder successive diagnosis.

The key insight behind our method is that anomaly measurements that are negatively affected by the same root cause exhibit similar deviations from the standard measurements. If an anomaly is occurring in component C in the example shown in Fig. 1, the measurements of requests R_1 and R_2 are expected to deviate similarly. If components A and B are anomalous, the measurements are expected to deviate differently because two different anomalies affect R_1 and R_2 differently. We use an existing anomaly detector [4], [5], [6], [7], [8], [9], [10], [11] as a back-end in our clustering method to collect the measurements and detect anomalies. The monitoring is done at the granularity of the back-end anomaly detector.

The clustering method proposed in this paper involves three steps. First, we distill a *performance anomaly signature* from the measurements of anomalous monitoring regions. A performance anomaly signature characterizes how the "distribution" of the measurements has changed after an anomaly has occurred. To distill a signature, we calculate the difference between cumulative distribution functions (CDFs) of the measurements before and after a performance anomaly has been detected. Since the distribution of measurements can be expressed in a CDF, the difference between CDFs naturally captures the change in the distribution of the measurements. A signature in our method is represented as a bar graph. Second, we calculate the *similarity* in the signatures. The similarity is a scalar that represents the degree to which two signatures, i.e., two bar graphs, overlap each other. We prepare two similarity functions to capture different features in similarity. Finally, we cluster anomalies based on the similarities. If two or more anomalies are clustered together, this implies that they are affected by the same root cause. Otherwise, they are affected by different root causes.

To investigate the usefulness of our method, we conducted three case studies using RUBiS [1], which is an auction site prototype modeled after eBay.com [2]. RUBiS has a typical three-tier structure and is based on the Java EE platform. We used the anomaly detector proposed by Iwata and Kono [4] to detect performance anomalies in RUBiS. This detector measures the end-to-end processing time of each request and raises an alarm when measurements deviate statistically from the standard. We clustered anomalous requests with our clustering method using the measured processing times and the results we obtained were encouraging. We extensively used the clustering results in all three case studies to narrow down suspicious components (servers, Enterprise Beans, or methods) and successfully identified root causes. We do not use any information which is unique to RUBiS, so we believe our method can be applied to various web applications.

The rest of this paper is organized as follows. Section 2 briefly explains the anomaly-detection method we used. Section 3 describes our clustering method and Section 4 reports case studies. Section 5 describes related work. Finally, Section 6 concludes the paper.

2. Detection of Performance Anomalies

We briefly explain the performance anomaly detector that we used in our case studies before describing our method. Although our clustering approach was developed for the method, we believe that it can be applied to other methods.

This method monitors the end-to-end processing time available from outside a web application. Thus, as it is easy to introduce

into existing systems, we chose it as our back-end detector in this research. Statistics are collected for each *request type* in this method. Since the statistics are collected on the basis of request types, performance anomalies are detected in the granularities of request types.

A request type can be defined naturally based on its job. Each request type for RUBiS is associated with an HTML file or a servlet (e.g., `index.html`, `ViewItem.java`, or `PutBid.java`). If a servlet is shared by some request types, we can distinguish the types from their parameters. For example, `BrowseCategories` and `BrowseCategoriesInRegion` commonly use `BrowseCategories.java`. If a request type includes a region parameter, it belongs to the latter. Otherwise, it belongs to the former. Request types in this method, are assumed to be defined beforehand. Actually, RUBiS defines a list of request types. Even if no definitions are given, request types can be easily defined from URLs and their parameters.

The method proposed by Iwata and Kono [4] applies *control charts* [12] to detect performance anomalies. Control charts have been developed to determine whether a manufacturing or business process is in a state of statistical control or not. Since control charts are intended to detect deviations from the standard quality of products, they are suitable for detecting performance anomalies, which are deviations from standard performance.

Figure 3 has an example of a control chart. We measure characteristic values (e.g., processing times of one request type) and calculate and plot the statistics (e.g., average, median, or defective fraction) on the control chart to detect anomalies. There are three baselines in the chart: upper control limit (UCL), lower control limit (LCL), and center line. The center line represents the center of the measured statistics. UCL and LCL indicate 3σ above/below the center of the statistics, where σ denotes the standard deviation. These baselines are calculated in advance from the data measured in a controlled environment.

The theory of control charts defines statistical deviations, which should be considered to be anomalous. For example, a plot outside the limits (UCL and LCL) suggests an anomaly. Systematic patterns within the limits are also considered to be an anomaly. Readers who are interested in control charts can refer to Ref. [12] for more details.

In the method in Ref. [4], the end-to-end processing time for each request type is measured, and statistics, such as the average, median, maximum, and minimum are calculated. In this method, one control chart is prepared for each request type and for each statistic. The method in Ref. [4] detected many performance anomalies in RUBiS with these control charts.

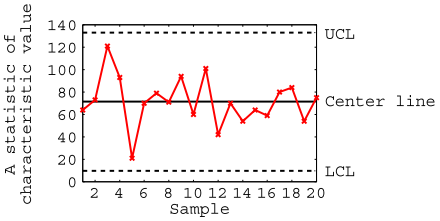


Fig. 3 Example of control chart.

3. Our Method

The key insight behind our method is that the measurements of request types that are negatively affected by the same root cause exhibit similar deviations from the standard. If an anomaly is occurring in component C in the example in Fig. 1, the measurements of request types R_1 and R_2 are expected to deviate similarly. If components A and B are anomalous, the measurements are expected to deviate differently because two different anomalies affect R_1 and R_2 differently.

Our method involves three steps. First, after a detector detects anomalous request types, we distill a *performance anomaly signature* from the measurements of each request type. A performance anomaly signature characterizes deviation from the standard. A signature in our method is represented as a bar graph. Second, we calculate the *similarity* of the signatures. The similarity is a scalar that represents the degree to which two signatures, i.e., two bar graphs, overlap each other. Finally, we cluster request types based on their similarities. If two or more request types are clustered together, this implies that they have been affected by the same root cause. Otherwise, they have been affected by different root causes.

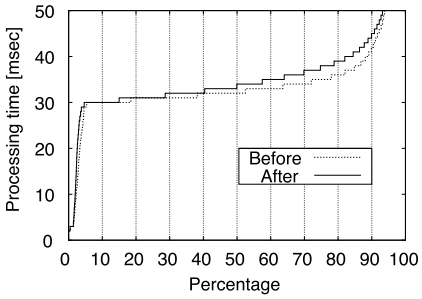
Our design of performance signatures is derived from our experience [4]. We are not going to claim that our design is the best. We will explain our method based on the processing times of each request type in this paper. We believe our method is independent of processing times and can be applied to other statistics such as the number of accesses or request/response size to web pages. In this paper, we employ the processing time as one example of the indicator of performance anomalies.

3.1 Signature of Performance Anomalies

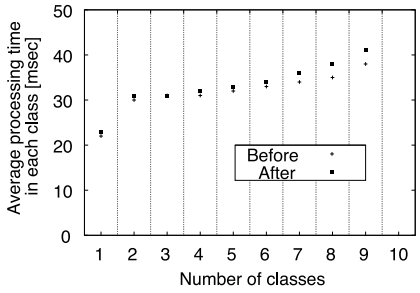
An anomaly signature should indicate how the "distribution" of the processing times has changed after an anomaly has occurred to characterize deviation from the standard processing times. This is not appropriate for signatures using the difference in simple statistics, such as averages and medians, because simple statistics cannot capture enough of the difference to distinguish root causes. For example, the average does not capture enough characteristics of the deviation. Even if the average increases similarly in two request types, there are many cases that should be distinguished. In one case, all requests' processing time may have uniformly increased. In another case, some requests may have exhibited spikes to increase the average. These two cases should be considered to have been caused by different root causes.

We calculate the difference between cumulative distribution functions (CDFs) of the processing times before and after a performance anomaly has been detected to distill an anomaly signature. Since the distribution of processing times can be expressed in a CDF, the difference between CDFs naturally captures the change in the processing time distributions. Hence, it is appropriate for signatures. Instead of CDFs, we can use histograms to represent the distribution of processing times because the histogram is equivalent to CDF in nature. As seen in **Fig. 4**, the difference in CDFs is calculated in the three steps.

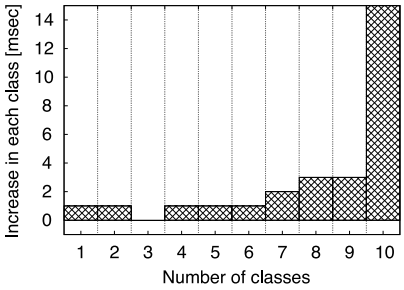
(1) We plot two inverse CDFs (before and after an anomaly has



(a) Inverse CDFs before and after an anomaly is detected.



(b) Simplified inverse CDFs.



(c) Our signature: increases in each class.

Fig. 4 Distilling performance anomaly signature.

been detected) of the processing times for an anomalous request type as shown in Fig. 4 (a).

(2) The inverse CDFs are simplified to simplify calculation. The X-axis of the inverse CDF is divided into n classes, each of which is represented by the average processing times in the class. A simplified CDF plots the representative of each class as plotted in Fig. 4 (b), where the X-axis is divided into 10 classes. The n in our case studies was set to 100.

(3) A performance anomaly signature is derived from the simplified CDFs. A signature is represented as a bar graph as plotted in Fig. 4 (c). A signature plots the representative of the "before" CDF subtracted from that of the "after" CDF for each class.

Our design of performance anomaly signatures is not perfect for three reasons. First, a single root cause may affect the processing times of two (or more) request types in a totally different way. When this occurs, a signature extracted for one request type is totally different from the one extracted for the other. As a result, these two request types are clustered differently to imply different root causes. This is not a serious problem because the anomalies in the two request types disappear if we fix the single root cause. Second, two root causes independent of each other happen to similarly affect the processing times of two (or more) request types. When this occurs, the signatures extracted for anomalous request types happen to be similar to each other. This can be compensated for by setting the resolution of clustering higher, as will be explained later in Section 3.3. If we set the resolution higher, anomalous request types are clustered differently due to a slight difference in signatures. Finally, two or more root causes may negatively affect a single request type. In this case, our method cannot determine the number of root causes; the root causes must be discovered and resolved individually. For example, there can be three more different pairs of root causes other than both A and B when request types R_1 and R_2 are clustered into different clusters in case of Fig. 1. In this case, we can concentrate on A and B at first, since either A or B has to be anomalous. Then, after fixing discovered problem, we are required to recursively cluster two request types and fix problems until both request types become healthy.

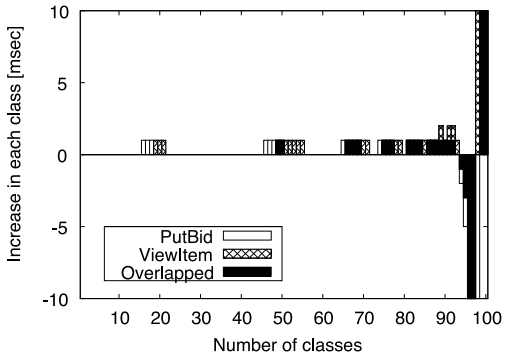


Fig. 5 Two signatures calculated for `PutBid` and `ViewItem` request types in RUBiS are superimposed. Black area indicates overlap between two signatures.

3.2 Similarities between Signatures

Our method calculates how much two signature bar graphs overlap to identify the similarity between two signatures. **Figure 5** depicts the area overlapped by two superimposing signatures, which have been calculated for `PutBid` and `ViewItem` request types in RUBiS. The larger the overlapping area, the more similar the two anomalies are. As described in the next paragraph, we prepare two variants of the similarity calculation to capture different features of performance anomalies. If we use an existing, more general method such as covariance matrix, it would be difficult to tune the similarity calculation for different types of anomalies. Again, we do not claim that our method is the best.

The similarity is normalized between 0 and 100. Below, we in-

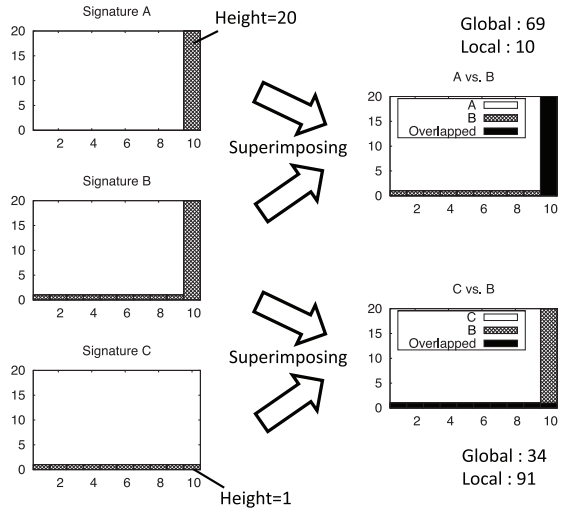


Fig. 6 Example signatures and overlapping areas. Three signatures A, B, and C are shown on left. Overlapping area of A and B and that of B and C are shown on right.

troduce two normalization algorithms for calculating similarities, i.e., *global* and *local* normalization. A similarity with global normalization captures the similarity of spikes in signatures, while a similarity with local normalization captures the similarity of slight but prevailing changes in signatures. The combination of these two algorithms enables us to distinguish root causes more precisely. If we can capture the similarity of spikes and that of slight but prevailing changes, we can employ another method of similarity calculation.

3.2.1 Similarities with Global Normalization

A similarity with global normalization captures the similarities of spikes in signatures. **Figure 6** outlines three signatures A, B, and C. Signatures A and B have similar spikes at the 10th class. A global normalization is designed to capture this similarity. A similarity with global normalization $G(X, Y)$ of two signatures X and Y is computed as

$$G(X,Y) = \frac{\sum_{1\le i\le n} X_i \cap Y_i}{\sum_{1\le i\le n} X_i \cup Y_i} \times 100,$$

where n denotes the number of classes in a signature (n is 10 in Fig. 6). Here, $X_i \cap Y_i$ means the intersection, i.e., the overlapping area, of the i-th bars in signatures X and Y. The $X_i \cup Y_i$ means the area of the union of the i-th bars in signatures X and Y, which is covered by the i-th bar in X or Y. In the example in Fig. 6, $G(A, B) = 20/29 \times 100 = 69$ and $G(B, C) = 10/29 \times 100 = 34$. Since signatures A and B have similar spikes, the resulting similarity is higher than that of signatures B and C. The area of spikes accounts for a large amount of both the denominator and numerator by globally normalizing similarity. As a result, spikes largely contribute to the results.

3.2.2 Similarities with Local Normalization

A similarity with local normalization captures the similarities of signatures whose anomalies are slight but prevailing across many classes. Signatures B and C in Fig. 6, have a similarity where the heights of the 1st-to-9th bars are constant and equal to one. This similarity cannot be captured with global normalization because it focuses on the similarity in spikes. A local normalization is designed to capture similarities that may be missed with global normalization. A similarity with local normalization $L(X, Y)$ can be calculated as

$$L(X,Y) = \sum_{1\le i\le n, \neg(X_i=Y_i=0)} \frac{X_i \cap Y_i}{X_i \cup Y_i} \times \frac{100}{n - \#z},$$

where $\#z$ is the number of zero-height bars in both signatures, i.e., the number of i's such that $X_i = Y_i = 0$. In the example in Fig. 6, $L(A, B) = (0/1 \times 9 + 20/20 \times 1) \times 10 = 10$ and $L(B, C) = (1/1 \times 9 + 1/20 \times 1) \times 10 = 91$. Since a local normalization is designed to capture similarities across many classes in signatures, $L(B, C)$ becomes higher than $L(A, B)$. All n classes equally contribute to the resulting similarity by normalizing similarity locally. As a result, $L(X, Y)$ captures the similarities in signatures in which an anomaly prevails across many classes.

Special attention must be paid when both X_i and Y_i are zero. If $X_i = Y_i = 0$, $X_i \cup Y_i$ becomes zero; thus, we cannot define $X_i \cap Y_i / X_i \cup Y_i$. Intuitively, if $X_i = Y_i = 0$, the i-th bars in signatures X and Y are exactly the same in the sense that there are no bars in either signature. Therefore, it seems natural to regard $X_i \cap Y_i / X_i \cup Y_i$ as one when $X_i = Y_i = 0$. Unfortunately, this does not work well. Suppose that there are two signatures S and T, where all bars in S are zero and one half of the bars in T are zero and the other half are one. Here, the similarity between S and T increases because the zero bars in S and T contribute to increase their similarity. However, these two signatures should be dissimilar. Signature S indicates a non-anomalous request type because the processing times have not changed. Signature T, on the other hand, is anomalous because the processing times have changed. To deal with this problem, our definition of a similarity with local normalization excludes zero-height bars from the summation. Then, $100/(n - \#z)$ is multiplied with the summation, to maintain $L(X, Y)$ ranging between 0 and 100.

A local normalization is also useful to ignore *natural fluctuations* in processing times. Processing times in web applications fluctuate extensively due to CPU or I/O scheduling, resource contention, and so on. Since these fluctuations are occasionally larger than anomalous changes in processing times, similarities with global normalization have possibility to erroneously conclude that two signatures affected by the same root cause are dissimilar, and vice versa; they have possibility to conclude that two signatures affected by different root causes are similar. In contrast, local normalization ignores such fluctuations. If a spike in the 10th class in Fig. 6 is caused by fluctuations, local normalization ignores the spike and concludes signatures B and C are more similar than A and B.

There are many alternatives in signature design. For example, to ignore natural fluctuations, it is attractive to drop the long tail of processing times. Unfortunately, this approach does not work well in the cases where the performance anomalies appear in the maximum processing times. As shown in the case study in Section 4.2, the maximum processing time is sometimes a good indicator of performance anomalies.

3.2.3 Aligning

If we apply two similarity-calculating equations as they are, similar but slightly shifted signatures are regarded as being dissimilar. The classes around 20 and 50 in Fig. 5 look similar, but

our basic method overlooks these. We compute the overlapping area by moving around the classes in a signature to regard these slightly shifted classes as similar. Basically, our method searches for a class of signature Y that provides the largest overlapping area to each class of signature X. Thus, overlapping area $X_i \cap Y_i$ in the two similarity-calculating equations $G(X, Y)$ and $L(X, Y)$ formulated in Section 3.2 becomes $X_i \cap Y_j$, where j denotes the number of classes that provide the largest overlapping area to X_i.

To prevent shifted signatures from gaining superfluous similarity (overlapping area), we introduce two penalties. The first is for classes that are far from their own matched classes. We decrease similarity by penalizing the overlapped area depending on how far one class is from the matched class. The second is for classes that are competing against each other for one class, e.g., X_1 and X_2 have the possibility of competing for Y_1. We decrease similarity by penalizing the overlapping area depending on how many classes are competing for one class. Even though these two penalties can be seen as being heuristic and not ideal, they worked well in our case studies and we thus believe they are reasonable.

3.3 Clustering

Similarities with global and local normalization were calculated for all pairs of signatures before clustering. We apply a method of clustering a set of similarities with global normalization and a set of similarities with local normalization and obtain two clustering results. The clustering method we used was the *group average method in hierarchical clustering*. This is a well known and widely used method of clustering.

The input for this method is a table that describes the similarities of all request types. The table to the left of **Fig. 7** informs us that the similarity value between request types A and B is 70, the similarity value between B and D is 20, and so on. The output from the method is a binary tree that is to the right of Fig. 7. Leaves of this binary tree corresponds to request types. Each node represents a cluster whose members are the descendant leaves of the node.

The algorithm works as follows. Each leaf at the beginning is considered to be one cluster and the method groups two clusters that are most similar to each other into one cluster, and recursively repeats the grouping until there is only one cluster. We calculate the similarity between two clusters to group them, which is calculated as the average similarity between all pairs of request types in the two different clusters. Request types D and E in our example in Fig. 7 are grouped since they have the greatest similarity. Then, request types A and B are clustered together. Now, we have three clusters: [A, B], [C], and [D, E]. Next, since [C] and [D, E] have the greatest similarity, [C] and [D, E] are grouped together; the similarity in this new cluster is the average similarity between all pairs of request types in [C] and [D, E] (C and D, C and E). This process is repeated until there is only one cluster.

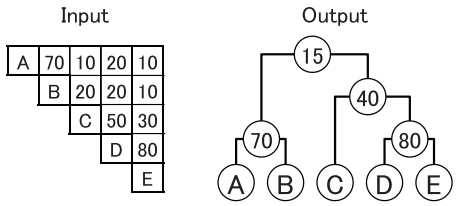


Fig. 7 Example of clustering input and output. Output tree at right can be obtained from input table at left.

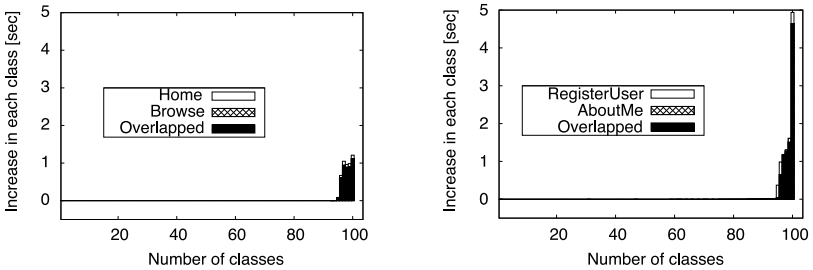
The grouping can be stopped when the similarity between two clusters becomes lower than a *clustering threshold*. If we set the threshold to 50, the resulting clusters are [A, B], [C], and [D, E] in our example. If the threshold is set to 30, the resulting clusters are [A, B] and [C, D, E]. This threshold was set to 60 in our case studies described in Section 4. As mentioned earlier in Section 3.1, we can control the resolution of clustering by changing this threshold. If the threshold is set higher, each cluster contains signatures very similar to one another; a slight difference in signatures is considered to be caused by different root causes. If the threshold is set lower, a slight difference in signatures is ignored and we can obtain rough distinction of root causes. An operator can start analyzing root causes with a lower clustering threshold as an option and repeatedly increase the threshold until root causes can be determined. A lower threshold yields fewer possible causes to an operator, since a cluster contains more request types and shared components are thus fewer.

4. Case Studies

To demonstrate the usefulness of our method, we conducted three case studies, where we used RUBiS [1], which is an auction site prototype modeled after eBay.com [2]. Our anomaly detector detected anomalous request types during the experiments. We clustered all the request types with our method to narrow down possible causes and deduced which components (machines, servers, Enterprise Beans, or methods) were anomalous with the help of the clustering results. Then, we manually extracted root causes from the narrowed down possible causes. After the root causes had been determined, we repaired them to confirm that the anomalies had disappeared.

We insist that our goal is to cluster anomalies based on root causes. Although manual effort is required to successively diagnose root causes, this is beyond the scope of this work. First, narrowing down possible causes is not trivial work; we need to search for components exclusively shared by a cluster. The example cases we presented in Section 1 using Figs. 1 and 2, in which narrowing down can easily be done, were simplified cases. As inside of real applications is much more complex, searching for shared components is not easy. However, readers who want to automate this task can refer to existing work, such as that by Ref. [8]. Second, even though manually extracting root causes from narrowed down possible causes is not trivial, we believe that the list of narrowed down possible causes would depend on monitoring granularity. Clustering results will become more useful if we apply finer-grained methods of monitoring.

We demonstrate in the first case study that our method was helpful in finding anomalous servers that caused performance anomalies, where our method clustered request types into two clusters. Two root causes (the first was in Apache and the second was in JBoss) were discovered and repaired using this information. Our method was demonstrated to be helpful in the second case study in discovering anomalous methods in RUBiS. We encountered a situation in the third case study to which our



(a) Superimposed signatures typical for C_a. (b) Superimposed signatures typical for C_b.

Fig. 8 Two typical superimposed signatures for C_a and C_b.

method was not easy to apply. We demonstrated that our method was still helpful in determining root causes. Since performance anomalies appear and disappear depending on server settings and workloads, we prepared many scenarios that differed in server settings and workloads, and chose three case studies in which performance anomalies occurred.

4.1 Experimental Setup

Our target was RUBiS [1] on the Java EE platform. RUBiS had a typical three-tier architecture that consisted of a front-end web server, an application server, and a back-end database server. RUBiS had 27 types of requests (e.g., `Home`, `SearchItemsInCategory`, and `PutBid`). We used a four-machine cluster in the experiments. Each machine consisted of four 3.00-GHz CPUs and 2 GB of main memory, and ran the Linux 2.6.27 kernel distribution by Red Hat. The four machines ran a RUBiS client emulator, Apache 2.2.11, JBoss 5.0.1, or MySQL 5.1.34, respectively. The RUBiS client emulator was slightly changed to fit our purposes. The default emulator caused a load peak for RUBiS because it activated all the client threads at startup. To avoid this load peak, we added 5-second intervals between the activation of each client. In addition, we introduced 8-second timeouts to avoid the clients waiting for server replies even when the server was not responding.

Similarity with global normalization was used in clustering when an anomaly was detected in maximum processing times, while similarity with local normalization was used when an anomaly occurred in the other statistics. As explained in Section 2, our anomaly detector observed the average, median, maximum, and minimum of the end-to-end processing time, i.e., the time between the receipt and the reply of each request on the server side. Since similarity with local normalization is insensitive to fluctuations in processing times, the similarity with local normalization was usually appropriate to ignore the noise caused by natural fluctuations. In contrast, when the maximum processing time increased without the other statistics changing, there were a few erroneous requests that took a long time to finish processing and thus, a spike appeared in the processing times. When this occurred, the similarity with global normalization was appropriate because it was designed to detect spikes in processing times. Note that if the overall processing time increases, other statistics (average or median) than the maximum will also increase and thus, we can use the similarity with local normalization in clustering. Throughout our case studies, we set the threshold for clustering to 60 as explained in Section 3.3.

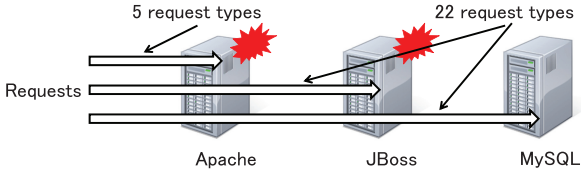


Fig. 9 Our hypothesis about root causes in case 1.

We note that we are not going to list the full input tables and output trees as in the form illustrated in Fig. 7. The full input and output of each case study were too large to be put in an academic paper (recall that RUBiS had 27 types of request). In addition, most of the input and output was useless in diagnosis.

4.2 Case 1: Finding Anomalous Servers

We ran servers with their default settings in this case study and calculated the baselines of control charts with the number of clients set to 200. After the baselines were calculated, we increased the number of clients from 200 to 300.

(Anomalous request types) When the number of clients was increased, our anomaly detector detected anomalies in the maximum processing times of 26 out of 27 request types; the maximums increased by several seconds.

(Clusters including one or more anomalous request type) The request types based on our method were clustered into two clusters C_a and C_b. Cluster C_a included five request types ([`Home`, `Browse`, `Register`, `Sell`, `AboutMe (auth form)`]) and cluster C_b included the rest ([`RegisterUser`, `AboutMe`, `SelectCategoryToSellItem`,...]). **Figure 8** depicts two pairs of superimposed signatures obtained in this case study. Pairs of signatures from clusters C_a and C_b are superimposed on the left and right of Fig. 8. Since the anomalies were detected in maximum processing times, the similarity with global normalization was used to cluster the request types.

(Narrowing down possible causes with the help of clustering result) To narrow down the possible causes, we investigated all the request types to identify which layers in RUBiS handled which request types, and noticed that all the request types in cluster C_a were handled by Apache and never passed through to either JBoss or MySQL. All the request types in cluster C_b, on the other hand, passed through to either JBoss or MySQL. This situation is shown in **Fig. 9**. Since the request types in cluster C_a were affected by the anomaly, there should be at least one root cause in the Apache layer. Although all the request types in cluster C_b were also affected by the anomaly, there should be another root

cause in either the JBoss or MySQL layer because C_b was a different cluster than C_a. Since all the request types in C_b were handled by JBoss but some were not handled by MySQL, we assumed that there was another root cause in the JBoss layer.

After we narrowed down possible causes guided by our clustering results, we started manually hunting for root causes in the Apathe and the JBoss layer. To begin with, we investigated the logs generated by JBoss, which informed us that the number of threads used to communicate with Apache reached the maximum number specified by the `maxThreads` parameter. To confirm whether improperly setting `maxThreads` was a root cause, we changed the parameter from 200 (the default) to 250 and ran the same workload that caused the anomaly.

(Anomalous request types No. 2) The number of anomalous request types decreased to 11; this number was not zero since there would have been another root cause in the Apache layer.

(Clusters including one or more anomalous request type No. 2) We clustered the request types again to find the root cause in the Apache layer. We only obtained one cluster, which indicated that all the request types were negatively affected by the same root cause.

(Narrowing down possible causes with the help of clustering result No. 2) This clustering result was the same as the one we had expected. We had assumed there had been two root causes; the first was in the JBoss layer and the second was in the Apache layer. Therefore, as we had repaired the root cause in the JBoss layer, there would only be one root cause in the Apache layer that affected all the request types because Apache handled all the request types.

We then probed the root cause in the Apache layer apart from the clustering results. First, we investigated the Apache logs to find the root cause. Unfortunately, as there was no useful information, we used the following heuristics. When the maximum processing time increases, there is shortage in resource in most cases. If a request arrives when the resource is available, it is processed immediately without increasing the maximum processing time. If it arrives when the resource is not available, it is delayed until the resource becomes available again and the maximum processing time increases enormously; a spike appears in the processing times because only *unlucky* requests are affected by the anomaly.

Based on this heuristics, we changed the performance parameters in Apache that were related to resource allocation. We discovered by trial and error that `KeepAliveTimeout` should be set smaller than the default in this case study. If `KeepAliveTimeout` is set longer, Apache keeps unnecessary connections and runs short of network connections. To test and confirm this, we changed `KeepAliveTimeout` from five (the default) to two and repeated the same experiment. The anomaly disappeared and all the request types (except one) raised no alarms.

4.3 Case 2: Finding Anomalous Methods

We demonstrated that our method was helpful in this case study for finding anomalous methods, where we used a scenario in which `KeepAliveTimeout` was set to one and `maxThreads` was set to 400. These settings were considered reasonable since the first case study indicated `KeepAliveTimeout` should be less than five (the default) and `maxThreads` should be larger than 200 (the default). In addition, two indexes were added to the `items` and `users` tables in the database since the case study discussed in the next section indicated the overall performance improved with these indexes.

(Anomalous request types) When we increased the number of clients from 200 to 300 in this scenario, anomalies were detected in median processing times in five request types (`SearchItemsInCategory`, `SearchItemsInRegion`, `ViewItem`, `ViewUserInfo`, and `ViewBidHistory`).

(Clusters including one or more anomalous request type) Each request type with our clustering method was clustered into one cluster ([`SearchItemsInCategory`], [`SearchItemsInRegion`], [`ViewItem`, `PutBid`], [`ViewUserInfo`], and [`ViewBidHistory`]).

There was one interesting cluster, [`ViewItem`, `PutBid`]. Its superimposed signatures have already been described in Fig. 5. This cluster contained request type `PutBid`, which was not considered anomalous according to our anomaly detector. We will concentrate our explanation on this cluster in this case study for two reasons. First, the investigation into root causes was simplified to some extent because `PutBid` had been included. Second, this cluster indicated an anomaly in the method shared by these request types. We used similarity with local normalization to cluster the request types since the anomalies were detected in median processing times.

(Narrowing down possible causes with the help of clustering result) We investigated the source code of RUBiS to narrow down possible causes and our strategy for the investigation was as follows. Since the cluster only contained `ViewItem` and `PutBid`, our focus was on the components that were used exclusively by these two request types. The granularity in this quest for root causes should be fine-grained because anomalies in coarser-grained components such as Apache and JBoss would affect many request types as in the first case study. Consequently, we focused on EJB components. We searched for components exclusively used by `ViewItem` and `PutBid`. They are possible causes of anomalies.

Then, we sought out root cases. We chose a method that did relatively complex operations from the list of possible causes and that was prone to be erroneous. This method (`SB_ViewItemBean.getItemDescription`) displayed information about a requested item (recall that RUBiS is an auction site). Investigating this method carefully, we noticed that method `getItemDescription` displayed the minimum bid for a requested item. If the quantity (n) of the item was greater than one, i.e., $n > 1$, the minimum bid was the n-th maximum bid plus one dollar. If the quantity was one, i.e., $n = 1$, the minimum bid was always the last bid plus one dollar. Since the former case needed much more complex calculations, the ratio of the total to multiple items affected the median processing time of

getItemDescription.

The RUBiS workload was designed so that the percentage of multiple items was around 20%. If workload was generated expectedly, the median processing time for getItemDescription had to be almost constant. Unfortunately, the current implementation of the RUBiS workload had two bugs that set the initial ratio around 10% and made the ratio gradually approach the correct value (around 20%). This gradual change was considered anomalous by our anomaly detector, which was based on control charts. As explained in Section 2, the control charts do not regard random changes within a fixed range as being anomalous (the measured statistics fall within LCL and UCL). However, gradual change is considered as being anomalous because gradually increasing (or decreasing) statistics eventually exceed UCL (or LCL).

We fixed the bugs in the RUBiS workload generator to test and confirm that these bugs were root causes. We repeated the same experiment after the bugs had been fixed. The ratio throughout the experiment kept fluctuating randomly around 20%; thus, the median processing time for the two request types remained constant. Hence, our anomaly detector raised no alarms.

There are two things to be noted from this case study. The first is that all request types, whether anomalous or non-anomalous, should be clustered in our method. PutBid was not anomalous in this case study according to the detector but it was clustered together with ViewItem. If PutBid was not clustered, we had to investigate all the source code related to ViewItem. By clustering PutBid together with ViewItem, we could narrow down possible causes into the components (servers, Enterprise Beans, or methods) shared by them and found getItemDescription, which was the key to analyzing root causes.

The other thing to be noted is that this case study did not deal with real anomalies; it only dealt with bugs in workload generators. The most important thing here is whether our method was helpful in identifying root causes. It is not important where the root cause was. Finding a bug in workload generators is useful in many scenarios. Imagine you are doing a performance test using a workload generator and a performance anomaly is detected during the test. This anomaly may be caused by the workload generator but you have to hunt for the root cause to ensure your product is not causing the anomaly. We believe our method would be helpful in this situation.

4.4 Case 3: 9 Anomalies Happening Simultaneously

We encountered a situation to which our method was not easy to apply in this case study but still helpful in identifying root causes. We used a scenario in which KeepAlive was turned off in this experiment, which conformed to our experience in the first case study that KeepAliveTimeout should be short. We found and fixed a bug in the RUBiS client emulator during the case studies and used this modified client emulator in this experiment.

(Anomalous request types) Performance anomalies were detected in this scenario in average processing times in nine request types (RegisterUser, SearchItemsInCategory, BrowseRegions, SearchItemsInRegion, ViewUserInfo, ViewBidHistory, BuyNow, PutComment, and AboutMe), while we were calculating the baselines for the control charts when there were 200 clients.

Recall that the baselines must be calculated in our anomaly detector before the performance test. Since the anomalies were detected in the baseline-calculating phase, we did not know when they had appeared; we could not calculate anomaly signatures because the signatures needed to be calculated by comparing the distribution of processing times before and after an anomaly had occurred. We divided the baseline-calculating phase into two halves to apply our method and calculated the signatures by comparing the earlier and the later halves. In other words, the earlier half was regarded as healthy and the later was regarded as anomalous. This was not strictly correct but our method worked well in this case study.

(Clusters including one or more anomalous request type) After clustering, we obtained nine clusters each of which contained only one request type from the nine anomalous request types. Similarity with local normalization was used in clustering since the anomalies appeared in the average processing times.

(Narrowing down possible causes with the help of clustering result) This implied that nine anomalies had been incurred by different root causes. **Figure 10** has an example of the superimposed signatures, from which we can see that they were different.

Let us summarize the results obtained from analysis before describing the root causes for each cluster. We could identify root causes for five out of nine clusters. Each cluster corresponded to different root causes; thus, our method was helpful in narrowing down possible causes. Unfortunately, we could not find the root causes for the other four clusters. This was because these anomalies appeared and disappeared randomly when we repeated the experiment. Therefore, we believe these anomalies were caused by natural fluctuations. **Figure 11** plots the control charts used to detect these anomalies. The control chart for RegisterUser at the left shows an anomaly that we could not solve, while the control chart for ViewUserInfo at the right shows an anomaly that we could solve. As you can see from these charts, the chart for RegisterUser has spikes and thus, RegisterUser is supposed to be affected by natural fluctuations. This was a problem with our anomaly detector, rather than our clustering method. Introducing a simple filter that drops spikes into our anomaly detector can be used to suppress these alarms. The chart for

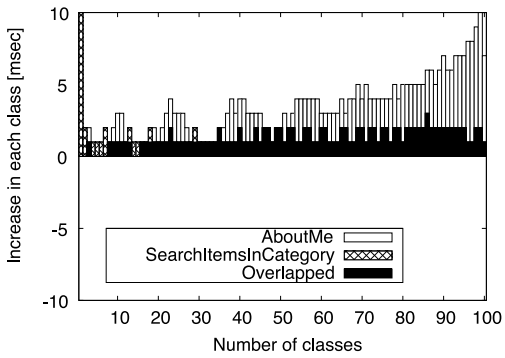
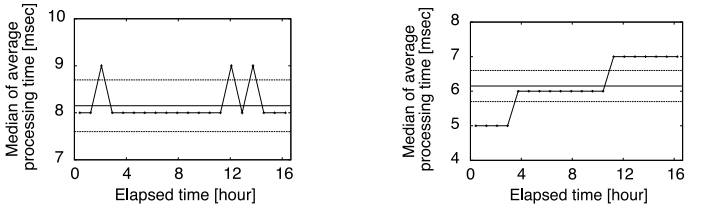
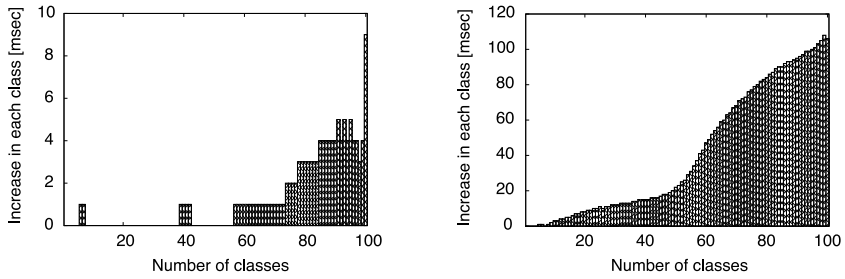


Fig. 10 Superimposed signatures: AboutMe and SearchItemsInCategory.



(a) A control chart of `RegisterUser`. (b) A control chart of `ViewUserInfo`.

Fig. 11 Typical control charts in case study 3.



(a) A signature of `ViewUserInfo`. (b) A signature of `ViewBidHistory`.

Fig. 12 Two signatures whose request types shared coarse-grained updates of `end_date`. Note that scales on Y-axis are different.

`ViewUserInfo` shows a gradual increase in the processing times and thus, `ViewUserInfo` was considered to be anomalous.

(Narrowing down possible causes with the help of clustering result continued) We will briefly describe the root causes for five anomalous request types (or clusters) in the following. We could narrow down possible causes for the five request types (`SearchItemsInCategory`, `SearchItemsInRegion`, `AboutMe`, `ViewUserInfo`, and `ViewBidHistory`) in this case study. We used the following strategy to narrow down possible root causes for each cluster. Nine anomalous request types were clustered differently into nine clusters in this case study, each of which contained one anomalous request type. Therefore, each request type was supposed to be affected by different root causes. This was the same in the second case study, where we tried to find fine-grained components, such as Enterprise Beans and methods that were used exclusively by each request type. This strategy worked well and we were able to find the root causes from the list of possible causes.

`SearchItemsInCategory` and `SearchItemsInRegion` were affected by the slowdown caused by the growth in the `items` table in the database. The lack of a proper index in `SearchItemsInCategory` for a table slowed down the processing time. As a result, we created an index from `category` and `end_date` columns, and the anomaly disappeared. The problem was more complicated in `SearchItemsInRegion`. `SearchItemsInRegion` accessed the `users` table in addition to the `items` table. We had to duplicate some columns from the `users` table to the `items` table to resolve the slowdown. This solution was not ideal since a single datum needed to be duplicated over two tables. We gave up on trying to fix this problem.

An anomaly in `AboutMe` was caused by a bug in RUBiS. This request type displayed a list of items won in a bid by a specified user apart from other pieces of information. Our investigations into the RUBiS source code revealed that RUBiS issued an SQL that gathered all the items bid for by the specified user, without checking whether they had been in the winning bid made by him/her. We were able to successfully remedy this anomaly by correcting this SQL.

`ViewUserInfo` and `ViewBidHistory` were caused by improper updates to the `end_date` column on the `items` table in the database. This column indicated when the auction would be closed for each item. Improperly setting this column caused a large number of comments to be handled in `ViewUserInfo` and a large number of bids to be handled in `ViewBidHistory`, and this resulted in performance anomalies. `ViewUserInfo` displayed the comments stored for the user specified in the request. `ViewUserInfo` and `StoreComment` (for storing comments) were sent to users in the RUBiS workload who had sold items, which were displayed and sorted by `end_date` in ascending order. `end_date` was always set to seven days by default and thus the same items were always displayed on the front page and their sellers received an enormous number of `StoreComment` requests. As a result, `ViewUserInfo` took a long time to display a large number of comments. The situation in `ViewBidHistory` was similar to what occurred in `ViewUserInfo`; the history of bids suddenly increased for the same items. We changed the update script to set `end_date` randomly from zero to seven days with a resolution of one second to mitigate this performance anomaly.

Some readers may have expected that `SearchItemsInCategory` and `SearchItemsInRegion` were clustered together because both were affected by the growth in the `items` table. `ViewUserInfo` and `ViewBidHistory` should be clustered together in the same way. If we introduce another method of calculating signature similarities, these request types can be clustered together. **Figure 12** shows the signatures of `ViewUserInfo` and `ViewBidHistory`, which look similar in shape but dissimilar in height (up to 10 in `ViewUserInfo` and up to 120 in `ViewBidHistory`). A similarity function that normalizes the height of each bar and ignores the height, or one that normalizes the area of each signature and ignores the area,

may cluster them together. However, this similarity function may also introduce many false positives; request types affected by different root causes may be clustered together. Keeping in mind this trade-off, we concluded that our current similarity functions were reasonable.

5. Related Work

To the best of our knowledge, there has been no work that has focused on the clustering of performance anomalies. The methods most related to ours are those by Bodík et al. [13], Cohen et al. [14], and Yuan et al. [15] who determined whether current anomalies were previously observed ones. If a current anomaly is similar to a previous one, we can reuse our experience in diagnosing the previous anomaly. These methods are related to ours because they attempted to determine if two anomalies were similar. However, we cannot rely on these methods to cluster performance anomalies because we must cluster anomalies that occur *simultaneously*.

The methods proposed by Bodík et al. [13] and Cohen et al. [14] to determine the similarities between anomalies used performance metrics, such as CPU loads or disk I/O rates. The collected metrics were used to construct anomaly signatures. Unfortunately, these metrics did not fit in with our purposes since they monitored the overall behavior of the entire system; thus, the negative effects of different root causes were combined.

The method proposed by Yuan et al. [15] used system call traces to distill anomaly signatures. A signature consists of a system call name, parameters, a return value, and so on. The goal with their method was totally different from ours, which was not to diagnose performance anomalies but to diagnose *errors* that caused incorrect behaviors, such as "The page cannot be displayed in web browsers." Since performance anomalies do not appear in the traces, the signatures used in their method cannot be applied to fulfill our objectives.

Our clustering method was inspired by other research efforts. Many researchers have pointed out that the processing times include valuable information on diagnosing performance anomalies. Joukov et al. [16] applied the similarity between distributions of processing times to operating system profiling. This method collects the histograms of processing times in each kernel function and searches for valuable pairs whose histograms are slightly different. This information is useful in finding performance bugs. For example, if the histogram of the `read` system call issued by a single process differs slightly from that issued by multiple processes, this slight difference can imply race conditions. Our use of CDFs was inspired by a method proposed by Chen et al. [8], in which CDFs were used to discern differences in processing times and to detect performance anomalies. Our clustering method can be seen as being an extension to their approach because it involves performance anomaly signatures that distill the difference in two CDFs.

Many methods of detecting anomalies have already been proposed. We believe most of these techniques can be used as a back-end detector if we tailor our clustering algorithm to the measurements obtained with these methods. A method proposed by Bodík et al. [5] observes and analyzes the number of accesses to each web page with two statistical methods. Combined with the results of statistical analysis, a visualizer that displays the number of accesses to each web page helps the operators identify the root causes of performance anomalies. Aguilera et al. [6] and Tak et al. [7] use inter-machine requests to discover how requests flow through a cluster. This information enables the administrators to find time-consuming or overloaded machines. System metrics such as CPU loads or I/O rates are monitored to construct a classifier model which determines if the current state is healthy [10]. Logs generated by applications are automatically analyzed to model the internal states of the applications [11].

To enable us to monitor system behaviors at fine-grained granularity, the methods by Chen et al. [8] and Chanda et al. [9] augment servlets, EJB containers, or servers' libraries. Since fine-grained components are monitored in detail, these methods can find anomalous Enterprise Beans or library functions. As previously discussed in Section 1, it is almost impossible to augment all components so that no root causes share any components due to the increased overhead and cost of implementation. We believe our method is also useful for these fine-grained monitoring systems.

After the root cause of a performance anomaly is determined, we can use the anomaly avoidance methods proposed by Xi et al. [17] and Sugiki et al. [18] to avoid this problem.

6. Conclusion

Performance anomalies in web applications are becoming a huge problem and the increasing complexity of modern web applications demands techniques for narrowing down possible causes of these anomalies. We proposed a method which helps us narrow down suspicious components that cause performance anomalies. In this method, performance anomalies, i.e., anomalous monitoring regions, that are negatively affected by the same root cause are clustered together.

We could narrow down suspicious components that were used exclusively by the monitoring regions in the same cluster from these clustering results. We demonstrated that our clustering method was helpful in narrowing down possible causes through three case studies where RUBiS was used, which is an auction site prototype. We searched for components commonly used by request types in other clusters to reach suspicious components throughout the case studies. By manually investigating the suspicious components, we could determine root causes, i.e., improper settings for Apache and JBoss servers, bugs in RUBiS workload generators, and bugs in SQL handling, and so on.

The back-end anomaly detector in our current implementation was limited in its detection capabilities because it only monitored the end-to-end processing time of each request. By extending our method to use other measurements such as CPU loads and I/O rates, it can provide information that is much more useful for the diagnosis of root causes.

The evaluation using real workloads is left for future work. We believe that our approach works well in real workloads if the CDF represents the distribution of processing times correctly. If the number of anomalous requests is too small to create reliable CDFs, our method does not work well due to the incorrectness in

the signatures.

Reference

[1] RUBiS: Rice University Bidding System, available from ⟨http://rubis.objectweb.org/⟩.
[2] eBay.com, available from ⟨http://www.ebay.com/⟩.
[3] Kohavi, R., Henne, R.M. and Sommerfield, D.: Practical Guide to Controlled Experiments on the Web: Listen to Your Customers not to the HiPPO, *Proc. International Conference on Knowledge Discovery and Data Mining*, ACM (2007).
[4] Iwata, S. and Kono, K.: Narrowing Down Possible Causes of Performance Anomaly in Web Applications, *Proc. European Dependable Computing Conference* (2010).
[5] Bodík, P., Friedman, G., Biewald, L., Levine, H., Candea, G., Patel, K., Tolle, G., Hui, J., Fox, A., Jordan, M.I. and Patterson, D.: Combining Visualization and Statistical Analysis to Improve Operator Confidence and Efficiency for Failure Detection and Localization, *Proc. International Conference on Autonomic Computing*, IEEE (2005).
[6] Aguilera, M.K., Mogul, J.C., Wiener, J.L., Reynolds, P. and Muthitacharoen, A.: Performance Debugging for Distributed Systems of Black Boxes, *Proc. Symposium on Operating Systems Principles*, ACM (2003).
[7] Tak, B.C., Tang, C., Zhang, C., Govindan, S., Urgaonkar, B. and Chang, R.N.: vPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities, *Proc. Annual Technical Conference*, USENIX (2009).
[8] Chen, M.Y., Accardi, A., Kiciman, E., Lloyd, J., Patterson, D., Fox, A. and Brewer, E.: Path-Based Failure and Evolution Management, *Proc. Symposium on Networked Systems Design and Implementation*, USENIX (2004).
[9] Chanda, A., Cox, A.L. and Zwaenepoel, W.: Whodunit: Transactional Profiling for Multi-Tier Applications, *Proc. European Conference on Computer Systems*, ACM (2007).
[10] Cohen, I., Goldszmidt, M., Kelly, T., Symons, J. and Chase, J.S.: Correlating instrumentation data to system states: A building block for automated diagnosis and control, *Proc. Symposium on Operating Systems Design and Implementation*, USENIX (2004).
[11] Xu, W., Huang, L., Fox, A., Patterson, D. and Jordan, M.I.: Detecting Large-Scale System Problems by Mining Console Logs, *Proc. Symposium on Operating Systems Principles*, ACM (2009).
[12] Xie, M., Goh, T. and Kuralmani, V.: *Statistical Models and Control Charts for High Quality Processes*, Springer (2002).
[13] Bodík, P., Goldszmidt, M., Fox, A., Woodard, D.B. and Andersen, H.: Fingerprinting the Datacenter: Automated Classification of Performance Crises, *Proc. European Conference on Computer Systems*, ACM (2010).
[14] Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T. and Fox, A.: Capturing, Indexing, Clustering, and Retrieving System History, *Proc. Symposium on Operating Systems Principles*, ACM (2005).
[15] Yuan, C., Lao, N., Wen, J.-R., Li, J., Zhang, Z., Wang, Y.-M. and Ma, W.-Y.: Automated Known Problem Diagnosis with Event Traces, *Proc. European Conference on Computer Systems*, ACM (2006).
[16] Joukov, N., Traeger, A., Iyer, R., Wright, C.P. and Zadok, E.: Operating System Profiling via Latency Analysis, *Proc. Symposium on Operating Systems Design and Implementation*, USENIX (2006).
[17] Xi, B., Liu, Z., Raghavachari, M., Xia, C.H. and Zhang, L.: A Smart Hill-Climbing Algorithm for Application Server Configuration, *Proc. International World Wide Web Conference* (2004).
[18] Sugiki, A., Kono, K. and Iwasaki, H.: Tuning mechanisms for two major parameters of Apache web servers, *Software: Practice and Experience*, Vol.38, No.12, pp.626–634 (2008).

Satoshi Iwata was born in 1983. He received his B.E. and M.E. from Keio University in 2007 and 2009, respectively. He is currently a Ph.D. student in Keio University. His current research interest is dependable computing. He is a student member of IPSJ and ACM.

Kenji Kono received his B.Sc. degree in 1993, M.Sc. degree in 1995, and Ph.D. degree in 2000, all in computer science from the University of Tokyo. He is an associate professor of the Department of Information and Computer Science at Keio University. His research interests include operating systems, system software, and Internet security. He is a member of IEEE/CS, ACM and USENIX.