An Unsupervised Diagnosis Tool for Locating fine-grained Performance Anomalies in Cloud Computing Systems

Saleem Malik S[1], Maruthi G.B[2], Pratap M.S[3], S.P Prashanth Kumar [4]
[1,3,4]Asst.Professor, CSE Dept, [2]Asst. Professor, ECE Dept
[1,2,3,4]KVGCE, Sullia, Karnataka, India
[1]baronsaleem@gmail.com [2]maruthgb@gmail.com [3]pratapms@gmail.com [4]prasahantheshwar2010@gmail.com

Abstract **— Among cloud computing systems, performance diagnosis is labor intensive in production. These type of systems face many real world challenges, which the existing diagnosis techniques for such distributed systems cannot effectively solve. An efficient, unsupervised diagnosis tool for locating fine-grained performance anomalies is still lacking in production cloud computing systems. This paper proposes CloudDiag to bridge this gap. Combining a statistical technique and a fast matrix recovery algorithm, CloudDiag can efficiently pinpoint fine-grained causes of the performance problems, which does not require any domain-specific knowledge to the target system.**
.

Keywords- Cloud computing, performance diagnosis and request tracing

I. INTRODUCTION

Performance diagnosis is labor-intensive, especially for typical production cloud computing systems. In such systems, a lot of software components bear a large number of replicas (component instances) distributed in different physical nodes in the cloud. They can be assembled into multiple types of services, serving large amounts of user requests. The services provisioned by the cloud are often prone to various performance anomalies caused by software faults, unexpected workload, or hardware failures. Such defects may however be manifested only in a small part of component replicas, hiding themselves in a large number of normal component replicas.

Main objective of this paper is to achieve fine grained, unsupervised and scalable performance diagnosis for production cloud computing systems. In order to achieve this objective we propose CloudDiag. CloudDiag periodically collects the end-to-end tracing data (In particular, execution time of method invocations) from each physical node in the cloud. It then employs a customized Map-Reduce algorithm to proactively analyze the tracing data. Specifically, it assembles the tracing data of each user request, and classifies the tracing data into different categories according to call trees of the requests. When the cloud system is suffering performance degradation (*e.g.*, average response time of user requests is larger than a threshold), a cloud operator can access CloudDiag with its web interfaces to conduct a performance diagnosis. With the request tracing data, CloudDiag will perform a fast customized matrix recovery algorithm to instantly identify the method invocations (together with the replicas they locate) which contribute the most to the performance anomaly. The whole process requires no domain specific knowledge to the target service.

II. RELATED WORK

Extensive work has employed explicit annotation basedinstrumentation to conduct performance monitoring, tuning and debugging for distributed systems. Work from past, applies application-specific event schemas to correlate the resource consumption of individual requests with the goal of understanding performance also compare users' actual behavior with self-defined expectation to determine whether a request is anomalous or not. It is very hard to construct these models because they require much specific domain knowledge. Compared to them, CloudDiag considers the intrinsic characteristics of request latencies to determine the anomalous method invocations, which requires no specific domain knowledge[1][2].

Existing system traces request call relationship in multi-layers of Web service components and adopts a clustering algorithm to group failure and success logs. It can only find out the anomalous components. In comparison, CloudDiag can identify the latency anomalous methods together with corresponding physical replicas. Methods in past can be utilized to identify the performance anomalies that manifest themselves as the change in the ratios of the chosen call trees, while CloudDiag can localize the latency-anomalous methods within call trees[3][2].

There are also many performance diagnosing techniques that rely on specific types of data, combine console logs with source codes to construct performance features and conducts the principal component analysis (PCA) to detect problems. However, these techniques generally focus on locating anomalous logical components instead of replicas.

III. FRAMEWORK OF CLOUD-DIAG

Performance anomalies in cloud systems will manifest themselves as anomalous response time of user requests. Since a service is composed of a lot of components, a service with anomalous performance must have involved some components with performance anomalies. A component typically has a lot

of replicas in a production cloud system; however, the performance anomaly of a component may be manifested only in a small part of its replicas. This will cause the performance degradation of the involving service. Such performance problems are the most difficult to locate, because the anomalous methods hide themselves in numerous well-functioning replicas. Therefore, to reduce human efforts in pinpointing performance anomaly, a performance diagnosis tool must first identify which component methods contribute to the performance anomaly, and then locate the component replicas that execute the methods. An efficient, unsupervised diagnosing tool that can pinpoint the fine-grained causes of performance anomalies is of critical importance to production cloud computing systems. To this end, we propose CloudDiag, a tool for performance diagnosis in production cloud computing systems. Fig 1 provides a system-level overview of CloudDiag. CloudDiag is composed of three major parts, *i.e.*, 1) collecting the performance data, 2) assembling the performance data, and 3) identifying the primary causes of the anomalies. These steps are summarized as follows.

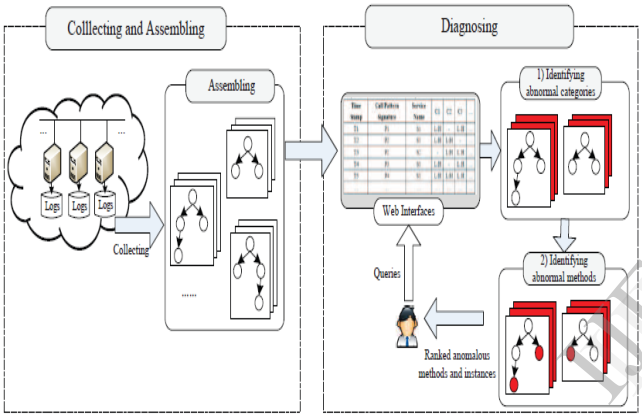

Fig 1: Overview of CloudDiag

First, CloudDiag traces user requests at a given sampling rate to expose performance data. For the sampled requests, each component replica records the performance data and saves them in its local storage. An important consideration is what kind of performance data CloudDiag should collect and how. CloudDiag adopts an instrumentation-based approach that collects the execution time of each component
method. Second, Assemble performance data. CloudDiag should first assemble the performance data distributed in numerous component replicas in a requestoriented way. In other words, the performance data belonging to the same requests are correlated together. CloudDiag will then analyze such request-oriented performance data and infer the call tree of each sampled request. A customized map-reduce process is utilized to group requests into different categories based on their call trees. Requests within one category share the same call tree. Third, Identify the primary causes of anomalies. CloudDiag then identifies the anomalous categories according to their latency distribution. Then for each anomalous category, a fast customized matrix recovery algorithm is employed to identify the anomalous method invocations together with the replicas they are located. [6][7]

IV. PERFORMANCE DATA COLLECTION

We introduce what kind of performance data that CloudDiag should collect and how to collect them. Our instrumentation-based tracing approach will produce performance data when a sampled request is being processed in each component replica. Specifically, each component method, when being invoked or returning, will generate a log entry. The data structure of a tracing log entry is shown in Fig 3, which contains five items. *Host* indicates the machine where the component replica locates. *Timestamp* records the time of the event occurrence (*i.e.*, a method invocation or a method return). *RequestID* is the global identifier of a request. *MID* is a unique identifier for request tracing purpose, which will be discussed later. The *Method* field saves the name of the method invoked. Lastly, *Flag* indicates whether this is a method invocation or a method return. Fig 1 also shows two example log entries that record the invocation of the *AliStorage.readFile* method and its return. *RequestID* should be unique for every request. It is assigned when a request arrives the system. Typically a cloud service may have multiple entry nodes for the same type of requests. To guarantee the uniqueness of *RequestID*, an entry node will assign the *RequestID* (a unique 64-bit integer) as the concatenation of two integers: one is the unique number to identify the entry node *per se*, and the other is incremental with each new request. CloudDiag uses the invocation relationship between methods to model a request. To obtain such relationship, CloudDiag resorts to the chronological order of the method invocations. One challenge of request tracing is to cope with the clock drifts of nodes, which are inevitable in production cloud systems. Previous approaches generally assume that the clock drifts are negligible. However, this is not true in production cloud computing systems[6][7].

Even with a Network Time Protocol (NTP) to synchronize the clocks, the deviations between the clocks of different cloud nodes are still in millisecond-level. Previous approaches simply employ only a global identifier (*i.e.*, *RequestID*) to mark the distributed tracing record and use their timestamps to infer the invocation relationships with methods. As a result, clock drifts may lead to the wrong order of method invocations when merging the distributed logs generated in processing a request. For example, the start time-stamp of the callee in one host may be earlier than the start timestamp of the caller in another host. To guarantee the order of the invoked methods, we design hierarchical identifiers to trace a request. Specifically, before a node calls a method in another node, besides passing the *RequestID* to the callee, the caller also generates an *MID*, a unique integer, for the callee. When a request enters the first component method in the system (*i.e.*, the request entry method), the *MID* of the method is initially set identical to the *RequestID*. CloudDiag then records the *MID*s of the caller and the callee in the logs of the caller as well. Fig 3 shows such a log format. Thus, CloudDiag can recover the caller-callee relationships according to the *MID*s. The order of method invocations can then be correctly recovered and an entire performance trace of a

request can thus be obtained.

V. DIAGNOSING ANOMALIES WITHOUT DOMAIN KNOWLEDGE

In this section, CloudDiag first employs a statistical technique to detect anomalous categories that contain latency-anomalous requests. Then, from anomalous categories, a fast matrix recovery algorithm, namely, robust principal component analysis (RPCA), is adopted to identify the anomalous methods and instances. Details are as follows.

A) Identifying Anomalous Categories

Normal and anomalous replicas exist simultaneously when performance problems occur. For a component, the same service requests may pass through normal instances as well as anomalous instances. The response latency of a request will be influenced by anomalous instances that it passes through. Normal and anomalous requests may share the same call tree and be grouped into one category. Hence, the latency distribution of requests within the same category
could be utilized to detect whether it contains latency anomalous requests or not. Requests within one category have the same call tree; hence, the response latencies should be close to each other. A category is considered to be normal if the latencies of requests within the category are clustered in a specific range; on the contrary, a category is considered to be anomalous if the latencies are over dispersed. In this regards, we choose the coefficient of variation (CV) to measure the distribution of a set of data. Let α be the threshold. A category is defined to be anomalous if its CV is larger than α.

B) Identifying Anomalous Methods

In an anomalous category of requests, our aim now is to pinpoint the anomalous method invocations that are responsive for the performance anomaly of the requests. For such a category of requests, we can create an m × n matrix M, where n is the number of the invoked methods in the corresponding call tree and m is the number of the requests that bear the same call tree. Mij denotes the execution time of the jth method when depth-first traversing the call tree ofthe ith request. Column M(j) denotes the invocation time vector of the jth method. Intuitively, we can identify the anomalous method by measuring the execution time deviation of each method one by one. However, this cannot capture the correlations of the invoked methods, and will hence cause imprecise diagnosis results. Furthermore, such a statistical analysis can only identify anomalous methods, but cannot find out on which replicas the anomalous methods are executed. Hence, we design an unsupervised machine learning algorithm to automatically learn the characteristics of the invoked methods and identify which methods are anomalous together with on which replicas they are executed.

Although PCA is one of the most popular algorithms for anomaly detection, it only works well to the data in which the errors (in our problem domain, errors in the data are caused by the performance anomaly) follows the Gaussian distribution. However, the execution time of the anomalous methods is actually corrupted by large errors, which does not follow the Gaussian-distribution assumption of PCA. Such large errors caused by anomalous methods will cause PCA to produce imprecise diagnosing results .

To solve the problem, we propose to use the robust principal component analysis (RPCA) for the anomaly detection task. RPCA is an algorithm for high dimensional matrix completion. When a matrix M is corrupted by gross sparse errors, RPCA can decompose the full matrix M as: M = L+E, where L is a low-rank matrix with non-corrupted columns and E is a sparse matrix with a few non-zero corrupted columns. The matrix M is the input of RPCA. Therefore, the problem of identifying anomalous methods in a category is transferred into the process of recovering a matrix with unknown corrupted latency columns. After obtaining the non-corrupted matrix L and error matrix E, we can identify the corrupted columns (*i.e.*, the anomalous methods) from E. The anomalous methods refer to those columns that are farthest from the true column space. For the i-th column in original matrix M and non-corrupted matrix L, the extension of deviation can be measured as:

$$\beta=\cos\theta= \|M(i) \cdot L(i)\|_1 \,/\, \|M(i)\|_2\|L(i)\|_2 \qquad (1)$$

where θ represents the angle between column M(i) and column L(i). The larger the angle is, the more deviation the column L(i) is away from the true space. A method is defined to be anomalous if β is smaller than a given threshold. For each anomalous method (*i.e.*, the corrupted column), anomalous replicas are located by checking the entries of the corrupted column in Matrix E. With the row and column indices of the corrupted entries, we can get the physical addresses of anomalous replicas from physical paths. Since the same method (running on the same component replica) may be identified to be anomalous in different categories, we calculate the times that it is identified to be anomalous. The larger the number of times is, the more suspicious the method is. Cloud-Diag can then rank the methods in descending order of the number of times that they are identified to be anomalous, which can direct the operators to localize the primary cause of performance anomaly

V. FUTURE WORK AND CONCLUSION

Request tracing technologies have been proven effective in performance debugging. In this paper, Cloud-Diag resorts to a white-box instrumentation mechanism to trace service requests, since the source codes of services are generally available in typical production cloud systems. Note that such a white-box performance data acquisition component of CloudDiag can also be substituted with another tracing mechanism if it can obtain the latency data of method invocations. Another way to trace requests is via black-box mechanisms.

Black-box tracing mechanisms assume no knowledge of the source codes. But, existing approaches generally cannot directly obtain the latency data of method invocations. In this regard, a black-box tracing mechanism can be deemed as a tracing mechanism with the large granularity (*e.g.*, in node

level). There is a tradeoff between tracing granularity and debugging effort. As a result, more effort will be

increased in trouble-shooting the performance anomalies if a black-box tracing mechanism is applied. To incorporate black-box tracing mechanisms with CloudDiag, a future direction is to explore blackbox tracing mechanisms so that a fine granularity (*i.e.*, in method invocation level) can be achieved. To this end, the runtime instrumentation can be a promising technique[7].

This paper proposes CloudDiag, an efficient, unsupervised diagnosis tool for locating finegrained performance anomalies.

REFERENCES

[1] A. Chanda, A. Cox, and W. Zwaenepoel, "Whodunit: Transactional profiling for multi-tier applications," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, 2007, pp. 17–30.

[2] B. Sang, J. Zhan, G. Lu, H. Wang, D. Xu, L. Wang, and Z. Zhang, "Precise, scalable, and online request tracing for multi-tier services of black boxes," *IEEE Transactions on Parallel and Distributed Systems*, no. 99, pp. 1–16, 2010.

[3] H. Mi, H. Wang, Y. Zhou, M. R. Lyu, and H. Cai, "P-tracer: Path-base performance profiling in cloud computing systems," in *Proceedings of IEEE COMPSAC*, 2012, pp. 509–514.

[4] A. Chanda, A. Cox, and W. Zwaenepoel, "Whodunit: Transactional profiling for multi-tier applications," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, 2007, pp. 17–30. Z. Lin, M. Chen, L. Wu, and Y. Ma, "The augmented lagrange multiplier method for exact recovery of corrupted low rank matrices," *Arxiv preprint arXiv:1009.5055*, 2010.

[5] B. Sigelman, L. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Technical report dapper-2010-1. Google, Tech. Rep., 2010.

[6] V. Emeakaroha, M. Netto, R. Calheiros, I. Brandic, R. Buyya, and C. De Rose, "Towards autonomic detection of sla violations in cloud infrastructures," *Future Generation Computer Systems*, vol. 28, pp. 1017–1029, 2011.

[7] M. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based faliure and evolution management," in *Proceedings of USENIX NSDI*, 2004, pp. 23– 36.