2012 IEEE 36th International Conference on Computer Software and Applications

P-Tracer: Path-based Performance Profiling in Cloud Computing Systems

Haibo Mi[1], Huaimin Wang[1], Hua Cai[2], Yangfan Zhou[3], Michael R Lyu[1,3], Zhenbang Chen[1]
National Lab for Parallel & Distributed Processing, National Univ. of Defense Technology, Changsha, China[1]
Computing Platform, Alibaba Cloud Computing Company, Hangzhou, China[2]
Dept. of Computer Science & Engineering, The Chinese Univ. of Hong Kong, Hong Kong[3]

***Abstract*—In large-scale cloud computing systems, the growing scale and complexity of component interactions pose great challenges for operators to understand the characteristics of system performance. Performance profiling has long been proved to be an effective approach to performance analysis; however, existing approaches do not consider two new requirements that emerge in cloud computing systems. First, the efficiency of the profiling becomes of critical concern; second, visual analytics should be utilized to make profiling results more readable. To address the above two issues, in this paper, we present P-Tracer, an online performance profiling approach specifically tailored for large-scale cloud computing systems. P-Tracer constructs a specific search engine that adopts a proactive way to process performance logs and generates particular indices for fast queries; furthermore, P-Tracer provides users with a suite of web-based interfaces to query statistical information of all kinds of services, which helps them quickly and intuitively understand system behavior. The approach has been successfully applied in Alibaba Cloud Computing Inc.[1] to conduct online performance profiling both in production clusters and test clusters. Experience with one real-world case demonstrates that P-Tracer can effectively and efficiently help users conduct performance profiling and localize the primary causes of performance anomalies.**

***Keywords*-Performance profiling; performance anomaly; visual analytics.**

I. Introduction

Performance maintenance is one of the most essential processes in software performance engineering [1]. One requirement of performance maintenance is to ensure that system performance complies with the Service Level Agreement [2], [3]. However, modern distributed systems (e.g., cloud computing systems) are continuously growing in scale, along with the complexity of component interactions, which creates great challenges for operators to understand the characteristics of system performance.

For instance, in Alibaba cloud computing platform, user requests will pass through many kinds of service components that are deployed on different hosts, which generates many types of execution paths. As a result, operators have much difficulty in knowing the system behavior. Examples include: *1) Where do the requests spend most of their time? 2) Which types of execution paths are the critical paths that are most frequently passed through? 3) When the system performance changes beyond expectation, which service components might be the primary causes of the problem?* Therefore, it could be helpful for them to have an effective tool to profile system performance.

Request tracing approaches [4], [5], [6], [7] are effective to performance debugging. It can record the execution information of individual requests, for example, the entering and exiting time when individual requests go through service components. These approaches are useful for operators to understand the casual relationships of component invocations. However, they are not capable to performance profiling for cloud systems due to the following reasons.

First, the efficiency of a performance profiling tool is of critical concern. In cloud computing systems, thousands of requests are served per second, which generates tremendous trace logs. For example, more than 12 million lines of trace logs are generated for a 350-host cluster in an hour in Alibaba cloud computing platform. It may take a long time for a simple query to calculate the aggregate statistical information for one type of execution paths in the past 24 hours, which is undesirable for operators. Second, one graph is worth a thousand logs [8]. It is more helpful to visualize the information for operators to identify hidden performance anormalies. However, most current approaches do not provide visualization techniques for massive trace logs.

To support fast performance analysis for large-scale cloud computing systems, in this paper we propose an online performance profiling tool, namely P-Tracer. P-Tracer is based on end-to-end request tracing technologies and provides a statistical insight into exectution time consumed by requests in each part of the systems. The contributions of this paper are as follows:

- A particular search engine is constructed for trace logs, which adopts a proactive way to process trace logs and generates particular indices for fast queries.
- Statistical profiling information for requests are visualized to operators with a high readable way, which helps them quickly understand system behaviors and reason about performance changes.

This paper is organized as follows. In Section II, we briefly introduce end-to-end request tracing technologies and

[1]Alibaba Cloud Computing Inc. is a subsidiary of Alibaba Inc., one of the largest e-commerce companies in the world. Our work is carried out in Alibaba Cloud Computing Inc.

0730-3157/12 $26.00 © 2012 IEEE
DOI 10.1109/COMPSAC.2012.69

```
main(...){
    ENABLE_TRACE();
    MethodA();
}

MethodA(){
    TRACE_LOG();
    MethodB();
    ……
}

MethodB(){
    TRACE_LOG();
    MethodC();
    ……
}

MethodC(){
    TRACE_LOG();
    ……
}
```

Figure 1. An example of explicit instrumentation.

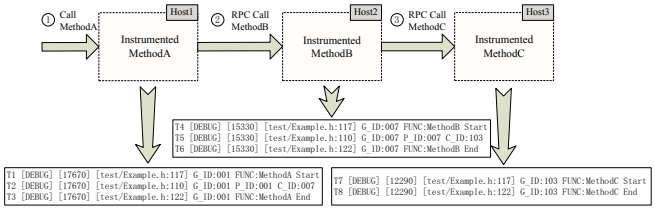


Figure 2. A request passes through three instrumented methods that are in different hosts. Instrumentation points record contextual information into logs. These logs contain the current time stamp, log level, process number, line number of statement generating the log, GID for the request, name of the invoked method and flag signifying the start or end of the invocation. When the request spans between hosts, the GID for the request changes. The first GID (i.e., 001) for the request is defined as root GID.

the structure of distributed trace logs. Section III discusses the design details of the search engine. Section IV presents visual dimensions of statistical information of system performance. In section V, we report one real-world case to validate the effectiveness of P-Tracer. Section VI compares our approach with the related work. In Section VII, we conclude this paper.

II. Background

A. End-to-end request tracing technology

Our profiling tool relies on end-to-end request tracing technology to generate distributed trace logs. One characteristic of end-to-end request tracing technology is to record the execution information of individual requests. This information can be utilized to profile services. Note that the tracing technology is not our research scope in this paper. Hence, although there have been advanced dynamic instrumentation approaches (e.g., [9], [4]), for simplicity, we just select the source-level instrumentation approach. Figure 1 shows an instrumentation example of our target system. The interface of $ENABLE\_TRACE()$ invoked in $main$ is utilized to activate a tracing process, which generates a global identifier for the request. The interface of $TRACE\_LOG()$ is utilized to record contextual information into logs when instrumented methods are invoked.

B. Request trace logs

The trace logs could be utilized to profile system performance. In large-scale cloud computing systems, a request may span many hosts. The upper half of Figure 2 shows an process in which a request spans three hosts and invokes three instrumented methods. When a user request is tagged with a global identifier (GID) that is randomly generated, all instrumented methods that it passes through will record contextual information into local log files, as shown in lower half of Figure 2. When the invocation of an instrumented method starts, a line of log is recorded that sequentially contains the current time stamp, log level, process number, line number of statement generating the log, GID (a unique 64-bit integer), name of the invoked method and flag signifying the start of the invocation, as shown in the first line of three local log files in Figure 2. When the invocation ends, another line of log will be recorded. The content is almost the same as that of the first line except that the flag changes to signify the end of the invocation, as shown in the last line of three local log files in Figure 2.

Former researches (e.g., Stardust [10]) always use one global identifier to mark a request and construct call relationships of instrumented methods depending on the sorted time-stamps in trace logs. However, it does not work in large-scale cloud computing systems because of the clock skews of hosts. Although the clocks of hosts have been synchronized by Network Time Protocol [11], there are still millisecond-level deviations between the clocks. Because requests may span many hosts and the time-stamps in logs are recorded according to local clocks, it may lead to disorder of time-stamps when dispersed logs are merged together. For instance, the start time-stamp of the callee in one host is earlier than the start time-stamp of the caller in another host. To avoid clock drift, we change the GID for a request when it spans networks between hosts and use the parent-child relationship of GIDs within one request to construct its call relationship. The change process is recorded as a line of log that has the same structure with other logs except explicitly marking the original and new value of the GID. For example, when the instrumented $methodA$ makes a RPC call for the instrumented $methodB$, the GID changes from 001 to 007, as shown in the second line of the lower left part in Figure 2.

III. Constructing Search Engine

In order to online provide operators with the statistical information of services, we need to construct a specific search engine to preprocess execution paths. The challenge is how to effectively merge distributed massive trace logs and generate suitable indices.

In this section, we discuss the design details of the search engine. As shown in Figure 3, it contains four parts: 1) extract the key parameters from raw trace logs and collect the refined logs from production and test clusters to a analysis cluster; 2) parse the refined logs into call trees and

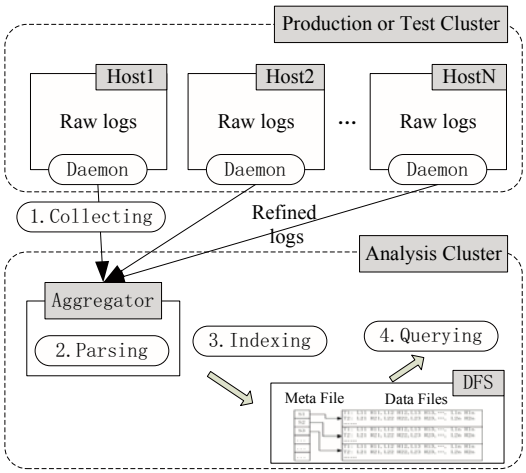


Figure 3. The structure of search engine.

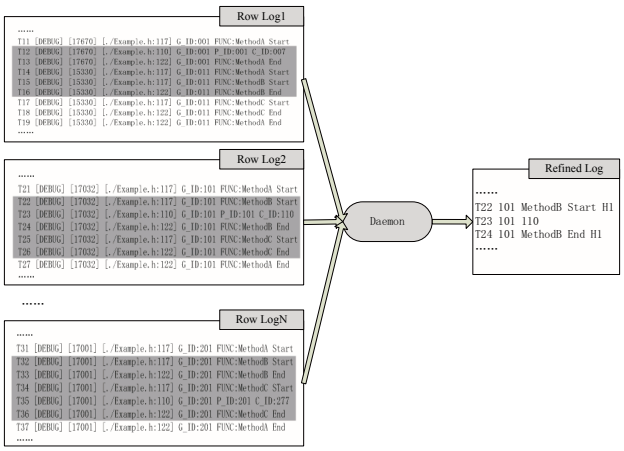
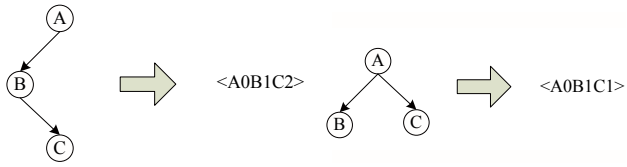


Figure 4. An example of refining raw logs in one host. All related lines under the desired time window (i.e., the grey parts) are merged into a refined log in an ascending order of time stamps.



(a) Example one (b) Example two

Figure 5. Examples of generating call sequences from call trees. without the depth information, the same signature will be constructed for the two call trees.

call sequences; 3) group structure-identical call sequences together and generate the indices for them; 4) provide users with web-based interfaces to do online query.

A. Collecting

A log collecting daemon that collects raw trace logs at a controlled time interval is deployed for each host in production and test clusters. In order to reduce the overhead of network traffic, daemons do not send the raw logs to the analysis cluster directly, but first extract the primary parameters from them.

Figure 4 shows an example of the extracting process in one host where numerous raw logs are generated. First, the selected lines (i.e., grey parts) that are in the desired time window are extracted by the daemon from all raw logs; then the daemon filters out all redundancy information (e.g., process number and log level) and puts the refined lines into one file (called refined file). Note that the daemon appends the host address to the refined lines in order to keep physical information. Finally, the daemon compresses the file and sends it to the aggregator daemon in the analysis cluster. After refining, the volumes of network transmission are decreased by 9/10.

B. Parsing

After collecting the refined logs together, an aggregator uses them to generate call trees of individual requests and corresponding call sequences. One call tree corresponds one execution path pattern.

The process is as follows. First, the aggregator traverses all refined logs to generate one temporary file, which is a pair list of parent GIDs and child GIDs. The GIDs without parent GIDs are the root GIDs. Second, the aggregator assigns them to distributed computing nodes, together with the temporary file of parent-child GID relationships. Those computing nodes use a map-reduce [12] process to generate call trees and corresponding call sequences.

Call sequences are constructed by adopting a depth-first traversal of call trees. Elements in the call sequence are comprised of the nodes in the call tree plus corresponding depths. Through concatenating the method name and the depth of each node, we can get a string representation that is the signature of the call tree. Using the logs in Figure 2, Figure 5(a) plots a call tree and its corresponding signature where X is short for $methodX$. Without adding the depths of the nodes, different the call trees may generate the same signature. For instance, Figure 5(b) plots another the call tree and its corresponding signature. If removing the depths of the nodes, the two call trees shown in Figure 5 will generate the same signature $\langle ABC \rangle$.

C. Indexing

Next, the aggregator makes an index for the call sequences of requests. Recall that each call trees (i.e., execution path pattern) has a unique signature that is the string representation of invoked methods. For each time interval of log collecting, all kinds of signatures are merged together and stored into a meta file. Call sequences with the same signatures are kept into one data file. Figure 6 shows an example of the structures of a meta file and three data files.

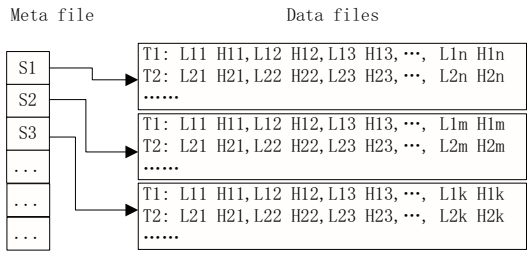


Figure 6. The structure of indexing.

SX in the meta file is short for the value of signature X. TX in the data files is the time stamps of requests when they enter the system. Elements in one call sequence are kept sequentially into a row of one data file. LX and HX denote the response latency and host address of the invoked method respectively. Meta and data files are stored into the distributed file system of the analysis cluster.

D. Querying

Last, the search engine provides users with web-based interfaces to facilitate query operations. It not only supports the query of statistic information for a time window (e.g., types of call sequences and the latency distribution), but also allows users to compare performance changes between any two time periods. It is quite useful for operators to find differences between two load tests. The result is represented in a user-friendly way, which will be introduced in the next Section in detail.

IV. Visualization for Performance Profiling

P-Tracer provides users with web-based interfaces to construct ad hoc queries, which makes it easy to access and retrieve information from the search engine. Figure 7 displays the primary interfaces and parts of results that match the query parameters. Additionally, users can modify any of the parameters (e.g., filter out requests with latencies smaller than 100 millisecond) to the current query to create a detailed profile view.

P-Tracer supplies multi-dimensional statistical information to help operators in-depth understand the system performance. For space consideration, we introduce two typical metrics. Note that all figures are drawn according to the application of mail service. Call trees in figures have been simplified and all real method names are also replaced. To facilitate the presentation, we will use the term of call tree to refer to the execution path pattern.

A. Call Tree Overviews

In cloud computing systems, for a kind of service, there will involve thousands of requests per second that may take different kinds of call trees. Different call trees imply different semantics. For example, two call trees can be constructed depending on whether the required file is in the cache or not. P-Tracer supplies users to query the statistical information of overall call trees for one kind of service in a desired time range. The left part of Figure 7 shows a query result for the ListMail service, which includes statistical information about each call tree, such as the request count in each type, frequency and average latency. The types are listed in the descending order of frequency. From this figure, operators can easily observe which call trees are the critical paths. Ideally, requests within the same call tree should have approximate latencies, however, many factors may cause latency fluctuation. Types in which latencies of requests are over dispersed are defined to be suspicious. We use the measure of coefficient of variation (by default, the threshold is set to be 1) [13] to pick suspicious types and highlight them. As shown in the left part of Figure 7, there are two suspicious types (i.e., type C and type E) that are labeled as red color.

B. Shape and Statistics of a Particular Call Tree

When users click on a particular call tree, P-Tracer not only visualizes its shape, but also shows statistical information of each node in the call tree, including maximum, minimum, average latency and the ratio of the average latency of the node to that of the call tree. The latency of root node is defined as the latency of corresponding call tree. The latency ratios of nodes in one call tree are plotted as swimlanes, which helps users observe statistical time consumption in every part of invocation and provides an intuitive way of understanding the bottlenecks of the services, as shown in the right part of Figure 8.

C. Shape Comparison between Call Trees

P-Tracer supports to distinguish shape differences between call trees. Figure 9 shows a comparison of two call trees for the ListMail service. We can see the first type has two more RPC call processes than the second type. Through comparing the differences between execution paths, operators can learn whether methods are being invoked as their expectation and further infer whether the latency of one call tree is suspicious or not.

V. Validation

P-Tracer has been successfully applied in Alibaba Cloud Computing Inc. to employ online performance profiling both in production clusters and test clusters. These clusters are under different workloads and load conditions. All clusters are equipped with the Alibaba cloud computing platform, which contains a series of service components, such as distributed scheduler, storage, communication, monitor. There are about one hundred instrumented points in the systems. Default sampling policy for tracing in clusters is at 1 out of 200 requests. All the trace logs from those clusters are dumped into a 10-host analysis cluster at every controlled time interval (1 hour by default). There are about 2 million

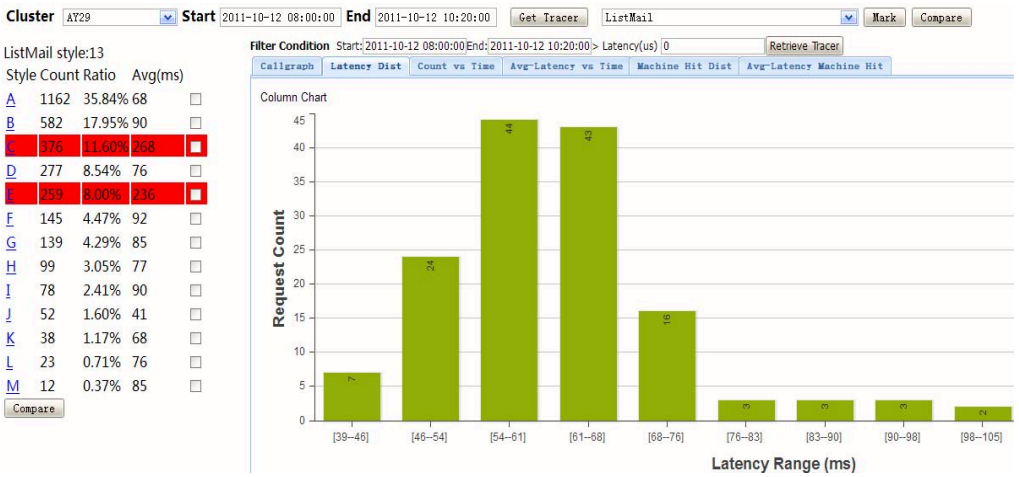


Figure 7. The main page of P-Tracer contains primary interfaces to let users query what they are interested in. Users can query a specific metric within a desired time range. It also supplies links for users to modify parameters of the current query, such as, filtering out requests with latencies smaller than 100 millisecond.

Function name	Max(ms)	Min(ms)	Avg(ms)	Ratio(%)
ListMail	55.12	3.57	6.52	100%
--ClientRead	11.99	0.59	1.04	16%
----RPCCall	1.32	0.06	0.17	3%
-------SeverReadRow	21.97	0.91	2.02	31%
--ClientRead	13.72	0.68	0.91	14%
----RPCCall	1.31	0.85	0.26	4%
-------SeverReadRow	21.86	0.79	2.09	32%

Figure 8. Shape and statistical information for a particular call tree.

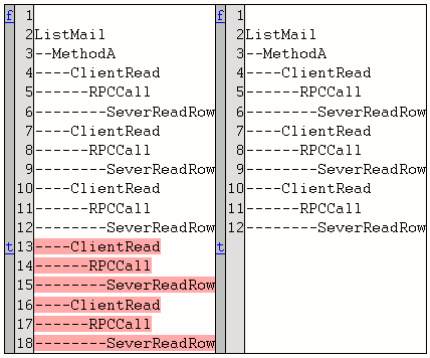


Figure 9. Shape comparison of two call trees.

lines of raw logs generated in a 100-host cluster. It costs less than 5 minutes to construct new call sequences and import them into distributed file system of the analysis cluster. In this section, we introduce one experience of using P-Tracer to do performance profiling for the mail service in production and test clusters.

After a system upgrade, an operator utilized P-Tracer to do performance profiling and found a design defect. When checking the overall call trees of the SendMail service (used to send mails), he observed that there was a call tree that contained a method labeling the transaction aborted. The frequency of this call tree was more than 25 percent and the average latency was about 3 times larger than those of other call trees. After trying different scales of time windows (from one hour to two weeks), he found the ratio of this call tree was steady.

He presented this result to the relevant developers. They were surprised and created a bug report to investigate it. Afterwards, they found that the root cause was a design defect of sending group mails. When sending mails to a group, the redundancy mails were not wiped off in the application level but directly dispatched to the system level. In the system level, each sending mail would invoke a transaction to store the content. When a mail was kept by the former transaction, other transactions to store the same mails would be aborted. After the developers transferred the logic of wiping off redundancy mails to the application level, the performance of the SendMail service increased about one fifth.

Not like functional problems that will directly cause break down of the systems, performance problems are hard to detect as they are influenced by many factors. Without P-Tracer, developers have much difficulty in understanding the behavior of requests in a fine granularity and inferring hiding performance bottlenecks. Furthermore, P-Tracer supports user to online profile system performance in a large time scale, which can statistically reflect the system behavior more precisely.

VI. Related Work

Diverse single process-oriented performance profiling tools are proposed to help developers identify time-consuming parts of the computation within one node. DTrace

[4] and DARC [14] visualize the execution of systems as call graphs to signify where requests spend time in a single node. Misailovic et al. [15] uses a loop perforation approach to optimize the tradeoff between execution time and quality of service. Our work builds on some ideas from these researches, however more factors have to be considered in cloud computing systems, such as large volumes of trace logs, clock drift and complex execution paths.

There are many tracing approaches [6], [16], [17] that focus on performance debugging and diagnosing in distributed systems. The most related work to P-Tracer is Dapper [16], which is a distributed performance analysis tool in Google. There is a major difference between Dapper and P-Tracer. Dapper keeps trace logs of one request into Bigtable as a row, with the global identifier as the primary key; whereas, P-Tracer groups the requests with the same type into a data file and designs an index to search them, which is more efficient to compute statistical information.

VII. Conclusion

Currently, distributed systems are continuously growing in scale and complexity of component interactions, which poses great challenges for operators to online capture the characteristic of system performance. This paper presents P-Tracer, an approach to support the proformance profiling in cloud computing systems. Experience with one real-world case demonstrates P-Tracer can effectively help users do performance profiling and localize the primary causes of performance anomalies.

Acknowledgment

This research is supported by the National Basic Research Program of China under Grant No.2011CB302600, the National High Technology Research and Development Program of China under Grant No.2012AA011201, the National Natural Science Foundation of China (NSFC) under grant No. 91018004, 60903043, 61100077, and an NSFC/RGC Joint Research Scheme sponsored by the Research Grants Council of Hong Kong and NSFC (Project No. N_CUHK405/11)

References

[1] M. Woodside, G. Franks, and D. Petriu, "The future of software performance engineering," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 171–187.

[2] J. Trienekens, J. Bouman, and M. Van Der Zwan, "Specification of service level agreements: Problems, principles and practices," *Software Quality Journal*, vol. 12, no. 1, pp. 43–57, 2004.

[3] L. Chung and J. do Prado Leite, "On non-functional requirements in software engineering," *Conceptual modeling: Foundations and applications*, pp. 363–379, 2009.

[4] B. Cantrill, M. Shapiro, and A. Leventhal, "Dynamic instrumentation of production systems," in *Proceedings of the annual conference on USENIX Annual Technical Conference (ATC)*. USENIX Association, 2004, pp. 2–15.

[5] M. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based faliure and evolution management," in *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2004, pp. 23–36.

[6] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *Networked Systems Design and Implementation*, 2007.

[7] Z. Zhang, J. Zhan, Y. Li, L. Wang, D. Meng, and B. Sang, "Precise request tracing and performance debugging for multi-tier services of black boxes," in *Proceedings of 39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN),*. IEEE, 2009, pp. 337–346.

[8] M. Aharon, G. Barash, I. Cohen, and E. Mordechai, "One graph is worth a thousand logs: Uncovering hidden structures in massive system event logs," *Machine Learning and Knowledge Discovery in Databases*, pp. 227–243, 2009.

[9] Ú. Erlingsson, M. Peinado, S. Peter, and M. Budiu, "Fay: extensible distributed tracing from kernels to clusters," in *Proceedings of the 23th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2011, pp. 311–326.

[10] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. Ganger, "Stardust: tracking activity in a distributed storage system," *ACM SIGMETRICS Performance Evaluation Review*, vol. 34, no. 1, pp. 3–14, 2006.

[11] D. Mills, "Network time protocol (version 3) specification, implementation and analysis," 1992.

[12] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[13] H. Abdi, "Coefficient of variation," *Encyclopedia of research design/Ed. N. Salkind*, pp. 1–5, 2010.

[14] A. Traeger, I. Deras, and E. Zadok, "Darc: Dynamic analysis of root causes of latency distributions," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 1, pp. 277–288, 2008.

[15] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of service profiling," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 25–34.

[16] 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.

[17] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, "Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers," *Micro, IEEE*, vol. 30, no. 4, pp. 65–79, 2010.