Front. Comput. Sci., 2013, 7(3): 431–445
DOI 10.1007/s11704-013-2193-4

RESEARCH ARTICLE

An online service-oriented performance profiling tool for cloud computing systems

Haibo MI(✉)[1], Huaimin WANG[1], Yangfan ZHOU[2], Michael Rung-Tsong LYU[2], Hua CAI[3], Gang YIN[1]

1 National Lab for Parallel & Distributed Processing, National University of Defense Technology, Changsha 410073, China
2 Shenzhen Research Institute, The Chinese University of Hong Kong, Shenzhen 518000, China
3 Computing Platform, Alibaba Cloud Computing Company, Hangzhou 310000, China

Abstract The growing scale and complexity of component interactions in cloud computing systems post great challenges for operators to understand the characteristics of system performance. Profiling has long been proved to be an effective approach to performance analysis; however, existing approaches confront new challenges that emerge in cloud computing systems. First, the efficiency of the profiling becomes of critical concern; second, service-oriented profiling should be considered to support separation-of-concerns performance analysis. To address the above issues, in this paper, we present P-Tracer, an online performance profiling tool specifically tailored for cloud computing systems. P-Tracer constructs a specific search engine that proactively processes performance logs and generates a particular index for fast queries; second, for each service, P-Tracer retrieves a statistical insight of performance characteristics from multi-dimensions and provides operators with a suite of web-based interfaces to query the critical information. We evaluate P-Tracer in the aspects of tracing overheads, data preprocessing scalability and querying efficiency. Three real-world case studies that happened in Alibaba cloud computing platform demonstrate that P-Tracer can help operators understand software behaviors and localize the primary causes of performance anomalies effectively and efficiently.

Received June 1, 2012; accepted November 3, 2012

E-mail: rainmhb@gmail.com

Keywords cloud computing, performance profiling, performance anomaly, visual analytics

1 Introduction

In recent years, cloud computing has surged in popularity in software industry to deliver Internet-based services. The growing scale and complexity of typical cloud computing systems post great challenges for system operators to understand system performance and diagnose problems. Profiling has long been proved to be an effective approach to software analysis [1]. Although, many profiling tools for stand-alone softwares [2], high-performance computing systems [3] and distributed systems [4] have been proposed recently, ranging from profiling kernel functions [5] to application-level methods [4], an urgent and critical concern is whether these existing tools can be applicable to cloud computing systems, i.e., whether there are emerging design requirements of performance profiling tools for cloud computing systems.

To answer this problem, we interview over 30 operators in Alibaba Cloud Computing Company[1)], which designs and maintains a data-centric cloud computing service platform to the public. Their experiences reveal that many new special design requirements of performance profiling for large-scale cloud computing systems. We summarize them as the following two aspects.

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

First, the efficiency of the performance profiling tool is of critical concern in cloud computing systems. A cloud computing system requires providing non-stop services to a huge number of concurrent users. For a typical service, thousands of requests are served every second, which can generate massive performance data. For example, more than 12 million lines of performance logs can be generated in a 350-host cloud platform in Alibaba Cloud Computing Company in an hour. Moreover, the huge volume of performance logs is distributed across a large number of cloud hosts. The performance profiling tool should therefore handle such massive, distributed logs efficiently. In particular, when an operator analyzes a performance problem, the tool should retrieve and combine the performance logs of interest (e.g., those for a type of service in a given time period) to generate a report in a timely manner.

Second, to cope with the specifics of cloud computing systems, more comprehensive performance information should be captured, organized, and visualized by the profiling tool. A typical cloud computing system provides multiple applications to the public simultaneously. Each application (e.g., mail application) will handle user requests of many different types, each of which conducts a particular service (e.g., sending an email). The internal components of the systems can be simultaneously accessed by user requests of different services. Furthermore, requests to the same service may involve different call trees. Different call trees have different response latencies. For example, consider two SendMail requests: one sends an email with an attachment and the other sends an email without. The two requests, although both being SendMail requests, have different response latencies. Hence, service-oriented profiling should be considered, so as to support separation-of-concerns performance analysis. In other words, requests should be analyzed separately according to their service types and call trees. These can greatly facilitate operators to understand system behavior or conduct a quick diagnosis of performance problems.

Unfortunately, these emerging requirements towards a handy performance profiling tool for large-scale cloud computing systems are not the focus of previous profiling approaches. Those approaches basically target to aggregated functions profiling [6] or system resources profiling [7], rather than the behavior of services. Furthermore, in order to help the operators timely understand the behavior of the large-scale cloud computing systems, a profiling tool needs to provide the functionality of real-time statistical analysis. Since it works on the massive tracing data, query efficiency is one of key factors for a profiling tool to be utilized in such production system. However, few approaches (e.g., [4]) consider the query efficiency since they do not need to handle so large volumes of the distributed logs.

Our work aim at advancing the current state-of-the-art of performance profiling for large-scale cloud computing systems. In this paper, we present P-Tracer, an online performance profiling tool, which is specifically tailored for large-scale cloud computing systems. P-Tracer relies on the end-to-end request tracing technique to record the execution information of individual requests. Then, P-Tracer constructs a specific search engine that adopts a proactive way to process tracing logs and generates a particular index for fast queries; third, for each service, P-Tracer adopts a statistical insight to its characteristic of performance from multi-dimensions and provides operators with a suite of web-based interfaces to query such information. It helps them quickly and intuitively understand the system behavior. Currently, P-Tracer has been successfully launched in Alibaba Cloud Computing Company for performance profiling in its production clusters.

The contributions of this paper are as follows:

- We propose a profiling tool that can help operators conduct real-time performance analysis in large-scale cloud comouting systems. Compared to existing approaches, the tool can precisely trace requests without being affected by clock drifts; furthermore, the tool designs a specific index to help operators conduct statistical analysis more efficiently.
- We report three real-world cases in which P-Tracer helps operators conduct performance profiling and localize the primary causes of performance anomalies to demonstrate P-Tracer effectiveness.

This paper is organized as follows. In Section 2, we briefly introduce end-to-end request tracing technologies and our clock drifts avoided tracing approach. Section 3 discusses the design details of the search engine. Section 4presents visual dimensions of statistical information of system performance. In Section 5, we evaluate P-Tracer in the aspects of overheads, data preprocessing scalability and querying efficiency. Section 6 reports three real-world cases to validate the effectiveness of P-Tracer. Section 7 compares our approach with the related work. Section 8 provides some further discussions and concludes this paper.

2 Background

Cloud computing systems generally consist of a lot of software components that can be assembled into multiple types of services, serving tremendous user requests. Operators typically have much difficulty in knowing the system behavior for a given service. Examples include: 1) where do the user 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 components might be the primary causes of the problem? The end-to-end request tracing technology is effective for operators to understand the casual relationships of component invocations. It can record the execution information of individual requests, for example, the entering and exiting time when individual requests go through service components. This information can be utilized to profile services. Hence, our profiling tool relies on this technology to profile system performance.

2.1 End-to-end request tracing technology

Basically, there are two kinds of instrumentation mechanisms, white-box-based mechanism and black-box-based mechanism. A white-box-based mechanism (e.g., [4,8,9]) assumes source codes are available and utilizes explicit global identifiers to correlate runtime events; while a black-box-based mechanism (e.g., [10–14]) assumes no knowledge of the source code and adopts probabilistic correlation methods or statistical regression techniques to infer the casual paths. Since the source codes of services are generally available in typical production cloud systems, in this paper, we utilize an annotation-based tracing mechanism to capture the system activities of a request when it is processed in the system.

Figure 1 shows an instrumentation example of our target system. The interface of ENABLE_TRACE( ) invoked in main is engaged to activate a tracing process, which generates a global identifier for the request. The interface of TRACE_LOG( ) is used to record contextual information into logs when instrumented methods are invoked.

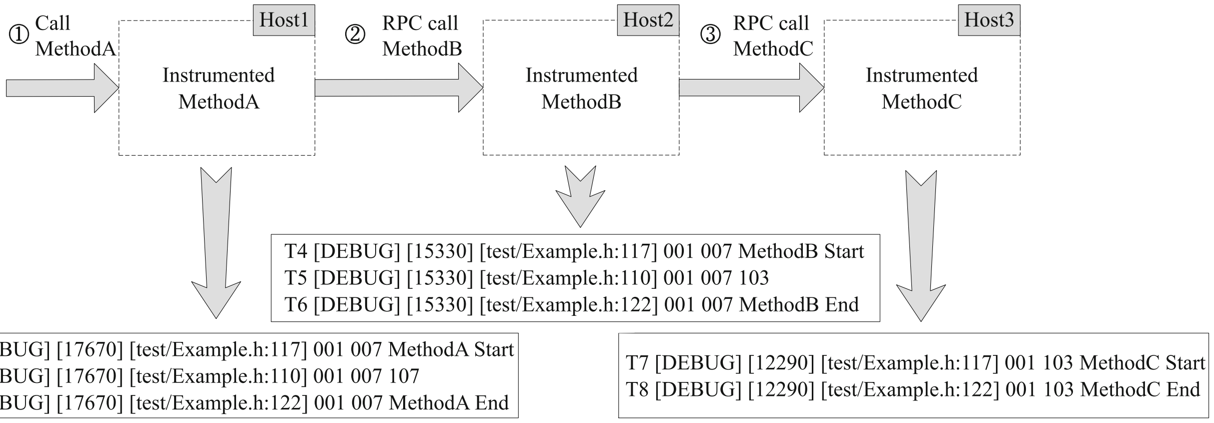
```
main(...){
    ENABLE_TRACE( );
    MethodA( );
}

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

MethodB(){
    TRACE_LOG( );
    MethodC( );
    ...
}
MethodC(){
    TRACE_LOG( );
    ...
}
```

Fig. 1 An example of explicit instrumentation.

2.2 Clock drifts avoided request tracing

In large-scale cloud computing systems, a request may span many hosts. The sequence of events (i.e., method invocations) along one execution is recorded into logs that are distributed across the system. The distributed tracing logs could be utilized to profile system performance.

The upper half of Fig. 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 through which it passes will record the contextual information into local log files, as shown in the lower half of Fig. 2. When the invocation of the 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 for the request, GID for the invoked method, name of the invoked method, and flag signifying the



```
T4 [DEBUG] [15330] [test/Example.h:117] 001 007 MethodB Start
T5 [DEBUG] [15330] [test/Example.h:110] 001 007 103
T6 [DEBUG] [15330] [test/Example.h:122] 001 007 MethodB End

T1 [DEBUG] [17670] [test/Example.h:117] 001 007 MethodA Start
T2 [DEBUG] [17670] [test/Example.h:110] 001 007 107
T3 [DEBUG] [17670] [test/Example.h:122] 001 007 MethodA End

T7 [DEBUG] [12290] [test/Example.h:117] 001 103 MethodC Start
T8 [DEBUG] [12290] [test/Example.h:122] 001 103 MethodC End
```

Fig. 2 A request passes through three instrumented methods which 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, GID for the invoked method, name of the invoked method, and a flag signifying the start or end of the invocation. When the request spans between hosts, P-Tracer assigns new GIDs for methods in remote hosts. The GID of the first invoked method is initially set identical to the GID of the request (i.e., 001).

start of the invocation, as shown in the first line of three local log files in Fig. 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 the three local log files in Fig. 2.

Former research investigations (e.g., Stardust [4]) always utilize one global identifier to mark a request and construct the call relationships of instrumented methods depending on the sorted time-stamps in tracing logs. However, this approach does not work in cloud computing systems because of the clock drifts among the hosts. Although the clocks of the hosts have been synchronized by Network Time Protocol [15], there are still millisecond-level deviations between the clocks. Figure 3 shows an example of the clock drifts in a 100-host cluster. From the figure, we select one host randomly and consider the local clock of this host as the standard time. The biggest difference is larger than 20 ms. Because requests may span many hosts and the time-stamps in logs are recorded according to the local clocks, it may lead to the 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.

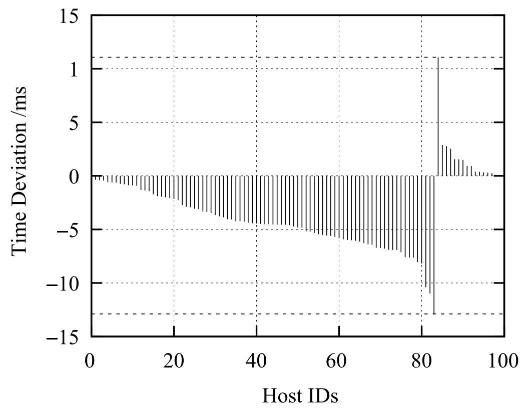


Fig. 3 The clock drifts of a cluster with 100 hosts

In order to precisely trace requests without being affected by the clock drifts, for a sampled request, besides the GID of the request, P-Tracer also assigns GIDs to its invoked methods. When a method in host A calls another method in host B, besides passing the request GID to the callee, the caller also generates and sends a new GID for the callee. Then, P-Tracer records the GID relationship between the caller and callee into the tracing log to explicitly mark the original and new value of the method GIDs. To reduce the overheads of generating GID, invoked methods within the same host share the same identifiers.

For example, when the instrumented MethodA makes an RPC call for the instrumented MethodB, it generates a new GID (i.e., 007) and stores the change process into the tracing log, as shown in the second line of the lower left part in Fig. 2. Relying on the parent-child relationship of the method GIDs within one request, P-Tracer can precisely retrieve its call tree.

3 Constructing the search engine

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

In this section, we discuss the design details of the search engine. As shown in Fig. 4, our approach contains four parts: 1) extract the key parameters from the raw tracing logs and collect the refined logs from production and testing clusters to an analysis cluster; 2) retrieve call trees of requests from the refined tracing logs, and classify structure-identical call trees together; 3) generate the index for them; and 4) provide operators with web-based interfaces to perform online query.

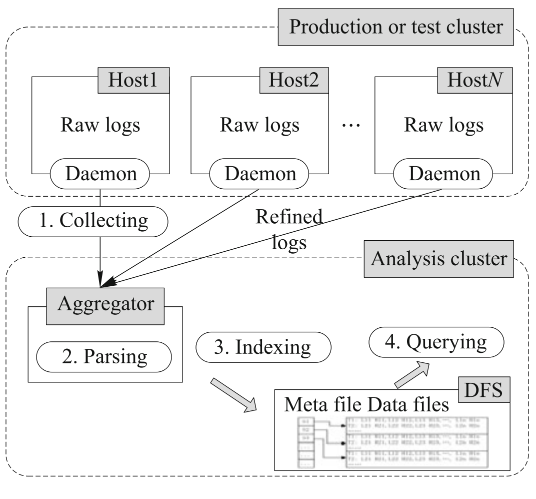
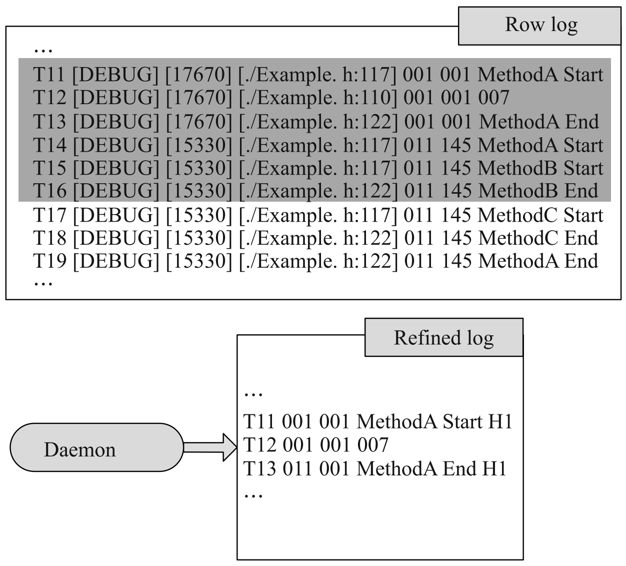


Fig. 4 The structure of the search engine

3.1 Collecting

A log collecting daemon is deployed for each host in production and testing clusters. It collects raw tracing logs at a controlled time window. In order to reduce the overheads of the network traffic, the daemon does not send the raw logs to the analysis cluster directly, but first extracts the primary parameters from them.

Figure 5 shows an example of the extracting process in one host. First, the selected lines (i.e., the grey parts) in the desired time window are extracted by the daemon from the raw log; then the daemon filters out all redundancy information (e.g., process number and log level) and puts the refined lines into one temporary file (called refined file). Note that the daemon appends the host address to the refined lines in order to keep the 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 90%.



```
…
T11 [DEBUG] [17670] [./Example. h:117] 001 001 MethodA Start
T12 [DEBUG] [17670] [./Example. h:110] 001 001 007
T13 [DEBUG] [17670] [./Example. h:122] 001 001 MethodA End
T14 [DEBUG] [15330] [./Example. h:117] 011 145 MethodA Start
T15 [DEBUG] [15330] [./Example. h:117] 011 145 MethodB Start
T16 [DEBUG] [15330] [./Example. h:122] 011 145 MethodB End
T17 [DEBUG] [15330] [./Example. h:117] 011 145 MethodC Start
T18 [DEBUG] [15330] [./Example. h:122] 011 145 MethodC End
T19 [DEBUG] [15330] [./Example. h:122] 011 145 MethodA End
…
```

```
…
T11 001 001 MethodA Start H1
T12 001 001 007
T13 011 001 MethodA End H1
…
```

Fig. 5 An example of refining a raw log in one host. All related lines under the desired time window (i.e., the grey parts) are merged into a refined temporary file in ascending order of time stamps.

3.2 Parsing

After collecting the refined logs together, the aggregator daemon deployed on the analysis cluster dump them into the distributed file system. Then, the aggregator daemon begins to correlate tracing logs to call trees of requests and classify them. In order to make the preprocessing scalable to the massive volumes of tracing logs, a customized map-reduce process is [16] designed. First, map tasks assign correlated tracing logs that belong to the same requests to corresponding reduce tasks. Second, reduce tasks generate and classify requests.

The map task sends the tracing logs to the reducers, with a guarantee that the logs belonging to the same request are sent to the same reducer. This ensures that the entire call tree of each request can be worked out by only one reducer. First, each map task is assigned a portion of tracing logs (called a split) from the distributed file system. Then, it parses the split into key/value pairs (called records). The key of a record is the GID of the request. The value contains two kinds of formats according to the log formats. The first one is a five-tuple. Its items are sequentially the time stamp, method GID, method name, start/end flag and host address. The second one is a three-tuple that is composed of the time stamp, GID of caller and GID of callee. Finally, the map task uses a modular arithmetic to decide which reduce task the record should be assigned to. Specifically, suppose the number of the reduce tasks is m. The map task assigns the log entry to the ith reducer, where $i = (k \; mod \; m) + 1$ and *mod* denotes the modular arithmetic.

The reduce task first retrieves call trees of individual requests from those records and generates the corresponding call sequences. It then classifies the requests into different categories according to their call sequences such that the requests within one category bear the same call sequence. The call sequences are constructed by adopting a depth-first traversal of call trees. Through concatenating the method name and depth of each node of a call tree, we can get a string representation. The string representation is the signature of the call tree and its corresponding call sequences. Using the logs in Fig.2, Fig. 6(a) plots a call tree and its corresponding signature where X is an abbreviation for Method$X$. Without adding the depths of the nodes, different types of the call trees may generate the same signature. For instance, Fig. 6(b) plots another type of the call tree and its corresponding signature. If removing the depths of the nodes, the two types of the call trees (shown in Fig. 6) will generate the same signature ⟨ABC⟩.

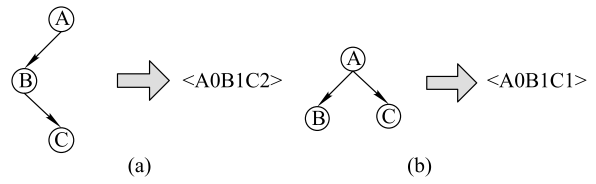


Fig. 6 Examples of generating signatures from call trees. Note that without the depth information, the same signature will be constructed for two types of call trees. (a) Example 1; (b) Example 2

3.3 Indexing

In order to support operators to timely understand the behavior of the system which serves large amounts of user requests, P-Tracer needs to provide the functionality of real-time statistical analysis. Since it works on the massive tracing data, for efficiency considerations, P-Tracer constructs a specific index of categories for fast access. The categories are the output of the map-reduce procedure. Requests within the one category belong to the same service and share the same call sequence. With this index, P-Tracer can help operators conduct statistical analysis more efficiently.

Recall that each type of call trees has a unique signature. For each time window of log collection, all kinds of signatures are merged together and stored into a meta file. The call sequences with the same signature (i.e., within the same category) are kept into one data file. Figure 7 shows an example of the structures of a meta file and three data files. SX in the meta file is an abbreviation 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 as 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 in the distributed file system of the analysis cluster.

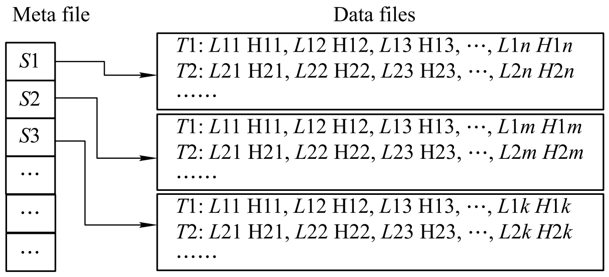

Meta file
Data files
S1
S2
S3
…
…
…
T1: L11 H11, L12 H12, L13 H13, ···, L1n H1n
T2: L21 H21, L22 H22, L23 H23, ···, L2n H2n
……
T1: L11 H11, L12 H12, L13 H13, ···, L1m H1m
T2: L21 H21, L22 H22, L23 H23, ···, L2m H2m
……
T1: L11 H11, L12 H12, L13 H13, ···, L1k H1k
T2: L21 H21, L22 H22, L23 H23, ···, L2k H2k
……

Fig. 7 The structure of an index

3.4 Querying

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

4 Visualization for performance profiling

P-Tracer offers multi-dimensional statistical information to help operators deeply understand the system performance. For space consideration, we introduce some typical metrics which are summarized in Table 1. Note that all figures are drawn according to the ListMail service of the Aliyun mail application. Due to the confidentiality agreement with Alibaba Cloud Computing Company, call trees in the figures have been simplified and all real method names are also replaced.

P-Tracer provides operators with web-based interfaces to construct ad hoc queries, which makes it easy to access and retrieve information from the search engine. Figure 8 displays the primary interfaces and parts of results that match the query parameters. Additionally, the operators can modify any of the parameters (e.g., filter out requests with latencies smaller than 100 milliseconds) of the current query to create a detailed profile view.

4.1 Call tree characteristics

4.1.1 Call tree overviews

In cloud computing systems, for a kind of service, it will involve thousands of requests per second that may take different types of call trees. Different call trees imply different semantics. For example, two types of call trees can be constructed depending on whether the required file is in the cache or

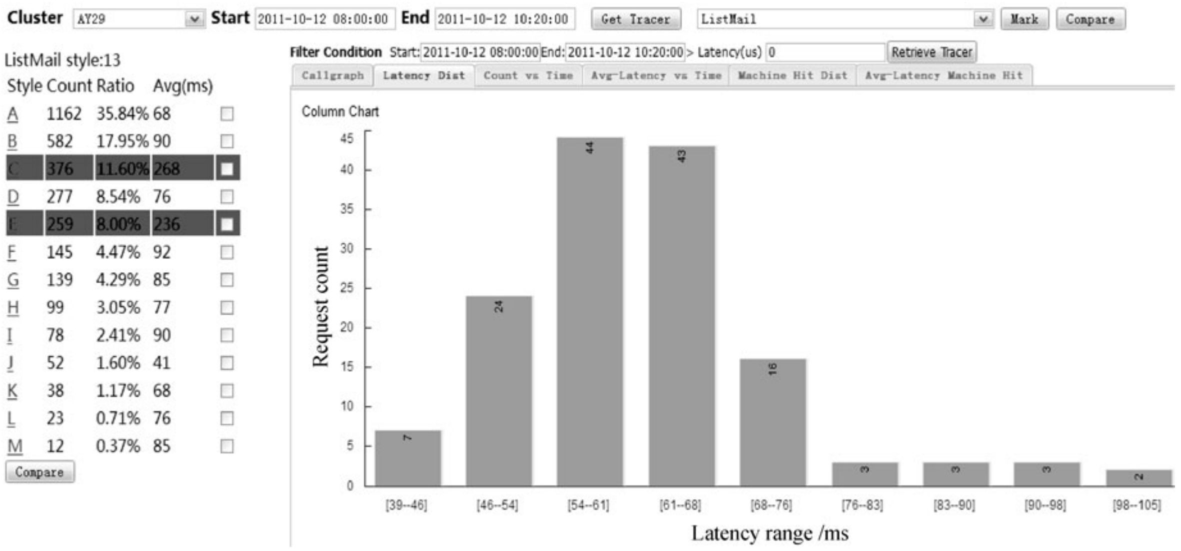
Table 1 Summary of the dimensions of statistical analysis

Category	Metric	Description	Example
Call tree characteristics	Overviews of call trees	Type list of call trees for one kind of service in a desired time window, including the request count in each type, frequency and average latency for each type	Figure 8
	Statistics of a particular type of call tree	Structure of a call tree as well as statistics of each node in the call tree	Figure 9
	Structure comparison between call trees	Highlighting the differences with two call trees	Figure 10
Temporal and spatial characteristics	Histogram of request latencies	Latencies distribution of requests with the same type of call tree	Figure 8
	Variation tendency of request latencies	Trend of latency change for a particular type of call trees	Figure 11
Comparison characteristic	Overall comparison	Overall change for a kind of service between two data sets, such as the types of call trees, latency deviation	Figure 12
	Detailed comparison	Latency deviation of a particular type of call tree between two data sets	Figure 13

not. P-Tracer allows the operators to query the statistical information of the overall call trees for one kind of service in a desired time window. The left part of Fig. 8 shows a query result for the ListMail service, which includes the statistical information about each type of the 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 types of call trees are the critical paths. Ideally, requests within the same type of call tree should have the approximate latencies; however, many factors may cause latency fluctuation. The types in which latencies of requests are overly deviated are defined to be anomalous. We utilize the measure of coefficient of variation (by default, the threshold is set to be 1) [17] to pick anomalous types and highlight them. As shown in the left part of Fig. 8, there are two anomalous types (i.e., type *C* and type *E*) that are labeled in dark background.

4.1.2 Statistics of a particular type of call tree

When operators click on a particular type of call tree, P-Tracer not only visualizes its structure, but also shows statistical information of each node in the call tree, including maximum, minimum, average latencies and the ratio of the average latency of the node to that of the call tree. The latency of the root node is defined as the latency of the entire call tree. The latency ratios of nodes in one call tree are plotted as swimlanes, which helps operators 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 Fig. 9.


Cluster AY29 Start 2011-10-12 08:00:00 End 2011-10-12 10:20:00 Get Tracer ListMail Mark Compare

Filter Condition Start: 2011-10-12 08:00:00 End: 2011-10-12 10:20:00 > Latency(us) 0 Retrieve Tracer

ListMail style:13

Style	Count	Ratio	Avg(ms)
A	1162	35.84%	68
B	582	17.95%	90
C	376	11.60%	268
D	277	8.54%	76
E	259	8.00%	236
F	145	4.47%	92
G	139	4.29%	85
H	99	3.05%	77
I	78	2.41%	90
J	52	1.60%	41
K	38	1.17%	68
L	23	0.71%	76
M	12	0.37%	85

Fig. 8 The main page of P-Tracer contains the primary interfaces to let operators query what they are interested in. Operators can query a specific metric within a desired time window. It also supplies links for operators to modify parameters of the current query, such as filtering out requests with latencies smaller than 100 ms

Function name	Max/ms	Min/ms	Avg/ms	Ratio/%
List Mail	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

Fig. 9 Structure and statistical information for a particular type of call tree

4.1.3 Structure comparison between call trees

P-Tracer provides operators with an interface to distinguish structure differences between call trees. For example, Fig. 10 shows a comparison of two call trees for the ListMail service. We can see the first call tree has two more RPC call

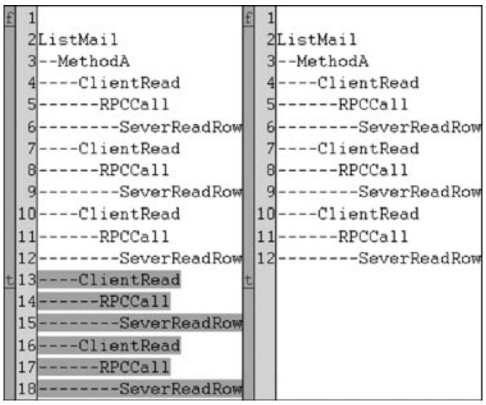

```
 1
 2ListMail
 3--MethodA
 4----ClientRead
 5------RPCCall
 6--------SeverReadRow
 7----ClientRead
 8------RPCCall
 9--------SeverReadRow
10----ClientRead
11------RPCCall
12--------SeverReadRow
13----ClientRead
14------RPCCall
15--------SeverReadRow
16----ClientRead
17------RPCCall
18--------SeverReadRow
```
```
 1
 2ListMail
 3--MethodA
 4----ClientRead
 5------RPCCall
 6--------SeverReadRow
 7----ClientRead
 8------RPCCall
 9--------SeverReadRow
10----ClientRead
11------RPCCall
12--------SeverReadRow
```

Fig. 10 Comparison of two types of call trees

processes than the second one. Through comparing the call trees, operators can learn the differences between execution paths with finer granularities.

4.2 Temporal and spatial characteristics

Apart from call tree characteristics, P-Tracer provides operators to analyze temporal and spatial characteristics of requests to help them better understand the system performance.

4.2.1 Latency distribution

P-Tracer plots a latency histogram of requests that share the same type of call tree in a desired time window, which allows operators to quantify the latency distribution in a finer granularity. The horizontal axes are the latency bins and the vertical axis represents the request count in each bin. The right part in Fig. 8 shows an example of the latency histogram of type F. We can see that most latencies aggregate into the former 5 bins.

4.2.2 Latency variation tendency

P-Tracer provides the operators to observe the latency variation tendency of requests (i.e., the root nodes of the call trees) as well as invoked methods (i.e., the son nodes of call trees) in a desired time window. It is different to performance counters [18] that show the changes of a given metric as a function of time; however, P-Tracer could differentiate the requests to the same service base on the types of call trees. Figure 11 plots the latency variation tendency of requests in the abnormal type E. The horizontal axis is the time tick and the vertical axis is the average latency of requests within the same time tick. It shows the latency of first time tick increases by nearly 30% comparing with those of other ticks, which may push operators to analyze why this happens.

4.3 Performance comparison

Next, P-Tracer allows the operators to compare two groups of tracing data and analyze the root cause of performance changes. It is quite helpful when they need to detect why the performance of the systems with two versions is different under the same load.

4.3.1 Overall comparison

Given a kind of service, P-Tracer visualizes the total average latencies for two datasets as well as the average latencies for particular types of call trees. Figure 12 shows the overall comparison of two datasets for the ListMail service. Both datasets are composed of the tracing logs for two days. On the horizontal axis, each tick represents the number of requests with the same type of call tree. The vertical axis represents the latency. From the figure, we can see that the average latencies of the 5th type in both data sets are larger than those corresponding total average latencies by nearly 30%, which means the fifth type of call trees is perhaps a bottleneck for the ListMail service. Furthermore, the average latency of the 5th type in the second dataset is larger than that in the first dataset by 25%. This detailed information could help operators localize the abnormal types which cause the performance changes.

4.3.2 Detailed comparison

For a suspicious type of call tree, P-Tracer further calculates the statistical information in two datasets respectively and

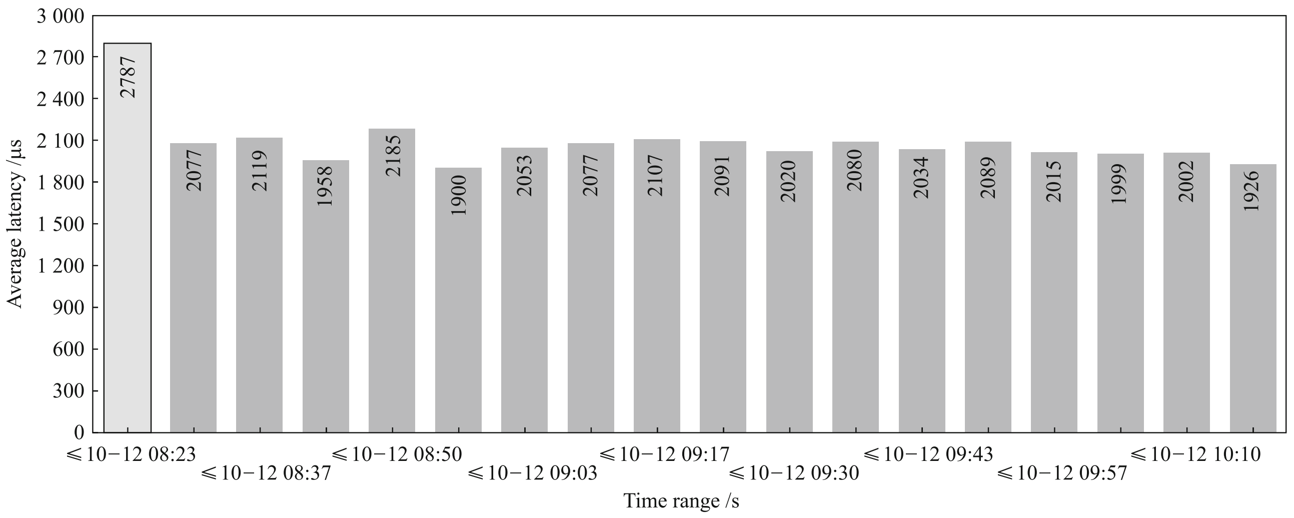


Fig. 11 Evolution of the average latency of requests in the type E along the time proceeding

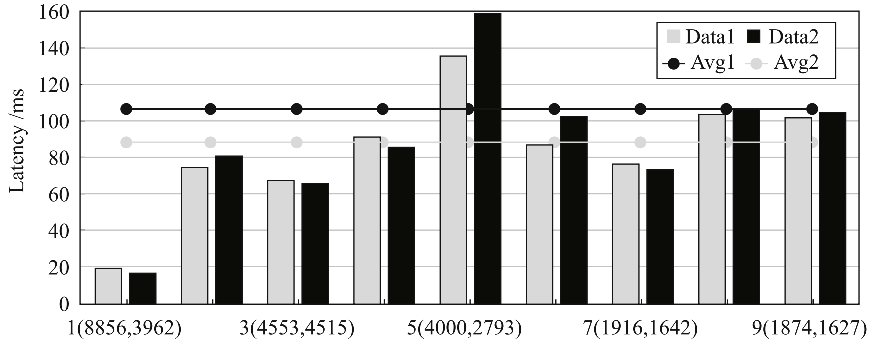
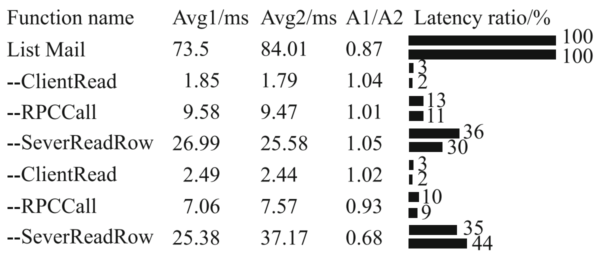


Fig. 12 Overall comparison of two data sets for the ListMail service, which visualizes total average latencies for two datasets as well as average latencies for particular types of call trees.

then compares the latency deviations between the same nodes. In order to identify which methods are the causes of performance changes, for each method, we apply the standard Mann-Whitney U test [19] to quantify the difference between the two datasets. A method is defined to be anomalous if the difference is calculated to be statistically significant. Figure 13 shows the detailed comparison of the fifth type in Fig. 12. The left is the structure of this type of call tree and the right shows the changes of the average latency for each node. The latency ratios in two datasets are shown in the form of swimlanes. From the figure, we can see that the last method highlighted in the last line is the anomalous one, which is the primary cause of the performance change.



Function name	Avg1/ms	Avg2/ms	A1/A2	Latency ratio/%
List Mail	73.5	84.01	0.87	100 / 100
--ClientRead	1.85	1.79	1.04	3 / 2
--RPCCall	9.58	9.47	1.01	13 / 11
--SeverReadRow	26.99	25.58	1.05	36 / 30
--ClientRead	2.49	2.44	1.02	3 / 2
--RPCCall	7.06	7.57	0.93	10 / 9
--SeverReadRow	25.38	37.17	0.68	35 / 44

Fig. 13 Detailed comparison of the same type of call tree in two datasets

5 Evaluation

P-Tracer has been successfully applied in Alibaba Cloud Computing Company to conduct online performance profiling in both production clusters and testing clusters. The scales of clusters range from 40 hosts to 350 hosts. These clusters are under different 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. All the tracing logs from those clusters are dumped into a 10-host analysis cluster at every controlled time window (1 h by default). For each cluster, P-Tracer creates an index and adopts an independent map-reduce procedure to preprocess its tracing logs. All indices together with the retrieved call sequences are imported into the distributed file system of the analysis cluster. In this section, we evaluate P-Tracer in the aspects of tracing overheads, data preprocessing scalability and query efficiency.

5.1 Evaluation of tracing overheads

As the request tracing process will inevitably cause disturbance to the target systems; in this section, we first evaluate the extra overheads that P-Tracer brings to systems. Figure 14 shows the extra overheads to the CPU usage that P-Tracer causes under the conditions of different sample rates when tracing requests. The horizontal axis represents the sample rate and the vertical axis denotes the CPU overhead.

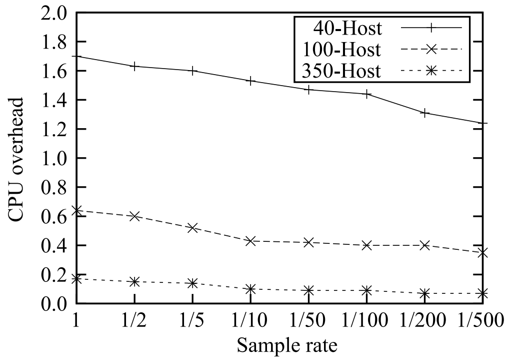


Fig. 14 Extra overheads brought by the request tracing process

In total, we adopt eight kinds of sample rates (from tracing per request to tracing one request on every 500 requests). As shown in the figure, the overheads decrease as the sample rate decreases in the three clusters with different scales. Meanwhile, we can see that the larger the system scale increases, the smaller the overheads are caused. P-Tracer generates on average less than 1.7% CPU overheads to the 40-host cluster. When the scale of the cluster increases to 100 hosts, the overheads will decrease below 0.7%, and P-Tracer brings negligible overheads to the 350-host cluster. It demonstrates the effectiveness of such a request tracing mechanism. In the production environment, the default sampling policy for tracing in the clusters is at 1 out of 200 requests.

5.2 Evaluation of data preprocessing scalability

The customized map-reduce procedure is designed to guarantee the scalability of the tracing data preprocessing step. Figure 15 plots the computation time of the map-reduce procedure when processing different volumes of tracing logs. We can see that the computational time is linearly-related to the data volume. This is further confirmed by a regression analysis on the data, which finds that the linear dependency between the computational time and the data volume is around 0.93. This shows the scalability of the customized map-reduce procedure for the massive tracing data preprocessing. Actually, it takes less than 340 seconds to process the 240 million lines (about 60GB) of tracing logs in our experiment.

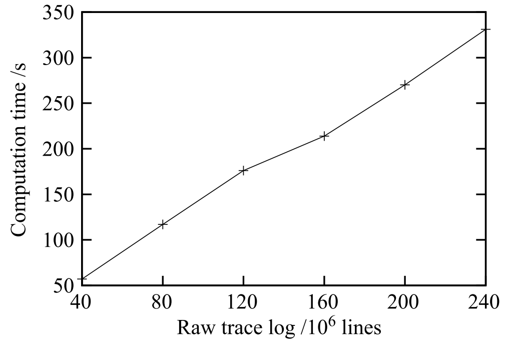


Fig. 15 Scalability of the map-reduce based tracing logs preprocess

5.3 Evaluation of querying efficiency

Tremendous user requests are simultaneously served per second in the cloud computing environment; hence, P-Tracer needs to provide the operators with the functionality of statistical analysis to understand system behaviors. Query efficiency is one of key factors for P-Tracer to be utilized in such production system and the query index is one of the core mechanisms to support the real-time profiling in the environment of massive requests. In order to highlight the engineering merit of the index, we evaluate the query efficiency of P-Tracer with and without the index. The experiment is to calculate the average latencies of requests that share the same call sequences. From Fig. 16, we can see that the computation time with the index is on average one order of magnitude smaller than that without the index. Without the index, the requests have to be reclassified to get the query results, which wastes extra time. On the contrary, with the index, since all requests are proactively organized based on their call sequences, it is more efficient for the operators to conduct statistical analysis.

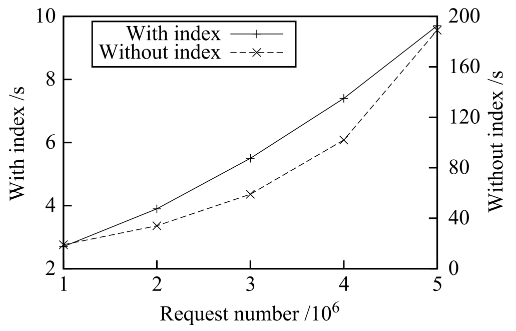


Fig. 16 Querying time of P-Tracer with and without the index under different request numbers

6 Validation

In this section, we report three real-world cases of applying P-Tracer to analyze performance for the mail service in the production and testing clusters.

6.1 Detecting design defects

After a system upgrade, we utilized P-Tracer to conduct performance profiling and found a design defect. While checking the overall call trees of the SendMail service (used to send mails), we 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 three times larger than those of other call trees. After trying different scales of time windows (from one hour to two weeks), we found the ratio of this call tree was steady.

We presented this result to the relevant developers. They opened a bug report to investigate the case. Afterwards, they found that the root cause was a design defect of sending group mails. When sending mails to a group, the redundant 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 mail would be aborted. After the developers added the logic of wiping off redundant mails in the application level, the performance of the SaveMail 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 will experience difficulty in understanding the behavior of requests in such a fine granularity to discover performance bottlenecks. Furthermore, P-Tracer supports operators to conduct online profiling of system performance in a large time scale, which can statistically reflect the system behavior more precisely.

6.2 Detecting code bugs

This is another problem detected by P-Tracer. During a performance test that last two days, a developer used P-Tracer to check the top ten types of call trees of the ReadMail service, and he found that the average latency taken in the client was about two times larger than that taken in the server, which was quite an unexpected result.

After analyzing the source code, he localized the main cause of the latency disproportion. A developer mistakenly accessed a deprecated method in the new version of the system. The method tried to access a security authentication file which had been renamed. Moreover, this performance test was done in a test environment, but the security class was mistakenly set high in the configuration file. Therefore, large mounts of thrown-out exceptions were generated and dumped into the alarm log files, which caused the performance degradation.

P-Tracer can visualize the latency ratios of the invoked methods through which requests passed, facilitating developers to easily check if the behavior of the requests matches their expectation and infer the possible root causes when performance degrades.

6.3 Performance comparison

As shown in Fig. 17, during a performance test under steady load, the performance sharply decreased. To diagnose this degradation, we compared the tracing logs in the non-problem period with that in the problem period and found that the proportion of request count in one type of call trees increased by 25%. Then, we made a detailed comparison about this type. From visualizing the structure of this type of call trees, and we found that it reflected the behavior of requests when some replicated instances of the service component cashed.

In large-scale cloud computing systems, one service component runs many replicated instances deployed on different hosts. When an instance of one service component crashes, the platform provides a failover mechanism to restart it. The

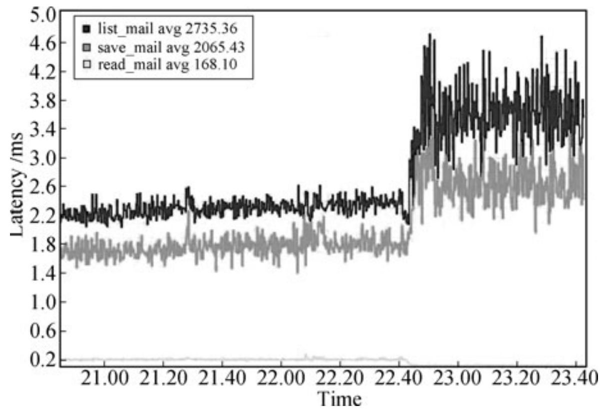


Fig. 17 Performance degradation during a load test

crashed instance cannot be accessed during the recovery, which causes the failure of parts of requests. These failed requests generate a type of call trees the structure of which is to retry the server with RPC call for three times and then abort. Most of time the failover mechanism guarantees the recovery of crashed instances; however, from the increased ratio of the failure type of call trees, we inferred that the failover mechanism did not work at this time. We presented the result to the developer, who confirmed our inference after checking the event logs. The load of the host with the crashed instance was so high that there were not enough resources left to recovery the instance. This case again demonstrates that P-Tracer can help developers identify the possible primary cause of the performance changes.

7 Related work

7.1 Performance counter vs end-to-end request tracing

Extensive research efforts try to utilize the explicit annotation-based instrumentation to conduct performance monitoring, tuning and debugging for large-scale distributed systems. There are mainly two types of instrumentation based approaches: performance counter based and end-to-end tracing based schemes. Performance counter based approaches [18,20,21] aggregate the utilization of system resources (e.g., CPU utilization, disk I/O, and network traffic) or user-defined operations (e.g., the number of requests in queue and hit ratio of cache). However, developers tend to add so large numbers of counters (in Alibaba Cloud Computing Company, more than three hundred of counters are added to the system) that it is quite difficult to infer which ones are the most relevant with the performance changes. Furthermore, performance counters just expose an aggregated view of different workloads, which provides little assistance for operators to understand the resource demands of individual workloads. Compared to performance counters, end-to-end tracing approaches record the execution information of requests in the system, e.g., the entering and exiting time stamps when the requests go through service components, which directly reflects the causal dependency among component interactions for individual requests and is informative for developers to understand system performance.

7.2 Performance profiling in a single process

Diverse single process-oriented performance profiling tools are proposed to help developers identify time-consuming parts of the computation in a single node. DTrace [5], gprof [2], and DARC [6] visualize the execution of systems as call graphs to signify where requests spend time in a single node. Misailovic et al. [22] propose a loop perforation approach to optimize the tradeoff between execution time and quality of service. Our work builds on some ideas from these approaches; however, more factors have been considered in large-scale cloud computing systems, such as large volumes of tracing logs, clock drifts and complex execution paths.

7.3 Performance analysis in distributed systems

Extensive work has employed the explicit annotation-based instrumentation mechanism to conduct performance profiling in distributed system. Magpie [23] applies application-specific event schemas to correlate events and captures the resource consumption of individual requests with the goal of understanding system performance. Stardust [4] captures resource demands for per workload and stores them into a relational database. Pinpoint [24] traces causal relationships of requests in multi-layers of web service components and adopts a statistical analysis process to identify abnormal requests. Chen et al. [25] engage runtime request paths to manage and assess performance issues in the evolution of systems. These techniques generally assume that the clock drifts are negligible, and then utilize a global identifier to trace requests that spans many hosts. Since the clock drifts are millisecond-level in large-scale distributed systems, they cannot retrieve the call tree of requests precisely. Compared to these techniques, besides the global identifiers of the sampled requests, P-Tracer also assigns global identifiers to the invoked methods. In order to reduce the overheads of generating identifiers, for each sampled request, invoked methods within the same host share the same identifiers. Therefore, P-Tracer can precisely retrieve the call trees of requests based on the parent-child relationships of identifiers.

Dapper [8] is a distributed performance analysis tool in Google, which is the most similar to P-Tracer. However, there is a major difference between Dapper and P-Tracer. Dapper keeps one request into Bigtable [26] as a row, whereas P-Tracer proactively groups user requests into different categories according to their call trees and constructs a specific index for these categories. With this index, P-Tracer can help operators conduct statistical analysis more efficiently.

A number of techniques have been proposed for analyzing the system performance based on the request tracing logs. Spectroscope [27] aims to isolate the root cause of performance changes through identifying latency anomaly and structure anomaly of requests. Pip [28] and Ironmodel [29] compare users' actual behavior with self-defined expectation to determine whether a request is abnormal or not. Huang et al. [7] rely on a layered queuing network to model the relationship between time-varying workloads and system resources in order to capture system behaviors. Mann et al. [30] utilize the latencies of request traces to construct a directed acyclic graph to infer the parent RPC latencies of parallel services. Relying on this directed acyclic graph model, Ostrowski et al. [31] conduct an automated hierarchical detection to identify the system elements that cause the changes of end-to-end latencies. The above algorithms can be easily plugged into the framework of P-Tracer and the tracing data generated by P-Tracer can be the input of those algorithms.

Compared to our previous work [32], this paper systematically discusses the functionalities of statistical analysis, evaluates P-Tracer in the aspects of tracing overheads and data preprocessing scalability, and extends two real-world cases to demonstrate the effectiveness of P-Tracer.

There are also many black-box-based performance tuning and diagnosing techniques (e.g., [11, 13, 14, 33]). These approaches do not need the domain-specific knowledge and can work when the source codes of applications are unavailable or uninstrumented. However, they have to spend more time on training logs in exchange for sufficient accuracy of inference. Furthermore, there is a tradeoff between tracing granularity and debugging efforts. A black-box mechanism can be deemed as a tracing mechanism with the coarse granularity (e.g., in node level) and will increase more human efforts in analyzing the system behaviors.

8 Discussion and conclusion

8.1 Discussion

Although P-Tracer has provided operators with web-based query interfaces that cover most of their requirements, sometimes specific tailor-made computation may be needed. In order to support self-defined analysis, we have opened trace data store to engineers and supplied APIs for operators to directly access meta files and data files.

Moreover, tracing logs are invaluable to conduct performance analysis and finer granularity information could be mined. For example, relying on the historical tracing data, we could construct an online detection tool that automatically report alarms to operators when performance degrades.

8.2 Conclusion

Currently, distributed systems are continuously growing in scale and complexity of component interactions, which posts great challenges for operators to online capture the characteristic of system performance. This paper presents P-Tracer, a service-oriented profiling tool to support separation-of-concerns performance analysis in real time. P-Tracer is evaluated in the aspects of tracing overheads, data preprocessing scalability and query efficiency. Experiences with three real-world cases demonstrate that P-Tracer can effectively help operators conduct performance profiling and localize the primary causes of performance anomalies.

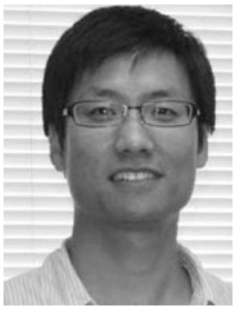
Acknowledgements This research was supported by the National Basic Research Program of China (2011CB302600), the National High Technology Research and Development Program of China (2012AA011201), the National Natural Science Foundation of China (Grant Nos. 61161160565, 90818028, 91118008, 60903043), and an NSFC/RGC Joint Research Scheme sponsored by the Research Grants Council of Hong Kong, China and National Natural Science Foundation of China Project (JC201104220300A).

References

1. Ren G, Tune E, Moseley T, Shi Y, Rus S, Hundt R. Google-wide profiling: a continuous profiling infrastructure for data centers. IEEE Micro Magazine, 2010, 30(4): 65–79
2. Graham S, Kessler P, McKusick M. Gprof: a call graph execution profiler. ACM SIGPLAN Notices, 2004, 39(4): 49–57
3. Mohr B, Wylie B, Wolf F. Performance measurement and analysis tools for extremely scalable systems. Concurrency and Computation: Practice and Experience, 2010, 22(16): 2212–2229
4. Thereska E, Salmon B, Strunk J, Wachs M, Abd-El-Malek M, Lopez J, Ganger G. Stardust: tracking activity in a distributed storage system. ACM SIGMETRICS Performance Evaluation Review, 2006, 34(1): 3–14
5. Cantrill B, Shapiro M, Leventhal A. Dynamic instrumentation of production systems. In: Proceedings of the 2004 USENIX Annual Technical Conference. 2004, 2–15
6. Traeger A, Deras I, Zadok E. DARC: dynamic analysis of root causes of latency distributions. ACM SIGMETRICS Performance Evaluation Review, 2008, 36(1): 277–288
7. Huang X, Wang W, Zhang W, Wei J, Huang T. An adaptive performance modeling approach to performance profiling of multi-service web applications. In: Proceedings of the 35th IEEE Computer Software and Applications Conference. 2011, 4–13
8. Sigelman B, Barroso L, Burrows M, Stephenson P, Plakal M, Beaver D, Jaspan S, Shanbhag C. Dapper, a large-scale distributed systems tracing infrastructure. Technical Report, Google, 2010
9. Park I, Buch R. Event tracing-improve debugging and performance tuning with etw. MSDN Magazine-Louisville. 2007, 81–92
10. Sang B, Zhan J, Lu G, Wang H, Xu D, Wang L, Zhang Z, Jia Z. Precise, scalable, and online request tracing for multitier services of black boxes. IEEE Transactions on Parallel and Distributed Systems, 2012, 23(6): 1159–1167
11. Tak B, Tang C, Zhang C, Govindan S, Urgaonkar B, Chang R. Vpath: precise discovery of request processing paths from black-box observations of thread and network activities. In: Proceedings of the 2009 Conference on USENIX Annual Technical Conference. 2009, 19–32
12. Koskinen E, Jannotti J. Borderpatrol: isolating events for black-box tracing. ACM SIGOPS Operating Systems Review, 2008, 42(4): 191–203
13. Reynolds P, Wiener J, Mogul J, Aguilera M, Vahdat A. WAP5: black-box performance debugging for wide-area systems. In: Proceedings of the 15th International Conference on World Wide Web. 2006, 347–356
14. Aguilera M, Mogul J, Wiener J, Reynolds P, Muthitacharoen A. Performance debugging for distributed systems of black boxes. ACM SIGOPS Operating Systems Review, 2003, 37(5): 74–89
15. Mills D. Network time protocol (Version 3) specification, implementation and analysis. RFC Editor, 1992
16. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. Communications of the ACM, 2008, 51(1): 107–113
17. Abdi H. Coefficient of variation. Sage Publications, 2010
18. Massie M, Chun B, Culler D. The ganglia distributed monitoring system: design, implementation, and experience. Parallel Computing, 2004, 30(7): 817–840
19. Fay M, Proschan M. Wilcoxon-mann-whitney or t-test? on assumptions for hypothesis tests and multiple interpretations of decision rules. Statistics Surveys, 2010
20. Malik H, Adams B, Hassan A. Pinpointing the subsystems responsible for the performance deviations in a load test. In: Proceedings of the 21st International Symposium on Software Reliability Engineering. 2010, 201–210
21. Bodik P, Goldszmidt M, Fox A, Woodard D, Andersen H. Fingerprinting the datacenter: automated classification of performance crises. In: Proceedings of the 5th European Conference on Computer Systems. 2010, 111–124
22. Misailovic S, Sidiroglou S, Hoffmann H, Rinard M. Quality of service profiling. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. 2010, 25–34
23. Barham P, Donnelly A, Isaacs R, Mortier R. Using magpie for request extraction and workload modelling. In: Proceedings of the 6th Symposium on Opearting Systems Design and Implementation (OSDI). 2004, 259–272
24. Chen M, Kiciman E, Fratkin E, Fox A, Brewer E. Pinpoint: Problem determination in large, dynamic internet services. In: Proceedings of the 32nd International Conference on Dependable Systems and Net-

works. 2002, 595–604

25. Chen M, Accardi A, Kiciman E, Lloyd J, Patterson D, Fox A, Brewer E. Path-based faliure and evolution management. In: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation. 2004, 23–36
26. Chang F, Dean J, Ghemawat S, Hsieh W, Wallach D, Burrows M, Chandra T, Fikes A, Gruber R. Bigtable: a distributed storage system for structured data. ACM Transactions on Computer Systems, 2008, 26(2): 1–26
27. Sambasivan R, Zheng A, De Rosa M, Krevat E, Whitman S, Stroucken M, Wang W, Xu L, Ganger G. Diagnosing performance changes by comparing request flows. In: Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation. 2011, 43–56
28. Reynolds P, Killian C, Wiener J, Mogul J, Shah M, Vahdat A. Pip: detecting the unexpected in distributed systems. In: Proceedings of the 3rd Symposium on Networked Systems Design and Implementation. 2006, 115–128
29. Thereska E, Ganger G. Ironmodel: robust performance models in the wild. ACM SIGMETRICS Performance Evaluation Review, 2008, 36(1): 253–264
30. Mann G, Sandler M, Krushevskaja D, Guha S, Even-Dar E. Modeling the parallel execution of black-box services. In: Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing. 2011, 20–24
31. Ostrowski K, Mann G, Sandler M. Diagnosing latency in multi-tier black-box services. In: Proceedings of the 5th Workshop on Large Scale Distributed Systems and Middleware. 2011
32. Mi H, Wang H, Zhou Y, Lyu M R, Cai H. P-tracer: service-oriented performance profiling in cloud computing systems. In: Proceedings of IEEE 36th Annual Computer Software and Applications Conference. 2012
33. Zhang Z, Zhan J, Li Y, Wang L, Meng D, Sang B. Precise request tracing and performance debugging for multi-tier services of black boxes. In: Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems & Networks. 2009, 337–346

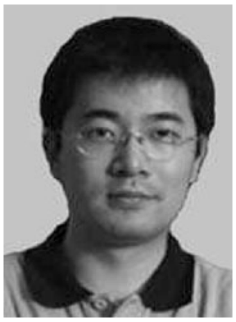


Haibo Mi received the BEng and MEng degrees from Communication Command University of Wu Han, in 2005 and 2008, respectively. He is currently working toward the PhD degree in National Laboratory for Parallel & Distributed Processing, National University of Defense Technology (NUDT), Changsha, China. His thesis focuses on performance maintenance in large-scale distributed systems. He has been worked with the engineers and operators of Alibaba Cloud Computing Company for two years. His research interests include distributed computing, cloud computing, performance monitoring and fault localization.

Huaimin Wang received his PhD in computer science from NUDT in 1992. He is now a professor and chief engineer in department of educational affairs, NUDT. He has been awarded the "Chang Jiang Scholars Program" professor by Ministry of Education of China, and the Distinct Young Scholar by the National Natural Science Foundation of China (NSFC), etc. He has worked as the director of several grand research projects and has published more than 100 research papers in international conferences and journals. His current research interests include middleware, software agent, trustworthy computing.

Yangfan Zhou is currently a research staff member with the Shenzhen Research Institute, The Chinese University of Hong Kong (CUHK) and Department of Computer Science and Engineering, CUHK. He received an MPhil and a PhD from CUHK in 2006 and 2009, respectively, and a BSc from Peking University in 2000. His current research is on software engineering issues (e.g., fault management, fault tolerance, reliability engineering, testing, and debugging) and their applications.

Michael Rung-Tsong Lyu received his PhD degree in computer science from University of California, Los Angeles, in 1988. He is now a professor in the Department of Computer Science & Engineering. He initiated the 1st International Symposium on Software Reliability Engineering (ISSRE) in 1990. He was the program chair for ISSRE 1996, the general chair for ISSRE 2001, the program cochair for PRDC 1999, WWW 2010, SRDS 2005, and ICEBE 2007, the general cochair for PRDC 2005, and a program committee member for many other conferences. Dr. Lyu's research interests include software reliability engineering, distributed systems, fault-tolerant computing, data mining, and machine learning. He has published over 400 refereed journal and conference papers in these areas. Dr. Lyu is an IEEE Fellow, an AAAS Fellow, and received IEEE Reliability Society 2010 Engineer of the Year Award.



Hua Cai received the BS degree from the Shanghai Jiaotong University, Shanghai, China, in 1999, and the PhD degree from the Hong Kong University of Science and Technology (HKUST) in 2003, all in electrical and electronic engineering. He is a member of the IEEE and ACM. He joined Microsoft Research Asia, Beijing, China, in December 2003 and was an associate researcher in the Media Communication Group. He is now a senior expert in Alibaba Cloud Computing Company and leads the teams of cloud monitoring and computing platform. His research interests include distributed system, cloud computing, and mobile media computing.