2014 IEEE 6th International Conference on Cloud Computing Technology and Science

TraceBench: An Open Data Set for Trace-Oriented Monitoring

Jingwen Zhou*†, Zhenbang Chen*†, Ji Wang*†, Zibin Zheng‡§, and Michael R. Lyu†‡§
*Science and Technology on Parallel and Distributed Processing Laboratory
National University of Defense Technology, Changsha, China
†College of Computer, National University of Defense Technology, Changsha, China
‡Shenzhen Research Institute, The Chinese University of Hong Kong, Shenzhen, China
§Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China
Email: {jwzhou, zbchen}@nudt.edu.cn

***Abstract*—User request trace-oriented monitoring is an effective method to improve the reliability of cloud systems. However, there are some difficulties in getting traces in practice, which hinder the development of trace-oriented monitoring research. In this paper, we release a fine-grained user request-centric open trace data set, called TraceBench[1], collected on a real world cloud storage system deployed in a real environment. During collecting, many aspects are considered to simulate different scenarios, including cluster size, request type, workload speed, *etc*. Besides recording the traces when the monitored system is running normally, we also collect the traces under the situation with faults injected. With a mature injection tool, 14 faults are introduced, including function faults and performance faults. The traces in TraceBench are clustered in different files, where each file corresponds to a certain scenario. The whole collection work lasted for more than half a year, resulting in more than 360,000 traces in 361 files. In addition, we also employ several applications based on TraceBench, which validate the helpfulness of TraceBench for the field of trace-oriented monitoring.**

***Keywords*-data set; trace-oriented monitoring; workload generation; fault injection; cloud computing**

I. Introduction

As an effective method to improve the reliability of cloud systems, trace-oriented monitoring systems [1]–[3] track the processes of handling user requests and record the contexts of each step, in the form of user request traces, or simply called traces. The traces naturally reflect the behaviors of the cloud systems under monitoring and can be used for failure detection, fault diagnosis, system recovery, *etc*.

Currently, more and more attentions are paid to the research topic of trace-oriented monitoring. However, it is not easy to get traces from real cloud systems, which provide necessary data for conducting related research. First, collecting traces by hand is a tedious and time-consuming process, involving instrumenting a target system, redeploying the instrumented system, setting up the workloads, injecting faults if needed, *etc.* Moreover, one needs to repeat these steps if the collected traces are insufficient. Second, for safety or other considerations, companies do not want to release the data of traces that record the internal details of their systems. Also, there are few freely available data sets of traces existing in academia and industry. Finally, the manually synthesized traces are weak in authenticity, which influences the usability. Actually, all these difficulties hinder the development of trace-oriented monitoring research, which also motivate this paper.

To address this challenge, we release a fine-grained user request-centric open trace data set, called TraceBench, for supporting trace-oriented monitoring research. TraceBench is collected with a trace-oriented monitoring tool we developed, called MTracer [4], by monitoring the Hadoop Distributed File System (HDFS) [23]. The cluster is deployed on a real environment, containing 50 datanodes, 50 clients and some other nodes. The whole size of TraceBench is about 3.2GB, recording more than 360,000 traces stored in 361 files. The whole collection work lasted for more than half a year. TraceBench consists of three classes: (1) *Normal*: in this class, the traces are collected when the HDFS is running normally in different cluster sizes and responding to various kinds of user requests in multiple speeds; (2) *Abnormal*: we inject permanent faults into the HDFS during the collection, which result in function and performance exceptions, such as data block missing and network slowdown; (3) *Combination*: we randomly inject faults during HDFS running and then perform the recovery, so that the traces record both the normal and abnormal system behaviors.

TraceBench is a well-designed trace data set, considering different user requests, various speeds of workloads, multiple scales of clusters, many types of injected faults, and so on, which is helpful for many trace-based research topics. Take the fault diagnosis as an example, the *Normal* class and *Abnormal* class can be treated as knowledge bases for training the algorithms to learn the features of normal behaviors and abnormal behaviors, respectively, and the *Combination* class can be employed as a test set for validating the effectiveness of the algorithms. Since TraceBench is collected in a real environment, other research topics, such as system understanding, feature extracting and bug finding, would also benefit from this data set.

The rest of this paper is structured as follows. Section II introduces the traces in TraceBench and Section

[1]TraceBench is freely available at: http://mtracer.github.io/TraceBench/

978-1-4799-4093-6/14 $31.00 © 2014 IEEE
DOI 10.1109/CloudCom.2014.79

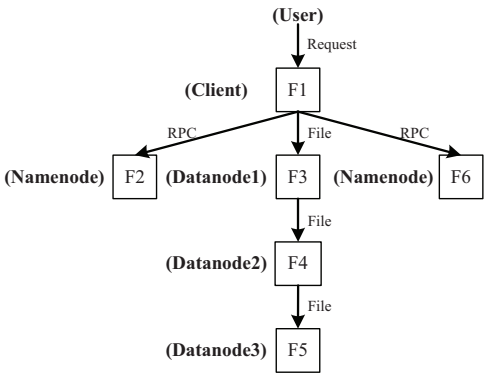

Figure 1. A trace sample

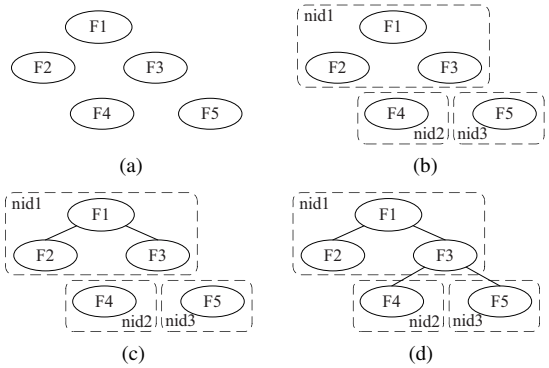

Figure 2. A sample of trace reconstruction in MTracer [4]

III describes the details. In Section IV, we introduce the collection process. Section V gives some applications based on TraceBench. In Section VI, we discuss the invalidities. Section VII reviews the related work and Section VIII concludes the paper.

II. Traces in TraceBench

A trace in the field of this paper records the execution path of a user request. For example, Figure 1 is a trace sample that sketches the process of uploading a file from a client to an HDFS service, involving 5 hosts: one client, one namenode and three datanodes. When receiving a user request of uploading, the client first asks the namenode where to store using the Remote Procedure Calls (RPCs), then uploads the file to datanodes in a pipe-like style, and finally notifies the namenode after completion. The trace, in the form of a tree, clearly illustrates the whole process.

Generally speaking, a trace can be formalized as (E, R) [6], where E and R are the set of events and the set of the relationships between the events, respectively. An event records the context of a request step, where a step stands for the execution process of a function or a routine. An event e is a triple (tid, eid, I). The first element tid is the identity of the trace, which means all the events with the same tid belonging to the same trace. Each event has a unique eid for distinguishing from each other in a trace. I records the detailed information of the step, such as function name and execution latency. On the other hand, a relationship records the causality between two events, and can be expressed in a quadruple $(tid, feid, ceid, T)$, which means the event identified by $feid$ is the father of the event identified by $ceid$, in the trace identified by tid. T indicates the type of the relationship, such as a local or remote function call. Using the events and the relationships, a trace can be reconstructed as a trace tree, where the nodes in the tree correspond to the events, and the edges correspond to the relationships.

Traces in TraceBench are collected by MTracer [4], which is a lightweight efficient trace-oriented monitoring platform for medium-scale distributed systems. In MTracer, the detailed information I of an event is $(name, ip, st, et, d, O)$, representing the event name, IP address, start time stamp, end time stamp, description and other information, respectively. For the efficiency and overhead consideration, MTracer introduces nid rather than eid for an event, inspired by P-Tracer [5]. Each time the local node communicates with a remote node, a new nid is generated in the remote node. So, an nid can be understood as a temporary node ID. Each event can be identified by nid and st, *i.e.*, $eid = (nid, st)$, since the events with the same nid are generated on the same node. A special strategy is employed to record the relationships between events. If a father event and a child event are generated on the same node, the two events have the same nid and their relationship is not recorded explicitly. Actually, the relationship can be calculated using the start and end time stamps. If a father event and a child event belong to different nodes, the relationship is recorded explicitly in the form of an edge, *i.e.*, $(tid, (fnid, fst), cnid)$, where $(fnid, fst)$ identifies the father event and $cnid$ indicates the child events. This strategy saves much of the time for generating random IDs, which is a time-consuming operation [4], thus reducing the overhead on the monitored systems. Additionally, this strategy avoids using the global clock and generates accurate traces.

As Figure 2 illustrates, a trace can be reconstructed to a trace tree in four steps using the events and edges recorded by MTracer: (a) pick out all the events and edges with the same tid identifying the trace; (b) classify the picked events into classes using nid, which means all the events in the same class are generated on the same node; (c) calculate the relationships in each class using the start and end time stamps (*e.g.*, *F1* is the father of *F2*, since *F1* starts earlier and finishes later than *F2* and there is no other ancestors of *F2*); (d) construct the relationships between classes using edges, where the event identified by $(fnid, fst)$ is the father of all the root nodes in the class identified by $cnid$.

III. Introduction of TraceBench

Using MTracer, we collected the traces in TraceBench on an HDFS cluster [23], deployed in a real environment, which consists of virtual machines (VMs) hosted on our private

Table I
OVERVIEW OF TRACEBENCH

Class	*Type*	*Fault*	*Workload*	*Variable*[a]	*#File*	*Collection Time(min)*	*#Trace*	*#Event*	*#Edge*
Normal	Workload	-	r/w/rw/rpc/rwrpc	1,5i Clients	55	2,761	209,326	3,848,191	1,963,129
	Datanode	-	r/w/rw	1,5i DNs	33	1,980	17,440	2,310,699	941,681
Abnormal	Process	killDN	r/w	0,1,2,3,4,5i FDNs	30	600	6,469	682,595	240,668
		suspendDN	r/w	1,2,3,4,5i FDNs	28	515	2,754	317,498	124,630
	Network	disconnectDN	r/w	1,2,3,4,5i FDNs	28	560	5,189	536,810	192,276
		slowHDFS	r/w/rpc	0,10i/2i/100i ms	33	506	41,200	669,692	348,912
		slowDN	r/w	1,2,3,4,5i FDNs	28	560	4,395	575,425	232,270
	Data	corruptBlk	r	0,1,2,3,4,5i FDNs	15	300	5,354	636,023	244,296
		corruptMeta	r	0,1,2,3,4,5i FDNs	15	300	5,285	671,185	214,446
		lossBlk	r	1,2,3,4,5i FDNs	14	280	4,920	573,148	186,554
		lossMeta	r	1,2,3,4,5i FDNs	14	280	4,789	590,902	256,211
		cutBlk	r	1,2,3,4,5i FDNs	14	280	4,982	619,235	194,904
		cutMeta	r	1,2,3,4,5i FDNs	14	280	5,020	595,579	188,863
	System	panicDN	r/w	1,2,3,4,5 FDNs	10	150	1,977	260,660	103,155
		deadDN	r/w	1,2,3,4,5 FDNs	10	150	1,751	228,550	90,693
		readOnlyDN	w	1,2,3,4,5 FDNs	5	75	368	61,944	27,820
Combination	Single	Process	rwrpc	-	3	210	7,037	237,338	111,247
		Network	rwrpc	-	3	210	6,539	219,692	103,192
		Data	rwrpc	-	3	210	7,392	251,902	117,475
		System	rwrpc	-	3	210	7,056	235,379	110,519
	Multiple	All	rwrpc	-	3	480	17,244	602,512	280,556
Total	**-**	**14 faults**	**5 workloads**	**-**	**361**	**10,897**	**366,487**	**14,724,959**	**6,273,497**

a. $i = 1, 2, ..., 10$

IaaS platform [24]. Our environment contains 50 clients for generating user requests, an HDFS cluster with 50 datanodes and one namenode for processing the requests from clients, and other nodes. The whole size of TraceBench is about 3.2GB and the total effective collection time is 10,897 minutes. TraceBench consists of 361 files in which 97 files contain failed user requests, 366,487 traces, 14,724,959 events, and 6,273,497 edges. In TraceBench, the number of contained events of a trace, or say trace length, spreads from 5 to 420, and the number of involved nodes in a trace spans from 2 to 44. Different aspects are considered during collection, like workloads and faults, to simulate various scenarios and to collect different behaviors of HDFS. In this section, we first describe the structure of TraceBench, and then introduce the workloads and faults.

A. Stucture

As shown in Table I, TraceBench includes three classes: *Normal* (NM), *Abnormal* (AN) and *Combination* (COM), and each trace class consists of different types. When collecting a trace type, we introduce different faults, workloads and variables, showing in the column *Fault*, *Workload* and *Variable*, respectively. For example, when collecting traces in *Workload* type of *Normal* class, we introduce 5 workloads, and in each workload we respectively set the client number to be $1, 5, 10, ..., 50$, to simulate various requests at different speeds. According to the fault, workload and variable, a trace type contains many trace files. The traces in a file are stored in the form of events and edges, introduced in the previous section. Note that, the latency of an event can be calculated with *st* and *et*, and the description *d* depicts the result of the execution, including exception information. In this paper, we name a set of traces according to the context in Table I, *e.g.*, the trace set named as *Normal_Workload_r_1Client*, or *NM_WL_r_1C* for short, contains the traces collected under a workload of file read and with one client in the *Workload* type of *Normal* class.

The traces in the *Normal* class record the request paths when the HDFS system is running normally, without injecting any faults. The *Normal* class consists of *Workload* (WL) type and *Datanode* (DN) type. The *Workload* type takes the speed of workloads as the variable, and can be used to study the capacities of the HDFS system in dealing with different workloads in a certain scale (*i.e.*, 50 datanodes). Since each client generates the workload in the same speed, we use the number of valid clients to represent the total workload speed in Table I. "1, 5*i* Clients" means setting the number of valid clients to be 1, 5, 10, ..., 50, respectively. In contrast, the *Datanode* type fixes the speed of workload (*i.e.*, 30 clients) and changes the number of valid datanodes, to study the behaviors with different system scales.

The traces in the *Abnormal* class are collected when a permanent fault is injected into the HDFS system with a fixed scale (*i.e.*, 50 datanodes) and a fixed workload speed (*i.e.*, 30 clients). The trace files contain the information about function and performance faults, and can be used to study the behaviors when the system is running abnormally. According to the types of faults, the *Abnormal* class can be divided into four types, *i.e.*, *Process* (Proc), *Network* (Net), *Data* and *System* (Sys), which will be discussed later. Except the fault *slowHDFS* affects on the whole network, all the faults are injected into datanode(s), *e.g.*, the fault *lossBlk* deletes all the data blocks on some datanodes. So, the FDN in Table I represents the datanodes with a fault injected.

When collecting the traces in *Combination* class, faults are randomly picked and injected into the HDFS system, and then recovered automatically or manually. Thus, the *Combination* class contains the traces collected both when the system is running normally and abnormally. In this class, together with a trace file, a text file is also given, recording the information about injected faults, including the fault name, injection time, recovery time, *etc*. The *Combination* class consists of the *Single* type, in which faults are chosen from only one fault type, and the *Multiple* type, in which faults are selected from all the four fault types.

B. Workloads

There are about 30 different requests in HDFS, such as *rm*, *copyFromLocal* and *mkdir*, which can be basically divided into three kinds: file read, file write and RPC. A read request downloads a file from HDFS to local, like *copyToLocal*, while a write request uploads a file from local to HDFS, like *copyFromLocal*. Both of them involve the communications with the namenode to get the information of data blocks, and the communications with some datanodes to download or upload data blocks. The RPC requests only include RPC operations with the namenode and without data block accessing, such as *rm* and *ls*. The workloads containing only read, write or RPC requests are indicated as *r*, *w* and *rpc*, respectively. We also introduce two other workloads: *rw*, containing both read and write requests, and *rwrpc*, containing all the three. When collecting the traces in each trace type, we introduce different workloads as needed. For example, since the faults in the *Data* type do not influence write requests, the workload *w* is not introduced in the *AN_Data* set.

C. Faults

During trace collection, we introduce 14 faults of 4 types, showing in the row *Abnormal* and the column *Type* and *Fault* in Table I. The faults in the *Process* type affect the HDFS processes on HDFS nodes. The fault *killDN* kills the HDFS processes on some datanodes, while *suspendDN* suspends some processes. *Network* faults bring anarchies to the network in the cluster, such as *slowHDFS* slows the whole network by milliseconds, and *slowDN* decreases the speeds of sending and receiving packets on some datanodes. The faults in *Data* introduce errors in the data blocks or the metadata files on some datanodes. Such as *corruptMeta* changes the values of some bits in the metadata files, and *cutBlk* removes parts of bits in the data blocks. The *System* faults introduce problems to the OSs of the HDFS nodes, such as making OSs to be panic, dead or read-only. Some of these faults bring function exceptions when handling requests, *e.g.*, *killDN* may cause a failure of reading a file, while others may introduce performance exceptions, *e.g.*, *slowHDFS* increases the latency of data exchanging.

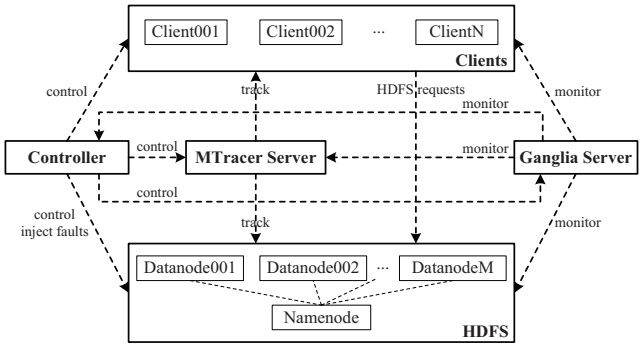

Figure 3. The architecture of trace collection

There are some other aspects need to be pointed out. First, except the fault *slowHDFS* making the namenode slowdown, all the rest faults only affect the datanodes. The reason is that the whole HDFS system would crash in case some problems happen on the namenode, such as the HDFS process is killed and the OS is panic. In this situation, the data collected is meaningless. Second, injecting *Data* faults should be carried out at datanode level rather than the files inside a datanode, because the exceptions caused by these faults have a probability to happen. For example, if the fault *lossBlk* only deletes one data block rather than all data blocks on a datanode and the requests do not involve this block, no exception will occur. Finally, most of the variables in the *Abnormal* class are the number of FDNs. The purpose is to make the trace files more usable for studying the HDFS features with different numbers of abnormal datanodes, *e.g.*, to know how many requests will fail when 20% of HDFS processes are killed on datanodes.

IV. Details of Trace Collection

To collect the traces in HDFS, we instrumented the source code using the interfaces provided by MTracer, and then deployed the modified HDFS and MTracer in a real environment [24] composed of more than 100 VMs. The workloads are generated with *bash* scripts on clients, and the faults are injected using AnarchyApe [25]. This section introduces the architecture and the process of trace collection.

A. Architecture

Figure 3 shows the architecture of trace collection, involving the following components:

- HDFS, the instrumented HDFS system used to process different user requests, containing many datanodes and one namenode.
- Clients, used to generate workloads for simulating the real usages of HDFS, containing many client nodes.
- MTracer Server, the server of MTracer which is in charge of receiving, storing and visualizing traces.
- Controller, the node which controls the whole process of collection and is also in charge of injecting faults.

- Ganglia Server, the server of Ganglia [7], monitoring the whole environment to ensure no unexpected issues happen during collection.

The MTracer Server is deployed on a VM with 4GB memory and $8 \times$ 1GHz CPU, while all the rest nodes are deployed on the VMs with 2GB memory and $4 \times$ 1GHz CPU, and the OS that all the nodes use is CentOS 6.3.

B. Process of Collecting Traces

Bash scripts are employed on each client for generating workloads. Each script takes charge in one kind of workload, *e.g.*, *rpc.sh* produces the *rpc* workload. Using a loop, requests are sent to the HDFS service continuously by a client. All the requests from the clients form the global workload. Intervals are introduced between neighbour requests generated by a script, *i.e.*, after finishing a request, a script waits for a moment and then starts the next one, to control the speeds of requests.

During collecting the traces in a trace file, we first start the MTracer server and the HDFS service with a certain number of datanodes, and then launch the workload on some clients concurrently. When finishing the collection, workload is stopped first. Then, the HDFS service and the MTracer server are shut down after waiting for a few minutes, to guarantee that all requests are finished smoothly and no fragmentary traces are collected. Therefore, in Table I, the values of the column *Collection Time* are composed of the running time of clients and the waiting time. Moreover, when collecting a trace file in *Abnormal* class, a fault is injected before starting the workload, and the recovery will be done after handling all the requests, to ensure all the traces are collected when the system is running abnormally. In contrast, when collecting a trace file in *Combination* class, a randomly chosen fault is injected after starting the workload and the system is later recovered, and the next fault acts in the same way after an interval, to simulate the occasionally occuring faults in the system.

In addition, some parameters should be clarified. First, except the *NM_DN* set, the datanode number is set to 50, corresponding to the maximal capability of the HDFS system in our environment. Except the *NM_WL* set, the client number is fixed to 30, which generates plenty of requests that can be processed in time. Second, the collection time of trace files in *Normal* class is 60 minutes, which is enough for collecting plenty of traces to reflect related statistical features. The collection time of trace files in *Abnormal* class is 20 minutes, since the number of the faulty traces may significantly reduce after 20 minutes due to the mechanisms in HDFS for detecting and avoiding faults, like the heart-beating protocol. Third, most step sizes of variables are set to 5, *i.e.*, changing 10% each time comparing to 50, which balances between reflecting statistical features and controlling the number of trace files. Finally, we choose 8 regular RPC requests for the *rpc* workload, involving file directory creating, modifying, querying and removing, which cover the most frequently used RPC operations in HDFS.

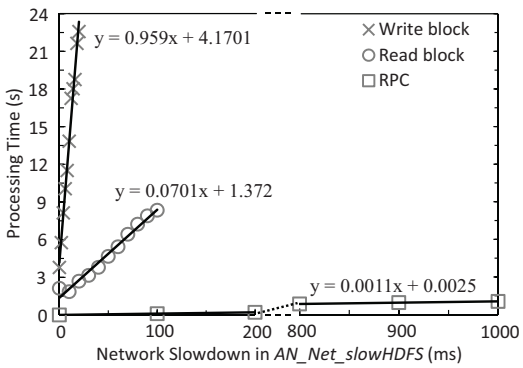

Figure 4. Analysing the fault influence

V. Applications

To validate the usability and authenticity, we carried out extensive data analyses on TraceBench and employed it in several scenarios, including failed requests detection, temporal invariants mining and performance anomalies diagnosis.

A. Data Analyses

Based on TraceBench, we analysed the behaviors of HDFS on the aspects of request handling, workload balancing, fault influence, *etc*. Figure 4 is a sample of analysing the influence of the fault *slowHDFS*. The figure shows the average processing time of reading a block, writing a block and handling an RPC, with different slowdowns introduced into the network of the HDFS cluster. The processing time of each operation increases as the network becomes slower. The slope of each curve reflects the sensitivity of the corresponding operation to this fault. Because requiring plenty of network communications, writing a block is the most sensitive operation, whose slowdown is about $1,000$ times of the network slowdown, *i.e.*, one millisecond slowdown in network results in one second increase in writing a block. Since only a few network interactions are required, RPC is the least sensitive one, where one millisecond slowdown results in one millisecond increase.

B. Detecting failed requests

The same kind of user requests usually result in the traces with similar topologies. The similarities can be extracted as properties, which are useful in failure detection, fault diagnosis and other fields. For example, when reading a file, the client downloads the data blocks of the file from HDFS one by one, where downloading a data block starts with invoking the function "blockSeekTo" (short for *B*), and ends with calling "checksumOk" (*K*) if success. If any data block fails to be downloaded after some retries, the whole request aborts and fails. In other words, in a successfully handled read request, the last data block should be downloaded successfully, which can be expressed by the

Table II
RESULTS OF TEMPORAL INVARIANTS MINING

Log Type	*Trace Set*	$\rightarrow$	$\leftarrow$	$\not\rightarrow$	*Total*
TO	*NM_WL_r_1C*	123	120	42	285
	NM_WL_w_1C	136	120	91	347
	NM_WL_rpc_1C	50	68	697	815
PO	*NM_WL_r_1C*	899	2,103	7,368	10,370
	NM_WL_w_1C	263	1,551	6,002	7,816
	NM_WL_rpc_1C	49	58	688	795

Linear Temporal Logic (LTL) [8] as the following property, where the operators $\Box$, $\Diamond$, $\bigcirc$ represent for *Always*, *Exist* and *Next*, respectively. If a trace of read request violates the following property, we say a failure happens.

$$\Diamond B \wedge (\Box((B \rightarrow \bigcirc(\Box\neg B)) \rightarrow \bigcirc(\Diamond K)))$$

Similarly, we also extracted the properties for write and RPC requests. To validate these properties, we checked the traces in the *AN* set in the form of SQL queries to detect failures. All of the failed requests are picked out correctly. Besides the properties for detecting failures, we have also extracted many properties for diagnosing various faults, such as datanode invalid, data block missing, operation latency anomaly, *etc*. All of these properties can be in turn used to monitor an HDFS system with different methods, *e.g.*, the techniques in runtime verification (RV) [9], which is part of our future work.

C. Mining temporal invariants

As an important respect of system features, temporal invariants record the rules of the orders obeyed by system operations, which can be obtained by mining system logs and can be used to understand systems, detect abnormal behaviors, diagnose deadlocks, infer higher-level properties, *etc*. Ref. [10] defines 5 types of invariants, including $a \rightarrow b$ (Always followed by), $a \leftarrow b$ (Always precedes of), $a \not\rightarrow b$ (Never followed by), $a \parallel b$ (Always concurrent with), and $a \nparallel b$ (Never concurrent with), and gives three algorithms for mining, called transitive closure algorithm, co-occurrence counting v1 algorithm and co-occurrence counting v2 algorithm, respectively. In this subsection, we first mine temporal invariants in TraceBench with the corresponding tool Synoptic [11] (revision id: cc864f389c71), and then compare the mining speeds of these three algorithms.

After formatting the traces both into the totally ordered (TO) format and the partially ordered (PO) format, we mined plenty of invariants in different trace sets with various request kinds in the *NM_WL* set. Table II shows the numbers of mined invariants, where we are interested in $\rightarrow$, $\leftarrow$ and $\not\rightarrow$. Both TO logs and PO logs imply useful invariants, *e.g.*, $INITIAL \rightarrow blockSeekTo$ and $blockSeekTo \rightarrow checksumOk$ mined from the *NM_WL_r_1C* set, respectively mean every read request contains at least one data block reading operation and each successfully handled reading operation invokes "checksumOk", which coincide to the property introduced in previous subsection. Actually, many properties can be inferred from these invariants.

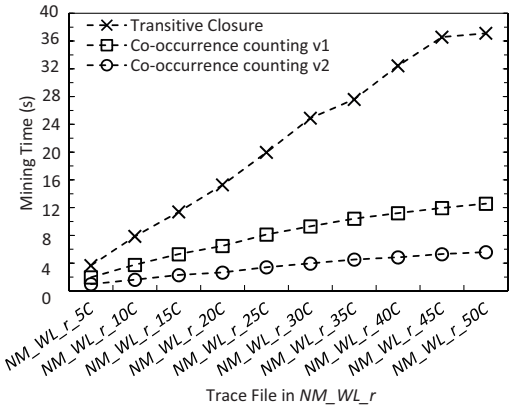

Figure 5. Results of comparing mining algorithms in [10]

When dealing with PO logs, Synoptic treats the same kind of events generated from different nodes as different events, which makes it possible to mine more useful invariants for concurrent systems. However, this also has several limitations. First, too many invariants are generated due to the massive event types, *e.g.*, more than ten thousand in the *NM_WL_r_1C* set, which makes it difficult to search desirable features. Second, since the occasionally happened or occasionally not happened event combinations are treated as inherent system features, some false invariants arise. For example, the invariant $receiveBlock_{datanode001} \not\rightarrow receiveBlock_{datanode006}$ mined from the *NM_WL_w_1C* set, where the function "receiveBlock" running on datanodes receives a copy of data block from the client, represents that there does not exist the situation in the *NM_WL_w_1C* set that HDFS stores a data block on datanode006 after storing on datanode001, which is still possible in other executions. Third, many invariants contain the same information. For example, the invariants $Conn_{datanode001} \rightarrow receiveBlock_{datanode001}$, ..., $Conn_{datanode050} \rightarrow receiveBlock_{datanode050}$ mined from the *NM_WL_w_1C* set, where "Conn" is short for "OP: connect next Datanode", all mean that when receiving a copy of data block, the related datanode first builds connection with the client and then receives the copy. To summarize, when mining temporal invariants in PO logs, Synoptic seems to be more suitable for the systems with few nodes.

In addition, the experiments in [10], which vary a certain dimension and keep others, can be well supported by TraceBench, such as choosing the traces with different datanodes to vary the node numbers, selecting different amounts of traces to change the execution lengths. To synthetically compare the aforementioned three algorithms, we redo the experiments with the traces in the *NM_WL_r* set, which mainly varies the number of traces (or called executions as in [10]) and the number of nodes, on a VM with 8×1 GHz CPU and 4GB memory, by running each algorithm 5 times

Table III
RESULTS OF PERFORMANCE ANOMALIES DIAGNOSING

#Trace	100	200	300	400	500
Time (ms)	31	65	95	130	154
#Anomaly	12/12/0	17/17/0	19/2/0	30/1/0	30/1/0
#Event	33	66	99	134	167
Time (ms)	84	1,413	8,359	35,269	100,023
#Anomaly	0/0/0	1/1/0	1/1/0	1/1/0	0/0/0

on each trace file and reporting the average values. Figure 5 shows the results, which indicates the improvements in the efficiency of the two co-occurrence counting algorithms, and the results are more convincing than the results in [10].

D. Diagnosing performance anomalies

We also implemented a principal component analysis (PCA) based performance anomalies diagnosing algorithm in [12], [13], which finds the traces with abnormal latencies and locates the faulty events as root causes. We evaluated it with different trace sets selected from the *COM_All_rwrpc* set. Since this algorithm can only process the traces with a same topology, we divided the experiment into two parts, *i.e.*, with trace sets of *rmr* requests, which fixes the trace length to 7 and varies the trace amount, and with trace sets of *copyToLocal* requests, which sets the trace amount to 7 and changes the trace length. The two parts of Table III respectively record the results of the two dimensions of the experiment, where *#Trace* and *#Event* respectively represent the trace amount and the trace length, the row *Time* shows the handling time of each trace sets, and the items in row *#Anomaly* are expressed as the form of (total anomalies)/(found anomalies)/(incorrectly found anomalies). In addition, the handling time on a trace set is again the average value of 5 executions, and the machine employed is the same as that in the last subsection.

This algorithm finds all anomalies in some cases, and however sometimes only a small part. Actually, the results depend on the features of data, since only some outliers exceeding the related thresholds are reported as anomalies. So, when some very abnormal traces exist, other less abnormal ones would be ignored. For example, only 1 of 30 anomalies is found when the trace length is 400 in Table III, and after removing this one, another 27 anomalies can be correctly picked out. In addition, this algorithm is pretty accurate with no false alarm in our experiments.

On the other hand, the analysis time increases very fast when increasing the trace length, but slowly when growing the trace amount, which indicates that this algorithm is more sensitive to trace length rather than trace amount. The primary cause is the process of calculating eigenvalues and eigenvectors of a square matrix for getting principal components, where the calculating time is mainly related to the matrix size that exactly equals to the trace length and has no relation with the trace amount. Therefore, this algorithm is maybe more feasible for short traces.

VI. THREATS TO VALIDITY

Some aspects may threaten the validity of TraceBench, such as the monitored system, the cluster size and the injected faults. We discuss these aspects in this section.

We collected our traces only on HDFS. However, the HDFS system is a widely used cloud storage system in academia and industry, and many mechanisms in HDFS, like RPC, are commonly used in other systems. Therefore, TraceBench is representative and we believe TraceBench is helpful for the research of trace-oriented monitoring.

The traces mainly record the communications between nodes, which may make TraceBench unsuitable for the scenarios concerning concurrent operations within one node. However, for a multi-threads program in one node, MTracer can also be used to record the causal relations.

The scale of our HDFS cluster is smaller than real production systems. However, it is enough to exhibit the different features of HDFS for research purpose since a user request in HDFS always involves limited nodes. Moreover, it is really difficult for academic to deploy very large-scale clusters in practice. In addition, the clients can be configured to send requests in a batch mode, which simulates multiple users in one client node.

Regarding fault injection, indeed besides the faults we introduce, many others exist in real systems, like dropping packets in network. Nevertheless, the faults we use are from a mature injection tool of HDFS and we have injected all the types of faults in this tool. Hence, we believe that the faults we inject are representative.

Lastly, there are no explicit tags exist in traces for indicating whether the trace is normal or not. However, we always can exactly estimate a trace from the trace topology, the corresponding text file which describes the injected faults and the logged information in events, such as description field and latency field.

VII. RELATED WORK

TraceBench records the fine-grained information of handling user requests, *i.e.*, the details and casual relations of function calls among multiple nodes, and hence is user request-centric. The traces in TraceBench supply more detailed information than some other approaches, such as job-centric and resource-centric. To the best of our knowledge, TraceBench is the first fine-grained user request-centric open trace data set. Following, we introduce some related public data sets in other categories.

Resouce-centric data sets record the availability of nodes and components or the usages of resouces, focusing on external, rather than internal, information of system programs. For example, the Failure Trace Archive (FTA) [14], collected from parallel and distributed systems, records the times of resource failures; the Computer Failure Data Repository (CFDR) [15] provides the failure data in supercomputers and clusters, *e.g.*, the storage failure data; and the Repository

of Availability Traces [16] contains the events that indicate whether and when a node is available.

Job-centric data sets give a coarse-grained view of system running, such as when a job starts and ends, which nodes involve, without the details about how a job works. For example, the Grid Observatory (GO) [17], collected on a grid infrastructure, includes many data sets and records the information around jobs, *e.g.*, jobs lifecycle; the Parallel Workloads Archive (PWA) [18] provides job-level usage data and is widely used in the research of job scheduling strategies for parallel systems; the Google Cluster Data [19] describes hundreds of thousands of jobs, composed by lots of tasks, together with the task resource usages.

Additionally, TraceBench also exceeds some data sets in certain aspects. For example, the PWA lacks failure information, where the same problem exists in the Grid Workloads Archive (GWA) [20] and the Game Trace Archive (GTA) [21], while the FTA and the CFDR are weak in the aspect of normal data, and the Repository of Availability Traces has limited information besides availability.

VIII. Conclusion

In this paper, we provide a fine-grained user request-centric open trace data set, called TraceBench, collected with a trace-oriented monitoring platform we developed. Our traces are collected in a real environment considering different scales, including the size of the monitored system, workload type, injected fault, and so on. Since TraceBench is collected in a real environment, where the usability and authenticity of this data set are validated by several applications, it would be helpful to user request tracing-oriented monitoring research, such as online fault detection and localization. A lot of other related research, such as system understanding, feature extraction and bug finding, can also employ our data set.

In the future, we plan to study several trace-based technologies based on TraceBench, to improve the reliability of cloud systems, like fault tolerant [22].

Acknowledgment

We thank the anonymous reviewers for helpful comments and feedback. We are indebted to Haibo Mi for suggestions in designing MTracer, to Ivan Beschastnikh for discussions about Synoptic, and to Yongquan Fu for assistances in deploying on and using with the IaaS platform.

This work is supported by the National 973 Program of China under the Grant No.2011CB302603, the National Natural Science Foundation of China (NSFC) under the Grant No. 61161160565 and No. 61103013, and the SRFDP under the Grant No. 20114307120015.

References

[1] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, *et al*, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Tech. Rep., 2010.

[2] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-Trace: A pervasive network tracing framework," in *USENIX NSDI'07*, pp. 271–284, 2007.

[3] Zipkin. http://twitter.github.io/zipkin/

[4] J. Zhou, Z. Chen, H. Mi, and J. Wang, "MTracer: A trace-oriented monitoring framework for medium-scale distributed systems," in *IEEE SOSE 2014*, pp. 266–271, 2014.

[5] H. Mi, H. Wang, H. Cai, Y. Zhou, M. R. Lyu, *et al*, "P-Tracer: Path-based performance profiling in cloud computing systems," in *IEEE COMPSAC 2012*, pp. 509–514, 2012.

[6] H. Mi, "Research on key techniques of performance maintenance for cloud services," Ph.D. dissertation, National University of Defense Technology, 2012.

[7] M. L. Massie, B. N. Chun, and D. E. Culler, "The Ganglia distributed monitoring system: Design, implementation, and experience," *Elsevier Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[8] A. Pnueli, "The temporal logic of programs," in *IEEE SFCS'77*, pp. 46–57, 1977.

[9] M. Leucker and C. Schallhart, "A brief account of runtime verification," *Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[10] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson, "Mining temporal invariants from partially ordered logs," *ACM SIGOPS Operating Systems Review*, vol. 45, no. 3, pp. 39–46, 2011.

[11] Synoptic. https://code.google.com/p/synoptic/

[12] H. Mi, H. Wang, G. Yin, H. Cai, Q. Zhou, *et al*, "Performance problems diagnosis in cloud computing systems by mining request trace logs," in *IEEE NOMS 2012*, pp. 893–899, 2012.

[13] A. Lakhina, M. Crovella, and C. Diot, "Diagnosing network-wide traffic anomalies," in *ACM SIGCOMM 2004*, pp. 219–230, 2004.

[14] D. Kondo, B. Javadi, A. Iosup, and D. Epema, "The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems," in *IEEE/ACM CCGrid 2010*, pp. 398–407, 2010.

[15] B. Schroeder and G. Gibson, "The computer failure data repository (CFDR): Collecting, sharing and analyzing failure data," in *ACM SC'06*, p. 154, 2006.

[16] Repository of availability traces. http://www.cs.illinois.edu/~pbg/availability/

[17] C. Germain-Renaud, A. Cady, P. Gauron, M. Jouvin, C. Loomis, *et al*, "The grid observatory," in *IEEE/ACM CCGrid 2011*, pp. 114–123, 2011.

[18] D. G. Feitelson, D. Tsafrir, and D. Krakov, "Experience with the parallel workloads archive," Hebrew University, Tech. Rep., 2012.

[19] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: Format + schema," Google, Tech. Rep., 2011.

[20] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, *et al*, "The grid workloads archive," *Future Generation Computer Systems*, vol. 24, no. 7, pp. 672–686, 2008.

[21] Y. Guo and A. Iosup, "The game trace archive," in *IEEE NetGames 2012*, pp. 1–6, 2012.

[22] Z. Zheng, T. C. Zhou, M. R. Lyu, and I. King, "Component ranking for fault-tolerant cloud applications," *IEEE Transactions on Services Computing*, vol. 5, no. 4, pp. 540–550, 2012.

[23] Hadoop. http://hadoop.apache.org/

[24] CloudStack. http://cloudstack.apache.org/

[25] AnarchyApe. https://github.com/ynroberts/anarchyape