PerfSig: Extracting Performance Bug Signatures via Multi-modality Causal Analysis

Jingzhu He
ShanghaiTech University
Shanghai, China
hejzh1@shanghaitech.edu.cn

Yuhang Lin
North Carolina State University
Raleigh, NC, USA
ylin34@ncsu.edu

Xiaohui Gu
North Carolina State University
Raleigh, NC, USA
xgu@ncsu.edu

Chin-Chia Michael Yeh
Visa Research
Palo Alto, CA, USA
miyeh@visa.com

Zhongfang Zhuang
Visa Research
Palo Alto, CA, USA
zzhuang@visa.com

ABSTRACT

Diagnosing a performance bug triggered in production cloud environments is notoriously challenging. Extracting performance bug signatures can help cloud operators quickly pinpoint the problem and avoid repeating manual efforts for diagnosing similar performance bugs. In this paper, we present PerfSig, a multi-modality performance bug signature extraction tool which can identify principal anomaly patterns and root cause functions for performance bugs. PerfSig performs fine-grained anomaly detection over various machine data such as system metrics, system logs, and function call traces. We then conduct causal analysis across different machine data using information theory method to pinpoint the root cause function of a performance bug. PerfSig generates bug signatures as the combination of the identified anomaly patterns and root cause functions. We have implemented a prototype of PerfSig and conducted evaluation using 20 real world performance bugs in six commonly used cloud systems. Our experimental results show that PerfSig captures various kinds of fine-grained anomaly patterns from different machine data and successfully identifies the root cause functions through multi-modality causal analysis for 19 out of 20 tested performance bugs.

KEYWORDS

Debugging, Bug signatures, Software reliability, Performance

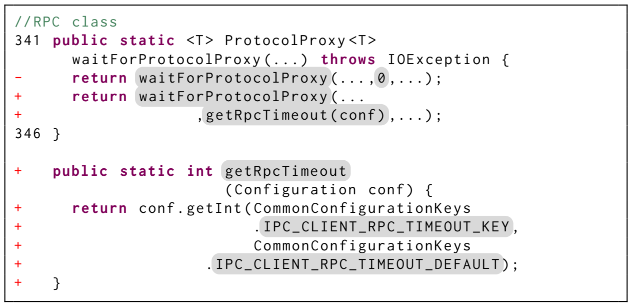
ACM Reference Format:
Jingzhu He, Yuhang Lin, Xiaohui Gu, Chin-Chia Michael Yeh, and Zhongfang Zhuang. 2022. PerfSig: Extracting Performance Bug Signatures via Multi-modality Causal Analysis. In *44th International Conference on Software Engineering, May 21–29, 2022, Pittsburgh, PA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510110

1 INTRODUCTION

Cloud systems are becoming increasingly complex, which dramatically increase the occurrence chance of various software bugs. In this work, we focus on those performance bugs [23, 33] which cause cloud systems to get stuck in a hang state or experience performance slow down. Performance bugs triggered in production cloud environments are notoriously difficult to diagnose and fix due to the lack of diagnostic information. When a performance bug occurs in production cloud environments, system operators and developers often need to put a lot of manual efforts to diagnose and fix the problem under time pressure. For example, it took more than 12 hours for Amazon to recover its membership service outage caused by a performance bug [4]. The bug was triggered by a limit on the allowable thread count, that is, the server hung when the number of server threads exceeded its pre-defined limit.

During our empirical bug study using popular bug repositories such as Jira and Bugzilla [2, 5], we observe that many performance bugs repeatedly occur in different versions of open source systems, which causes the community to perform redundant debugging over the same bug. Moreover, micro-services using containers [6] make the bug replication easier than ever – the same bug occurs in multiple containers that are created from the same container image. To this end, we believe creating signatures for different performance bugs can help system operators quickly identify recurrent bugs and expedite debugging process. A performance bug signature uniquely characterizes a performance bug in both symptoms (i.e., anomalous resource usages and/or abnormal log sequences) and root cause functions.

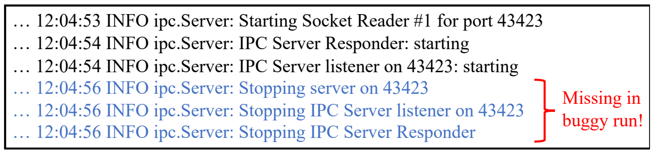
Previous work on performance bug detection and diagnosis (e.g., [14, 19, 20, 23, 25, 57, 59]) has two major limitations when applying to the production cloud environment. First, previous work (e.g., [14, 19, 20, 23]) mainly focuses on depicting performance bugs via analysis over single data type such as system metrics, system calls, system logs, or performance counters. However, a performance bug may manifest as anomalies in different data types. For example, an infinite loop bug may cause a persistently high CPU usage while a timeout bug can cause abnormal log sequences. Thus, it is likely that we may fail to extract bug signatures for some performance bugs if we only focus on analyzing one data type. Moreover, extracting anomalies alone often cannot uniquely characterize a performance bug because different performance bugs may exhibit similar anomaly patterns in one data modality. For example, different infinite loop bugs can all show increased CPU consumption. Thus, it is necessary to perform multi-modality analysis to extract representative signatures for different performance bugs. Second,



```
//RPC class
    public static <T> ProtocolProxy<T>
      waitForProtocolProxy(...) throws IOException {
-     return waitForProtocolProxy(...,0,...);
+     return waitForProtocolProxy(...
+                    ,getRpcTimeout(conf),...);
    }

+   public static int getRpcTimeout
                     (Configuration conf) {
+     return conf.getInt(CommonConfigurationKeys
+                        .IPC_CLIENT_RPC_TIMEOUT_KEY,
+                        CommonConfigurationKeys
+                   .IPC_CLIENT_RPC_TIMEOUT_DEFAULT);
+   }
```

Figure 1 The code snippet of the Hadoop-11252 Bug. The buggy code is invoked at the DataNode.



```
... 12:04:53 INFO ipc.Server: Starting Socket Reader #1 for port 43423
... 12:04:54 INFO ipc.Server: IPC Server Responder: starting
... 12:04:54 INFO ipc.Server: IPC Server listener on 43423: starting
... 12:04:56 INFO ipc.Server: Stopping server on 43423
... 12:04:56 INFO ipc.Server: Stopping IPC Server listener on 43423
... 12:04:56 INFO ipc.Server: Stopping IPC Server Responder
```

Figure 2 Logs generated by Hadoop-11252 bug. The logs are produced at the NameNode.

the existing tools [25, 57, 59] are not application-agnostic. The existing tools often require domain knowledge extracted from the source code or binary code. However, such information is not easily accessible in production systems. Therefore, it is essential to design a light-weight performance bug signature extraction tool without requiring domain knowledge.

1.1 A Motivating Example

We use Hadoop-11252 [1] bug to illustrate how a performance bug happens and how it manifests in different machine data types. The root cause of Hadoop-11252 bug is that the DataNode does not properly timeout the connection with the NameNode. Timeout is a commonly used failover mechanism to close the broken connection. As shown in Figure 1, the root cause function is `waitForProtocolProxy` function which passes 0 timeout value (0 means never timeout) to the timeout configuration incorrectly at the DataNode side. When the NameNode experiences some unexpected problems such as network outage, the DataNode hangs on waiting for the response from the remote server without producing any error information. As shown in Figure 2, we observe that the log entries that are typically produced by server stopping are missing at the NameNode side. Even if developers can discover the missing log anomaly at the NameNode side, it is still difficult for them to pinpoint the root cause function which is actually located at the DataNode side.

1.2 Contribution

In this paper, we present PerfSig, an automatic performance bug signature extraction tool which performs multi-modality analysis across different machine data including system metrics, system logs, and function call traces. When a performance alert or service level objective (SLO) violation is detected, PerfSig is triggered to analyze a time window of recent machine data. PerfSig first employs signal processing techniques and unsupervised machine learning methods to identify fine-grained anomaly patterns in various machine data. For example, for system metrics such as CPU usage time series, we employ fast Fourier transform (FFT) and time series discord mining to identify anomaly patterns such as fluctuation pattern changes, persistent increase, and cycle period changes. For system logs, we identify abnormal log sequences such as missing log entries in a certain common sequence or overly long time span for certain sequences. Next, PerfSig performs causal analysis between abnormal metric/log patterns and function call traces using information theory method mutual information (MI) [42]. Our causal analysis reveals the Granger causality (i.e., dependencies) [26] between the anomalies detected in different monitoring data (e.g., system metrics, system logs, function call traces). The goal is to identify the root cause function which is the top contributor to the metric or log anomaly. PerfSig outputs the performance bug signature as the combination of the detected anomaly pattern and the pinpointed root cause function.

Specifically, this paper makes the following contributions.

- We present a new multi-modality performance bug signature extraction framework which can precisely depict a performance bug using both fine-grained anomaly patterns and root cause functions.
- We describe a set of fine-grained anomaly detection methods to capture specific manifestation of a performance bug in system metrics or logs.
- We introduce an information theory based causal analysis approach to pinpointing root cause functions by discovering the causal relationship between function call traces and anomaly patterns of system metrics or logs.
- We have implemented a prototype of PerfSig and evaluated it over 20 real-world bugs that are discovered in six commonly used cloud systems. The results show that PerfSig can produce precise signatures for 19 out of 20 performance bugs.

The rest of the paper is organized as follows. Section 2 discusses the system design details. Section 3 presents the experimental evaluation. Section 4 discusses the threats to validity. Section 5 discusses the related work. Section 6 concludes the paper. Section 7 presents the data availability.

2 SYSTEM DESIGN

In this section, we describe the system design of the PerfSig system in details. We first give an overview of the system. Next, we present how we identify various fine-grained anomaly patterns in system metric and system log data, respectively. Finally, we discuss how we perform causal analysis between system metric/log anomalies and function call traces to identify the root cause function which contributes to the anomaly.

2.1 Approach Overview

PerfSig adopts a two-phase approach to extracting signatures for a performance bug, shown by Figure 3. When a performance alert is generated or a service level objective (SLO) violation is detected, PerfSig is triggered to extract performance bug signatures on-the-fly by analyzing a recent time window of system metrics, system logs,

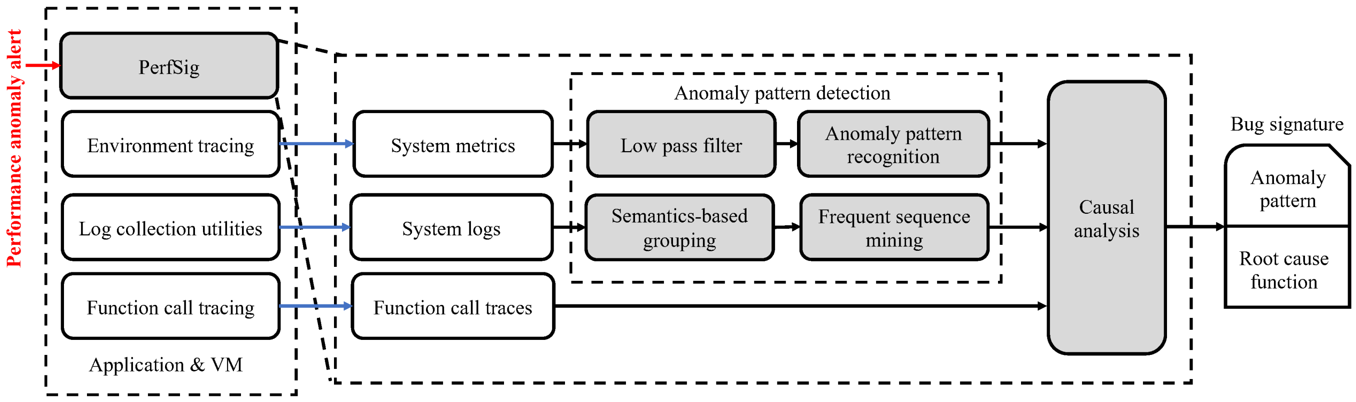


Figure 3 The architecture overview of PerfSig.

and function call traces. During the first phase (Phase I), PerfSig employs various signal processing and machine learning techniques to extract fine-grained anomaly patterns to capture the manifestation of the performance bug in either system metrics or system logs. Specifically, for system metrics, PerfSig uses time series analysis schemes to extract principal features such as fluctuation pattern changes, persistent increases, and cycle period changes (Section 2.2). For system logs, we leverage classification and frequent sequence mining to extract anomalous log patterns such as missing certain log entries or overly long time span of certain log sequences (Section 2.3). During the second phase (Phase II), PerfSig uses information theory approach to performing causal analysis between function call traces and detected abnormal system metric or system log patterns to pinpoint root cause functions (Section 2.4). Performance bugs often manifest as abnormal resource consumption and/or system log outputs (i.e., performance bug symptoms), which are typically caused by the root cause functions (e.g., infinite loops, missing timeout, costly operations). Therefore, we leverage the Granger causality between anomaly patterns and function time span anomalies to identify the root cause functions. Combining the phase I and Phase II results, PerfSig outputs the performance bug signature as the combination of the fine-grained anomaly pattern and the pinpointed root cause function.

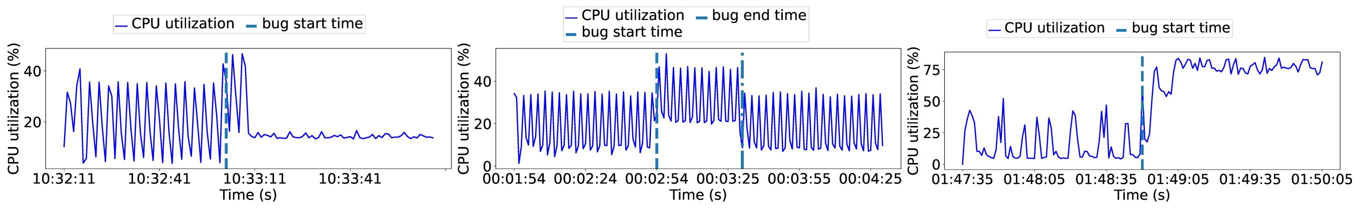
2.2 System Metric Anomaly Pattern Detection

Many performance bugs can manifest as changes in system metrics such as CPU utilization, memory utilization, and network traffic. However, different performance bugs can exhibit different anomaly patterns. For example, Figure 4 shows different CPU usage abnormal patterns for three real performance bugs. To extract distinctive signatures for different performance bugs, we need to not only detect anomalies but also extract fine-grained anomaly patterns.

We observe that the system metrics such as CPU consumption are inherently fluctuating. In order to extract principal anomaly patterns, we first leverage low pass filters to remove random noises from the raw system metric time series. The low pass filter performs data denoising by filtering out high frequency signals in original system metric time series. The rationale is that random fluctuations usually manifest as the high frequency signals. We transform the time series to the signals in frequency domain and drop high frequency signals. Note that we use relational values instead of absolute values, which avoids setting manual thresholds. If we choose a too large filtering percentage, we filter out too many signals which might include anomalies. If we choose a too small filtering percentage, we cannot filter out noises. In our experiment, we filter out top 50% high frequency signals. After that, we transform the signals in the frequency domain back to the time series. We conduct extensive experiments to compare the time series patterns before and after performing low pass filters. The results show that the anomaly patterns become more salient after filtering. For example, Figure 4b and Figure 9a show the same CPU usage patterns before and after the filtering, respectively. We can see the anomaly pattern is much clearer in Figure 9a. Next, we employ signal processing methods over denoised time series to extract principal anomaly patterns.

Fluctuation pattern changes. For dynamic data-intensive computing systems such as Hadoop, CPU utilization usually has periodical large fluctuations during normal run. It is because the application workload contains different types of interleaving jobs. For example, Figure 4a shows the CPU utilization's fluctuation change when Hadoop-15415 bug occurs. During the normal run (the first half of the figure), we observe large fluctuations. After the bug is triggered (the second half of the figure), the system hangs inside an infinite loop which fully consumes one CPU core and then CPU utilization stays at a steady value. We observe that many hang bugs in dynamic data-intensive systems often manifest as fluctuation pattern changes which refer to the cases when the system usage changes from normal fluctuating patterns caused by dynamic workloads during normal runs to nearly non-fluctuating patterns caused by the hang bugs during buggy runs. To capture this anomaly pattern, PerfSig calculates the standard deviations of a moving window in the system metric time series and identify the time when the moving window standard deviation experiences significant changes (e.g., dropping from a large value to a small value).

Persistent increases. Besides software hang bugs, slowdown bugs are another common category of performance bugs. We observe slowdown bugs caused by code inefficiency often consumes a large amount of computational resources (e.g., CPU) during the abnormal period. For example, Figure 4b shows the CPU increase



(a) Hadoop-15415 bug's fluctuation pattern change. The bug starts at 10:33:02. **(b) Hadoop-6133 bug's persistent increase. The bug starts at 00:02:54 and ends at 00:03:30.** **(c) Cassandra-7330 bug's cyclic pattern change. The bug starts at 01:48:50.**

Figure 4 Three commonly seen system metric anomaly patterns.

caused by the Hadoop-6133 bug. PerfSig detects such abnormal pattern using time series discord [56]. A sliding window is applied on the filtered system metric time series where the nearest neighbor distance [56] between the previous sliding windows and the current sliding window is computed. PerfSig detects the persistent increase pattern when the nearest neighbor distance shows significant increases.

Cyclic pattern changes. Many production server systems exhibit cyclic resource consumption patterns. It is because production server resource usage patterns are typically driven by production workload patterns. When the production workload exhibits regular patterns, the corresponding system usage patterns show cyclic patterns. We observe that when a performance bug is triggered, the workload changes, leading to the cyclic resource usage pattern changes. Figure 4c shows the CPU cyclic pattern change caused by the Cassandra-7330 bug. During normal run, CPU shows a cycle of nine seconds, while during buggy run, CPU does not show cyclic pattern. To detect such cyclic pattern changes, we employ fast Fourier transform (FFT) algorithm on a sliding window of system metric time series to extract the dominating frequencies whose magnitude values are in the top rank list. We detect the cyclic pattern change when the top frequency values experience changes.

2.3 System Log Anomaly Pattern Detection

We now describe how PerfSig extracts anomaly patterns from system logs. Much existing work focuses on detecting abnormal error logs that only appear in a buggy run. However, we observe that when a performance bug is triggered, the system usually does not produce any error log message. Instead, some log entries included in the normal run are missing during the buggy run, which are quite common among software hang bugs. In other cases like slowdown bugs, some log sequences could exhibit longer time span (i.e., the time duration from the start time of the first log entry in a sequence to the end time of the last log entry in a sequence) during the buggy run when compared with normal run.

Previous work in reconstructing execution path from the system logs contains three limitations: 1) focusing on sequential task execution [25]; 2) is combined with domain knowledge extracted from binary code [59]; and 3) is not generic for all the systems [57]. Compared with the existing work, PerfSig considers concurrent task execution and only takes the log entries and vector timestamp as the input. PerfSig does not require any application-specific knowledge. To discover the log sequences from interleaving log entries, PerfSig first classifies the log entries generated by different tasks.

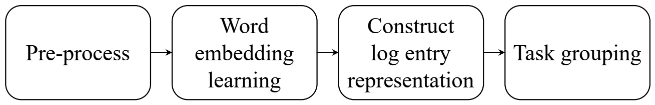


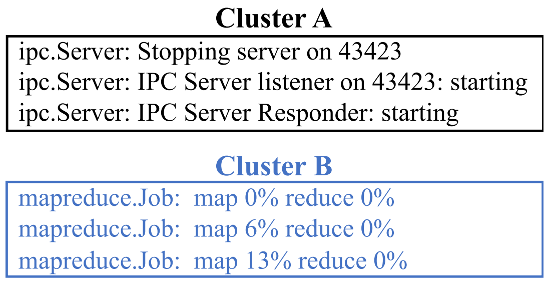
Figure 5 The log entry classification framework.

After the task classification is done, we further separate the log entries based on the time gaps. Then we perform frequent sequence mining to extract the log sequences.

Semantics-based Grouping. After we collect logs from distributed hosts, PerfSig classifies logs generated by different tasks. The idea is that logs generated by the same task have similar semantic meanings. To achieve this goal, we use the word embedding vector [45] to represent each word's contextual meaning. After that, we use the average of the word embedding vectors of the words' to represent each log entry, which is common for generating representation for natural language sentences [39]. Intuitively, log entries which have similar meanings have similar word embedding vectors. Therefore, we can apply clustering algorithm to grouping the similar log entries together. Figure 5 summarizes the classification procedures. Specifically, PerfSig pre-processes the logs to split the log entries into words, extracts word embedding vectors for each word, builds log entry representation by aggregating the word embeddings associated with each log entry, and classifies the entries generated by different tasks with the Self-Organizing Map (SOM) algorithm [36].

First, we pre-process each log entry to extract the words for learning word embedding vectors. The first step is to split the log entries based on brackets and parentheses, because content inside a pair of brackets and parentheses often represents one command or operation, e.g., database query command. After that, we split the log entry based on comma, full stop, colon, and semi-colon. Then we further separate them according to the spaces between words.

Once we have extracted words from the log entries, we treat each entry as a sentence and learns the word embedding representation for each word. The major advantage of using word embedding is that it considers words' semantic meaning in the contexts. It is based on the hypothesis that words occurred in the similar contexts tend to be semantically similar. We choose to use word embedding as the feature vector because log entries generated by the same task typically use short sentence with similar terms. We use word2vec with Continuous Bag of Words (CBoW) [45] to learn the word embedding vector to represent each word's meaning. Each word embedding is initialized as a n-dimensional vector. In each iteration,

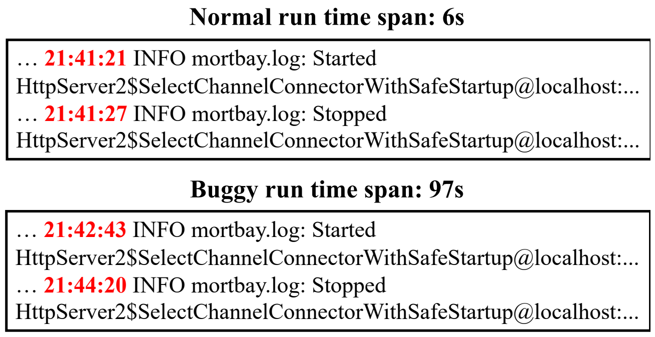

Cluster A

```
ipc.Server: Stopping server on 43423
ipc.Server: IPC Server listener on 43423: starting
ipc.Server: IPC Server Responder: starting
```

Cluster B

```
mapreduce.Job:  map 0% reduce 0%
mapreduce.Job:  map 6% reduce 0%
mapreduce.Job:  map 13% reduce 0%
```

Figure 6 The log entries' embedding vectors forms two clusters, one for each task.


Normal run time span: 6s

```
… 21:41:21 INFO mortbay.log: Started
HttpServer2$SelectChannelConnectorWithSafeStartup@localhost:…
… 21:41:27 INFO mortbay.log: Stopped
HttpServer2$SelectChannelConnectorWithSafeStartup@localhost:…
```

Buggy run time span: 97s

```
… 21:42:43 INFO mortbay.log: Started
HttpServer2$SelectChannelConnectorWithSafeStartup@localhost:…
… 21:44:20 INFO mortbay.log: Stopped
HttpServer2$SelectChannelConnectorWithSafeStartup@localhost:…
```

Figure 7 Logs generated by HDFS-4301 bug.

we train each word's embedding using the sum of m surrounding context words' embedding vectors. We update each word's embedding vector until the convergence is reached. In our experiment, n is set to 100 and m is set to five.

We construct each log entry's embedding vector by taking the average of all the words' embedding vectors in the log entry. After that, we apply the SOM clustering algorithm [36] to clustering log entries that belong to different tasks. Compared with the traditional distance-based methods, SOM model has better performance to cluster high-dimensional vectors.

Figure 6 shows an example of classification results. The log entries are split into two clusters and each cluster represents one type of task. For example, log cluster A represents Hadoop system establishes IPC connection between different processes. Log cluster B represents Hadoop system runs the map and reduce computational jobs.

Frequent Sequence Mining: After task classification, we successfully separate interleaving logs into log entry clusters, representing different tasks. The goal of this step is to extract frequent system log sequences with each log cluster, which often represent a set of execution such as client-server connection or threads communication. Performance bugs manifest as missing log entries in the log sequence or the log sequence has abnormal time span. For example, Figure 2 shows a complete log sequence. When Hadoop-11252 bug happens, the last three log entries are missing. Our idea is to only extract the constants (log keys) from the log entries. Then we split the log entries based on the time gaps and extract frequently occurred log sequences.

The log entries contain variables like socket reader number and port number, which are different in each IPC connection. We extract the constants (log keys) by adopting simple regular expressions. After we split the log entries into words, we keep the words that only contain alphabetic letters and replace other words with "$*$". Then we concatenate the words with a space as the log key. For each log entry cluster, we separate it based on the time gaps. The rationale is based on the observation that the same log sequence occurs after long time gap compared with its time span. We calculate the time gaps between each two consecutive log entries and derives the average value and standard deviation of each cluster. If the time gap between two consecutive log entries exceeds the average value + 2.0 $\times$ standard deviation, we separate the log cluster.

After we get separated log clusters and replace each log entry with the log key, we perform frequent sequence mining to extract log sequences from normal run data. We use PrefixSpan [28] to perform frequent sequence mining because it is more efficient compared with other methods. After we extract top frequent log sequences from normal run, we use them to detect anomalies during buggy run. Specifically, we detect two anomaly patterns, i.e., missing logs and abnormal log sequence time span.

Missing log entries anomaly pattern: During the anomaly detection phase, we first perform semantics-based grouping. In each cluster, we extract log keys and check whether the log keys belong to any extracted log sequence. If the answer is yes, we check whether other log keys of the log sequence also appear in the log cluster. For example, Figure 2 shows how we detect Hadoop-11252 bug. During normal run, we extract the complete log sequence from starting the socket reader to stopping the server responder. During buggy run, we perform semantic-based grouping to cluster log entries generated from IPC connection tasks together. For the extracted the log sequence, We find the log keys to start the server but we cannot find the log keys to stop the server.

Excessive time span anomaly pattern: Similar to missing log pattern detection, we perform semantics-based grouping and log key extraction. If all the log keys of one particular log sequence appear in one cluster, PerSig extracts the log sequence time span which starts from the first log key's occurring time to the last log key's occurring time. If the time span is excessive long, PerSig reports the abnormal log sequence time span pattern. For example, Figure 7 shows how we detect HDFS-4301 bug. During normal run, we extract the log sequence and the log sequence spans six seconds. During buggy run, the log sequence spans 97 seconds and PerSig reports the abnormal log sequence time span pattern.

2.4 Root Cause Analysis via Multi-modality Causal Analysis

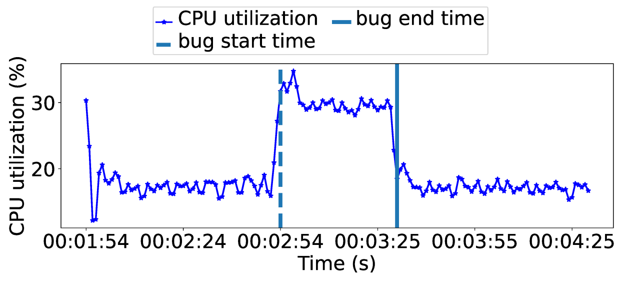
After we identify the anomaly patterns, we perform causal analysis to localize the root cause function.

2.4.1 Causal Analysis between System Metrics and Function Call Traces. After we identify three anomaly patterns of the system metric time series, we retrieve a window of the filtered time series which contains the anomaly patterns. Suppose the bug happens between time $[t_1, t_2]$, then we retrieve the system metric between $[t_1 - w, t_2 + w]$, where w is the parameter to control the window size. Then we retrieve all the function call traces which occurs in

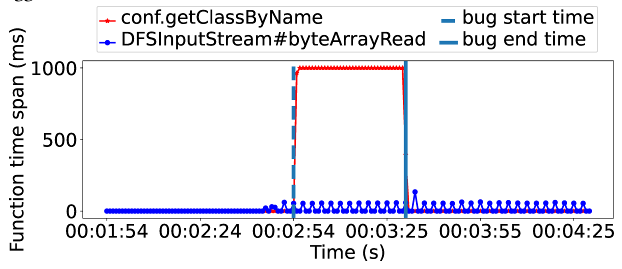
```
{"i":"2960606995495142","s":"ab3c06709e3bbab4",
"b":1622001774031,"e":1622001774058,
"d":"DFSInputStream#byteArrayRead",
"r":"YarnChild","p":[],
"n":{"path":".../job_1621397508582_0005/job.split"}}
...

{"i":"37dada28edec1ea0","s":"5cbe565f9c5a3c8b",
"b":1622001775036,"e":1622001810393,
"d":"conf.getClassByName","r":"Test","p":[]}
```

Figure 8 The extracted function call traces. "b" represents function's start timestamp, "e" represents function's end timestamp and "d" represents the function name. All the traces have the start time no earlier than 00:02:54 and end time no later than 00:03:30.



(a) The CPU utilization time series after filtering. The bug starts at 00:02:54 and ends at 00:03:30. We retrieve one minute normal run data before and after bug is triggered.



(b) The function call time series in Hadoop-6133 bug. `conf.getClassByName` **is the root cause function and** `DFSInputStream#byteArrayRead` **is an non-root cause function.**

Figure 9 Hadoop-6133 bug's time series.

the time window, i.e, $[t_1 - w, t_2 + w]$. For example, Figure 9a shows the CPU utilization time series. It contains the CPU spike occurred between 00:02:54 and 00:03:30. We set w to one minute, therefore, we include the normal run data from 00:01:54 to 00:02:54 and data from 00:03:30 to 00:04:30, because we want to include the sharp CPU changes at 00:02:54 and 00:03:30. Including normal run data can increase the accuracy of causal analysis. If we only consider the buggy run data, the frequently used functions also occur in the buggy run and may infer a high causality score. However, if we consider both normal run and buggy run, the frequent used functions always appear, which decreases the causal relationship between the system metrics and them. Figure 8 shows the retrieved function call trace snippet. All the function call traces are in the chronological order. We extract all the function call traces occurred between 00:02:54 and 00:03:30, i.e,, their start times are no earlier than 00:02:54 and end time no later than 00:03:30. We extract three kinds of information, i.e, beginning timestamp, ending timestamp and function name, to formulate the function call trace into time series.

Next, we formulate the function call traces into the time series. we extract the aggregated time span of each function between two consecutive time points of system metric time series. If CPU utilization are sampled at $t_1, t_2, ..., t_N$, then the function time series can be formulated as $T[t_1, t_2), T[t_2, t_3), ..., T[t_N, t_{N+1})$, where $T[t_i, t_{i+1})$ represents the function's aggregated time span during $[t_i, t_{i+1})$ period. For example, Figure 9b shows the `conf.getClassByName` function time series. During the time [00:02:25, 00:02:26), the bug happens and `conf.getClassByName` is invoked for all the one second period. Therefore, `conf.getClassByName` time series has the value of 1000 milliseconds at the time point 00:02:25. If M functions are invoked during the whole time window, then there are M function time series. In this way, the function time series is aligned with system metric time series, and we can apply causal analysis algorithm on them. We extract the time span as the function call's feature because performance issues such as hang or slowdown, usually manifest as the changes in function time spans.

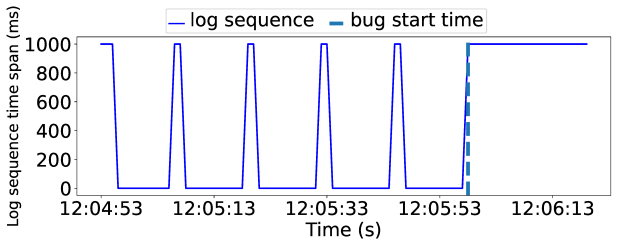
After we get all the function time series, we normalize each function time series and the system metric time series. After normalization, the sum of all the data points in one time series is equal to one. We adopt information theoretic method, i.e., mutual information (MI) [3, 42, 44], to infer the causal relationship between each function's time series and CPU time series. Because performance bugs often manifest as abnormal system usages and/or abnormal system log outputs, we leverage the Granger causality between system anomalies and function time span anomalies to identify root cause functions. Mutual information is one of the entropy-based methods which are important methods to perform exploratory causal analysis for time series data [42, 52]. Compared with linear correlation methods such as Pearson and Spearman coefficient, mutual information captures non-linear causal relationship between two time series. Mutual information measures how much knowing one of the two time series reduces uncertainty about the other. Mutual information not only consider the absolute value of each data point, but also consider the data distribution across the whole time series. Therefore, we can filter out the frequently invoked functions. It is because the frequently invoked functions have long time span during both buggy run and normal run. Considering the data distribution, they cannot have a larger mutual information than those functions which are only frequently invoked during buggy run. In comparison, the frequent invoked functions can have high correlation scores with system metric time series because they have long time spans during buggy run.

Mutual information between two time series X and Y is defined as:

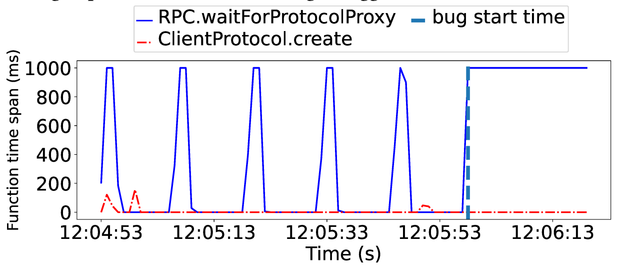
$$MI(X, Y) = \sum_{x,y} p(x, y) \log(\frac{p(x, y)}{p(x)p(y)}) \tag{1}$$

where $p(x)$ and $p(y)$ represent the probabilities of X and Y occurred at the sampling point. $p(x, y)$ represents the probability that X and Y occur at the same time.

We calculate the mutual information between each function time series and the system metric time series. We rank all the candidate functions based on the mutual information scores. We



(a) The log sequence time series. The bug is triggered at 12:05:58.



(b) The function call time series. `RPC.waitForProtocolProxy` **is the root cause function and** `ClientProtocol.create` **is an non-root cause function.**

Figure 10 Hadoop-11252 bug's time series.

then determine the function that contributes to the anomaly most as the one with the largest mutual information.

2.4.2 Causal Analysis between System Logs and Function Call Traces. We perform causal analysis on function time series and log sequence time series to localize the root cause function that contributes to the log anomaly. Even though system logs contain rich information, e.g., the class name that generates the logs, it is still essential to perform causal analysis, because the function that generates the log sequence is usually not the root cause function.

After we identify the missing log or longer execution patterns, we retrieve the log sequence's information, i.e., beginning time and end time, during both normal run and buggy run. The start time is the first log entry's invoking time and the end time is the last log entry's invoking time. For example, in Figure 2, the log sequence start time is 12:04:53 and end time is 12:04:56. If missing log pattern happens, we regard its time span as infinity because the log sequence never ends. Besides that, we retrieve all the function call trace within the same time window, i.e, during the period that the log sequences are produced.

We formulate the log sequences and the function call traces into time series. We sample them every one second and collect the aggregated time span during the one second interval. It is similar to how we formulate the function time series in Section 2.4.1. Note that the function time series still needs to be aligned with the log sequence time series. Figure 10a shows the log sequence time series and Figure 10b shows the function call time series. The root cause function has similar shape with the log sequence.

After generating log sequence and function time series, we calculate the mutual information between each function's time series and the log sequence to identify the root cause function, similar to Section 2.4.1.

3 EXPERIMENTAL EVALUATION

In this section, we first present the evaluation methodology followed by the experimental results. Next, we present several real bug examples including one negative case study where PerfSig fails to extract a signature for the bug.

3.1 Evaluation Methodology

Cloud systems: We studied 20 real performance bugs from six commonly used open-source cloud systems: Hadoop common library, Hadoop MapReduce big data processing framework, Hadoop HDFS file system, Hadoop Yarn resource management service, HBase database system, and Cassandra database system. Four Hadoop systems are set up in distributed modes to evaluate PerfSig's effectiveness over distributed system performance bugs.

Benchmarks: We use the "hang", "stuck", "block", "log", "CPU", "performance" and "slowdown" keywords to search for performance bugs. We manually examine each bug to determine whether it is a real performance bug triggered in production environments and whether it is reproducible in deterministic ways. To the best of our efforts, we successfully reproduced 20 bugs in six cloud systems. Table 1 shows our collected bug benchmark.

Setup: All the experiments were conducted in our lab machine with an Intel i7-4790 Octa-core 3.6GHz CPU, 16GB memory, running 64-bit Ubuntu v16.04 with kernel v4.4.0.

3.2 Implementation

Function call tracing: The function call traces are collected using Google Dapper framework. Dapper has various implementations on different production systems. For example, an implementation of Dapper, HTrace [7] is integrated into Hadoop since version 2.7.0. Another implementation of Dapper, Zipkin [9] is integrated into Hadoop, HBase, and Cassandra. Those implementations collect traces for error-prone functions in cloud systems. We can configure the parameters for Dapper tracing in the configuration files directly and deploy the production systems to trace the function calls.

Anomaly pattern analysis: We implement the anomaly pattern detection in Python 3.9. We use scikit-learn [46] package to implement FFT analysis and log classification. We use the Word2vec-CBoW implemented in Gensim [47] for embedding learning.

Hyperparameter in word embedding learning: We use the Word2vec-CBoW algorithm [45] with the embedding vector size of 100 and the window size of 5 words. The SOM is set to be 5x5 grid map and the weight vector length in SOM is set to be 100, the same as the embedding vector size.

3.3 Alternative Approaches

Performance bug signature is quite new. Previous tools only addressed partial problems. PerfSig first provides a comprehensive end-to-end solution to extract both symptoms and root causes as the bug signatures. To compare with PerfSig, we implement several alternative approaches to perform log analysis and causal analysis.

3.3.1 Logs Anomaly Pattern Detection. For system log anomaly pattern detection, we implement four alternative approaches to compare with PerfSig.

Table 1 Bug benchmark. bug ID* **represents it is a distributed system performance bug.**

Bug ID	Version	Symptom	Description
Cassandra-7330	2.0.8	Hang	The corrupted InputStream returns error code, causing an infinite loop.
Cassandra-9881	2.0.8	Hang	Improper exception handling skips loop index-forwarding API, causing an infinite loop.
Hadoop-11252*	2.5.0	Hang	the RPC connection timeout is missing, leading to system hanging.
Hadoop-15415	2.5.0	Hang	Misconfigured parameter indirectly affects loop index, causing an infinite loop.
Hadoop-5318	0.19.0	Slowdown	The AtomicLong operations cause contention with multiple threads.
Hadoop-6133	0.20.0	Slowdown	Extra calls cause 80x performance slowdown.
Hadoop-8614	0.23.0	Hang	Skipping after EOF returns error code, affecting loop stride.
Hadoop-9106*	3.0.0-alpha1	Slowdown	IPC connection timeout is hard-coded, causing a much longer failure recovery time.
MapReduce-5066*	2.0.3-alpha	Hang	Timeout is missing when JobTracker calls a URL, causing system hanging.
MapReduce-4089	2.0.0-alpha	Hang	Task status updates cause hung task never timeout.
MapReduce-3862	0.23.1	Hang	NodeManager hangs on shutdown due to struggling DeletionService threads.
MapReduce-7089	2.5.0	Hang	Misconfigured variable causes loop stride to be set to 0.
MapReduce-6990	2.5.0	Hang	Skipping on a corrupted InputStream returns error code, affecting loop stride.
HDFS-1490*	2.0.2-alpha	Hang	Timeout is missing for image transfer operations, causing system hanging.
HDFS-4301*	2.0.3-alpha	Slowdown	Timeout value on image transfer operation is large.
HDFS-7005*	2.5.0	Hang	DFS input streams do not timeout.
HBase-17341	1.3.0	Slowdown	Timeout is missing for terminating replication endpoint.
Yarn-163	2.0.0-alpha	Hang	Skipping on a corrupted FileReader returns error code, affecting loop stride.
Yarn-1630	2.2.0	Hang	YarnClient endlessly polls the state of an asynchronized application.
Yarn-2905	2.5.0	Hang	Skipping on a corrupted aggregated log file returns error code, affecting loop stride.

DBScan-embedding: We use the same embedding representation. However, instead of the SOM clustering algorithm, we use DBScan [49] to group the log entries generated by different tasks. DBScan is a popular non-parametric density-based clustering algorithm. It automatically determines the number of clusters when trained.

SOM-TFIDF: We use Term-Frequency-Inverse-Document-Frequency (TFIDF) [34] to extract features from each log entry. Term frequency (TF) captures the frequent words in a log entry. Inverse document frequency (IDF) is used to weight down the frequently used meaningless words such as "the", "an", and "on". The SOM algorithm is used to classify the TFIDF vectors associated with different log entries into different task groups.

DBScan-TFIDF: We use TFIDF to represent each log entry and DBScan [49] to classify the log entries into different task groups.

Topic-LDA: Latent Dirichlet Allocation (LDA) [16] is a popular technique to analyze the topics of natural language documents. When log entries are fed into the LDA algorithm, the algorithm estimates the probability of each log entry belonging to different topics. We choose the topic with the largest probability as its topic. Then we classify the log entries with the same topic into one task group.

3.3.2 Causal Analysis. We implement two alternative approaches for causal analysis for comparison with PerfSig. Both alternative approaches use correlation coefficients to predict causality. Specifically, we test the Pearson Correlation Coefficient [27] and Spearman's Rank Correlation Coefficient [27]. Pearson Correlation Coefficient measures the correlation using normalized co-variances of absolute values. Spearman's Rank Correlation Coefficient measures the correlation using normalized co-variances of ranked values. A larger causal score means the two time series are strongly correlated.

3.4 Results

Table 2 shows the results of signature extraction for bugs which manifest as system metric anomalies. PerfSig can identify three different anomaly patterns, i.e, fluctuation pattern changes, persistent increases and cyclic pattern changes, from all the 11 tested bugs. Moreover, PerfSig is capable of detecting the true root cause functions for all the tested bugs.

We also evaluated three different causal analysis methods (i.e., MI, Pearson and Spearman) under two different settings: with low-pass filter and without low-pass filter. Overall, the low-pass filter improves the quality of causal analysis result, no matter which causal analysis method is used. It is because smoothing the time series reduces irrelevant fluctuations brought by dynamic workloads and makes the anomaly pattern more salient. For example, in Hadoop-15415 bug, the `hflush` function ranks the highest when filtering is not employed. The `hflush` function is CPU-intensive, which is mis-detected as the root cause function due to a low causal score. In our experiments, we observe that setting a 50-90% filtering threshold produces the same results, so we choose 50% as our default filtering threshold.

When comparing the three different causal analysis methods, we can see that the proposed MI scheme outperforms the alternative Pearson and Spearman methods. The experimental results show that good causal analysis techniques needs to consider data distribution across the whole time series. Moreover, entropy based method such as MI is better than co-variance based methods.

Table 3 shows the results of bug signature extraction results for bugs which manifest as log anomalies. Specifically, We compare PerfSig (i.e., SOM-embedding) with four other alternative designs: DBScan-embedding, SOM-TFIDF, DBScan-TFIDF, and Topic-LDA. Overall, PerfSig is capable of detecting the anomaly pattern for eight out of nine bugs which show log anomalies.

First, we compare PerfSig with DBScan-embedding where the workload classification method, SOM, is replaced with DBScan. PerfSig outperforms DBScan-embedding because the word embedding vector is 100-dimensional. DBScan has poor performance on high-dimensional vectors due to the curse of dimensionality [37].

If we replace the embedding representation with the TFIDF representation, we can see PerfSig outperforms both alternative approaches with TFIDF, i.e, SOM-TFIDF and DBScan-TFIDF. It is because TFIDF only considers individual word occurrence and fails to

Table 2 Signature extraction results for bugs which have system metric anomalies. Pearson^f is pearson method with filter. Similarly, Spearman^f is spearman method with filter. For each method, we present the root cause function's rank using causal analysis.

Bug ID	Anomaly Pattern	Root Cause Function	Number of Candidate Functions	PerfSig	Pearson^f	Spearman^f	MI	Pearson	Spearman
Hadoop-8614	Fluctuation pattern change	IOUtils#skipFully	91	1	31	1	6	14	12
Hadoop-15415	Fluctuation pattern change	IOUtils#copyBytes	91	1	3	1	6	13	20
Yarn-163	Fluctuation pattern change	ContainerLogsPage#printLogs	91	1	24	2	8	20	81
Yarn-2905	Fluctuation pattern change	AggregatedLogsBlock #readContainerLogs	91	1	37	2	6	46	35
Yarn-1630	Fluctuation pattern change	YarnClientImpl#submitApplication	91	1	2	1	12	10	21
MapReduce-7089	Fluctuation pattern change	ReadMapper#doIO	91	1	4	1	5	6	48
Mapreduce-6990	Fluctuation pattern change	TaskLog#Reader	91	1	41	1	2	58	20
Hadoop-5318	Persistent increase	FSDataOutputStream#write	188	1	1	1	1	1	18
Hadoop-6133	Persistent increase	conf#getClassByName	115	1	1	1	21	24	32
Cassandra-9881	Cyclic pattern change	Scrubber#scrub	41	1	1	1	1	1	1
Cassandra-7330	Cyclic pattern change	StreamReader#drain	41	1	1	1	1	1	1

Table 3 Signature extraction results for bugs which have system log anomalies. ✗ represents the anomaly pattern and the root cause function cannot be identified. For each method, we present the root cause function's rank using causal analysis.

Bug ID	Signature (Anomaly Pattern, Root Cause Function)	Number of Candidate Functions	PerfSig	DBScan-embedding	SOM -TFIDF	DBScan -TFIDF	Topic-LDA
Hadoop-11252	Infinite log sequence timespan due to missing closing server log, RPC.waitForProtocolProxy	91	1	1	1	1	✗
MapReduce-5066	Infinite log sequence timespan due to missing closing server log, JobEndNotifier.localRunnerNotification	81	1	✗	✗	3	✗
MapReduce-4089	Infinite log sequence timespan due to missing stopping service log, PingChecker.run	79	1	1	1	1	1
Mapreduce-3862	Infinite log sequence timespan due to missing stopping service log, DeletionService.delete	81	1	1	1	1	✗
HDFS-7005	–	–	✗	✗	✗	✗	✗
HDFS-1490	Infinite log sequence timespan due to missing closing server log, TransferFsImage.getFileClient	93	1	✗	1	1	✗
Hadoop-9106	Abnormal log sequence timespan, Client.call()	84	1	1	✗	✗	✗
HBase-17341	Abnormal log sequence timespan, ReplicationSource.terminate()	8	1	1	✗	✗	✗
HDFS-4301	Abnormal log sequence timespan, TransferFsImage.getFileClient()	93	1	✗	1	1	✗

capture the semantics of each word. The embedding representation conversely captures the semantic information by modeling the contextual surrounding words. As system logs are written in a human readable format for developers to diagnose problems of the system, the advantage of embedding representation over TFIDF is clear and also is observed from the experimental results. We observe semantic grouping works well because the system log semantics are much simpler than natural language textual data. SOM and DBScan clustering have similar performance on TFIDF representation, because number of TFIDF dimensions is not large. Log entries typically are short sentences with limited key words.

Next, we replace the classification framework with topic extraction model (i.e., Topic-LDA), which is another kind of semantic analysis. We observe a much worse performance compared with the other methods. Topic-LDA has poor performance because the log entries are usually very short. According to a study on text documents from micro-blogging platform Twitter [8, 32], the performance of Topic-LDA degrades when the input text documents are short. According to [32], it is hard for Topic-LDA to extract semantics from short documents as Topic-LDA cannot obtain sufficient statistics from short documents.

PerfSig considers all functions invoked around the performance alert detection time as candidate root cause functions and ranks all candidate root cause functions based on the MI scores. As shown in Table 2 and Table 3, for 80% bugs, the number of candidate functions exceeds 70. In our experiments, the true root cause functions have the highest MI scores (102% higher than the second rank candidate functions on average) in 19 out of 20 bugs.

Table 4 The runtime overhead and diagnosis time. Hadoop represents the four Hadoop system, i.e., Hadoop Common, Hadoop MapReduce, Hadoop HDFS and Hadoop Yarn.

System	Workload	Tracing Overhead	Metric Anomaly Detection Time	Log Anomaly Detection Time	Causal Analysis Time
Hadoop	π calculation	0.18%	0.13±0.01s	0.22±0.26s	0.28±0.16s
HBase	database query	0.67%	0.13±0.01s	0.17±0.01s	0.02±0.002s
Cassandra	database query	1.7%	0.13±0.01s	0.36±0.03s	0.15±0.01s

```
//ReflectionUtils class
104  public static <T> T newInstance(...) {
117    setConf(result, conf);
119  }

59   public static void setConf(...) {
64     setJobConf(theObject, conf);
66   }

74   private static void setJobConf(...) {
80      Class<?> jobConfClass =
81       conf.getClassByName(
            "org.apache.hadoop.mapred.JobConf");
95   }

//Configuration class
761  public Class <?> getClassByName(...)
                  throws ClassNotFoundException {
762    return Class.forName(name, true, classLoader);
       /* duplicated costly operations to search for
       the same configuration class */
763  }
```

Figure 11 The code snippet of the Hadoop-6133 Bug.

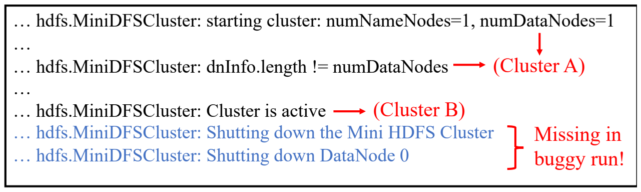
3.5 Overhead and Diagnosis Time

Table 4 shows the runtime overhead and diagnosis time. The runtime tracing overhead is below 2%. The diagnosis time are all less than one second. Note that although log anomaly detection uses the deep learning model, i.e., word embedding model, the diagnosis time is still very short because log entries are typically short sentences with repeated words.

3.6 Case Studies

We have described Hadoop-11252 bug's root cause in Section 1.1. PerfSig identifies the log sequence which starts from the first log entry of starting server to the last log entry of stopping server as in Figure 2. PerfSig identifies the bug as the missing log pattern. PerfSig extracts the log sequence time series during normal run and regard the log sequence's time span as infinity due to missing log during buggy run. PerfSig performs causal analysis on all the function call time series and the log sequence time series. The time series are shown in Figure 10. PerfSig determines the root cause function as `RPC.waitForProtoProxy` because it has the largest MI value.

Hadoop-6133 bug is caused by duplicated costly operations. As shown in Figure 11, when we use `ReflectionUtils.newInstance` function to initialize a new reflection instance at line 104, `setConf` function is invoked at line 117 to set the job configuration. `setConf` calls `setJobConf` function at line 64, then calls the `conf.getClassByName` function at line 81 to find a particular configuration class. However, the JDK function `Class.forName` invoked by `conf.getClassByName` is costly. When there are multiple threads which initialize the instances, `Class.forName` is frequently invoked to search for the same configuration class, which is unnecessary. When the bug is triggered, we observe 80x slowdown in the system. PerfSig identifies the bug anomaly pattern as the persistent increases, because `Class.forName` consumes a lot of CPU resources, as shown in Figure 9a. PerfSig then performs causal analysis on the CPU utilization time series and all the function time series. As shown in Figure 9b, PerSig determines the root cause function as `conf.getClassByName` because it has the largest MI value.



```
… hdfs.MiniDFSCluster: starting cluster: numNameNodes=1, numDataNodes=1
…
… hdfs.MiniDFSCluster: dnInfo.length != numDataNodes
…
… hdfs.MiniDFSCluster: Cluster is active
… hdfs.MiniDFSCluster: Shutting down the Mini HDFS Cluster
… hdfs.MiniDFSCluster: Shutting down DataNode 0
```

(Cluster A)
(Cluster B)
Missing in buggy run!

Figure 12 Logs generated by HDFS-7005 bug.

Negative Case Study: HDFS-7005 bug is caused by missing timeout for `DFSClient#newConnectedPeer`. When this bug happens, the Resource Manager hangs on waiting response from the HDFS cluster. We observe that the logs of shutting down HDFS cluster is missing as shown in Figure 12. PerfSig fails to classify the log entries to the same cluster. For example, the first two log entries are grouped in cluster A which represents DataNode's tasks. Since the log entries of the log sequence are classified to different tasks, we cannot extract the right log sequence from the log clusters.

4 THREATS TO VALIDITY

Benchmark bias: For the experimental evaluation, we have reproduced 20 performance bugs, which are all the bugs we can reproduce within a time limit. Up to the submission, we have exhaustively searched the bug reports in six common cloud systems from JIRA [5], selected all the true performance bugs and tried our best to reproduce them.

Parameter bias: The choice of several hyperparameters in the design can introduce bias on the system efficacy, such as the threshold of the low pass filter. In our experiment, we adjust the hyperparameters several times and choose the best one. The experimental results show that our design is less sensitive to the hyperparameters than the alternative approaches.

5 RELATED WORK

In this section, we discuss the existing work in the literature.

Single-modality data analysis: Previous work has worked on bug diagnosis by performing single modality data analysis. Cohen et al. [19, 20] leveraged Tree-Augmented Naive Bayes models and clustering methods to extract signatures from system metrics. PerfScope [23], PerfCompass [24] and TScope [30] diagnosed performance bugs by performing unsupervised machine learning on system call traces. Stitch [58] and lprof [59] reconstructed the domain knowledge and system model from the logs. CloudSeer [57] reconstructed the execution workflow entirely from interleaving

OpenStack logs. PLELog [55] performed semi-supervised learning combining HDBScan clustering and probabilistic label estimation to detect log anomalies. LogFaultFlagger [10] extracted vectors based on TF-IDF method and applied the kNN classifier to identifying abnormal logs. Deeplog [25] applied a deep neural network model, i.e., Long Short-Term Memory (LSTM), to detect anomalies from the system logs. The mystery machine [18] analyzed logs from Internet service to diagnose Facebook request latency. CSight [15] modeled the system behavior in the form of CFSM from system logs. Kabinna et al. [35] applied machine learning-based methods to determine the change risk of system logs based on certain metrics such as file ownership and log density. Li et al. [40] adopted the LDA method to extract topics from source codes automatically and studied the likelihood of topics to be logged. Lou et al. [43] constructed program workflow from event traces to understand system behaviors and verify system executions. PBI [14] and REPT [21] leveraged hardware traces, i.e., performance counters and Intel Processor Trace, to understand the software bugs. PerfSig performs multi-modality analysis to address the limitation that performance bugs manifest in different data types compared with the existing work.

Performance bug diagnosis and fixing: Previous work has proposed detection and fixing solutions for performance bugs. Hang doctor [17] detected soft hangs at runtime to address the limitations of offline detection. PerfChecker [41], and HangWiz [53] automatically detected soft hang bugs by searching the application code for known blocking APIs. Yang et al. [54] and He et al. [29] presented comprehensive empirical studies and detection solutions for two kinds of performance bugs, i.e., database-backed web application performance bugs and configuration-related performance bugs. PerfDebug [51] applied a data provenance-based technique to diagnose performance issues in applications that exhibit computation skew. DScope [22] and HangFix [31] adopted pattern-driven approaches to diagnosing and fixing software hang bugs in cloud systems. PerSig complements the existing work in providing a multi-modality signature extraction framework to depict performance bugs in a comprehensive fashion.

Causal analysis: Causal analysis attracts much attention in software debugging recently. For example, UniVal [38] transformed branch and loop predicates into variables and applied statistic causal analysis to infer the faulty component. REPTRACE [48] performed causality analysis on system call traces to identify the execution dependencies. Compared with the existing work, PerfSig performs multi-modality causal analysis among different types of data to extract the root cause functions of performance bugs, taking a further step in applying causal analysis in software debugging.

Causal analysis on time series data typically focused on identifying Granger-causal relationship among different time series [11, 13, 50, 52]. McCracken [44] presented a comprehensive review to perform exploratory causal analysis. The causal analysis techniques could be divided into two categories, i.e., regression-based and information theory-based methods. For the regression-based method, Arnold et al. [13] adopted linear lasso regression for identifying Granger-causal relationship among time series. Tank et al. [50] explored the possibility of detecting non-linear Granger-causal relationship among time series by training deep learning models (i.e., multi-layer perceptrons and recurrent neural networks) with sparsity constrain. For the techniques using information theory, the Granger-causal relationship was detected by entropy-based measures [11, 52]. Existing work leveraged different kinds of measures like directed information theory [11, 12] or transfer entropy [52]. PerfSig makes the first step to apply the information theory method mutual information to performance bug signature extraction.

6 CONCLUSION

In this paper, we present PerfSig, an automatic performance bug signature extraction tool. PerfSig can analyze various kinds of machine data including system metric, system logs, and function call traces to identify principal anomaly patterns and root cause functions as unique signature patterns for representing performance bugs. We have implemented a prototype of PerfSig and conducted extensive evaluations using 20 real world performance bugs on six commonly used cloud systems. Our results show that PerfSig can successfully extract unique signatures for 19 out of 20 tested performance bugs. PerfSig imposes low overhead to the cloud system, which makes it practical for production environments.

7 DATA AVAILABILITY

The data and the implementation of PerfSig are publicly available at https://github.com/jhe16/PerfSig.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable comments. This work was sponsored in part by NSF CNS1513942 grant, NSF CNS1149445 grant, and NSA H98230-17-D-0080 grant. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of NSF, NSA, or U.S. Government.

REFERENCES

[1] 2016. Hadoop-11252. https://issues.apache.org/jira/browse/HADOOP-11252.
[2] 2018. Bugzilla. https://www.bugzilla.org.
[3] 2018. Java Information Dynamics Toolkit (JIDT). https://github.com/jlizier/jidt.
[4] 2020. AWS outage. https://www.techrepublic.com/article/amazon-reveals-reason-for-last-weeks-major-aws-outage/.
[5] 2021. Apache JIRA. https://issues.apache.org/jira.
[6] 2021. Docker. https://www.docker.com/.
[7] 2021. HTrace. http://htrace.incubator.apache.org/.
[8] 2021. Twitter. https://twitter.com.
[9] 2021. Zipkin. https://zipkin.io/.
[10] Anunay Amar and Peter C Rigby. 2019. Mining historical test logs to predict bugs and localize faults in the test logs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 140–151.
[11] Pierre-Olivier Amblard and Olivier JJ Michel. 2011. On directed information theory and Granger causality graphs. *Journal of computational neuroscience* 30, 1 (2011), 7–16.
[12] Pierre-Olivier Amblard and Olivier JJ Michel. 2013. The relation between Granger causality and directed information theory: A review. *Entropy* 15, 1 (2013), 113–143.
[13] Andrew Arnold, Yan Liu, and Naoki Abe. 2007. Temporal causal modeling with graphical granger methods. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 66–75.
[14] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. 2013. Production-run software failure diagnosis via hardware performance counters. *Acm Sigplan Notices* 48, 4 (2013), 101–112.
[15] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, and Arvind Krishnamurthy. 2014. Inferring models of concurrent systems from logs of their behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering*. 468–479.
[16] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *the Journal of machine Learning research* 3 (2003), 993–1022.

[17] Marco Brocanelli and Xiaorui Wang. 2018. Hang doctor: runtime detection and diagnosis of soft hangs for smartphone apps. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.

[18] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. 2014. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 217–231.

[19] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. 2004. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *OSDI*.

[20] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. 2005. Capturing, indexing, clustering, and retrieving system history. *ACM SIGOPS Operating Systems Review* 39, 5 (2005), 105–118.

[21] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse debugging of failures in deployed software. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 17–32.

[22] Ting Dai, Jingzhu He, Xiaohui Gu, Shan Lu, and Peipei Wang. 2018. Dscope: Detecting real-world data corruption hang bugs in cloud server systems. In *Proceedings of the ACM Symposium on Cloud Computing*. 313–325.

[23] Daniel J Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. 2014. PerfScope: Practical Online Server Performance Bug Inference in Production Cloud Computing Infrastructures. In *SOCC*.

[24] Daniel J Dean, Hiep Nguyen, Peipei Wang, Xiaohui Gu, Anca Sailer, and Andrzej Kochut. 2015. Perfcompass: Online performance anomaly fault localization and inference in infrastructure-as-a-service clouds. *IEEE Transactions on Parallel and Distributed Systems* 27, 6 (2015), 1742–1755.

[25] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1285–1298.

[26] Michael Eichler. 2012. Causal Inference in Time Series Analysis. *Causality: Statistical Perspectives and Applications* (2012), 327–354.

[27] Isabelle Guyon et al. 2008. Practical feature selection: from correlation to causality. *Mining massive data sets for security: advances in data mining, search, social networks and text mining, and their applications to security* (2008), 27–43.

[28] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. 2001. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *proceedings of the 17th international conference on data engineering*. Citeseer, 215–224.

[29] Haochen He, Zhouyang Jia, Shanshan Li, Erci Xu, Tingting Yu, Yue Yu, Ji Wang, and Xiangke Liao. 2020. CP-detector: using configuration-related performance properties to expose performance bugs. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 623–634.

[30] Jingzhu He, Ting Dai, and Xiaohui Gu. 2018. Tscope: Automatic timeout bug identification for server systems. In *2018 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 1–10.

[31] Jingzhu He, Ting Dai, Xiaohui Gu, and Guoliang Jin. 2020. HangFix: automatically fixing software hang bugs for production cloud systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 344–357.

[32] Liangjie Hong and Brian D Davison. 2010. Empirical study of topic modeling in twitter. In *Proceedings of the first workshop on social media analytics*. 80–88.

[33] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices* 47, 6 (2012), 77–88.

[34] Karen Sparck Jones. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation* (1972).

[35] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, Mark D Syer, and Ahmed E Hassan. 2018. Examining the stability of logging statements. *Empirical Software Engineering* 23, 1 (2018), 290–333.

[36] Teuvo Kohonen. 1990. The self-organizing map. *Proc. IEEE* 78, 9 (1990), 1464–1480.

[37] Mario Koppen. 2000. The curse of dimensionality. In *5th Online World Conference on Soft Computing in Industrial Applications (WSC5)*, Vol. 1. 4–8.

[38] Yiğit Küçük, Tim AD Henderson, and Andy Podgurski. 2021. Improving fault localization by integrating value and predicate based causal inference techniques. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 649–660.

[39] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *International conference on machine learning*. PMLR, 1188–1196.

[40] Heng Li, Tse-Hsun Peter Chen, Weiyi Shang, and Ahmed E Hassan. 2018. Studying software logging using topic models. *Empirical Software Engineering* 23, 5 (2018), 2655–2694.

[41] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th international conference on software engineering*. 1013–1024.

[42] Joseph T Lizier. 2014. JIDT: An information-theoretic toolkit for studying the dynamics of complex systems. *Frontiers in Robotics and AI* 1 (2014), 11.

[43] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Jiang Li, and Bin Wu. 2010. Mining program workflow from interleaved traces. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. 613–622.

[44] James M McCracken. 2016. Exploratory Causal Analysis with Time Series Data. *Synthesis Lectures on Data Mining and Knowledge Discovery* 8, 1 (2016), 1–147.

[45] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[46] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[47] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA, Valletta, Malta, 45–50. http://is.muni.cz/publication/884893/en.

[48] Zhilei Ren, Changlin Liu, Xusheng Xiao, He Jiang, and Tao Xie. 2019. Root cause localization for unreproducible builds via causality analysis over system call tracing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 527–538.

[49] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. *ACM Transactions on Database Systems (TODS)* 42, 3 (2017), 1–21.

[50] Alex Tank, Ian Covert, Nicholas Foti, Ali Shojaie, and Emily Fox. 2018. Neural granger causality for nonlinear time series. *arXiv preprint arXiv:1802.05842* (2018).

[51] Jason Teoh, Muhammad Ali Gulzar, Guoqing Harry Xu, and Miryung Kim. 2019. Perfdebug: Performance debugging of computation skew in dataflow systems. In *Proceedings of the ACM Symposium on Cloud Computing*. 465–476.

[52] Raul Vicente, Michael Wibral, Michael Lindner, and Gordon Pipa. 2011. Transfer entropy—a model-free measure of effective connectivity for the neurosciences. *Journal of computational neuroscience* 30, 1 (2011), 45–67.

[53] Xi Wang, Zhenyu Guo, Xuezheng Liu, Zhilei Xu, Haoxiang Lin, Xiaoge Wang, and Zheng Zhang. 2008. Hang analysis: fighting responsiveness bugs. *ACM SIGOPS Operating Systems Review* 42, 4 (2008), 177–190.

[54] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. 2018. How not to structure your database-backed web applications: a study of performance bugs in the wild. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 800–810.

[55] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, and Wenbin Zhang. 2021. Semi-supervised log-based anomaly detection via probabilistic label estimation. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1448–1460.

[56] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Diego Furtado Silva, Abdullah Mueen, and Eamonn Keogh. 2016. Matrix profile I: all pairs similarity joins for time series: a unifying view that includes motifs, discords and shapelets. In *2016 IEEE 16th international conference on data mining (ICDM)*. Ieee, 1317–1322.

[57] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. 2016. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 489–502.

[58] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. 2016. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 603–618.

[59] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. lprof: A non-intrusive request flow profiler for distributed systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 629–644.