PerfScope: Practical Online Server Performance Bug Inference in Production Cloud Computing Infrastructures

Daniel J. Dean Hiep Nguyen Xiaohui Gu
North Carolina State University
{djdean2,hcnguye3}@ncsu.edu, gu@csc.ncsu.edu

Hui Zhang Junghwan Rhee Nipun Arora Geoff Jiang
NEC Labs America
{huizhang,rhee,nipun,gfj}@nec-labs.com

Abstract

Performance bugs which manifest in a production cloud computing infrastructure are notoriously difficult to diagnose because of both the difficulty of reproducing those bugs and the lack of debugging information. In this paper, we present PerfScope, a *practical* online performance bug inference tool to help the developer understand how a performance bug happened during the production run. PerfScope achieves *online* bug inference to obviate the need for offline bug reproduction. PerfScope does not require application source code or any runtime instrumentation to the production system. PerfScope is application-agnostic, which can support both interpreted and compiled programs running inside a cloud infrastructure.

We have implemented PerfScope and tested it using real performance bugs on seven popular open source server systems (Hadoop, HDFS, Cassandra, Tomcat, Apache, Lighttpd, MySQL). The results show that PerfScope can narrow down the search scope of the bug-related functions to a small percentage (0.03-2.3%) and rank the real bug-related functions within top five candidates in the majority of cases. PerfScope only imposes on average 1.8% runtime overhead to the tested server applications.

Keywords Testing and Debugging, Reliability, Performance

1. Introduction

It is a notoriously difficult task to diagnose a performance bug [22, 29, 35] that occurred inside a production cloud computing infrastructure[1]. The reasons are multifold. First, it is difficult, if not totally impossible, to reproduce a production-run performance bug outside the cloud infrastructure because those performance bugs often only manifest under specific conditions (e.g., particular user inputs, certain system configurations, non-deterministic system events). Thus, it is hard to apply existing offline debugging tools such as GDB directly to diagnose those performance bugs that cannot be reproduced outside the production cloud infrastructure. Second, unlike crash failures, performance bugs often provide little diagnostic information. Many performance bugs even do not produce any error message since they are *unexpected* by the developer. To exacerbate the problem, it is often impractical to perform detailed system execution tracing inside the cloud infrastructure due to concerns about prohibitive cost and user privacy. However, a modern server system typically consists of tens of thousands of functions. Searching the buggy functions among tens of thousands of functions without any clue becomes an impossible task in many cases.

In this paper, we propose PerfScope, a practical *online* bug inference tool to help the application developer understand why a performance bug occurs in the production cloud computing infrastructure. One big advantage of PerfScope over existing offline debugging tools is that PerfScope does not require production-run bug reproduction by achieving online bug inference. PerfScope also does not require application source code or any system instrumentation, which makes it practical for production cloud infrastructures. PerfScope is application-agnostic, which can be applied to any program (e.g., C/C++, Java, Python, etc.) running inside the cloud system. Our work focuses on diagnosing performance bugs [22] that are defined as software bugs causing performance anomalies (e.g., software hang, performance slowdown). Performance anomalies caused by other problems (e.g., hardware faults, configuration issues, kernel bugs) are outside the scope of this work.

When a performance anomaly is detected by an existing online anomaly detection tool [16, 37], PerfScope is trig-

SoCC '14, November 03 - 05 2014, Seattle, WA, USA.
Copyright 2014 ACM 978-1-4503-3252-1/14/11…$15.00.
http://dx.doi.org/10.1145/2670979.2670987

[1] In this paper, we do not distinguish different types of cloud systems (e.g., infrastructure-as-a-service clouds, platform-as-a-service clouds) since we believe that our problem and approach apply to different cloud systems.

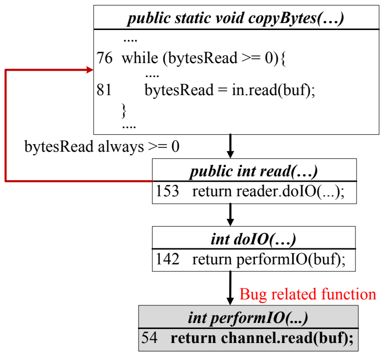

public static void copyBytes(…)
76 while (bytesRead >= 0){
81 bytesRead = in.read(buf);
bytesRead always >= 0
public int read(…)
153 return reader.doIO(...);
int doIO(…)
142 return performIO(buf);
Bug related function
int performIO(...)
54 return channel.read(buf);

Figure 1. A subset of function call graph for a real HDFS performance bug. When an input file larger than 2GB is used, the internal variable (`int`) representing the content length of the input file overflows. This causes the call to `in.read(buf)` at line 81 to never return -1 causing an infinite loop. The bug point is highlighted in bold.

gered to perform online bug inference by analyzing a window (e.g., five minutes) of recent system call traces collected on the faulty server pinpointed by a component-level fault localization tool [14, 30]. The rationale behind our approach is twofold. First, a system call trace can be easily collected via kernel tracing tool[2] such as Linux Tracing Tool next generation (LTTng) [18] in production cloud computing infrastructures with low overhead. Moreover, compared to other low-level tracing tools (e.g., hardware performance counter monitoring tool [34]), the system call tracing tool works in virtualized environments, which makes it suitable for virtualized cloud infrastructures. Second, we observe that performance anomalies are often caused by anomalous interactions between the application and the kernel. Since applications use system calls to interact with the kernel, we can detect anomalous application-kernel interactions by closely monitoring system calls. For example, Figure 1 shows a real performance bug in the HDFS system: an overflow issue causes the `while` loop to never end. This buggy execution produced 1.4 million system calls. As we will show in Section 4, we find that over 90% buggy functions produce system calls by performing static analysis over 228 real performance bugs.

However, it is a challenging task to identify bug-related functions from the massive, noisy raw system call traces. First, we need to develop efficient *online* system call trace analysis algorithms that can quickly identify the anomalous system call sequences from millions of system calls. Second, we need to develop fast and robust algorithms that can accurately map those anomalous system call sequences to specific bug-related *application functions*. Since our goal is to support different applications (e.g., both compiled and interpreted programs), we cannot rely on the program counter information to trace back to the application functions, which does not work for interpreted programs and some compiled programs using certain compilation optimizations.

To address those challenges, PerfScope first uses lightweight unsupervised learning techniques to identify anomalous system call sequences. Since the system call trace for a multithreaded server program is often huge, we first segment the raw system call trace into fine-grained units of closely related system calls called *execution units*. We then characterize different execution units using various features (e.g., system call appearance vector, system call execution time vector, system call frequency vector). Next, we use a hybrid anomaly detection scheme (i.e., hierarchical clustering plus outlier detection) to identify abnormal execution units.

PerfScope then uses those abnormal execution units as clues to identify the bug-related functions. We propose a signature-driven approach that creates a *robust* signature for each application function offline outside the production computing environment. Each function signature maps the function into a set of *closed frequent system call episodes* (e.g., `sys_write`, *, `sys_read`). When a production-run performance bug occurs, we extract the closed frequent system call episodes from those anomalous execution units and map those episodes back to a short list of candidate buggy functions using the function signatures. We also provide a ranking to those identified functions based on the abnormality degree observed in those frequent system call episodes. Note that we do not view PerfScope as the final debugging tool. Instead, PerfScope provides the developer with a small number of candidate buggy functions to examine, which makes production-run performance bug diagnosis much more tractable.

Specifically, this paper makes the following contributions:

- We propose an online performance bug inference tool that works for production cloud computing environments without requiring offline bug reproduction, application source code, or runtime system instrumentation.
- We present a robust application function signature extraction approach that works for different types of applications (e.g., both compiled and interpreted programs) running inside virtualized computing environments.
- We describe a set of online system call trace analysis algorithms that can quickly process millions of system calls to produce a short list of bug-related functions.
- We evaluated PerfScope using *real* performance bugs in popular open source systems (Hadoop, HDFS, Casandra, Tomcat, Apache, Lighttpd, MySQL). The results show that PerfScope is effective, which can narrow down the search scope for the bug-related functions to a small percentage (0.03-2.3%). The real bug-related functions are ranked within top five candidates in 8 out of 12 cases and are always ranked within top 15. PerfScope only

[2] Unlike the user level system call tracing tools such as ptrace [3], our experiments show that the kernel-based system call tracing only imposes on average 1.8% overhead to the tested server applications.

imposes on average 1.8% runtime overhead to the tested server applications.

Assumptions: We design PerfScope based on the following assumptions: 1) Performance bugs manifest as time or frequency changes in system calls. As mentioned earlier, over 90% of the real performance bugs we studied generate system calls; 2) We do not have access to application source code or any other application knowledge; and 3) We assume that function signatures can be profiled offline outside the production environments and remain stable under different inputs, workload intensities, and resource allocation settings. Although we cannot assure that this assumption holds on every system, our experiments with seven popular server systems under different workload conditions show that our approach is applicable to different types of functions without requiring application specific tuning.

The rest of the paper is organized as follows. Section 2 compares our work with related work. Section 3 describes the design of the PerfScope system. Section 4 presents our experimental evaluation. Finally, the paper concludes in Section 6.

2. Related Work

Our work is closely related to *Triage* [39] since both provide onsite production-run failure diagnosis framework. The key idea of Triage is to use checkpoint-replay with input/environment modification to perform just-in-time problem diagnosis by comparing normal runs and failure runs. In contrast, PerfScope does not require repeated replays, which makes PerfScope less intrusive to the production computing environment. Moreover, PerfScope specifically targets the performance bugs, which often do not manifest as code block changes between normal and failure runs. By detecting system call invocation changes and leveraging the robust function signatures, we can identify the buggy functions more effectively for performance bugs.

X-ray [12] provides automatic diagnosis support for performance anomalies caused by inputs or misconfiguraitons. It uses binary instrumentation to track the information flow in order to estimate the cost and the likelihood that a block was executed due to each potential root cause (e.g., configuration option). However, X-ray cannot diagnose the performance anomalies caused by software bugs. Similar to Triage, X-ray also requires replay support in order to offload the heavyweight binary instrumentation and analysis outside the production system. In contrast, PerfScope does not require any record and replay to identify the buggy functions.

Jin *et al.* [22] present a study of 109 real world performance bugs in 5 different systems. By manually checking the patches of known problems, they are able to then build an efficiency rule-based checker which was able to identify previously unknown performance problems in deployed software. However, the white-box approaches are only suitable for offline debugging due to the source code requirements. Additionally, existing white-box analysis tools mainly focus on detecting possible performance bugs instead of diagnosing a specific production-run performance bug. Directly applying existing bug detectors will lead to large false positives and false negatives as demonstrated by previous work [21].

Fay [19] proposes to use kernel drivers or DLLs along with hotpatching to collect and summarize detailed user-level and kernel-level traces without modifying the underlying system. Fay needs to combine user level and kernel level tracing to diagnose performance problems, imparting up to 280% overhead. In contrast, PerfScope only relies on kernel-level system call tracing, imparting on average 1.8% overhead.

Magpie [13] instruments middleware and packet transfer points to record fine-grained system events and correlates these events using an application specific event schema to capture the control flow and resource consumption of each request. DARC [38] identifies functions that are the main contributors to the latency peaks in a OSProf profile. Although both systems can be used to diagnose performance problems, they require instrumentations to the application or the middleware, which is difficult to be deployed in the production environment. In contrast, PerfScope does not require any application or middleware instrumentations, which makes it practical for production computing environments.

Fournier *et al.* [20] proposed to analyze the blocking behavior using kernel-level system events for diagnosing performance problems in multi-core systems. Similar to PerfScope, their approach only relies on the kernel level tracing, which imparts little overhead to the application. However, their approach cannot identify bug related application functions that can directly help the developer diagnose the performance bugs.

Previous work also explored black-box performance debugging schemes. Cohen *et al.* [15] proposed to use machine learning models to correlate the system-level metrics with the performance anomaly states for system metric attribution. Although black-box approaches are often light-weight and non-intrusive, they can only provide coarse-grained diagnosis (e.g., identifying fault related system metrics). PBI [11] used hardware performance counters along with pre-defined predicates to characterize concurrency and semantic bugs for compiled programs. In contrast, PerfScope focuses on performance bugs and can identify bug related functions for both compiled and interpreted programs.

Our previous work, PerfCompass [17], focuses on differentiating external faults (e.g., interference from co-located applications) from internal faults (e.g., software bugs) for production system performance anomalies. However, PerfCompass cannot localize buggy functions. Thus, PerfScope is orthogonal to PerfCompass.

3. System Design and Implementation

In this section, we present the design and implementation of the PerfScope system. We first describe the offline function signature extraction scheme. We then describe the online bug inference algorithms.

3.1 Offline Function Signature Extraction

We combine dynamic binary instrumentation with frequent episode learning techniques to extract the function signatures without requiring application source code. Our scheme constructs robust function signatures by learning what *closed frequent system call episodes*[3] can be generated by each function. Unlike other alternative approaches using low level system metrics (e.g., execution time, CPU usage) that often vary significantly among different platforms, we found closed frequent system call episodes remain stable under different workloads and resource allocation settings as shown by our experiments. Thus, we can perform function signature *offline outside* the production environment without requiring the exactly same workload or environment that trigger the production-run performance bug.

3.1.1 Binary-based Function Signature Profiling

To learn what system calls are produced by each function, we correlate the *function execution list* provided by the dynamic binary instrumentation tool and the system call trace provided by LTTng based on the timestamp information[4].

The *function execution list* records all the function entry points and exit points from an executing application. Each entry in a function execution list is in the form {`time stamp`, `enter function` f_i, `thread ID`} or {`time stamp`, `exit function` f_i, `thread ID`}. For C/C++ programs, we use the Pin tool [26] to insert the logging code into the binary to record when a function is called and when it returns with a timestamp. For Java programs, we insert hooks into the Java Virtual Machine (JVM) to intercept function invocations as they are being executed, logging when each function is called and when it returns with a timestamp.

Each system call entry consists of a pair of logs: 1) [`timestamp_entry`] `system call name`: {`procname`, `ppid`, `process ID`, `thread ID`, `cpu ID`} and 2) [`timestamp_exit`] `sys_exit`: {`procname ppid`, `process ID`, `thread ID`, `cpu ID`}. We can derive the system call execution time by computing the difference between timestamp_exit and timestamp_entry.

We then create a dynamic stack of active functions for each running thread to determine which function was active when each system call was generated. When a function is called, we push the function onto the corresponding active

[3] Closed frequent episodes mean that we only consider the maximum frequent episodes, not any sub-sequence of those frequent closed episodes.

[4] Although we can also use the dynamic binary instrumentation tool (e.g., the PIN tool) to collect the system call trace directly, those tools do not provide the function origin of each system call by themselves.

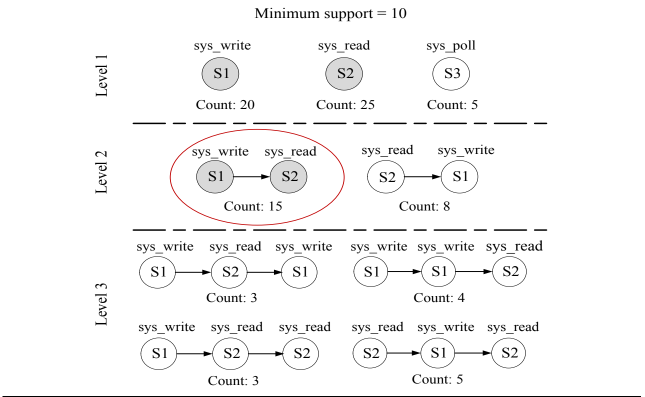


Figure 2. Frequent closed system call episodes discovery. All the frequent system call episodes (counter >10) are highlighted in gray color. The frequent closed episode is highlighted within a circle.

function stack based on the thread ID associated with the function. When a function returns, we pop the function off the active function stack. Given a system call s_i with a thread ID $tid = k$, we compare its exit timestamp T_{s_i} with the timestamp T_{top} of the top function f_i on the active function stack for the same thread $tid = k$. We also compare T_{s_i} with the timestamp T_{next} of the next function f_{i+1} in the function execution list. If the system call s_i occurs after the application enters f_i but before enters f_{i+1} (i.e., $T_{s_i} \geq T_{top}$ and $T_{s_i} < T_{next}$), we know s_i must have been generated by f_i. If we find the exit timestamp of the current system call s_i is larger than or equal to the timestamp of the next function in the function execution list (i.e., $T_{s_i} \geq T_{next}$), we push the next function f_{i+1} onto the stack and label that the system call s_i is produced by f_{i+1}. This process continues until either we have no more functions to process or no more system calls in the system call trace.

After the annotations for all the system calls are done, we group all the system calls produced by the same function together and sort them based on their timestamps into time ordered sequences. For example, if the system call `sys_write` annotated with function f_i is followed by `sys_read` annotated with the same function, we add the sequence {`sys_write`, `sys_read`} into the function profiling result for f_i. If a function consists of any branch statements, the profile of the function may consist of a set of system call sequences (e.g., {{`sys_write`, `sys_read`}, {`sys_poll`, `sys_futex`}}) when different branch paths are executed.

3.1.2 Frequent System Call Episode as Robust Function Signature

Although we could use the set of the *raw* system call sequences to represent the signature of the function, we found that those raw system call sequences are sensitive to execution environment changes. Since our goal is to find a robust function signature that is stable under different execution environments, we extract *closed frequent system call episodes* from those raw system call sequences, which can better characterize the key distinct behaviors of different applications. During our experiments, we observe that those frequent system call episodes are quite stable under different execution environments.

We extract closed frequent system call episodes from the raw system call sequences for each function using a common frequent episode mining algorithm called the *A-priori* method [6, 33]. The basic idea of the A-priori method is to start at the basic event type (e.g., a single system call) and then build up more complex sequences by taking all possible permutations using all the frequent episodes at all lower levels (i.e., frequent episodes of smaller size), shown by Figure 2.

The frequent episodes search iterates until a maximum level is reached where no frequent episode can be found. We perform pruning to reduce the number of items to be processed at each level. The intuition behind our pruning scheme is that a frequent episode at a higher level must be a frequent episode at a lower level. So it is only necessary to generate the permutations of the episodes we marked as frequent in the previous level. A *minimum support value* (*L*) is usually used to define the frequency lower bound for generating frequent episodes.

Figure 2 illustrates the frequent closed system call sequence discovery algorithm using a minimum support value $L = 10$. We start from three single system call episodes. We then drop `sys_poll` since its count does not meet the minimum support. At the level 2, we search all the permutations of the frequent episodes in the level 1 and identify (`sys_write`, `sys_read`) as a frequent episode. Moving up the level 3, we consider the permutations of all the frequent episodes found in the level 1 and 2. We found four possible permutations and none of them meets the minimum support. We stop the frequent episode search and outputs the (`sys_write`, `sys_read`) as the frequent closed episode.

We implement frequent closed episode frequency counting using finite state machine (FSM) based matching algorithm. During the candidate generation phase, we build a FSM for each candidate episode. Each state in the FSM represents a system call type. Given an input system call sequence, we match this sequence with all the candidates' FSMs simultaneously. When the input system call matches the current state in the FSM, we move the match pointer to the next state in the FSM. If the input system call does not match, we reset the match pointer to the start state. When all the states in the FSM are matched, we increase the frequency counter of the candidate episode represented by this FSM by 1 and reset the match pointer to the start state of the FSM.

In our experiments, we found that if we only consider contiguous frequent episodes, we often only discover trivial system call sequences that cannot help us distinguish different functions. Thus, we modified our algorithm to discover *non-contiguous* frequent episodes such as (`sys_write`, $*$, `sys_read`), where $*$ represents arbitrary system calls or zero system call. To discover non-contiguous frequent episodes, we only need to make a minor change to the above matching algorithm: if the input system call does not match the current state in the FSM, we do not reset the match pointer to the start state and wait until the next match occurs.

How to set a proper minimum support value (L) is a tricky issue. We calculate the minimum support value using the standard way of multiplying the number of items (i.e., system calls included in the execution unit N_{sys_call}) with a certain percentage p [40], that is $L = N_{sys\_call} \times p$. In our experiments, we use $p = 1\%$. We also conduct the sensitivity study on other possible values of p to evaluate its impact to our system. The general principle behind the minimum support value selection is that we should not use an overly high minimum support value, which prevents us from finding any frequent episodes; we should also avoid using a trivial minimum support value (e.g., $L = 1$) as this causes us to find too many frequent episodes (e.g., tens of thousands). So we set $L = max(min(N_{sys\_call} \times p, 10), 2)$ to cap the minimum support value at 10 and avoid the trivial minimum support value. We found our calculation scheme works well for all different systems we tested.

It can be time-consuming to perform the signature extraction for a complex server program consisting of tens of thousands of functions. We observe that the function signature extraction for different functions are independent from each other and can run in parallel. So we implement a distributed function signature extraction system to speed up the profiling process. Moreover, the function signature extraction is done offline, which will not affect our online bug inference time.

Note that PerfScope also does not require to profile all different function call paths but just individual functions. The function call path of the buggy run is inferred from the system call trace of the production-run directly, which will be explained in Section 3.2. We can also selectively profile a subset of user defined functions and exclude those uninteresting functions such as system libraries. For example, for C/C++ programs, we can use indexing tools such as cTags [1] to generate the list of profiled functions. For Java programs, we can use package names (e.g., Hadoop, catalina) to select the functions to be profiled.

3.2 Online Bug Inference

Our online bug inference scheme consists of three major steps: 1) extracting different execution units from the raw system call trace; 2) identifying anomalous execution units; 3) mapping anomalous execution units into a rank list of bug related functions. We will describe each step in detail in this section.

3.2.1 Execution Unit Extraction

The system call trace collected by LTTng is a massive stream of mixed system calls produced by different processes, threads, and functions. To identify bug-related functions, we first need to segment the raw system call trace into a set of execution units. Ideally, each execution unit should consist of a set of semantically related system calls which are produced by the same function, thread, and process. Since the system call trace collected by LTTng provides process ID and thread ID, it is easy to segment the trace into different threads/processes. However, during experiments, we found that the thread-based segmentation is sometimes insufficient. First, we observe that some server systems (e.g., Apache web server) use a pre-allocated thread pool in order to avoid the overhead of constantly creating new threads. The threads in the thread pool are reused for different tasks, which can cause the behavior of those threads to vary significantly over time. Second, some long running threads often execute multiple different functions. To address this issue, we propose to use the *time gap* to split those threads. The intuition is that a large time gap between two consecutive system calls in a thread implies a thread recycling or function switching. Therefore, we compute the intervals between each pair of contiguous system calls in the trace and use the mean $+ 2\times$ standard deviation as the segmentation threshold to split the per-thread trace. We found this segmentation threshold works well for all the tested systems and also conducted sensitivity study on this threshold value in the experiments.

During dynamic application runtime, we need to address one issue with the time gap based segmentation. We observe that the context switch between different threads caused by CPU scheduler sometimes creates a time gap between two semantically related system calls. Thus, we need to ignore those time gaps caused by the context switch. Luckily, it is easy to detect context switches in the LTTng collected system call trace. For example, when we detect either a thread ID change or a CPU ID change, we know a context switch has happened.

3.2.2 Abnormal Execution Unit (AEU) Identification

We use light-weight unsupervised learning methods to identify abnormal execution units. We chose unsupervised learning approach because it does not require any labelled training data which are hard to obtain the production environment. Specifically, we use a hybrid anomaly detection algorithm that combines hierarchical clustering and outlier detection.

We first apply a top-down hierarchical clustering [23] to group those execution units that perform similar operations together. This grouping is important, which can not only increase our accuracy but also reduce the processing time as the outlier detection step is more costly. For this purpose, we extract a *system call appearance vector* for each execution unit. The length of the system call appearance vector is equal to the number of unique system call types (e.g., `sys_write`, `sys_poll`) seen across all the execution units. Each system call type is assigned a position in the vector. The value is set to 1 if that type of system call is present in the execution unit, or 0 if it is not. For example, consider two execution units, {`sys_write`, `sys_read`} and {`sys_poll`, `sys_read`}. In this case, system call appearance vector follows the format of [`sys_write`, `sys_read`, `sys_poll`]. The system call appearance vector for the two execution units would be $[1,1,0]$ and $[0,1,1]$, respectively. We chose the system call appearance vector as the clustering feature vector to achieve robust grouping since two similar execution units might produce different numbers of the same system call type due to dynamic runtime environments.

Next, we perform outlier detection within each cluster to identify abnormal execution units. We observe that the abnormal behaviors of the affected execution units may manifest in both system call frequencies or execution time. For example, a loop bug may cause certain system calls to be executed more frequently whereas a synchronization bug may cause the lock acquiring system calls to take longer to complete their execution. Therefore we build a system call execution time vector (consisting of average execution time for each system call type) and a system call frequency vector (consisting of the count for each system call type) as two features for each execution unit. We use the nearest neighbor algorithm [36] to perform the outlier detection. Specifically, we compare the Euclidean distance from each execution unit to its nearest neighbor within the cluster using either the execution time vector or the frequency vector. If the nearest neighbor distance of an execution unit is larger than the mean nearest neighbor distance of the whole cluster plus two times the standard deviation, we say this execution unit is abnormal. We observe that some execution units might form a small cluster (e.g., size of the cluster < 4) by themselves. It is not meaningful to perform outlier detection within those small clusters. All the execution units in those small clusters are considered to be abnormal.

3.2.3 Bug Related Function List Generation

We now describe how we map those abnormal execution units to bug-related application functions and rank them based on an abnormality degree metric.

For each abnormal execution unit AEU_i, we extract a set of frequent closed system call episodes using the same frequent episode mining algorithm described in Section 3.1.2. Let FE_{AEU_i} denote the set of the frequent closed system call episode set produced by AEU_i. Similarly, our offline function signature profiling process produces a set of frequent closed system call episode FE_{f_i} for each function f_i. If any frequent episode in FE_{AEU_i} matches with any frequent episode in FE_{f_i}, we infer that AEU_i must execute part of the function f_i and output f_i as one candidate function.

For each candidate function f_i, we compute a total count of all the matching frequent episodes for f_i and its corresponding abnormal execution unit AEU_i, respectively. If the total frequent episode count of AEU_i is greater than or equal to that of f_i, we infer that AEU_i must have run some parts of

System name	Num of bugs studied	% of bugs generating system calls	% of slowdown bugs	% of hang bugs
Hadoop	40	83%	10%	90%
HDFS	32	100%	38%	62%
Cassandra	47	85%	28%	72%
Tomcat	22	95%	18%	82%
Apache	17	100%	41%	59%
Lighttpd	10	90%	30%	70%
MySQL	60	90%	38%	62%

Table 1. Statistics on the real performance bugs we examined.

f_i incorrectly (e.g., an incorrect loop) when the bug is triggered. We then output f_i as one identified function.

We then calculate a rank score for each identified function using a maximum percentage increase metric (i.e., the largest count increase percentage among all the matched frequent episodes between AEU_i and f_i) to quantify the abnormality degree of different functions during the buggy run. We sort all the identified functions using increasing rank scores. We break the tie between two functions with the same rank score using the percentage of matching frequent episodes between the function and its corresponding AEU because a higher matching percentage means we are more certain the function was executed by the corresponding AEU.

In addition, PerfScope can also infer possible call paths from the system call trace collected during buggy run. For each identified function, we examine the earliest and latest system call timestamps in all matching frequent episodes between the function and its corresponding AEU. Using those timestamps, we can infer the time when each function was active during the buggy run and use that information to create a time-based ordering among the identified functions. This information can be helpful for developers to further understand how the bug is triggered and propagates in the program. The developer can also refer to the function call graph that can be extracted via static code analysis to improve the precision of our function call path inference, which however is beyond the scope of this paper.

4. Experimental Evaluation

We tested PerfScope using seven popular open source server systems: 1) Hadoop [9]: a map-reduce framework; 2) HDFS [9]: a distributed file system; 3) Cassandra [8]: a distributed relational database system; 4) Tomcat [10]: a multi-threaded application server; 5) Apache web server [7]: a multi-threaded web server; 6) Lighttpd [24]: a light-weight web server; and 7) MySQL [27]: a relational database system. In this section, we first describe our static bug analysis result and the bug samples we could reproduce and use in our experiments. We then present our experiment setup followed by the bug inference results. We then discuss our sensitivity study results followed by the overhead evaluation.

4.1 Real Performance Bug Samples

We searched the bug repositories of each system we tested using performance terms (e.g., hangs, slowdown, etc.) and developer provided performance tags when available. We only examined those bugs which were confirmed by developers and subsequently fixed. Table 1 shows the results of our study. We manually examined each bug, along with any provided patch, to determine what kind of problem the bug caused (e.g., hang, slowdown) and whether the buggy function generated system calls. We found that over 90% bugs generate system calls. We found more hang bugs than slowdown bugs in each system.

We then try to reproduce each bug by following the instructions in the report and checking that the expected performance anomaly symptoms appeared (e.g., increased response time, 100% CPU usage, system is unresponsive). The bug reproduction is extremely time-consuming and tricky due to limited and often ambiguous information, which sometimes takes a month for us to reproduce one bug. Table 2 lists the 12 performance bugs we could reproduce given our time limit.

Although we only reproduced 12 out of the 228 bugs due to time constraints, the characteristics of the bugs we reproduced are similar to the other bugs we studied. Specifically, we found most of the bugs we examined involve some kind of loop problem, which is consistent with the observations of previous work [22, 31]. However, the bug points do not always reside in those loops. As we will show later, PerfScope can identify those bug-related functions that either include the bug points or are close to the functions that include the bug points. We found that several system hangs were the result of an application component waiting for a response which will never arrive or executing a blocking command without an appropriate timeout value. These types of problems can be difficult to track down. We also found the fix to these types of hang bugs was typically simple (e.g., adding timeout value). Lastly, we found I/O (e.g., file write) and locking operations were the most common system call generating operations. Based on our study, we believe the majority of the system call generating bugs we examined can be correctly handled by PerfScope.

4.2 Experiment Setup

The Hadoop, HDFS, Tomcat, and Cassandra systems were tested on a virtualized cloud test bed where each host is equipped with a quad-core Xeon 2.53Ghz CPU along with 8GB memory and runs 64bit CentOS 5.3 with KVM 0.12.1.2. The Apache, Lighttpd, and MySQL systems were tested on the virtual computing lab (VCL) [5], a production cloud infrastructure where each host has a dual-core Xeon 3.0GHz CPU and 4GB memory, and runs 64bit CentOS 5.2 with Xen 3.0.3. In both cases, each system call trace was

Bug name (system version)	**Server description (LOC)**	**Num. of error msg**	**Num. of threads**	**Num. of system calls (million)**	**Log size (MB)**	**Bug description**	**Bug symptom**
Hadoop bug (v0.23.1)	Data processing framework (1.3M Java)	0	729	1.5	222	Endless wait for an atomic variable to be set. (#MAPREDUCE-3738)	hang
HDFS bug (v2.0.0-alpha)	Data processing framework (1.3M Java)	1	340	1.4	227	Continuously read until timeout on a socket. (#HDFS-3318)	hang
Cassandra bug (v1.2.0-beta)	Database (168K Java)	0	95	1.1	241	Incorrect return value handling causes Cassandra to enter an infinite loop. (#5064)	hang
Tomcat-1 bug (v7.0.28)	Application server (287K Java)	0	20	0.2	18	Tomcat tries to upgrade a read lock to a write lock, which causes the server to hang. (#53450)	hang
Tomcat-2 bug (v7.0.27)	Application server (284K Java)	1	682	0.9	68	A shared counter value is not updated correctly causing all the threads to keep checking the value endlessly.(#53173)	hang
Tomcat-3 bug (v6.0.13)	Application server (228K Java)	0	26	0.4	28	A filter chain item is not set properly causing an infinite loop. (#42753)	hang
Apache-1 bug (v2.0.55)	Web server (161K C/C++)	0	19	3.7	142	Incorrect flag causes infinite loop. (#37680)	hang
Apache-2 bug (v2.2.4)	Web server (194K C/C++)	0	7	1	140	Incorrect flag causes proxy SSL connections to be constantly created and destroyed, slowing down performance. (#43238)	slowdown
Lighttpd-1 bug (v1.4.15)	Web server (38K C/C++)	0	10	5.3	160	Incorrect *errno* handling causes an infinite loop. (#1212)	hang
Lighttpd-2 bug (v1.4.23)	Web server (37K C/C++)	0	18	4.3	532	Large chunks of response data are repeatedly read and discarded while processing header information. (#1999)	slowdown
MySQL-1 bug (v5.5.5-m3)	Database server (914K C/C++)	0	551	1.9	138	Two threads try to execute the `INSERT DELAYED` statement but one of them has a locked table, causing two threads to become deadlocked. (#54332)	hang
MySQL-2 bug (v5.6.5-m8)	Database server (1.3M C/C++)	0	9	1.9	39	Flushing to disk abnormally frequently after truncating a large table. (#65615)	slowdown

Table 2. Reproduced real performance bugs that are used in our experiments.

collected in a virtual machine using LTTng 2.0.1 running 32-bit Ubuntu 12.04 kernel version 3.2.0.

We use PIN version 2.12 [26] to generate the function signature profiles for Apache, MySQL, and Lighttpd written in C/C++. For Hadoop, HDFS, Tomcat, and Cassandra, we instrumented the openJDK6 JVM [32] to perform the function signature profiling. The function signature extraction were done under the following workload conditions: 1) Hadoop: we use the Pi calculation application with 16 map and 16 reduce tasks; 2) HDFS: we transferred a 200MB file using hftp; 3) Cassandra: we use a simple workload which creates a table and inserts various entries into the table; 4) Tomcat: we randomly request different example servlets and JSPs included with Tomcat following a workload intensity observed in a NASA web server trace [2]; 5) Apache: we use httperf to request various pages from the Apache server; 6) Lighttpd: we use the same workload as Apache and configure Lighttpd to run in multi-process mode; 7) MySQL: we use a open source MySQL benchmark tool called Sysbench [4] and did the *oltp* test in *complex* mode. Note that the workloads we used for profiling are different from the bug-triggering workloads. We also intentionally vary the resource allocation setting for the profiling environment to test the robustness of our function signatures. We found that the workload and resource allocation changes did not significantly affect the frequent episodes generated. For example, varying the work-

Bug name, number of functions, (percent of identified functions)	**Top ranked bug-related functions (Rank)**	**Examples of detected abnormal application-kernel interactions**
Hadoop,85495 (0.52%)	getState (4)	Continuously produces {`sys_gettimeofday,sys_stat64, sys_stat64, sys_gettimeofday`} for acquiring locks to check an atomic variable.
HDFS,81306 (1.5%)	Reader.performIO (3), SocketIOWithTimeout.doIO (5)	Continuously produces {`sys_gettimeofday,sys_read,sys_read, sys_gettimeofday`} for attempting to perform I/O.
Cassandra,21765 (0.6%)	maybeSwitchMemtable (4), SSTableSliceIterator.hasNext (5)	Continuously produces {`sys_gettimeofday,sys_futex,sys_futex, sys_gettimeofday`} for synchronization from a buggy infinite loop.
Tomcat-1,32789 (1.5%)	addLifecycleListener (2)	Continuously produces {`sys_stat64,sys_gettimeofday, sys_gettimeofday,sys_stat64`} for upgrading from a read lock to a write lock.
Tomcat-2,32571 (0.8%)	LimitLatch.countDown (11)	Continuously produces {`sys_gettimeofday,sys_stat64, sys_stat64,sys_gettimeofday`} while checking an atomic counter value.
Tomcat-3,26471 (0.4%)	Poller.run (3)	Continuously produces {`sys_gettimeofday,sys_read,sys_read, sys_gettimeofday`} while reading and attempting to process an event in a buggy infinite loop.
Apache-1,17721 (0.7%)	apr_allocator_mutex_get (14)	Continuously produces {`sys_gettimeofday,sys_write`} for logging in a buggy infinite loop.
Apache-2,18761 (2%)	ssl_hook_pre_connection (9)	More {`sys_stat64,sys_write,sys_gettimeofday,sys_stat64`} are generated for logging in the buggy ssl_hook_pre_connection function.
Lighttpd-1,954 (0.1%)	fdevent_poll (1)	Continuously produces {`sys_poll,sys_time,sys_time,sys_poll`} while waiting for events to be processed in a buggy infinite loop.
Lighttpd-2,3577 (0.08%)	connection_handle_read_state (3)	Continuously produces {`sys_ioctl,sys_read,sys_read,sys_ioctl`} while repeatedly reading large chunks.
MySQL-1,25360 (0.6%)	ha_lock_engine (14)	Continuously produces {`sys_stat64,sys_llseek,sys_read,sys_stat64`} when checking for lock availability over the network.
MySQL-2,34850 (0.03%)	buf_flush_list (4)	More {`sys_futex,sys_gettimeofday,sys_gettimeofday,sys_futex`} are generated for synchronization while flushing lists to disk.

Table 3. Online bug inference result summary.

load intensity from 50 static page requests per second to 125 static page requests per second while building an Apache profile only caused 4% of the generated frequent episodes to change.

During each buggy run, PerfScope is triggered when the performance anomaly is detected by an external detector that checks for reported performance anomaly symptoms such as zero progress score or abnormal response time. Once triggered, PerfScope was run using the recent 1.5 minutes of system call traces for analysis. We can retrieve longer system call trace from an NFS server if needed. However, our experiments show that 1.5 minutes system call trace is sufficient for all the bugs we tested.

4.3 Bug Inference Result Summary

Table 3 shows a summary of our bug inference results for *all* tested bugs. We can see most server systems we tested consist of tens of thousands of functions. As shown in Table 2, most performance bugs do not produce any error message and some buggy runs also involve a large number of threads, it is a daunting task for the developer to figure out why the performance bugs occurred. Profiling by itself can help reduce the search scope as we only profile the functions called the system configuration described in the bug report, which was 0.7% to 10% of the total functions. However, this still equates to thousands of functions to search. The results show that PerfScope can identify a short list of bug-related functions for all tested bugs, which can greatly reduce the search scope to only 0.03-2.3% of total functions. We believe that this search scope reduction can significant expedite the debugging process.

To further validate the effectiveness of our bug inference results, we manually examined the source code of each tested server system to determine whether each of our short lists indeed includes the bug-related functions which can help the developer to locate the bug. We list the top ranked bug-related functions identified by PerfScope in Table 3. We found that the short list of identified functions indeed cover the bug-related functions in all the bugs we tested. We rank the bug-related function within the top five candidates in 8 out of the 12 bugs. In the worst case, we rank the bug-related function at 14. When multiple functions contribute to the problem (e.g., HDFS bug, Cassandra bug), PerfScope can identify them as well. Table 3 also shows one example of abnormal frequent system call episodes detected by PerfScope for each bug.

As shown in our results, PerfScope can handle both performance degradation bugs and hang bugs. PerfScope works best for detecting those bugs involving a buggy loop (e.g., an infinite loop or a loop iterates more times than usual). Previous work [22] along with our own study have shown most performance bugs do exhibit these characteristics.

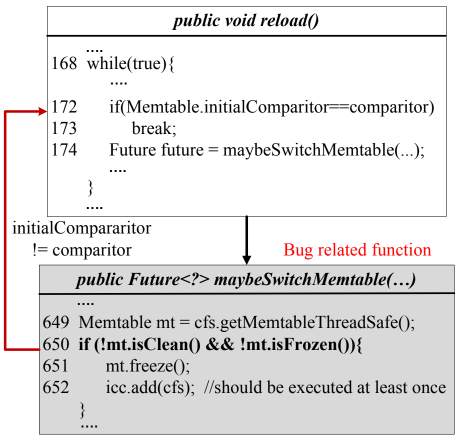
We also investigated the cases where PerfScope fails to rank the bug-related function within top 5 and identified two main reasons causing this. First, we found there are some frequent system call episodes which are common to several functions. When the abnormal frequent system call episodes also appear in a function called less frequently than the bug-related function during profiling, this can cause us to rank that function higher than the bug-related function. However, we found most of those mistakenly high-ranking functions (e.g., initialization functions) can be easily filtered out by the developer. Second, we sometimes give high rank to common utility functions (e.g., custom fast comparison function) used throughout the application. When those utility functions are called several times in the buggy function, this can cause the utility function to have a higher rank score than the buggy function. We are currently exploring ways of filtering out those common utility functions.

4.4 Bug Inference Case Studies

To further understand how the output of our bug inference tool can be used for debugging, we now discuss three of our bug inference results in detail. Due to the space limitation, we cannot describe the inference results for all the bugs in detail. We hope that these three case studies can show the usefulness of PerfScope for helping the developer diagnose the production-run performance bugs.

Cassandra bug: This Cassandra performance bug is related to a `memtable` mishandling problem. Figure 3 shows a subgraph of the bug's call graph. The root cause of this bug is the `if` statement at line 650 of the `maybeSwitchMemtable` function. The code block (lines 651 and 652) within the `if` statement should be executed at least once for every memtable object in order to add the memtable object into the memtable list. However, if a memtable is clean, the check at line 650 will be false, causing the memtable not to be added to the list. As the memtable has not been processed, the check at line 172 in the `reload` function never becomes `true`, causing the program never breaks out of the `while` loop. No error message is produced for this bug.

PerfScope analysis produced 1229 execution units, which were clustered into 49 clusters. Our outlier detection results identified 19 time based outlier units and 33 frequency based outlier units. PerfScope successfully identifies the root cause function `maybeSwitchMemtable` function as the fourth top ranked function. We found this function produced a set of


```
public void reload()
    ....
168 while(true){
        ....
172     if(Memtable.initialComparitor==comparitor)
173         break;
174     Future future = maybeSwitchMemtable(...);
        ....
    }
    ....
```

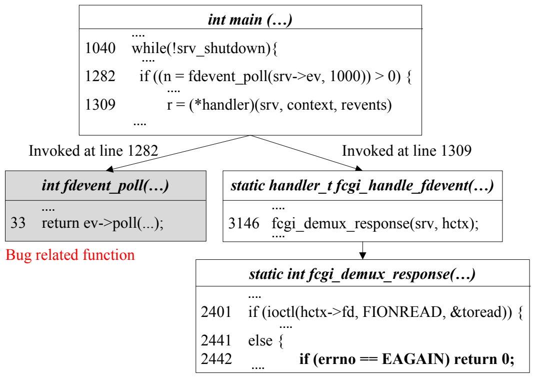
```
public Future<?> maybeSwitchMemtable(...)
    ....
649 Memtable mt = cfs.getMemtableThreadSafe();
650 if (!mt.isClean() && !mt.isFrozen()){
651     mt.freeze();
652     icc.add(cfs);  //should be executed at least once
    }
    ....
```

Figure 3. A subset of the call graph for the Cassandra bug. The bug causes a clean memtable not to be added to the list of memtables, resulting in an infinite loop in the `reload` function. The bug point is highlighted in bold. PerfScope identifies the root cause function `maybeSwitchMemtable` as the 4th top ranked bug-related function, which is responsible for adding all the memtables.

30 frequent episodes during profiling. We also found that this function is only called in a limited number of locations. Thus, identifying this function can help the developer quickly localize the problem. The fix to this bug is to force the code block within the `if` statement at line 650 of the `maybeSwitchMemtable` function to be executed at least once.

HDFS bug: The second bug we discuss in detail is a HDFS bug mentioned in the introduction, which is caused by an overflow of an internal variable. Figure 1 shows the subgraph of the bug's call graph. When copying a file using the `distcp` command, the `copyBytes` function is called to read 4096 bytes from the source stream and write those bytes to the destination stream. This causes the `read`, `doIO`, and `performIO` functions to be called in turn. The `performIO` function then makes a low level socket call and returns the number of bytes read back. This process is repeated until -1 is returned, signaling the end of the input stream has been reached. When this command is called with a file larger than 2GB in size, however, an internal variable (`int`) representing the length of the source stream overflows and the end of stream signal (i.e., -1) is never sent. This causes `while` loop at line 76 of the `copyBytes` function to never end until the operation times out several minutes later. The only error message generated throughout this process is a message saying the operation timed out. Additionally, as HDFS uses many different interfaces and abstract classes, it is difficult to identify which implementation subclass causes the problem.

PerfScope analysis produced 3817 execution units, which were clustered into 174 clusters. Our outlier detection results identified 92 time based outlier units and 113 frequency based outlier units. The third top ranked function PerfScope



```
int main (...)
....
1040   while(!srv_shutdown){
         ....
1282     if ((n = fdevent_poll(srv->ev, 1000)) > 0) {
           ....
1309         r = (*handler)(srv, context, revents)
....
```

```
int fdevent_poll(...)
....
33   return ev->poll(...);
```

```
static handler_t fcgi_handle_fdevent(...)
....
3146  fcgi_demux_response(srv, hctx);
....
```

```
static int fcgi_demux_response(...)
....
2401  if (ioctl(hctx->fd, FIONREAD, &toread)) {
      ....
2441  else {
2442  ....     if (errno == EAGAIN) return 0;
```

Figure 4. A subset of the call graph for the Lighttpd bug. The errno is handled incorrectly, causing the main function to process the same event again and again. The bug point is highlighted in bold. PerfScope identified the `fdevent_poll` as the bug related function which is responsible for controlling the event processing to continue as long as the number of events (n) is positive.

identified is the `SocketInputStream$Reader.performIO` function. We found this function produced a set of 30 frequent episodes during profiling. As shown in Figure 1, this is one of the functions directly responsible for reading data from the input stream and is directly related to the bug. By telling developers not only the function name, but also telling the developer that the bug is related to the `SocketInputStream$Reader` class, debugging time for this problem can be greatly reduced. Additionally, identifying that the problem is caused by an infinite loop could help developers immediately rule out other misleading causes, such as a network issue. The fix in this case is to ensure streams larger than 2GB is processed in chunks of appropriate size.

Lighttpd-1 bug: The third bug we want to discuss is a Lighttpd bug shown by Figure 4. The performance anomaly is caused by the `if` statement at line 1282 of the `main` function always stays true, causing the code block within the `if` statement executes forever. During this buggy run, the `fdevents` function is first invoked at the `if` statement at line 1282 of the `main` function. Next, the `fcgi_handle_dfevent` function is invoked at line 1309, which then calls the `fcgi_demux_response` function. The root cause of this problem is an improper errno handling (i.e., setting `errno` to EAGAIN) at line 2442 of the `fcgi_demux_response` function causes this function to return 0. When the function `fcgi_handle_fdevent` receives a 0 return value from `fcgi_demux_response`, it does not clean up this event from the event queue. This in turn causes `fdevent_poll` to return a value > 0, which causes the system to process this same event again and again. No error message is generated during this process.

PerfScope analysis produced 9380 execution units, which were clustered into 7 clusters. Our outlier detection results identified 33 time based outlier units and 31 frequency based outlier units. PerfScope identifies the `fdevent_poll` as the top ranked bug-related function, which controls the event processing code at line 1282 of the `main` function. We found this function produced a set of 900 frequent episodes during profiling. The large number of frequent episodes produced by this function are a result of the variety of tasks it is involved with. By starting debugging at this function, developers could quickly see that the same event was being processed over and over. This information is especially useful for Lighttpd, which has a complex main function responsible for performing many different tasks. Furthermore, the function we identify resides in the same `if` statement code block with the root cause function. This means developers would have significantly fewer areas to explore when trying to debug the problem. The fix for this issue would be to ensure `errno` is handled appropriately, ensuring the event is processed.

4.5 Sensitivity Study

We conducted sensitivity study experiments to evaluate how different parameter settings affect our bug inference results. Although we omit the results due to space limitations, we found that the parameter values do not significantly affect our bug inference results. We also examined the ranks of the true bug related functions, which also show little changes.

4.6 PerfScope Overhead

Bug name	Online bug inference	Offline function signature extraction
Hadoop	15.99 min	9.5 min
HDFS	29.94 min	38 min
Cassandra	3.34 min	12.9 min
Tomcat-1	5.14 min	42.7 min
Tomcat-2	12.34 min	2.9 hour
Tomcat-3	2.32 min	48.7 min
Apache-1	2.93 min	15.5 min
Apache-2	4.3 min	14.4 min
Lighttpd-1	9.91 min	1.02 hour
Lighttpd-2	5.87 min	1.15 hour
MySQL-1	12.8 min	44.5 min
MySQL-2	5.22 min	44.5 min

Table 4. Online and offline computation time for PerfScope.

We now evaluate the overhead of PerfScope to different server systems. For Hadoop/HDFS we ran the Pi sample application. For Cassandra, we ran a database insertion workload. We used httperf to send a number of requests to Tomcat, Apache, and Lighttpd. We used a request rate of 50 requests per second for Tomcat, 100 requests per second for Apache and Lighttpd. For MySQL, we run a constant workload of 20 select requests per second. Those request rates are set based on the maximum processing capacity of our host. We ran all overhead experiments 5 times, reporting the mean. We impart an average of 1.8% runtime overhead to the server system. Specifically, we impart 2.97% to Hadoop/HDFS, 3.33% to Cassandra, 1.4% to Tomcat, 0.8% to Apache, 0.3% to Lighttpd, and 2.2% to MySQL. We found Perfscope imparts between 2-3% CPU load and has a small memory footprint (about 256KB). We also found the storage overhead is reasonable, varying between about 18MB to 530MB.

We also ran an experiment to determine how PerfScope overhead scales under different workload intensity by comparing the overhead of tracing at 50, 75, 100, and 125 requests per second to an Apache web server. We found the log size scales linearly with request rate while the overhead of tracing remained constant. We used least squares linear regression to project log sizes for larger request rates. We found that the one-minute tracing log size will be around 450MB given 900 requests per second. Note that this log size can significantly reduced using standard compression techniques, which will be discussed further in Section 5. The low overhead we observed is consistent with the overhead results of previous projects [25]. Hence, we believe that PerfScope is light-weight and practical for online bug inference in production cloud computing infrastructures.

We now present the online and offline computation time of PerfScope shown by Table 4. The results show that PerfScope can complete the online bug inference within tens of minutes. The majority of the online inference time was spent on extracting the frequent episodes for different anomalous execution units in order to identify the bug-related functions. This time can be further shortened if we use multiple distributed hosts to perform the frequent episode mining. Our distributed offline function signature extraction time ranged from 9.5 minutes to about 2.9 hours as shown in Table 4. These times were obtained by running our distributed signature extraction algorithm on a 7 node cluster and could be further improved on larger clusters. Additionally, as the function signature profiling is performed offline outside the production environment, its overhead will not be a concern for using PerfScope to perform online bug inference.

5. Limitation Discussion

Although each individual system call log is relatively small and scales well with workload intensity, the overhead of storing traces for many different applications may be significant. In those cases, system call filtering can be used to reduce the trace log size. Assuming prior application knowledge, it is possible to modify PerfScope to only trace certain system call types or to only trace system calls generated by specific threads. We can also apply log compression to further reduce the trace log size. For example, after applying the standard gzip compression, the HDFS log size goes down from 227MB to 19MB.

In practice it may be necessary to move the collected traces from a storage node to a different node for processing, where the cost is dependent on the network bandwidth. However, as the size of each collected trace is relatively small, ranging from about 40MB to about 530MB, we believe these files can be quickly moved from host to host. Additionally, optimizations and log compression can further reduce the cost of moving these logs.

If two functions have identical frequent episode set, PerfScope will not be able to distinguish between them. However, in our experiments, we found this case never happened. Each function always generated unique frequent episodes. For example, as mentioned in the evaluation, the `SocketInputStream$Reader.performIO` produced 29 unique frequent episodes.

Our current evaluation is limited to single node server setup. For performance bugs in distributed systems, we believe that PerfScope can help with those bugs by identifying the affected functions on each affected component. In addition, performance bugs in distributed systems can also occur as the result of poor interactions between different system components (e.g., incorrect timeout value). Previous work [28] has shown that those bugs also exhibit anomalous application to kernel interactions on the faulty components in the form of either I/O or locking. How to apply PerfScope to distributed system performance bugs is part of our future work.

6. Conclusion

In this paper, we have presented PerfScope, an online performance bug inference tool for production cloud computing infrastructures. PerfScope can identify bug-related functions for both interpreted and compiled programs using light-weight kernel-level system call tracing and online system call trace analysis. We have implemented PerfScope and conducted experiments using real performance bugs in seven popular open source server systems. The results show that PerfScope can successfully identify true bug-related functions out of tens of thousands of application functions for all the tested performance bugs. PerfScope is light-weight, which only imposes 1.8% average runtime overhead to the tested server systems.

Acknowledgment

We thank the anonymous reviewers and our shepherd Haryadi Gunawi for their valuable comments. This work was sponsored in part by NSF CNS0915567 grant, NSF CNS0915861 grant, NSF CAREER Award CNS1149445, U.S. Army Research Office (ARO) under grant W911NF-10-1-0273, IBM Faculty Awards and Google Research Awards. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of NSF, ARO, or U.S. Government.

References

[1] Exuberant Ctags. `http://ctags.sourceforge.net/`.

[2] The IRCache Project. http://www.ircache.net/.

[3] Linux process trace (ptrace). http://linux.die.net/man/2/ptrace.

[4] *SysBench: a system performance benchmark*. http://sysbench.sourceforge.net/.

[5] NCSU Virtual Computing Lab. http://vcl.ncsu.edu/.

[6] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB*, 1994.

[7] Apache. Apache HTTP Server Project. http://httpd.apache.org/.

[8] Apache Cassandra. Apache Cassandra. http://cassandra.apache.org/.

[9] Apache Hadoop. Apache Hadoop. http://hadoop.apache.org/.

[10] Apache Tomcat. Apache Tomcat. http://tomcat.apache.org/.

[11] J. Arulraj, P. Chang, G. Jin, and S. Lu. Production-run software failure diagnosis via hardware performance counters. In *ASPLOS*, 2013.

[12] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.

[13] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, 2004.

[14] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN*, 2002.

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

[16] D. Dean, H. Nguyen, and X. Gu. UBL: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *ICAC*, 2012.

[17] D. J. Dean, H. Nguyen, P. Wang, and X. Gu. Perfcompass: Toward runtime performance anomaly fault localization for infrastructure-as-a-service clouds.

[18] M. Desnoyers and M. R. Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *Linux Symposium*, 2006.

[19] U. Erlingsson, M. Peinado, S. Peter, and M. Budiu. Fay: Extensible distributed tracing from kernels to clusters. In *SOSP*, 2011.

[20] P. Fournier and M. R. Dagenais. Analyzing blocking to debug performance problems on multi-core systems. In *SIGOPS*, 2010.

[21] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *OOPSLA*, 2010.

[22] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *PLDI*, 2012.

[23] L. Kaufman and P. J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 2009.

[24] Lighttpd. Lighttpd. http://www.lighttpd.net/.

[25] LTTng Success. LTTng Success. https://lttng.org/success-stories.

[26] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[27] MySQL. MySQL. http://www.mysql.com/.

[28] K. Nagaraj, C. E. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *NSDI*, 2012.

[29] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *OSDI*, 2004.

[30] H. Nguyen, Z. Shen, Y. Tan, and X. Gu. Fchain: Toward black-box online fault localization for cloud systems. In *ICDCS*, 2013.

[31] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: detecting performance problems via similar memory-access patterns. In *ICSE*, 2013.

[32] OpenJDK6. OpenJDK6. http://openjdk.java.net/projects/jdk6/.

[33] D. Patnaik, S. Laxman, B. Chandramouli, and N. Ramakrishnan. Efficient episode mining of dynamic event streams. In *ICDM*, 2012.

[34] perf. Performance counters for Linux. http://perf.wiki.kernel.org/index.php/Main_Page.

[35] X. Song, H. Chen, and B. Zang. Why software hangs and what can be done with it. In *DSN*, 2010.

[36] P. N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison Wesley, 2005.

[37] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *ICDCS*, 2012.

[38] A. Traeger, I. Deras, and E. Zadok. DARC: Dynamic analysis of root causes of latency distributions. In *SIGMETRICS*, 2008.

[39] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing production run failures at the user's site. In *SOSP*, 2007.

[40] M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine learning*, 2001.