ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 7, Issue 3, pp: (164-173), Month: July - September 2019, Available at: www.researchpublish.com

PerfTrace: Performance Anomaly Fault Localization & Performance Bug Diagnosis for Infrastructure-as-a-Service Clouds(IaaS)

Eashan Kaushik

Vellore Institute of Technology, Vellore, INDIA

Abstract: **-Server applications which are running inside production cloud infrastructures are prone to various performance problems like software hang, performance slowdown, etc. When those problems occur, developers often have little clue to diagnose and localise these problems. PerfTrace an anomaly Fault localization & diagnostic tool which builds on PerfCompass and Hytrace diagnostics tools and uses an amalgam of static analysis schemes and dynamic runtime analysis schemes. Both these schemes have advantages as well as limitations. PerfTrace tries to incorporate only the advantages of these schemes.**

Keywords: **IaaS, PerfTrace, Hytrace, PerfCompass, Fault Localization, Bug Diagnosis.**

I. INTRODUCTION

Infrastructure-as-a-service (IaaS) clouds have become increasingly popular by allowing its users to access computing resources in a highly cost-effective way. There are multiple tenants sharing the same physical resource as a result performance anomalies arise, these anomalies are laborious to diagnose because the user has restricted diagnostic information to localize the fault, as a result is of the main concerns for users.

This paper introduces PerfTrace an anomaly Fault localization & diagnostic tool. It is an amalgam of static analysis schemes and dynamic runtime analysis. Both these schemes have advantages as well as limitations. PerfTrace tries to incorporate only the advantages of these schemes.

PerfTrace builds on the work of – PerfCompass [19], Hytrace [20] and Pip [21]. This paper gives an overview of all these tools and take about their advantages, disadvantages and how integrating all these schemes into one idea will give remarkable results.

Summary of the State of the Art

There are a number of runtime performance anomaly fault localiztion tools avalable e.g., [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. These techniques are parochial and can only provide course grained fault localization.However some white-box and gray-box schemes, e.g, ., [15, 16, 17, 18] can provide fine-grained fault localiztion but due to expensive instrumentation these techniques are impractical.

Although techniques like [12, 13, 14] provide both course and fine grained fault localization, they need an extensive profiling phase to extract models which are not possible in case of Iaas Clouds.Debugging tools liks [18] does combine kernel and space tracing which causes a high overhead to applications. Moreover, all the existing work do not necesarlly distiguish between entern and intenal faults.

Apache httpd

In this section we talk about the challenges the user faces in diagnosis of a performance bug. For this purpose we take the example of Apache-httpd. This type of bug is one of the most common bug and arrises when the user restarts the Apache server. As a result the web server is changed from listening to two port to just one port. When the Apache server restarts httpd consumes complete 100% CPU and it wont respond to requests if at even one stage of execution it has more than one listening socket which is configured , it is then subsequently reconfigured with "Apachectl restart" or "Apachectl graceful" to have a single listening socket, which then corresponds to one

of the previously existing sockets. The problem here is that , 0_NONBLOCK option is set on the reused socket in previous initialisation and later that part of code is expected to make a blocking "accept" call on that socket e.g. /server/mpm/prefork/prefork.c:591. This call will return EWOULDBLOCK/EAGAIN where the case is not that specifically handled , it causes a loop in the MPM (e.g. at server/mpm/prefork/prefork.c:522) with full utilisation of CPU.

We can fix this by clearing the O_NONBLOCK option on the reused socket provided that there is only the one listening socket.

TABLE 1: Steps to reproduce

```
$ tar xjf httpd-2.0. 5.tar.bz2
$ cd httpd-2.0.5/
$ ./configure -prefix=/home/username/apache2-test
$ make
$ make install
$ cd home/username/apache2-test
$ vi /conf/httpd.conf
Change "Listen 80" to "Listen 10080"
$ /bin/apachectl start
Locate localhost:10080, and check if server still working.
$ vi /conf/httpd.conf
"Listen 10081" is added on a new line after the existing line "Listen 10080".
$ /bin/apachectl restart
Locate localhost:10081, and check if server still working.
$ vi conf/httpd.conf
$ vi /conf/httpd.conf
"Listen 10081"-delete.
$ /bin/apachectl restart
$top
Browse to local host:10080.
```

TABLE 2: httpd result table(Completely reproducible)

Actual Results	Expected Results	Problem Reproduced on
Normal CPU utilisation. Apache should serve the default installation page on local host: 10080.	100% CPU utilisation, shared amongst the five httpd child processes. Apache does not serve anything on any port.	Apache 2.0.54 and 2.0.55 on i686 Fedora Core 3 Linux kernel 2.6.12-1.1378_FC3 Apache 2.0.55 on Linux kernel 2.4.20, PowerPC MPC8555 CPU.

This bug reuses the socket from the previous running instance, without changing any of the socket setting. Unfortunately, the socket was actually set to not allow blocking calls in app_setup_listeners function through the appr_socket_opt_set function in previous running instance, when two ports were configured.

TABLE 3: Apache-httpd bug Patch

```
-if (app_listeners && app_listeners -> next) {
+ use_nonblock = app_listeners && app_listeners -> next;
 for (lr = app_listeners; lr; lr = lr -> next) {
       appr_status_t status;
       ....
        status = appr_socket_opt_set(lr -> sd,
                       APPR_SO_NONBLOCK,
-                                         1);
+                             use_of_nonblock);
        if (status != APPR_SUCCESS){
             apP_log_perror(APPLOG_MARK,          APPLOG_STARTUP          |
APPLOG_ERR, status, pool,
                             "app_listen_open: unable to "
-                        + "make socket non-block");
+                 + "control non-blocking status");
                  return (-1);
             }
        }
- }
```

This patch passes a variable use_nonblock to the invocation of appr_socket_opt_set inside the function app_setup_listeners. This changes allows to enable blocking calls on reused socket before actually making blocking calls**.**

Our Contributions

This paper presents PerfTrace. PerfTrace is an idea which is designed to use the strength of PerfCompass [19], Hytrace [20] and Pip [20]. It is a scheme that can be used for performance anomaly fault localization and bug diagnosis. The main challenge is to design a scheme which can incorporate the strengths of the three schemes and try to overcome their individual disadvantages. PerfTrace will perform anomaly fault localization using the studies of PerfCompass [19] and Bug diagnosis using the studies of Hytrace [20]. It also traces the behaviour of a running application using Pip [21].

This paper discusses in brief Hytrace [20], PerfCompass [19] and Pip [20] schemes and also presents their evaluations methodology and give limitations of each.

PerfTrace will be light weight and generic application like PerfCompass [19] and won't require any additional advance application profiling. It will be applicable to any multi-processed or multi-threaded applications. It will only use kernel-level system call trace only as a result will be light-weight and generic in nature. Particularly, PerfTrace like PerfCompass [19] will perform diagnosis, and try to find if performance anomaly is caused by an internal or external fault.

PerfTrace will be a static anti-pattern detection scheme like Hytrace [20] which can also detect dynamic abnormal behaviour but provide high coverage without the expense of precision unlike Hytrace [20]. Combining such schemes (static anti-pattern detector and a dynamic abnormal behaviour detector) will retain the high coverage and lost coverage can be regained, as most alarms would not be reported by both schemes that conduct diagnosis but from different perspective. To overcome this PerfTrace will implement within its functionality a generic rule checker which will detect function that bear patterns which might be vulnerable to performance anomalies. When performance problems such as hang and slowdown are observed by automated monitors, by the use of run-time analysis we identify a ranked list of functions that produce abnormal behaviour and as a result functions that appear suspicious from both static and dynamic

analysis are reported. It will be platform independent schema.

PerfTrace using Pip [21] traces the behaviour of a running application, checks its behaviour against programmer expectations, and then it displays the resulting behaviour (valid or invalid) in a GUI using several different visualization techniques.

II. PERFCOMPASS

A performance when occurs anomaly, the PerfCompass [19] analysis module is then dynamically triggered to perform an online fault localisation within the faulty VMs. We can identify the faulty VMs using the existing online black-box fault Localisation tools [22,23, 24]. However, when we ought to differentiate between the external and internal faults, we cannot then treat the entire application VM as a single black-box. PerfCompass will first extract the different groups of the closely related system calls, which are called execution units, from the continuous raw system call traces. We can then first easily group the system caperlls based on process/thread ID. Though, we observe that some server systems (e.g., Apache web server) also use a pre-allocated thread pool and often do reuse the same thread to perform different tasks. So when we further split the per-thread execution units based on inter-system-call time gap (e.g., 99th percentile value) to then mitigate in accurate system call grouping caused by the thread reuse.

After extracting different the execution units, we then perform change detection over the system call execution time or the system call frequency moving average values to detect if the execution unit is affected by the fault. We derive then a fault onset time for each of the affected execution unit to quantify how quickly the fault impacts the affected execution unit.

Fault Onset Time Identification

To distinguish between the external and internal fault, we first will extract the two pieces of the important information from raw system call traces of each of the execution unit: 1) which of the execution units are affected by the fault? and 2) when does the fault impact the first start in each of the execution unit? To detect if an execution unit is affected by the fault, we first have to analyse the system calls execution time. The observation is that if the fault causes the application performance to slowdown, the system call execution time should also be increased.

Therefore, when we compute the per system call execution time and detect if the fault affects the execution unit by identifying any outlier system call execution times. We then perform outlier detection using standard criteria (i.e., > mean + 2×standard deviation).

We also observe that if the different system calls often have varied execution time based on their functions. Therefore, it is necessary to distinguish the different kinds of system calls (e.g., sys read, sys write) and then compute the per system call execution time for each kind of system call separately. If any of the system call type is identified as an outlier, we then infer that this execution unit has been affected by the fault.

Some of the performance anomalies do not manifest as changes in the system call execution time. For example, if the performance anomaly is caused by one incorrect loop bug, we might then observe the system calls inside the loop iterate more number of times when the bug is triggered, than during the normal execution. So, then we maintain a frequency count for each of the system call type. An execution unit is said to be affected by the fault if we detect anomalous changes in either the system call execution time or the frequency for any type of system call.

Fault Localization Schemes

Our fault localisation schemes are then based on extracting and analysing the two fault features. We first will extract the fault impact feature to infer if the fault has a global or local impact. We then define the fault impact factor metric to calculate the percentage of threads that are then affected by the fault directly. If the thread consists of multiple execution units, then we only consider the fault onset time of the first execution unit that is affected by the fault since we only want to identify when the fault first affected each thread. We then discuss how to set fault onset time threshold later in the section below. If the fault onset time of the affected thread is smaller than any pre-defined threshold (e.g.,1second), we can say that this thread is affected by the fault directly. The fault impact factor if is close to 100%, we can infer that the fault would be an external fault; If it’s less than 100%, fault impact factor, we can them infer that the fault would be an internal fault.

However, if the fault impact factor has borderline value (e.g., 90%), then we extract a fault onset time dispersion feature to be able to identify whether the fault affects different threads at the same time or at a different time. Since the external fault affects all the running threads uniformly, the affected threads are then likely to show the fault impact at a similar time after the fault is triggered. We then use the standard deviation of the fault onset time among all the threads that are affected to quantify the fault on set time dispersion. If it is small, that is the fault onset dispersion, then we infer that this fault is external. However, if it’s an internal fault it is likely to directly affect a subset of threads executing the buggy code and then indirectly affect other threads that communicate with the directly affected threads. Therefore, if we observe a large fault onset time dispersion, we infer the fault is an internal one.

Our main observation is that if performance anomaly is caused by an external fault, then we often see that the fault directly affects all the running execution units simultaneously. However, an internal fault will only directly affect a subset of execution units at the beginning although the impact might propagate to other execution units through communications or shared resources at a later time. Therefore, we then use the impact factor to quantify the scope of direct impact from the fault to the different execution units and also the fault onset time dispersion to quantify the onset time difference among the different affected execution units. We can then infer if the performance anomaly is caused by an external or internal fault by checking if the fault has a global or local direct impact and also if the fault onset time durations of different execution units are similar.

Experimental Results

TABLE 2.1: Apache

Fault	**Fault Impact Factor**	**Fault on set time dispersion**	**Correct diagnosis**
CPU cap problem (external)	100±0%	7±1ms	**Yes**
Packet loss problem (external)	100±0%	4±1ms	**Yes**
Flag setting bug (internal)	50±0.5%	374±63ms	**Yes**

TABLE 2.2: SQL

Fault	**Fault Impact Factor**	**Fault on set time dispersion**	**Correct diagnosis**
I/O interference problem (external)	100±0%	15±7 ms	**Yes**
CPU cap problem (external)	94±2%	17.77±4 ms	**Yes**
Data flushing bug (internal)	62±3%	721±4 ms	
Dead lock bug (internal)	40±0.5%	38±3 ms	**Yes**

TABLE 2.3: Hadoop

Fault	**Fault Impact Factor**	**Fault on set time dispersion**	**Correct diagnosis**
I/O interference problem (external)	98±1%	16±3 ms	**Yes**
CPU cap problem (external)	98±0%	39±5 ms	**Yes**
Endless read bug (internal)	81±0%	23±6 ms	
Thread shutdown bug (internal)	85±0.5%	110±20 ms	**Yes**

TABLE 2.1, TABLE 2.2 and TABLE 2.3 [19] provide a summary of our fault localisation results for the Apache, MySQL and Hadoop under the various external and internal faults. The results show us that PerfCompass [19] accurately diagnoses all of external faults as external and all of internal faults as internal. All of the tested external faults have impact factors close to 100%, that clearly indicates each of them as external. Similarly, most of the internal faults for these systems have an impact factor significantly less than 100%, which then clearly indicates each of them as internal. We have also observed that most internal faults have significantly larger fault onset time dispersion values than the external faults of the same application.

PerfCompass Overhead

We will now evaluate the overhead of the PerfCompass [19] system. Figure 4 here shows us the runtime overhead imposed by PerfCompass on each of the tested systems. For Apache, MySQL, and Squid we have used httperf to send fixed number of requests, and recorded the average response time both with and without PerfCompass. We have used a request rate of 100 requests per second for Apache and Squid, and a request rate of 20 requests per second for MySQL1. For Cassandra, we then ran a simple database insertion workload, recording the average time of processing. For Hadoop we ran the Pi sample job, recording the average time of processing. We then ran all the tests 5 times, reporting the mean and standard deviation. We observed that PerfCompass imparts 1.03% runtime overhead on average. We also then measured their source consumption of PerfCompass. We found that the PerfCompass imparts between 2-3% CPU load and has a small memory footprint (about 256KB). We also believe that the PerfCompass is light-weight, which therefore makes it practical for online performance anomaly fault localisation in production IaaS clouds.

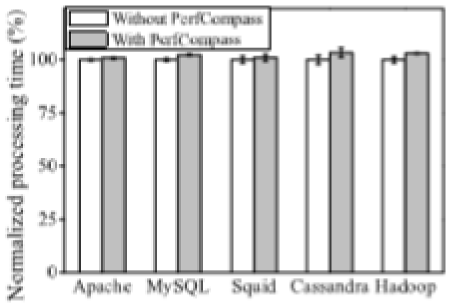
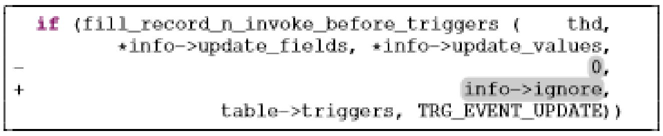


Fig. 1. The overhead imposed by PerfCompass on different server systems.

III. HYTRACE

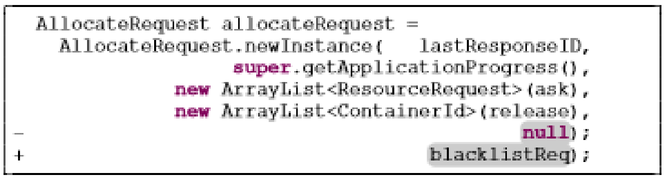
Hytrace Static Analysis

Our module of static analysis focuses on detecting the potential faulty functions that are prone to performance problems. Its design has two parts. First, designing the target for static analysis — identifying a few static code patterns that are vulnerable to performance problems, which we will refer to as rules. Second, designing the static analysis algorithm — designing how to analyse the program and discovering the code regions that matching the Rule Design Principles. Our rule designing follows two principles. First, different from a lot of stand-alone static checkers, our design favours the generality over precision. We should then look for code patterns that might be-indicators of the performance problems, not the patterns that are guaranteed to cause performance problems. This principle will help us avoid missing the true buggy functions. Since the runtime inference component of the Hytrace [20] can effectively filter out the many falsely identified functions that are detected by the static analysis, the final precision of the Hytrace [20] will be much better than the precisions of these static rules. Second, like in all the static checkers, we should find the statically checkable rules. That is, if a code region matches a rule or not should be decidable without any runtime information. For example, checking if a function call uses a constant value as the parameter is statically checkable. However, if a variable can take on a particular value during the program, execution often cannot be checked statically. We have randomly sampled 20 out of 133 performance bugs. Here we have derived a set of rules that will meet our design principles empirically based on our experience of studying those 20 real-world performance bugs in the server applications. For cross validation purpose , we use another disjoint set of 20 bugs to perform the same rule extraction process. (the details about all the 133 bugs are available online). We found that we will extract the same set of generic performance bug detection rules. The 40 sample bugs are our rule generation training set. The 133 bugs form Hytrace [20] testing set and are then used for our experimental evaluation. It should be noted that, Hytrace [20] can be extended easily with other rules that follow our design principles and are integrated with any static analysis tools that can help identify a set of candidate performance problem-prone functions. We will now specify the rules used by the Hytrace [20] static analysis component.



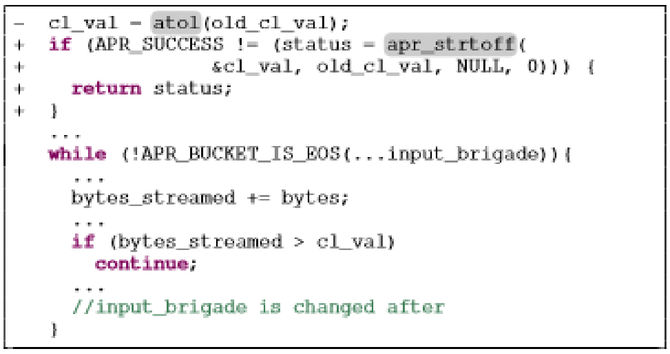
```
  if (fill_record_n_invoke_before_triggers (    thd,
        *info->update_fields, *info->update_values,
-                                                 0,
+                                      info->ignore,
              table->triggers, TRG_EVENT_UPDATE))
```

Fig. 2.: R1: Constant parameter function calls



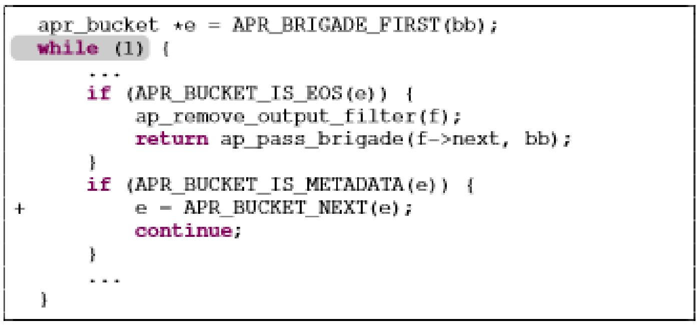
```
  AllocateRequest allocateRequest =
    AllocateRequest.newInstance(   lastResponseID,
                   super.getApplicationProgress(),
             new ArrayList<ResourceRequest>(ask),
             new ArrayList<ContainerId>(release),
-                                            null);
+                                    blacklistReq);
```

Fig. 3.: R2 Null parameter function calls



```
-  cl_val = atol(old_cl_val);
+  if (APR_SUCCESS != (status = apr_strtoff(
+               &cl_val, old_cl_val, NULL, 0))) {
+    return status;
+  }
   ...
   while (!APR_BUCKET_IS_EOS(...input_brigade)){
     ...
     bytes_streamed += bytes;
     ...
     if (bytes_streamed > cl_val)
       continue;
     ...
     //input_brigade is changed after
   }
```

Fig.4.: R3: Unsafe function calls.



```
  apr_bucket *e = APR_BRIGADE_FIRST(bb);
  while (1) {
       ...
       if (APR_BUCKET_IS_EOS(e)) {
           ap_remove_output_filter(f);
           return ap_pass_brigade(f->next, bb);
       }
       if (APR_BUCKET_IS_METADATA(e)) {
+          e = APR_BUCKET_NEXT(e);
           continue;
       }
       ...
  }
```

Fig. 5.: R4: Unchanged loop exit condition variables

Hytrace Dynamic Analysis

The design principle of the Hytrace [20] dynamic analysis component is very similar to that of the static analysis part. Our goal here is to relax the requirement for the precision and maximise the coverage (i.e., avoid miss detections) by including all of the potential root cause related functions. This, current Hytrace-dynamic module extends an existing dynamic cloud performance debugging tool PerfScope [25] to achieve our design goal. We chose PerfScope cause it imposes low overhead and it doesn't require the source code access, which makes it practical for the production of cloud infrastructures. However, like other dynamic techniques [26], [27], etc., PerfScope sacrifices the coverage in order to achieve high precision, which makes it inevitably miss identifying the buggy functions that have major contributions to the root cause. Based on this, the Hytrace-dynamic is proposed to address the coverage issue in PerfScope. When a performance anomaly is detected using an existing online anomaly detection tool [28], we first trigger a runtime system call analysis to identify the abnormal system call sequences produced by the server applications. Specifically, if we

ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 7, Issue 3, pp: (164-173), Month: July - September 2019, Available at: www.researchpublish.com

analyse a window of recent system call trace and identify which types of the system calls (e.g., sys_read, sys_futex) experience abnormal changes in either their execution frequency or their execution time. We first divide a window of the system call trace into multiple execution units based on the thread ID. We then have to apply a top-down hierarchical clustering algorithm [28] to group those execution units that perform similar operations together based on their appearance vector feature. Then, we use the nearest neighbour [29] algorithm to perform the outlier detection within each cluster to identify the abnormal execution units. Frequent episode mining is then used on those abnormal execution units, to identify the common abnormal system call sequences (i.e., S1). For example, from trace of the HDFS-3318, a sequence {sys_gettimeofday, sys_read, sys_read, sys_gettimeofday} is discovered which is executed more often than usual.

Next, we identify the application functions that have issued the abnormal system call sequences which are identified above.

We then again use frequent episode mining to extract the common system call sequences (i.e., S2) produced by the different application functions. These sequences (S2) are used then as signatures to match with the system call sequences (S1) whose execution frequencies or time are identified to be abnormal. For example, in the HDFS-3318, function Reader.performIO is found to be often producing system call sequence {sys_gettimeofday, sys_read, sys_read, sys_gettimeofday}, which is then used as its signature. When {sys_gettimeofday, sys_read, sys_read, sys_gettimeofday} is detected as one of the abnormal system call sequences, Reader.performIO is matched as one of the candidate buggy function. In comparison to the existing dynamic analysis tools (e.g., PerfScope), the Hytrace-dynamic integrates runtime execution path analysis with the abnormal function detection for the bug detection coverage to increase. Specifically, we then extend the candidate function list by adding the k-hop caller functions of the abnormal functions identified by the dynamic analysis tool. We also have conducted sensitivity study on the number of caller function hops (e.g., k) to evaluate the tradeoff between the coverage and precision.

Hybrid Scheme

The main idea of the Hytrace [20] is to combine the static and dynamic analysis techniques for achieving both the high coverage and high precision of performance diagnosis. Hytrace-static favours, coverage (i.e., completeness) over precision. It captures the potential buggy functions which are vulnerable to the performance problems over the static code pattern matching. The Hytrace-dynamic favours both the coverage and the “relaxed” precision. It identifies all the caller functions and the buggy functions which have abnormal practices during the runtime. The Hytrace approaches the leverages of the two carefully designed static and dynamic analysis components that are complementary to each other. Although each of the component is prone to false positives, the combination of these two leverages program semantic and the run-time behaviour information, and hence we can achieve much higher precision than the pure-static or pure-dynamic techniques. When performance symptom like a hang or a slowdown is observed by either of the users or an automated monitoring tool, the Hytrace then runs its dynamic component to identify a ranked list of functions that are behaving suspiciously, judging by the abnormality of the system-level metrics. Hytrace then compares this list which is produced by the dynamic component with the list of suspicious functions identified by the Hytrace static component, and then removes the functions that only appear in one list, and adjusts the ranks of the remaining functions accordingly. For example, if the Hytrace-dynamic identifies the three buggy functions foo (rank: 1), bar (rank: 2), and baz (rank: 3) but bar doesn’t include any static anti-patterns, the bar is removed and the final list becomes foo (rank: 1), baz (rank: 2). The rank of the baz gets improved because we remove the false positive function bar.

Limitations

The current evaluation of ours focuses on the single node performance bugs. For distributed performance bugs, the Hytrace’s diagnosis schemes are still quite preliminary. It generates a consolidated buggy function list using the intersection among all buggy function lists produced by the different faulty nodes. However, the distributed system bugs can manifest as a chain of the abnormal functions over multiple dependent nodes. Hytrace currently doesn’t consider such causal relationships between the distributed components. Previous work (e.g., FChain [34], PCatch [31]) has developed the distributed bug diagnosis tools based on the distributed system causal analysis. Hytrace can then integrate with those tools to achieve more precise distributed system performance bug diagnosis. The Hytrace-static component has currently five generic rules. Though our rule set can achieve 100% coverage on all the 133 performance bugs, we do not claim that these five rules can identify all the performance problems that are reported by the production cloud users. Hytrace framework allows the users to easily add new rules with a few code changes. Furthermore, we currently didn’t find any unsafe function in Java programs which matches with our rule R2. We do plan to extend this rule by adding more I/O related functions in Java, which is a part of our future work. The Hytrace-dynamic integrates the runtime execution path analysis with abnormal function detection to achieve the high coverage. Although, it can't identify all the root cause functions in every case.

IV. PERFTRACE

PerfTrace is an idea of combining the two above mentioned tools along with Pip scheme so that it can perform anomaly localization, bug diagnosis and behaviour tracing for running applications. We won't need to choose between these two scheme on the basis on our issue, PerfTrace can solve both the problems for us simultaneously. While developing PerfTrace we need to take care that we implement maximum advantages of Hytrace and PerfCompass, and at the same time minimize their individual limitations. PerfTrace is an ideal tool and works for multiple platform i.e. it is platform independent. As it is an amalgam of both these schemes it can easily differentiate between external and internal faults without the requirement of any source code or runtime instrumentation. PerfTrace uses a light weight kernel level system call for the purpose of tracing and performing online system call trace. PerfTrace is light weight and non-intrusive making it a perfect tool for IaaS clouds.

PerfTrace is a hybrid approach in diagnosing real world performance bugs. It combines rule based static analysis and runtime inference techniques in order to achieve higher accuracy, then what can one achieve using pure static or pure dynamic approaches.

V. CONCLUSION

In this paper we have presented an overview of Hytrace and PerfCompass understood there basic working and talked about their limitations. We also have introduced a new idea to concatenate these two tools- PerfTrace. PerfTrace can greatly improve coverage and precision comparing with existing techniques. PerfTrace will imposes less than 3% CPU overhead to the testing cloud environments. PerfTrace can be easily implemented and evaluated variety of commonly used open source server systems like Apache, MySQL, Hadoop, etc. PerfTrace can accurately diagnose all environment issues in the shared IaaS clouds and real software bugs.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments. With immense pleasure and deep sense of gratitude, I wish to express my sincere thanks to my supervisor Proff. Balamurugan R, Associate Professor, SCOPE, VIT University, without his/her motivation and continuous encouragement, this research would not have been successfully completed. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of VIT University.

REFERENCES

[1] M. Yu. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling network performance for multi-tier data center applications.

[2] R. Sambasivan, A. X. Zheng, M. D. Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G.R. Ganger. Diagnosing performance changes by comparing request flows.

[3] P. Reynolds, C. Killian, J. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems.

[4] R. Fonseca, G. Porter, H. Katz, S. Shenker, and I. Stoica. Xtrace: A pervasive network tracing framework.

[5] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services.

[6] Chanda, A. Cox, and W. Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. In SIGOPS.

[7] M. K. Aguilera, J. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes.

[8] P. Bahl, P. Barham, R. Black, R. Chandra, M. Goldszmidt, R. Isaacs, S. Kandula, L. Li, J. MacCormick, D. A. Maltz, R. Mortier, M. Wawrzoniak, and M. Zhang. Discovering dependencies for network management.

[9] Hussain, G. Bartlett, Y. Pryadkin, J. Heidemann, C. Papadopoulos and J. Bannister. Experiences with a continuous network tracing infrastructure. In SIGCOMM workshop on mining network data.

[10] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: black-box performance debugging for widearea systems.

[11] H. Nguyen, Z. Shen, Y. Tan, and X. Gu. Fchain: Toward blackbox online fault localization for cloud systems. In ICDCS.

[12] X.Ding, H. Huang, Y. Ruan, A. Shaikh, and X. Zhang. Automatic software fault diagnosis by exploiting application signatures. In LISA, 2008.

[13] K. Shen, C. Stewart, C. Li, and X. Li. Reference-driven performance anomaly identification.

[14] A. B. Sharma, H. Chen, M. Ding, K. Yoshihira, and G. Jiang. Fault detection and localization in distributed systems using invariant relationships.

[15] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs.

[16] P. Barham, A. Donnelly, Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling.

[17] A. Traeger, I. Deras, and E. Zadok. DARC: Dynamic analysis of root causes of latency distributions.

[18] U. Erlingsson, M. Peinado, S. Peter, and M. Budiu. Fay: Extensible distributed tracing from kernels to clusters.

[19] Daniel J. Dean, Hiep Nguyen, Peipei Wang, Xiaohui Gu. PerfCompass: Toward Runtime Performance Anomaly Fault Localization for Infrastructure-as-a-Service Clouds.

[20] Ting Dai, Member, IEEE, Daniel Dean, Member, IEEE, Peipei Wang, Member, IEEE, Xiaohui Gu, Senior Member, IEEE, and Shan Lu, Senior Member, IEEE. Hytrace: A Hybrid Approach to Performance Bug Diagnosis in Production Cloud Infrastructures.

[21] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems.

[22] H. Nguyen, Z. Shen, Y. Tan, and X. Gu. Fchain: Toward blackbox online fault localization for cloud systems.

[23] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes.

[24] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: black-box performance debugging for widearea systems.

[25] Daniel J. Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. PerfScope: Practical online server performance bug inference in production cloud computing infrastructures

[26] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. Productionrun software failure diagnosis via hardware performance counters.

[27] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software.

[28] Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. PREPARE: Predictive performance anomaly prevention for virtualized cloud systems.

[29] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. Introduction to Data Mining.