PerfCompass: Toward Runtime Performance Anomaly Fault Localization for Infrastructure-as-a-Service Clouds

Daniel J. Dean, Hiep Nguyen, Peipei Wang, Xiaohui Gu
{*djdean2,hcnguye3,pwang7*}*@ncsu.edu,gu@csc.ncsu.edu*
Department of Computer Science
North Carolina State University

Abstract

Infrastructure-as-a-service (IaaS) clouds are becoming widely adopted. However, as multiple tenants share the same physical resources, performance anomalies have become one of the top concerns for users. Unfortunately, performance anomaly diagnosis in the production IaaS cloud often takes a long time due to its inherent complexity and sharing nature. In this paper, we present PerfCompass, a runtime performance anomaly fault localization tool using online system call trace analysis techniques. Specifically, PerfCompass tackles a challenging fault localization problem for IaaS clouds, that is, differentiating whether a production-run performance anomaly is caused by an external fault (e.g., interference from other co-located applications) or an internal fault (e.g., software bug). PerfCompass does not require any application source code or runtime instrumentation, which makes it practical for production IaaS clouds. We have tested PerfCompass using a set of popular software systems (e.g., Apache, MySQL, Squid, Cassandra, Hadoop) and a range of common cloud environment issues and real software bugs. The results show that PerfCompass accurately diagnoses all the faults while imposing low overhead during normal application execution time.

1 Introduction

Infrastructure-as-a-service (IaaS) clouds [4] have become increasingly popular by providing a cost-effective resource sharing platform. While multi-tenant shared hosting has many benefits, there is also greater risk for performance anomalies (e.g., service outages [5], service level objective violations) to arise due to its inherent complexity and sharing nature. When an application in the IaaS cloud does not perform as expected, it is important to localize the cause of the performance anomaly quickly in order to minimize the performance penalty. However, diagnosing performance anomalies in a production IaaS cloud is extremely challenging. Shared computing environments mean that a performance anomaly can be caused by either external faults (e.g., interference from other co-located applications, improper resource allocations) or internal faults (e.g., application software bugs). Differentiating between these two types of faults is critical as the steps taken to handle external and internal faults are quite different. Particularly, as many external problems can be quickly fixed using virtual machine (VM) migration [13] or resource scaling [28, 21], we can avoid wasting time on unnecessary application debugging.

1.1 Summary of the State of the Art

Existing runtime performance anomaly fault localization tools (e.g.,[31, 25, 23, 17, 12, 11, 3, 9, 18, 24, 22]) can only provide *coarse-grained* fault localization such as identifying faulty components in a distributed application. In contrast, white-box or gray-box schemes (e.g., [19, 10, 30, 16]) can provide *fine-grained* diagnosis such as localizing the faulty functions or code block. However, those approaches often require application source code and expensive instrumentation, which makes them impractical for production IaaS clouds. Although previous work [15, 27, 26] also developed trace-based performance debugging tools, most of them need an extensive profiling phase to extract application performance models in advance, which are often infeasible in IaaS clouds. Some other performance debugging tools [16] need to combine user space and kernel space tracing, which can impart high overhead (up to 280%) to the application. Moreover, the existing performance anomaly diagnosis solutions do not address the unique fault localization problems in IaaS clouds such as distinguishing between external and internal faults.

1.2 Our Contributions

In this paper, we present PerfCompass, a light-weight runtime performance anomaly fault localization tool designed to be used by either IaaS cloud administrators or cloud service users. PerfCompass does not require application source code or any instrumentation, making it practical for production IaaS clouds. PerfCompass uses a kernel-level tracing tool [14] to collect system call traces with low overhead (1.03% on average) and performs *online* analysis over the system call trace to achieve fine-grained fault localization. Thus, PerfCompass does not require any advance application profiling, allowing it to diagnose previously seen or unseen anomalies.

By using kernel-level system call traces only, our approach is both light-weight and generic, applicable to any multi-processed or multi-threaded application written in different programming languages (e.g., both compiled and interpreted programs). Particularly, PerfCompass focuses on diagnosing whether a performance anomaly is caused by an internal or external fault. This diagnosis result is critical for achieving efficient performance anomaly correction in production IaaS clouds.

System call traces often have high volume. For example, an Apache web server can produce tens of thousands of system calls per second. Performing fault localization from such a large amount of system calls is like finding a needle in a haystack. We provide online system call trace analysis algorithms that can quickly extract useful *fault features* for us to identify whether the fault is external or internal. We propose to extract a *fault impact factor* feature and a *fault onset time dispersion* feature to infer whether the performance anomaly is caused by an external or internal fault.

When a performance anomaly occurs, the PerfCompass analysis module is dynamically triggered to perform online fault localization within the faulty VMs. We can identify faulty VMs using existing online black-box fault localization tools [22, 3, 9, 18, 24]. However, to differentiate between external and internal faults, we cannot treat the whole application VM as one black-box. PerfCompass first extracts different groups of closely related system calls, called *execution units*, from the continuous raw system call traces. We can first easily group system calls based on the process/thread ID. However, we observe that some server systems (e.g., Apache web server) use a pre-allocated thread pool and often reuse the same thread to perform different tasks. So we further split per-thread execution units based on the inter-system-call time gap (e.g., 99th percentile value) to mitigate inaccurate system call grouping caused by thread reuse.

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

Our key observation is that if the performance anomaly is caused by an external fault, we often see the fault directly affects *all* running execution units simultaneously. In contrast, an internal fault only directly affects 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. Thus, we use the *impact factor* to quantify the scope of the *direct* impact from the fault to different execution units and the *fault onset time dispersion* to quantify the onset time difference among different affected execution units. We can then infer whether the performance anomaly is caused by an external or internal fault by checking whether the fault has a global or local direct impact and whether the fault onset time durations of different execution units are similar.

In this paper, we make the following contributions:

- We make the first step toward runtime fine-grained performance anomaly fault localization in IaaS clouds using low-overhead kernel-level system call tracing and analysis techniques.
- We describe an online system call trace analysis algorithms that can extract useful fault features (e.g., fault impact scope, fault onset time dispersion) from massive raw system call traces quickly.
- We have implemented the PerfCompass system and tested it using five open source server systems under common environment issues and real software bugs. The results show that PerfCompass can successfully diagnose all the tested faults while imparting an average of 1.03% runtime overhead.

2 System Design

In this section, we present the design details of the PerfCompass system.

2.1 Fault Onset Time Identification

In order to distinguish between external and internal fault, we first extract two pieces of important information from raw system call traces of each execution unit: 1) *which* execution units are affected by the fault? and 2) *when* does the fault impact first start in each execution unit?

To detect whether an execution unit is affected by the fault, we first analyze the system call execution time.

Our observation is that if the fault causes application performance slowdown, the system call execution time should also increase. Therefore we compute the per-system call execution time and detect whether the fault affects the execution unit by identifying any outlier system call execution times. We perform outlier detection using the standard criteria (i.e., $>$ mean $+ 2 \times$ standard deviation).

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

Some performance anomalies do not manifest as changes in system call execution time. For example, if the performance anomaly is caused by an incorrect loop bug, we might observe the system calls inside the loop iterate more times when the bug is triggered than during normal execution. So, we maintain a frequency count for each 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.

If an execution unit is affected by the fault, we use a *fault onset time* metric to quantify how fast the fault affects the execution unit. We calculate the fault onset time using the time interval between the start time of the execution unit and the timestamp of the first affected system call in that unit. This time interval represents when the execution unit starts to be affected by the currently occurring fault after the execution unit is scheduled to run.

2.2 Fault Localization Schemes

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

However, if the fault impact factor has a borderline value (e.g., 90%), we extract a *fault onset time dispersion* feature to identify whether the fault affects different threads at the same time or at different time. Since an external fault affects all the running threads uniformly, the affected threads are likely to show the fault impact at a similar time after the fault is triggered. We use the standard deviation of the fault onset time among all the affected threads to quantify the fault onset time dispersion. If the fault onset dispersion is small, we infer that this fault is an external one. In contrast, an internal fault 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. Thus, if we observe a large fault onset time dispersion, we infer the fault is an internal one.

In our experiments, we found that a fixed fault onset time threshold (1 second) works well for all the applications and faults we tested. Generally speaking, it is more accurate to apply an application specific threshold. We observe that the fault onset time of different external faults are similar for the same application. Thus, we can easily calibrate the fault onset time threshold for each application by imposing a simple external fault to the application (e.g., imposing a low CPU cap by limiting the CPU consumption of the application). Such a calibration can be easily done without requiring any specific workload or fault type. We can calibrate the fault onset time dispersion threshold in a similar way.

3 Evaluation

We evaluate PerfCompass using real system performance anomalies caused by different external and internal faults. We first describe our experiment setup followed by a summary of our results.

3.1 Experiment Setup

We tested PerfCompass with five different systems: Apache [6], MySQL [20], Squid [29], Cassandra [7], and Hadoop [8]. Table 1 lists all the faults we tested. Each of the 9 external faults represents a common multi-tenancy or environment issue such as interference from other co-located applications, insufficient resource allocation, or network packet loss. We also tested 7 internal faults which are *real software bugs* found by searching the appropriate bug reporting repository (e.g., Bugzilla) for performance related terms. We then follow the instructions given in the bug report to reproduce the bugs.

We use Apache 2.2.22, MySQL 5.5.29, Squid 3.2.9, Cassandra 1.2.0-beta, and Hadoop 2.0.0-alpha for the

System name	Fault description	Fault impact factor	Fault onset time dispersion	Correct diagnosis
Apache	**CPU cap problem (external)**: improperly setting the VM's CPU cap to too low causes insufficient CPU allocation.	100 ± 0 %	7 ± 1 ms	✓
	Packet loss problem (external): using the *tc* command to randomly drop 10% packets.	100 ± 0 %	4 ± 1 ms	✓
	Flag setting bug (internal): deleting a port Apache is listening to and then restarting the server causes Apache to attempt to make a blocking call on a socket when a flag preventing blocking calls has not been cleared. The code does not check for this condition and re-tries endlessly (#37680).	50 ± 0.5 %	374 ± 63 ms	✓
MySQL	**I/O interference problem (external)**: a co-located VM causes a disk contention interference problem by running a disk intensive Hadoop job.	100 ± 0 %	15 ± 7 ms	✓
	CPU cap problem (external): improperly setting the VM's CPU cap to too low causes insufficient CPU allocation.	94 ± 2 %	17.77 ± 4 ms	✓
	Deadlock bug (internal): a MySQL deadlock bug that occurs when each of the two connections locks one table and tries to lock the other table. If one connection tries to execute a `INSERT DELAYED` command on the other while the other is sleeping, the system will become deadlocked (#54332).	40 ± 0%	38 ± 3 ms	✓
	Data flushing bug (internal): truncating a table causes a 5x slowdown in table insertions due to a bug with the InnoDB storage engine for big datasets. InnoDB fails to mark truncated data as deleted and constantly allocates new blocks. (#65615)	62 ± 3 %	721 ± 4 ms	✓
Squid	**Packet loss problem (external)**: using the *tc* command to randomly drop 10% packets.	100 ± 0 %	0.01 ± 0.001 ms	✓
	File access bug (internal): if /dev/null is made not accessible, by changing permissions for example, squid will loop endlessly trying to open it (#1484).	83 ± 1 %	0.35 ± 0.09 ms	✓
Cassandra	**CPU cap problem (external)**: improperly setting the VM's CPU cap to too low causes insufficient CPU allocation.	99 ± 1.4 %	28 ± 4 ms	✓
	I/O interference problem (external): a co-located VM causes a disk contention interference problem by running a disk intensive Hadoop job.	100 ± 0 %	9 ± 1 ms	✓
	Endless loop bug (internal): trying to alter a table when the table includes collections causes it to hang, consuming 100% CPU due to an internal problem with the way Cassandra locks tables for updating (#5064).	51 ± 5.7%	25 ± 0.98 ms	✓
Hadoop	**CPU cap problem (external)**: improperly setting the VM's CPU cap to too low causes insufficient CPU allocation.	98 ± 1 %	39 ± 5 ms	✓
	I/O interference problem (external): a co-located VM causes a disk contention interference problem by running a disk intensive Hadoop job.	98 ± 0 %	16 ± 3 ms	✓
	Endless read bug (internal): HDFS does not check for an overflow of an internal content length field causing HDFS transfers larger than 2GB to block and continuously try to read from an input stream (#HDFS-3318).	81 ± 0 %	23 ± 6 ms	✓
	Thread shutdown bug (internal): when the AppLogAggregator thread dies unexpectedly (e.g. due to a crash), the task waits for an atomic variable to be set indicating thread shutdown is complete. As the thread has died already, the variable will never be set and the job will hang indefinitely (# MAPREDUCE-3738).	85 ± 0.5 %	110 ± 20 ms	✓

Table 1: PerfCompass fault localization result summary for different systems under various external and internal faults.

Apache, MySQL, Squid, Cassandra, and Hadoop external faults, respectively. For each internal fault, we used the version specified in the bug report. In order to evaluate PerfCompass under workloads with realistic time variations, we used the following workloads during our experiments: 1) **Apache**: we use one day of per-minute workload intensity observed in a NASA web server trace starting at 1995-07-01:00.00 [2]; 2) **MySQL**: we use a MySQL benchmark tool called "Sysbench" [1]; 3) **Squid**: we configure Squid to act as a web proxy and use httperf to request various pages from an Apache web server using Squid as the proxy; 4) **Cassandra**: we use a workload which creates a table and inserts various entries into the table; and 5) **Hadoop**: we use the standard Pi calculation example with 16 map and 16 reduce tasks. We repeated each fault injection three times, reporting the average and mean standard deviation values over all 3 runs.

To evaluate our approach in different virtualization environments, we conducted our experiments on two different virtualized clusters. The Apache and MySQL experiments were conducted on a cluster 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. The Squid, Cassandra, and Hadoop experiments were conducted on a cluster where each node is equipped with a quad-core Xeon 2.53Ghz CPU along with 8GB memory with KVM 1.0. In both cases, each trace was collected on a guest VM using LTTng 2.0.1 running 32-bit Ubuntu 12.04 kernel version 3.2.0.

We calibrate the fault onset time threshold and fault onset time dispersion threshold values by injecting a low CPU cap external fault to different applications. Note that we do not need to perform the calibration under the same workload intensity as the performance anomaly occurrence time. The fault onset time thresholds we get are 0.06 second for Apache, 0.3 second for MySQL, 0.002 second for Squid, and 0.4 second for Cassandra and Hadoop. The fault onset time dispersion values we get are 7ms for Apache, 17ms for MySQL, 0.02ms for Squid, 28ms for Cassandra, and 39ms for Hadoop. We also conduct the experiments using a fixed fault onset time threshold (1 second) and found that using the simple threshold gives similar results.

3.2 Result Summary

Table 1 provides a summary of our fault localization results for different systems under various external and internal faults. The results show that PerfCompass correctly diagnoses each of the 9 external faults as external and each of the 7 internal faults as internal.

All the tested external faults have impact factors close to 100%, clearly indicating each of them as external. Similarly, most internal faults for these systems have an impact factor significantly less than 100%, clearly indicating each of them as internal. We also observe that most internal faults have significantly larger fault onset time dispersion values than the external faults of the same application.

3.2.1 PerfCompass Overhead

We now evaluate the overhead of the PerfCompass system. Figure 1 shows the runtime overhead imposed by PerfCompass on each of the tested systems. For Apache, MySQL, and Squid we used httperf to send a fixed number of requests, recording the average response time both with and without PerfCompass. We used a request rate of 100 requests per second for Apache and Squid, and a request rate of 20 requests per second for MySQL[1]. For Cassandra, we ran a simple database insertion workload, recording the average processing time. For Hadoop we ran the Pi sample job, recording the average processing time. We ran all the tests 5 times, reporting the mean and standard deviation. We observe that PerfCompass imparts 1.03% runtime overhead on average. We also measured the resource consumption of PerfCompass. We found PerfCompass imparts between 2-3% CPU load and has a small memory footprint (about 256KB). We believe that PerfCompass is light-weight, which makes it practical for online performance anomaly fault localization in production IaaS clouds.

[1]Those request rates were selected based on the capacity of the tested machine.

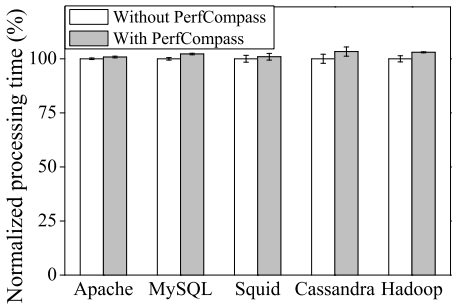

Figure 1: The overhead imposed by PerfCompass on different server systems.

4 Conclusions and Future Work

In this paper, we have presented PerfCompass, a runtime performance anomaly fault localization tool for production IaaS clouds. PerfCompass can distinguish between external and internal faults without requiring source code or runtime instrumentation. PerfCompass uses light-weight kernel-level system call tracing and performs online system call trace analysis to extract fault features from massive raw system call traces during runtime. We have implemented PerfCompass and evaluated it using a variety of commonly used open source server systems including Apache, MySQL, Squid, Cassandra, and Hadoop. We tested PerfCompass using a set of common environment issues in the shared IaaS clouds and real software bugs. The results show that PerfCompass accurately diagnoses all the tested faults. PerfCompass is light-weight and non-intrusive, which makes it practical for IaaS clouds. In the future, we plan to extend our system call analysis framework to extract other fault features for supporting different types of fine-grained runtime performance anomaly fault localizations such as narrowing down potential root cause functions.

Acknowledgment

We thank the anonymous reviewers 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] *SysBench: a system performance benchmark*. http://sysbench.sourceforge.net/.

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

[3] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SIGOPS*, 2003.

[4] Amazon elastic compute cloud. `http://aws.amazon.com/ec2/`.

[5] Amazon elastic compute cloud service outage. `http://money.cnn.com/2011/04/21/technology/amazon_server_outage/`.

[6] Apache. `http://httpd.apache.org/`.

[7] Apache Cassandra. `http://cassandra.apache.org/`.

[8] Apache Hadoop. `http://hadoop.apache.org/`.

[9] 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. In *Hotnets*, 2006.

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

[11] A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. In *SIGOPS*, 2007.

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

[13] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, 2005.

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

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

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

[17] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *NSDI*, 2007.

[18] A. 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*, 2005.

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

[20] MySQL. `http://www.mysql.com/`.

[21] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. AGILE: elastic distributed resource scaling for infrastructure-as-a-service. In *ICAC*, 2013.

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

[23] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, 2006.

[24] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: black-box performance debugging for wide-area systems. In *WWW*, 2006.

[25] R. 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. In *NSDI*, 2011.

[26] A. B. Sharma, H. Chen, M. Ding, K. Yoshihira, and G. Jiang. Fault detection and localization in distributed systems using invariant relationships. In *DSN*, 2013.

[27] K. Shen, C. Stewart, C. Li, and X. Li. Reference-driven performance anomaly identification. In *SIGMETRICS*, 2009.

[28] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *SOCC*, 2011.

[29] Squid-cache. `http://www.squid-cache.org/`.

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

[31] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling network performance for multi-tier data center applications. In *NSDI*, 2011.