PerfCompass: Online Performance Anomaly Fault Localization and Inference in Infrastructure-as-a-Service Clouds

Daniel J. Dean, *Member, IEEE,* Hiep Nguyen, *Member, IEEE,*
Peipei Wang, *Member, IEEE,*Xiaohui Gu, *Senior Member, IEEE,* Anca Sailer, *Senior Member, IEEE,*
Andrzej Kochut, *Senior Member, IEEE,*

Abstract—Infrastructure-as-a-service clouds are becoming widely adopted. However, resource sharing and multi-tenancy have made performance anomalies a top concern for users. Timely debugging those anomalies is paramount for minimizing the performance penalty for users. Unfortunately, this debugging often takes a long time due to the inherent complexity and sharing nature of cloud infrastructures. When an application experiences a performance anomaly, it is important to distinguish between faults with a global impact and faults with a local impact as the diagnosis and recovery steps for faults with a global impact or local impact are quite different. In this paper, we present PerfCompass, an online performance anomaly fault debugging tool that can quantify whether a production-run performance anomaly has a global impact or local impact. PerfCompass can use this information to suggest the root cause as either an external fault (e.g., environment-based) or an internal fault (e.g., software bugs). Furthermore, PerfCompass can identify top affected system calls to provide useful diagnostic hints for detailed performance debugging. PerfCompass does not require source code or runtime application instrumentation, which makes it practical for production systems. We have tested PerfCompass by running five common open source systems (e.g., Apache, MySQL, Tomcat, Hadoop, Cassandra) inside a virtualized cloud testbed. Our experiments use a range of common infrastructure sharing issues and real software bugs. The results show that PerfCompass accurately classifies 23 out of the 24 tested cases without calibration and achieves 100% accuracy with calibration. PerfCompass provides useful diagnosis hints within several minutes and imposes negligible runtime overhead to the production system during normal execution time.

Index Terms—Reliability, availability, and serviceability; Debugging aids; Distributed debugging; Performance

1 Introduction

Infrastructure-as-a-service (IaaS) clouds [5] provide cost-efficient access to computing resources by leasing those resources to users in a pay-as-you-go manner. Although multi-tenant shared hosting has many benefits, the complexity of virtualized infrastructures along with resource sharing causes IaaS clouds to be prone to performance anomalies (e.g., service outages [6], service level objective violations). When a performance anomaly occurs, it is important to identify the root cause and correct the performance anomaly quickly in order to avoid significant financial penalties for both the cloud service provider and the cloud user.

However, it is challenging to diagnose performance anomalies in a production cloud infrastructure is a challenging task. Due to the multi-tenancy and resource sharing nature of IaaS clouds, it is often difficult to identify the root cause of a production-run performance anomaly. On one hand, a performance anomaly fault can have a *global* impact, affecting almost all the executing threads of an application when the fault is triggered. For example, a VM with insufficient resources (e.g., insufficient memory) will cause all executing threads of an application to be affected. On the other hand, a fault can also have a *local* impact, affecting only a subset of the executing threads of an application right after the fault is triggered. For example, an infinite loop bug reading from a socket will only directly affect the threads executing the infinite loop directly (e.g., increasing the frequency of `sys_read`). *External* faults such as interference from other co-located applications or improper resource allocation typically have a global impact on the system. As identified by previous work, both external environment issues [54] and internal software bugs [32] are major problems existing in the cloud. It is critical to identify whether a fault has a global or local impact on the system as the steps taken to diagnose and correct those two different kinds of faults are quite different. When we find a fault has a global impact we can try a simple fix first such as alerting the infrastructure management system to migrate the application to a different host. If the fault has a local impact, however, the cause is likely to be an internal software bug, requiring the application developer to diagnose and fix it.

Although previous work [52], [39], [37], [23], [16], [15], [4], [12], [27], [38], [35] has studied the online fault localization problem under different computing contexts, existing approaches can only provide *coarse-grained* fault localization, that is, identifying faulty application components. However,

- *Daniel J. Dean, Hiep Nguyen, Peipei Wang, and Xiaohui Gu are with the Department of Computer Science, North Carolina State University, Raleigh, NC, 27603.*
  E-mail: {djdean2,pwang7,hcnguye3}@ncsu.edu,gu@csc.ncsu.edu
- *Anca Sailer and Andrzej Kochut are with the IBM T.J. Watson Research Center.*

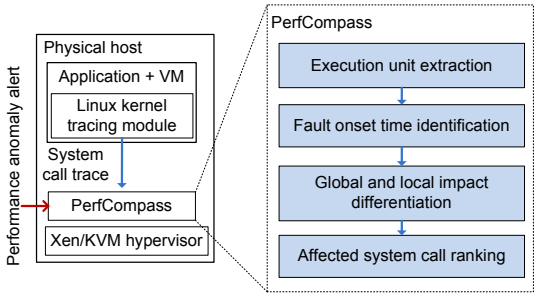


Fig. 1. PerfCompass overview.

they cannot distinguish between faults with a global or local impact since both global and local faults may manifest similarly at the application component level (e.g., increased CPU usage). Additionally, tools such as Fay [22] and DARC [47] can help developers debug the root cause of performance anomalies. However, these tools are difficult to adapt to a production cloud environments due to the large overhead they impart or application instrumentation they require. A detailed discussion of related work can be found in Section 5.

In this paper, we present PerfCompass, an online fine-grained performance anomaly fault localization and inference tool designed to differentiate faults with a global impact from faults with a local impact and provide diagnostic hints for either cloud system administrators or cloud users. Figure 1 illustrates the overall structure of the PerfCompass system. PerfCompass employs lightweight kernel-level system call tracing to continuously record system calls from the monitored application. PerfCompass performs fault localization and inference using four steps. First, we segment the large raw system call traces into groups of closely related system calls which we call *execution units*. Next, we process these execution units to extract a set of fine-grained fault features (e.g., which threads are affected by the fault, how quickly the fault manifests in different threads). Third, we use these fine-grained fault features to differentiate between faults with a global impact and faults with a local impact. PerfCompass then provides diagnostic hints by suggesting the root cause of the fault as being an external environment issue or internal software bug. Lastly, PerfCompass identifies the top affected system calls and provides ranks for those affected system calls. Knowing what system calls are affected by the fault allows developers to identify which subsystems (e.g., network, I/O, CPU) are affected and gives hints on how to perform further diagnosis on the fault.

In this paper, we make the following contributions:

- We develop a novel fine-grained fault feature extraction algorithm that can identify which threads are affected by the fault and how quickly the fault manifests in different threads.
- We describe how external environment issues typically have a global impact on the system while internal software bugs can have a global or local impact on the system. We show how to use this information to localize performance anomalies as external or internal.
- We present a system call ranking algorithm which identifies the top affected system calls and provides ranks for those affected system calls. This system call ranking is useful for system administrators or users to gain insight about the root cause of the performance anomaly.
- We describe a robust execution unit extraction scheme that can properly split a large raw system call trace into fine-grained groups of closely related system calls for different kinds of applications.
- We have implemented PerfCompass and conducted an experimental study using five open source systems (Apache, MySQL, Tomcat, Cassandra, and Hadoop). The results show that PerfCompass can correctly localize 23 out of the 24 tested faults (17 common environment issues and 7 real software bugs) without calibration and achieve 100% accuracy with calibration. Furthermore, PerfCompass provides useful hints for both external and internal faults. Our approach is lightweight, imparting an average of 2.1% runtime overhead to the tested server applications.

The rest of the paper is organized as follows. Section 2 discusses the preliminaries of our system. Section 3 describes the design of the PerfCompass system. Section 4 presents our experimental evaluation. Section 5 compares our work with related work. Section 6 discusses the limitations of our system along with our future work. Finally, the paper concludes in Section 7.

2 Preliminaries

In this section, we first describe our system model for IaaS clouds. We then describe our problem formulation followed by the key assumptions we make.

2.1 System Model

IaaS clouds are comprised of several physical hosts connected by networks. These physical hosts use virtualization technology (e.g., KVM [29], Xen [14]) to run several different virtual machines (VMs), which are then leased to end users. We instrument each VM with a lightweight kernel tracing tool called the Linux Trace Toolkit Next Generation (LTTng) [20] which continuously monitors the system calls generated by each application running on that VM.

The collected system call traces are then stored on a globally accessible NFS server. PerfCompass is decoupled from the VMs it monitors and only needs access to the system call trace of a monitored VM to perform its runtime analysis. Thus, PerfCompass can be encapsulated in special analysis VMs which could be dynamically placed on lightly loaded hosts to use the residual resources in the IaaS cloud for fault localization.

PerfCompass is triggered when an alarm is raised by an online performance anomaly detection tool [18], [46]. Fault localization is performed on the faulty components identified by existing online component-level fault localization tools (e.g., [16], [35]).

Table 1 shows the various terms we have defined and is intended as a reference to help avoid any ambiguity while discussing how each item is used.

Term	Definition
s_i	The type of system call (e.g., sys_write)
t_i	The timestamp of system call s_i
λ	An execution unit (closely related system calls)
σ	The fault onset time (how fast a fault effects a thread)
τ	The fault impact factor (percentage of affected threads over total threads)
Ω	The fault onset time dispersion (fault onset time standard deviation among all affected threads)
k	The moving average window length
C	The system call frequency count
T	The time interval of start of execution unit to system call si
W	The buffered window of recent system calls
α	The fault onset time threshold
β	The fault onset time dispersion threshold
δ	The external fault impact factor threshold
Δ	The internal fault impact factor threshold
Θ	The threshold for system call execution time and system call frequency outlier detection

TABLE 1
Notations.

2.2 Problem Formulation and Key Assumptions

The goal of PerfCompass is to provide two key pieces of information: 1) the performance anomaly impact results; and 2) performance anomaly root cause hints. Specifically, PerfCompass first quantifies the impact of a detected fault as either global or local, suggests the cause as external or internal, and then identifies the system calls which are most affected by the fault.

PerfCompass is designed specifically for performance anomalies which typically provide little information (e.g., no error messages, limited stack trace) for debugging. However, our work focuses on the anomalies which are caused by either infrastructure sharing issues or software bugs. Performance anomalies due to an application misconfiguration have been studied before [11] and are beyond the scope of this work.

PerfCompass targets multi-threaded or multi-processed server applications which are prone to performance anomalies. We observe that most modern server applications are multi-threaded or multi-processed in order to utilize the multi-core processors efficiently.

PerfCompass aims at providing fast online fault debugging and root cause inference within the production cloud infrastructure. Thus, PerfCompass needs to be light-weight without imposing high overhead to the production system. Note that PerfCompass is not designed to support detailed software debugging such as localizing root cause buggy functions, which require application knowledge and developer involvement. However, we believe that PerfCompass can avoid unnecessary software debugging effort by quantifying the fault impact as global or local, suggesting the root cause as either external or internal, and providing runtime fault debugging hints.

3 System Design

In this section, we present the design details of the PerfCompass system. We begin with an overview of our system. We then describe each component of PerfCompass. Lastly, we describe two enhancements which can help improve the fault localization accuracy.

3.1 System Overview

PerfCompass performs fault localization and inference using four components: 1) execution unit extraction; 2) fault onset time identification; 3) fault differentiation; and 4) affected system call ranking. PerfCompass first quantifies the fault impact as either global or local. PerfCompass next suggests root cause of the fault as either external or internal. Finally, PerfCompass identifies the system calls affected by the fault and ranks them in order to provide debugging clues to developers and system administrators.

Kernel-level system call tracing allows us to monitor application behavior without imparting significant overhead. However, the system call tracing tool typically produces a single large system call trace for each application. It is infeasible to perform fine-grained fault localization using the raw system call trace directly. Hence, PerfCompass first employs an *execution unit extraction* component to segment the raw system call traces into fine-grained groups of closely related system calls called *execution units*.

Next, we analyze the different extracted execution units to answer the following two questions: 1) which threads are affected by the fault? and 2) how quickly are they affected by the fault? To do this, we first compute a *fault onset time* metric to quantify how quickly a thread is affected by the fault. We then extract a *fault impact factor* metric, which computes the percentage of threads affected by the fault. We also extract a *fault onset time dispersion* metric, which quantifies the differences among the fault onset time values of the affected threads.

Third, we use the fault features to determine whether the fault has a *global* impact on the system or a *local* impact on the system. Intuitively we observe that external faults have global impact on the system while internal faults can have either a global or local impact. The reason is that an external fault, such as an insufficient CPU allocation, will directly affect all running threads regardless of what code they are executing. In contrast, only those threads executing the buggy code are directly affected by an internal fault with other threads being affected indirectly due to shared resources or inter-thread communication. Depending on the application logic, this means the fault could have a global or local impact on the system. We use this observation to suggest the root cause of the fault as either external or internal.

Lastly, PerfCompass identifies any system calls showing a significant increase in either execution time or frequency, providing a ranked list of these affected system calls based on the observed increase. Those top affected system calls can provide important hints about both external and internal faults. For example, when a co-located VM causes disk contention, I/O related system calls (e.g., `sys_read, sys_write`) will be more affected the other types of system calls. Similarly, when an internal bug causes an infinite loop in a blocking I/O function, we would expect to see an increase in the frequency of blocking I/O related system calls (e.g., `sys_futex, sys_write`).

We now discuss each component in detail in the following subsections.

3.2 Execution Unit Extraction

PerfCompass uses LTTng to collect system calls (denoted by s_i) issued by the monitored application from the guest OS. Each system call entry consists of 1) {timestamp 1, process/thread ID, system call name} and 2) {timestamp 2, process/thread ID, `sys_exit`}. We derive the system call execution time by computing the difference between timestamp 1 and timestamp 2.

We buffer a window (e.g., W = 1.5 minutes) of recent system calls in the kernel buffer. Older system call data are written into a NFS server in case longer trace-data are needed for fault localization. When a performance anomaly is detected, PerfCompass is triggered to perform online fault localization and inference using the buffered trace. Typically, the buffered data is sufficient for PerfCompass to perform fault localization as most faults manifest within a short period of time. However, if the buffered data are insufficient, PerfCompass can retrieve additional trace data from the remote NFS server.

The kernel tracing tool produces a single stream of system calls generated by all threads and processes for each application. In order to perform fine-grained fault localization, we need to segment this large raw trace into execution units. We first group system calls based on thread IDs. The rationale for doing this is that different threads are typically assigned with different tasks (e.g., logging, request processing). Therefore, the system calls generated by the individual threads are likely to be related to each other, though not necessarily from the same code block.

When processing the thread-based execution units, we found that system calls separated by relatively large time gaps were being grouped together into the same execution unit. These time gaps are problematic, as they give skewed fault onset time values, leading to incorrect fault localization results. Specifically, grouping two contiguous system calls with a large time gap into one execution unit causes us to derive an inflated fault onset time value for that execution unit. When exploring this issue, we found that a large time gap between two contiguous system calls can be caused by two reasons: 1) thread reuse, and 2) non-deterministic thread scheduling. For example, we observed that some server systems (e.g., Apache web server) use a pre-allocated thread pool in which the threads are reused

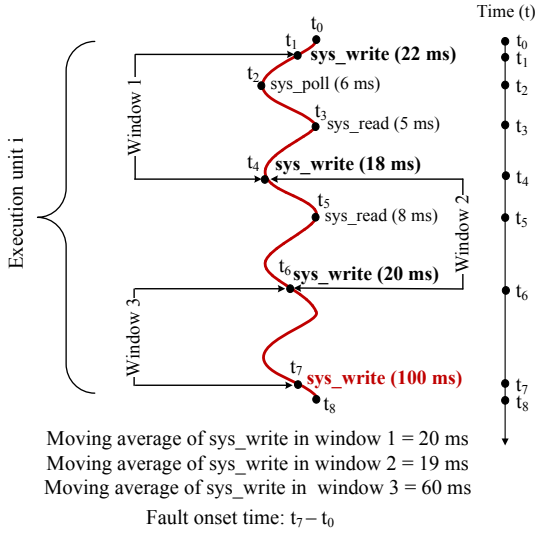


Fig. 2. Fault onset time calculation for one execution unit. The moving average for each system call type is calculated separately. The moving average of `sys_write` increases from 20ms in window 1 to 19ms in window 2 to 60ms in window 3. The time elapsed from the start of the execution unit λ_1 (t_0) to the currently processed system call (e.g.,`sys_write` in the current window (t_7)) is defined as the fault onset time (σ).

for different tasks to avoid the overhead of dynamically creating new threads. We also found a context switch, invoked by the CPU scheduler, could occur between two contiguous system calls in the same application, causing a large time gap.

To address this problem, we further divide each per-thread execution unit based on the time gaps between two contiguous system calls to mitigate the inaccuracies caused by thread recycling or the CPU scheduler. Specifically, if the time gap between two contiguous system calls s_1 and s_2 is larger than a certain threshold, we say the current execution unit λ_1 ends at s_1 and a new execution unit λ_2 begins at s_2. This ensures that each execution unit does not contain two system calls separated by a gap larger than the threshold. However, setting a proper threshold is a challenging task. In this work, we use the fault onset time threshold as the time gap threshold. The fault onset time threshold determines whether the fault has a direct or indirect impact on the execution unit. Hence, our approach ensures that each execution unit does not contain two contiguous system calls which are separated by a time gap larger than fault onset time threshold.

3.3 Fault Onset Time Identification

In order to distinguish between faults with a global and local impact, we need to analyze the execution units to identify which threads are affected by the fault and how quickly they are affected. We carefully chose a simple and fast outlier detection algorithm to make PerfCompass practical for online massive system call analysis. We also explored using a clustering algorithm. However we found this yields little accuracy gain with much higher computation overhead. To detect whether a thread is affected by the fault, we first analyze the system

call execution time of the execution units in that thread. When calculating system call execution time, it is important to note that different system call types have different execution time. As a result, it is necessary to distinguish different types of system calls (e.g., `sys_write`, `sys_gettimeofday`) and compute the execution time for each system call type separately.

We search for outlier system call execution time to detect whether the fault affects an execution unit. We use the standard criteria (i.e., $>$ mean $+$ 2 $\times$ standard deviation) to detect outliers. If *any* system call type is identified as an outlier, we infer that this execution unit is affected by the fault.

System call execution time is inherently fluctuating. For example, the execution time of `sys_write` can depend on both the number of bytes to be written as well as any blocking disk activity. These fluctuations can lead to the incorrect detection of spurious outliers. To avoid this, we use a k-length moving average filter to de-noise the system call execution time. Specifically, we compute an average value for a sliding window of k consecutive execution time samples. It is important to set k appropriately as too large of a value will over-smooth the execution time. We found a value of $k = 5$ works well for all the systems we tested. We have also conducted a sensitivity study on this parameter, which we present in Section 1.1 of the online supplementary material.

Performance anomalies do not always manifest as changes in system call execution time. For example, if a performance anomaly is caused by an infinite loop, we might observe an increase in the number of system calls generated within the loop when the bug is trigged, compared to the normal execution. As a result, we also include changes in system call frequency as part of the fault impact. We maintain a frequency count for each system call type. When a system call s_i is invoked, we increase the frequency count C for s_i by 1. Next, we compute the time interval T between the start time of the execution unit and the timestamp of s_i. We then compute the frequency using C/T. We compute frequency in this way because each execution unit represents some individual portion of the application with a certain system call generation rate. Finally, we de-noise and apply outlier detection over the system call frequency in a similar way as we do for the system call execution time.

If a thread consists of multiple execution units, we only consider the first execution unit affected by the fault. We also mark the thread containing the affected execution unit as affected by the fault.

When an execution unit is affected by the fault, we define a fault onset time metric to quantify how fast the fault affects the execution unit and the thread containing it. We compute the fault onset time using the time interval between the start time of the execution unit and the timestamp of the currently processed system call in the first moving average window where either a system call execution time or system call frequency outlier is detected. The rationale for this is we want to identify how long it takes the fault to show *any* effect on the execution unit.

Figure 2 shows a simple example for calculating the fault onset time. Suppose we set the moving average length $k = 2$. The first moving average execution time for `sys_write` in the first window is 20 ms. The second moving average execution time of `sys_write` in the second window is 19 ms. However, in the third window, the moving average execution time significantly increases to 60 ms. Hence, we infer that this execution unit λ_1 is affected by the fault during the third window. The time elapsed from the start of λ_1 (t_0) to the currently processed system call (e.g., `sys_write` in the third window) (t_7) is defined as the fault onset time (σ). Specifically, the fault onset time σ_1 for λ_1 is (t_7-t_0).

3.4 Fault Differentiation Schemes

Our fault differentiation scheme extracts and analyzes two features to determine whether a fault has a global or local impact. We first quantify the fault impact on each thread using the fault onset time to infer whether the thread is affected by the fault directly or indirectly. If the fault onset time of the affected thread is smaller than a pre-defined fault onset time threshold, we say this thread is affected by the fault directly. Otherwise, we say this thread is affected indirectly. We will describe how to set the fault onset time threshold properly in Section 3.6. We then define a *fault impact factor* to compute the percentage of threads affected by the fault directly. If the fault impact factor is close to 100% (e.g., >90%), we infer that the fault has a global impact and may be an external fault or internal fault. In this case, we suggest the fault is an external fault because the steps taken to fix an external fault are simple. If the simple fixes are ineffective, the problem is internal. However, if the fault impact factor is significantly less than 100% (e.g., 80%), we infer that the fault has a local impact and is thus an internal fault.

We can use the fault impact factor (τ) alone to correctly localize most faults as either external or internal. However, we found there are borderline cases where it is not clear whether the fault impact was global or local (e.g., $\tau \in$ [80%,90%]). To correctly localize those borderline cases, we compute a *fault onset time dispersion* metric to quantify the differences among the fault onset time values of different affected threads. We use the standard deviation of the fault onset time durations among all the affected threads to compute the fault onset time dispersion. A small fault onset time dispersion means that all threads are affected at roughly the same time, indicating an external fault. In contrast, a large fault onset time dispersion means the threads of the system are affected at different times, indicating an internal fault.

3.5 Affected System Call Ranking

In addition to distinguishing between external and internal faults, PerfCompass identifies the top affected system call types to provide hints for performance anomaly debugging. For example, in the Apache external memory cap case, we found an increase in the execution time of `sys_mmap_pgoff`, indicating a memory related problem. In the Hadoop infinite read bug, we found an increase in the frequency of `sys_read`, a direct result of the bug itself. Knowing which system calls are affected and which system calls are not affected can be helpful for both administrators

and developers to understand the root cause of a performance anomaly.

We rank each system call type based on the increase percentage of either the system call execution time or system call frequency. Our ranking algorithm only considers those system calls in the first affected execution unit in each affected thread in order to identify those system calls which are directly affected by the fault. Processing each thread in its entirety runs the risk of identifying system calls affected indirectly by the fault. For example, when a memory cap of a VM has been set too low, it is likely we will see memory related system calls (e.g.,`sys_mmap_pgoff`) are affected initially. However, once the machine starts swapping due to lack of memory, we may start to see scheduling-based system calls are more affected (e.g., `sys_sched_getaffinity`). We use the maximum observed system call execution time percentage increase and system call frequency percentage increase, for each system call type, among all threads, to provide an execution time rank and a frequency rank for each system call type.

Finally, we output both ranked lists in order to provide useful clues for system administrators or software developers to further diagnose external or internal faults. For example, knowing that blocking network I/O based system calls are heavily affected by an external fault could indicate a network contention problem. Alternatively, in the case of an internal fault, knowing blocking network I/O related system calls are affected could help developers localize the buggy segment of code (e.g., portions of the code performing blocking network I/O operations).

3.6 Fault Localization Enhancement

Unlike CPU and memory related problems, external disk I/O or network faults do not always affect all execution units. For example, disk contention only directly affects those execution units performing disk I/O operations. When the fault impact factor indicates a borderline case (i.e., $\Delta \leq \tau \leq \delta$) and when our system call ranking scheme indicates mainly disk I/O or network related system calls are affected, PerfCompass triggers a *filtering mode* to achieve more precise fault localization. Figure 3 describes our fault localization algorithm, enhanced with the filtering mode. As shown, PerfCompass first filters non-disk I/O and non-network related system calls and then recomputes the fault impact factor τ. We found that the filtering scheme gives more precise diagnosis for external disk I/O or network faults.

During our experiments, we found that the fault onset time threshold for distinguishing between direct and indirect impact varied from application to application. The reason is that the threads of different applications interact with the kernel and between each other in different ways. Although we found setting a static fault onset threshold (e.g., 0.5 sec) gave correct fault localization results for 23 out of 24 faults we tested, we incorrectly localized one fault. To further improve the accuracy of PerfCompass, we have developed calibration scheme using offline profiling. We run the application under a simple external fault (i.e., an overly low CPU cap) recording the observed fault onset time values. We then use the largest fault onset time value among all the affected threads as the fault onset time threshold for the whole application. We conducted several experiments on calibrating PerfCompass this way using different application workload intensities, mixes, and different application versions. We observe that although the raw calibrated values obtained may slightly vary, the accuracy of our fault localization scheme is not affected by the variations. We can also calibrate the fault onset time dispersion for distinguishing between external and internal faults in a similar way.

```
Inputs:
α: Fault onset time threshold
β: Fault onset time dispersion threshold
δ: External fault impact factor threshold (90%)
Δ: Internal fault impact factor threshold (80%)

FaultLocalization(α, β)
1. Calculate fault onset time σ for each thread
2. Compute fault impact factor τ using α
3. Rank affected system calls
4. if τ > δ
5.     return "external fault"
6. else if τ < Δ
7.     return "internal fault"
8. else { /* borderline cases */
9.     if top ranked system calls are I/O related
10.        Filter non-I/O related system calls
11.        Recompute τ
12.    if τ > δ
13.        return "external fault"
14.    else if fault onset time dispersion Ω > β
15.        return "internal fault"
16.    else
17.        return "external fault"
18.}
```

Fig. 3. PerfCompass external and internal fault localization algorithm.

4 Evaluation

We evaluate PerfCompass using real system performance anomalies caused by different external and internal faults. We first describe our experiment setup followed by the results. Next, we show the overhead imparted by our system. Finally, we discuss the limitations of our system.

4.1 Experiment Setup

We evaluated PerfCompass using five commonly used server systems: Apache [7], MySQL [34], Tomcat [10], Cassandra [8], and Hadoop [9]. Table 2 and 3 list all the external and internal faults we tested, respectively. Each of the 17 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 in real world bug repositories (e.g., Bugzilla) by searching for

System name	Fault ID	Fault description
Apache	1	**CPU cap problem**: improperly setting the VM's CPU cap to too low causes insufficient CPU allocation.
	2	**Memory cap problem**: improperly setting the VM's memory cap to too low causes insufficient memory allocation.
	3	**I/O interference problem**: a co-located VM causes a disk contention interference problem by running a disk intensive Hadoop job.
	4	**Packet loss problem**: using the *tc* command causes the network to randomly drop 10% of the packets.
MySQL	6	**I/O interference problem**: a co-located VM causes a disk contention interference problem by running a disk intensive Hadoop job.
	7	**CPU cap problem**: improperly setting the VM's CPU cap to too low causes insufficient CPU allocation.
	8	**Memory cap problem**: improperly setting the VM's memory cap to too low causes insufficient memory allocation.
	9	**Packet loss problem**: using the *tc* command causes the network to randomly drop 10% of the packets.
Tomcat	12	**Packet loss problem**: using the *tc* command causes the network to randomly drop 10% of the packets.
	13	**CPU cap problem**: improperly setting the VM's CPU cap to too low causes insufficient CPU allocation.
	14	**Memory cap problem:** improperly setting the VM's memory cap to too low causes insufficient memory allocation.
Cassandra	16	**CPU cap problem**: improperly setting the VM's CPU cap to too low causes insufficient CPU allocation.
	17	**Packet loss problem**: using the *tc* command causes the network to randomly drop 10% of the packets.
	18	**I/O interference problem**: a co-located VM causes a disk contention interference problem by running a disk intensive Hadoop job.
Hadoop	20	**CPU cap problem**: improperly setting the VM's CPU cap to too low causes insufficient CPU allocation.
	21	**Packet loss problem**: using the *tc* command causes the network to randomly drop 10% of the packets.
	22	**I/O interference problem**: a co-located VM causes a disk contention interference problem by running a disk intensive Hadoop job.

TABLE 2
Descriptions of the 17 external faults we tested.

System name	Fault ID	Fault description (type)
Apache	5	**Flag setting bug (hang)**: 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 continuously re-tries the call (#37680).
MySQL	10	**Deadlock bug (hang)**: 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).
	11	**Data flushing bug (slowdown)**: 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).
Tomcat	15	**Infinite wait bug (hang)**: A counter value is not updated correctly causing the request processing threads of Tomcat to hang (#53173).
Cassandra	19	**Infinite loop bug (hang)**: 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).
Hadoop	23	**Infinite read bug (hang)**: HDFS does not check for an overflow of an internal content length field causing HDFS transfers larger than 2GB to try to repeatedly read from an input stream until job time out (#HDFS-3318).
	24	**Thread shutdown bug (hang)**: 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).

TABLE 3
Descriptions of the 7 internal faults we tested.

performance related terms such as slowdown, hangs, and 100% CPU usage. We then follow the instructions given in the bug reports to reproduce the bugs.

Six of these seven internal software bugs are bugs which cause the system to hang and one bug causes a degradation in performance. We choose these bugs to evaluate PerfCompass in order to demonstrate that our tool is capable of handling bugs causing both a slowdown and system hang. We choose more bugs causing the system to hang because recent work [26], [53] has identified that those bugs are both difficult to debug and cause of a significant portion of problems in the cloud. Although previous work [43] has shown that performance anomalies can occur as the result of multiple bugs interacting with each other, we have evaluated our system using bugs in isolation. We discuss this further in Section 6.

We use Apache 2.2.22, MySQL 5.5.29, Tomcat 7.0.28, Cassandra 1.2.0-beta, and Hadoop 2.0.0-alpha for evaluating the external faults of those systems. For each internal fault, we used the version specified in the bug report. For all faults, PerfCompass is triggered when a performance anomaly is detected (e.g., response time $>$ 100 ms, progress score is 0), using the past 1.5 minutes of collected system call trace data for analysis. The accuracy of PerfCompass is not affected by the length of the history data as long as the data cover the fault manifestation. Our experiments show that 1.5 minutes is sufficient for all the faults we tested.

In order to evaluate PerfCompass under dynamic workloads with realistic time variations, we use the following workloads during our experiments.

- **Apache** - We employ a benchmark tool [1] to send requests to the Apache server at different intensity levels following the per-minute workload intensity observed in a NASA web server trace starting at 1995-07-01:00.00 [3].
- **MySQL** - We use a open source MySQL benchmark tool called "Sysbench" [2]. We use the *oltp* test in *complex* mode.
- **Tomcat** - We randomly request different example servlets

and JSPs included in Tomcat following the same workload intensity observed in the NASA web server trace [3] as Apache.
- **Cassandra** - We use a simple workload which creates a table and inserts various entries into the table.
- **Hadoop** - We use the Pi calculation sample application with 16 map and 16 reduce tasks.

PerfCompass has six parameters which need to be configured in order to achieve optimal results. They are the fault onset time threshold (α), the fault onset time dispersion threshold (β), the window size for calculating the moving average (k), the threshold for outlier detection (Θ), external fault impact factor threshold (δ), and the internal fault impact factor threshold (Δ). k, Θ, δ, and Δ are not sensitive and as a result, we use the same settings for different applications. We found that different α and β values are needed for each application to obtain optimal results.

During our experiments, the default parameter setting in PerfCompass is as follows: 1) k is 5; 2) Θ is mean + ($2 \times$ standard deviation); and 3) δ and Δ for differentiating external and internal faults are 90% and 80%, respectively. These settings are configurable and we have not attempted to calibrate or tune them to any system. We found those default settings work well for all the systems and faults we tested.

We obtained calibrated α and β values for each system using a simple external CPU cap fault (i.e., setting an overly low CPU cap) with different workload intensity levels from those used in the fault localization experiments[1]. We intentionally configure this way in order to evaluate whether our calibration requires the same workload as the production-run failure. Our results show that changing the workload intensity does not affect our fault localization results. The derived α are 60 ms for Apache and Tomcat, 300 ms for MySQL, and 400 ms for Cassandra and Hadoop. The calibrated β values are 7 ms for Apache, 17 ms for MySQL, 5 ms for Tomcat, 28 ms for Cassandra, and 39 ms for Hadoop.

We repeat each fault injection three times and report the mean and standard deviation of the fault impact factor values over all 3 runs.

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 Tomcat, 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.

4.2 Online Fault Localization Results

We now present our online fault localization and inference results. We begin by presenting our external and internal fault differentiation results. We then present our top affected system call ranking results followed by a sensitivity study on the key parameters of our system.

4.2.1 Fault Differentiation Results

Figures 4 shows the fault impact factor and fault onset time dispersion results using the calibrated fault onset time threshold values. As shown in Figure 4(a), the impact factors of all the external faults are greater than 90%, indicating them as external with no further analysis required. Similarly, 4 out of 7 of the internal faults are below the 80%, correctly indicating each case as internal. There are three borderline cases which require to use our fault onset time dispersion metric. Fault 15 is an internal Tomcat bug with an impact factor of 83%. Figure 4(b) shows the fault onset time dispersion values for the borderline cases we tested, normalized using the calibrated fault onset time dispersion threshold values. We observe that the fault onset time dispersion value of the internal case is significantly larger than the fault onset time dispersion threshold (i.e., $2\times$) for Tomcat, indicating it is an internal fault. Faults 23 and 24 have fault impact factors of about 80%, which makes them borderline cases as well. As shown in Figure 4(b), the fault onset time dispersion values for the two internal Hadoop cases are significantly higher than the fault onset time dispersion threshold (i.e., $2.3\times$,$2.2\times$) for Hadoop, correctly indicating both of them as internal bugs.

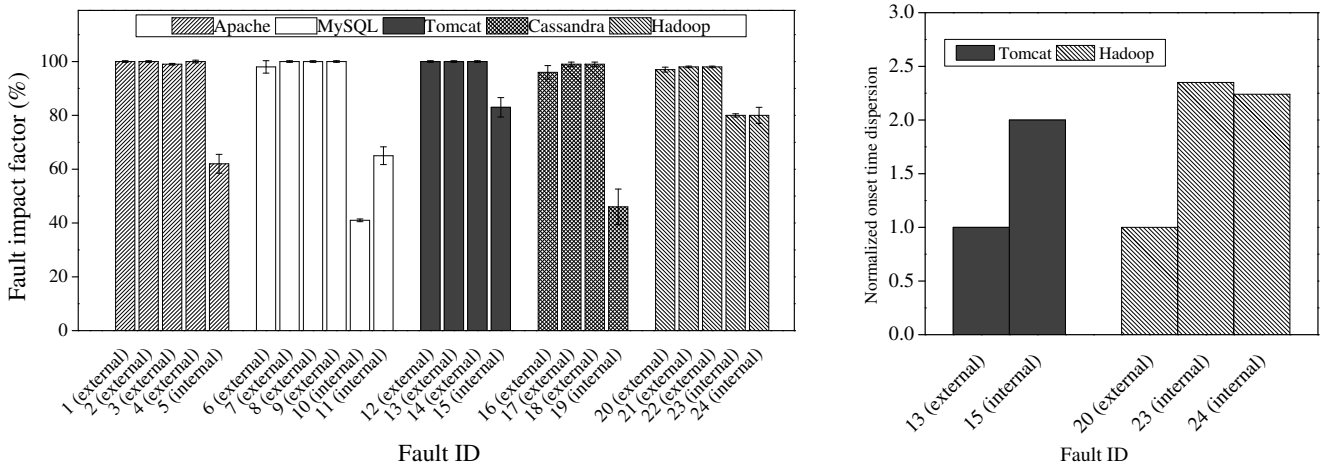
We also evaluated PerfCompass using a static 0.5 second fault onset time threshold. Due to space limitations, we discuss those results in Section 1.4 of the online supplementary material.

Setting a fault impact factor threshold of 90% would have allowed PerfCompass to correctly localize each fault as either external or internal without using our fault impact factor dispersion metric, thus reducing the overhead of PerfCompass. However, setting a relatively high fault impact factor threshold (e.g., 90%) would likely over-fit our scheme to our experimental data. This may increase the probability that PerfCompass would incorrectly localize an external fault as an internal fault for systems we have not tested. In addition, we want to avoid setting a hard threshold for distinguishing between internal and external faults (e.g., 90% being external and 89% being internal). As a result, we decided to use a conservative fault impact factor threshold and employ a hybrid two tier scheme to better capture the multi-dimensional difference between external and internal faults.

4.2.2 System Call Ranking Results

Tables 4 and 5 show the top three affected system calls identified by our ranking scheme for the external faults and internal faults, respectively. We consider both execution time and frequency changes while identifying top affected system calls. For example, "**Time:**`sys_write` (1)" means that `sys_write` is the highest ranked system call whose execution time increase is the largest. The last column in each table shows our inference results. For the external faults, this column describes how the affected system calls indicate which subsystems are most affected by the faults. For the internal faults, our system call ranking results give hints that may help

1. We used a static 1 second time segmentation gap threshold to segment each trace during calibration. We also tested other static time segmentation gap threshold values (e.g., 0.5 sec, 1.5 sec) and found it did not affect our calibration results.



(a) The computed τ values.

(b) The normalized Ω values for the borderline cases.

Fig. 4. The computed fault impact factor (τ) and normalized fault onset time dispersion (Ω) values using calibrated fault onset time threshold (α).

developers localize the buggy function. Note that PerfCompass is designed to be a light-weight online fault inference tool that focuses on providing useful clues for diagnosing both external and internal faults. Thus, we do not intend to use PerfCompass to pinpoint the exact root cause of a performance anomaly (e.g., buggy function name). We discuss this further in section 6. Due to space limitations, we only discuss a subset of our results in detail. We first discuss a subset external faults followed by a subset of internal faults.

Fault 8: This external fault is due to an incorrectly set external memory cap. The indicator of this fault is the `sys_fsync` system call. MySQL tries to keep as much of the database in physical memory as possible, flushing to disk only when necessary. The low amount of physical memory causes the system to use the disk more frequently, which can require additional memory to disk synchronization using the `sys_fsync` system call.

Fault 18: This is an external fault due to disk contention. We can see the execution time of `sys_futex` and `sys_write` are affected, indicating a blocking I/O problem. In turn, this indicates external disk contention.

Fault 21: This is an external fault due to packet loss. The increase in the execution time of `sys_futex` and `sys_socketcall` indicates a blocking network I/O problem. The execution time increase is a result of the dropped packets being re-transmitted.

Fault 5: Fault 5 is an internal CPU hog bug that occurs as a result of an infinite loop in which Apache attempts to make a blocking call on a socket. However, a flag preventing it from successfully making the call has not been cleared. The return value of the call is not checked correctly resulting in Apache re-trying the blocking call continuously. We found an increase in the frequency and execution time of the `sys_ipc` system call, which is a result of Apache repeatedly attempting to make the blocking call.

Fault 10: This internal fault is a deadlock bug caused by attempting to execute the INSERT_DELAYED statement on a locked table. The thread attempting to execute the INSERT_DELAYED statement tries to continuously access the locked table. We found an increase in the execution time of the `sys_select` and `sys_futex` system calls. `sys_select` is used to allow a program to monitor multiple file descriptors. Its execution time increase is due to the deadlocked function attempting to perform an I/O operation. Similarly, the execution time increase we observe with the `sys_futex` system call is a result of system trying unsuccessfully to acquire a lock on the already locked table. We also see an increase in the frequency of the `sys_stat64` system call, which is commonly used to first ensure a file exists before performing a requested I/O operation. This frequency increase is due to the repeated I/O attempts.

Fault 15: This is an internal fault caused by an incorrectly updated atomic counter value that is used to keep track of how many requests are being processed. This causes all other threads to hang indefinitely as the system incorrectly believes it cannot handle any more requests. Here we see an increase in both execution time and frequency of the `sys_futex` system call. This indicates a shared variable is being checked repeatedly.

Fault 23: This is an internal bug with HDFS where the system does not correctly check a header field. This causes HDFS to continuously attempt to read from a socket, waiting for the content that does not exist. This process continues until the job times out. Here, each of system calls affected is a result of the bug. The frequency increase of the I/O system calls, `sys_close`, `sys_read`, and `sys_stat64` are results of an infinite loop continuously attempting to read. We also see an increase in the execution time of the `sys_futex`, `sys_socketcall`, and `sys_epoll_wait` system calls as results of the read operations in the infinite loop.

System Name	Fault ID	Top affected system calls (rank)	Fault hints
Apache	1	**Time:**sys_waitpid(1),sys_mmap_pgoff(2),sys_socketcall(3) **Frequency:**None	Execution time increase of sys_waidpid indicates a CPU contention problem.
	2	**Time:**sys_clone(1),sys_mmap_pgoff(2),sys_socketcall(3) **Frequency:**None	Execution time increase of sys_mmap_pgoff and sys_clone indicates a memory allocation problem.
	3	**Time:**sys_select(1),sys_fcntl64(2),sys_socketcall(3) **Frequency:**sys_fcntl64(1)	Increase in frequency of sys_fcntl64 along with increase in execution time of sys_select indicates a disk contention problem.
	4	**Time:**sys_socketcall(1),sys_open(2),sys_close(3) **Frequency:**None	sys_socketcall is used for network I/O. sys_open and sys_close also related to sending data over the network. This indicates a non-blocking network I/O problem.
MySQL	6	**Time:**sys_futex(1),sys_read(2) **Frequency:**None	Increase in execution time of sys_futex and sys_read indicates blocking I/O problem.
	7	**Time:**sys_futex(1) **Frequency:**None	sys_futex affected by itself indicates issue related to locking. Indicates possible CPU contention or memory problem. Lack of any other system calls being affected indicates a CPU contention problem.
	8	**Time:**sys_futex(1),sys_fsync(2),sys_socketcall(3) **Frequency:**None	sys_fsync used to flushes buffer cache data to disk, which can occur when low on memory. Indicates a memory related problem.
	9	**Time:**sys_futex(1),sys_socketcall(2) **Frequency:**None	Increase in execution time of sys_futex and sys_socketcall indicates a blocking network I/O problem.
Tomcat	12	**Time:**sys_futex(1),sys_socketcall(2) **Frequency:**sys_stat64(1),sys_futex(2)	sys_socketcall and sys_futex execution time increase indicates a blocking network I/O problem.
	13	**Time:**sys_futex(1),sys_sched_getaffinity(2),sys_write(3) **Frequency:**sys_futex(1)	Execution time and frequency increase of sys_futex along with execution time increase of sys_sched_getaffinity indicates a CPU contention problem.
	14	**Time:**sys_futex(1),sys_socketcall(2), sys_stat64(3) **Frequency:**sys_sched_yield(1)	Increase in frequency of sys_sched_yield indicates increase in frequency of context switching, indicates a memory problem.
Cassandra	16	**Time:**sys_futex(1),sys_poll(2),sys_socketcall(3) **Frequency:**sys_close(1),sys_unlink(2),sys_futex(3)	Increase in execution time of sys_futex and sys_poll indicates a CPU contention problem.
	17	**Time:**sys_futex(1),sys_socketcall(2),sys_sched_getaffinity(3) **Frequency:**sys_close(1),sys_futex(2),sys_sched_yield(3)	Increase in execution time of sys_socketcall and sys_futex indicates a blocking network I/O problem.
	18	**Time:**sys_futex(1),sys_poll(2),sys_write(3) **Frequency:**sys_close(1),sys_futex(2)	Increase in execution time of sys_futex and sys_write indicates a blocking I/O problem. This indicates a disk contention problem.
Hadoop	20	**Time:**sys_futex(1),sys_socketcall(2),sys_epoll_wait(3) **Frequency:**sys_close(1),sys_open(2)	Increase in execution time of sys_epoll_wait and sys_futex indicates a CPU contention problem.
	21	**Time:**sys_futex(1),sys_socketcall(2),sys_read(3) **Frequency:**sys_close(1),sys_read(2)	sys_futex and sys_socketcall execution time increase indicates a blocking network I/O problem.
	22	**Time:**sys_futex(1),sys_epoll_wait(2),sys_read(3) **Frequency:**sys_close(1),sys_read(2)	Increase in execution time and frequency of sys_read indicates a disk contention problem.

TABLE 4
System call ranking results for the external faults we tested.

4.2.3 Filtering Mode

We now show the calibrated impact factor values with and without filtering mode enabled in Figure 5. The cases shown are external cases with highly ranked I/O related system calls. As shown, PerfCompass is still able to correctly quantify the global vs. local impact of most faults without filtering mode enabled. However, case 17 can be considered a borderline case as a result of the error bars of the average impact factor. When we apply filtering mode to that case, however, we find the impact factor increases substantially from 91+/-6.7% to 99+/-0.6%. Although this is the only case which shows a significant benefit from filtering mode, we believe this mode can be useful in allowing developers more flexibility when defining the fault impact factor threshold value.

4.2.4 Sensitivity Study

We conducted a sensitivity study in order to determine how different configuration parameters affect PerfCompass which we include in Section 1.1 of the online supplemental material due to space limitations.

4.3 PerfCompass Overhead

We evaluated the overhead imposed by PerfCompass during normal application execution which we present in Section 1.2 of the online supplemental material due to space limitations.

System Name	Fault ID	Top affected system calls (rank)	Fault hints
Apache	5	**Time:**sys_ipc(1),sys_close(2),sys_waitpid(3) **Frequency:**sys_ipc(1), sys_open (2)	Frequency increase of sys_ipc indicates significant increase in inter-thread signalling. *Hints: blocking operation (e.g., blocking socket call) in an infinite loop.*
MySQL	10	**Time:** sys_select(1),sys_futex(2),sys_fsync(3) **Frequency:**sys_stat64(1), sys_open(2),sys_read(3)	Increase in execution time of sys_futex and frequency of I/O related system calls indicate repeated attempts to acquire lock and perform I/O.*Hints: possible deadlocked I/O functions.*
	11	**Time:**sys_futex(1),sys_select(2),sys_socketcall(3) **Frequency:**None	Execution time increase of sys_select and sys_futex indicate increase of writing and reading to disk. *Hints: possible problem with data flushing functions.*
Tomcat	15	**Time:**sys_futex(1),sys_close(2),sys_pipe(3) **Frequency:**sys_close(1),sys_open(2),sys_futex(3)	Increase in execution time and frequency of sys_futex indicates issue with locking operations. *Hints: shared variable checking functions.*
Cassandra	19	**Time:**sys_futex(1),sys_fstat64(2),sys_stat64(3) **Frequency:**sys_close(1),sys_lstat64(2),sys_unlink(3)	Increase in execution time of sys_futex, sys_fstat64, and sys_stat64 along with frequency increase of sys_close indicates significant increase in I/O operations. *Hints: I/O functions using locking operations in an infinite loop.*
Hadoop	23	**Time:**sys_futex(1),sys_socketcall(2),sys_epoll_wait(3) **Frequency:**sys_close(1), sys_read(2), sys_stat64(3)	Increase in execution time of sys_futex, sys_socketcall, and sys_epoll_wait. along with increase in frequency of sys_close, sys_read, and sys_stat64 indicates issue with network I/O operation. *Hints: I/O operation involving functions reading over the network.*
	24	**Time:**sys_futex(1),sys_socketcall(2),sys_epoll_wait(3) **Frequency:**sys_close(1),sys_stat64(2),sys_rt_sigaction(3)	Increase of sys_futex and sys_epoll_wait indicates issue with locking operations. *Hints: shared variable checking functions.*

TABLE 5
System call ranking results for the internal faults we tested.

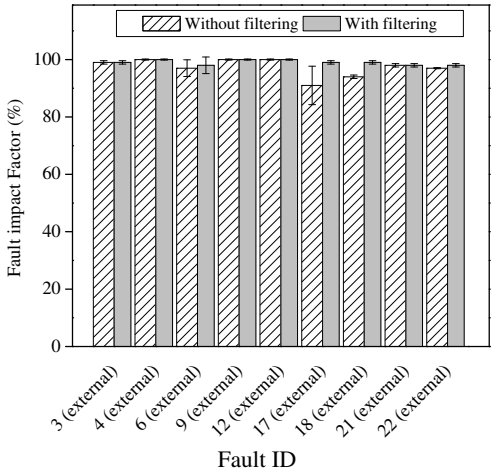


Fig. 5. The filtering mode effect using calibrated fault onset time thresholds for different faults where the filtering mode is triggered.

5 Related Work

In our previous work [19], we presented a preliminary version of PerfCompass. This paper extends our previous work in several major aspects. First, we provide a system call ranking scheme to extract useful hints about both external and internal faults. Second, we designed a robust execution unit extraction scheme that makes PerfCompass work well for different types of server systems and faults. Third, we performed a more thorough evaluation by extending our test cases from 16 faults to 24 faults.

Previous work has proposed to develop performance debugging tools using system event traces such as system system call traces and hardware performance counters. Fournier et al. [24] collected system level event traces and built a state transition model to analyze the blocking behavior on multi-core systems. Ding et al. [21] used system calls and command line options to build normal application signatures for diagnosing previously known problems. Shen et al. [41] construct change profiles to detect system anomaly symptoms by checking performance deviation between reference and target executions. DeepDive [36] first clusters low-level metrics to identify potential interference in virtualized environments. Potential interferences are then confirmed or denied by comparing the VM performance in isolation with that in production. Abhishek et al. [40] uses Auto-Regressive model to detect time-invariant relationships from monitoring data, and analyzes broken invariants during runtime to localize the fault. Lan et al. [31] use principle component analysis and independent component analysis along with outlier detection to automatically find faulty components in large-scale systems. In contrast, our work focuses on distinguishing faults with a globally direct impact from faults with a locally direct impact, which is critical for ef-

ficiently diagnosing performance anomalies in shared hosting infrastructures. Moreover, our approach does not assume any prior knowledge about the diagnosed problem. Thus, we can diagnose emergent production problems that are previously unseen.

Fay [22] uses probes, which are kernel drivers or DLLs, along with hotpatching to collect and summarize detailed user-mode and kernel level traces without modifying the underlying system. Although Fay is able to effectively determine performance problems, it does so through a combination of user space and kernel level tracing, imparting up to 280% overhead in the process. In contrast, our approach relies on only kernel level events, imparting 0.8-3.3% overhead in doing so. Thus, our approach is more suitable for diagnosing production-run problems within the large-scale hosting infrastructure.

Magpie [13] instruments middleware and packet transfer points in order to generate a series of low level network events. They then group these events together into requests via a series of temporal joins and finally cluster together requests based on behavior. DARC [47] identifies functions that are the main latency contributors to peaks in a OSProf profile. S^2E [17] uses selective symbolic execution and dynamic binary instrumentation to perform in-vivo multi-path analysis for finding bugs and profiling performance. The above approaches require instrumentations to the application or the middleware, which is difficult to be deployed in the production hosting infrastructure. In contrast, PerfCompass is designed to run in the production hosting infrastructure without any application or middleware instrumentation.

Researchers also proposed white-box approach to analyzing performance bugs. Jin et al. [28] 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. CloudDiag [33] automatically identifies and ranks anomalous function calls using robust principle component analysis along with white-box instrumentation to provide fine-grained performance debugging information to developers. However, it is difficult to obtain the source code access for production systems running inside the virtualized hosting infrastructures. Thus, PerfCompass strives to perform online performance anomaly localization and inference without requiring source code.

Previous work has analyzed RPC messages to identify faulty components. For example, Aguilera et al. [4] analyze RPC traces to infer causal paths for debugging. Similarly, WAP5 [38] links logical messages between different components in distributed systems in order to identify causal paths for debugging. In contrast, PerfCompass uses system call traces to achieve fine-grained fault localization within each faulty component identifying whether the fault comes from the environment or the component software itself and extracting hints about the fault. Previous black-box fault localization schemes cannot provide that type of fine-grained diagnosis.

The idea of tracing distributed systems and networks for debugging has been well studied [52], [39], [37], [23], [16], [15], [12], [27]. These systems perform network level tracing in order to infer paths or dependency information for diagnosing large-scale distributed systems. In comparison, our work focuses on fine-grained root cause inference within a single server, which is complementary to those network tracing approaches.

Much work has been done to diagnose configuration problems. X-ray [11] uses Pin to dynamically instrument binaries 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). Autobash [44] proposed to use the pattern of success and failure of known predicates to diagnose misconfigurations. Chronus [51] periodically checkpoints disk state and automatically searches for configuration changes that may have caused the misconfigurations. PeerPresure [49] and Strider [50] identify differences in configuration states across several different machines to diagnose misconfigurations on an affected machine. Our work is complementary to the above work by focusing on performance anomalies caused by environment issues or internal software bugs.

Record and replay techniques [42], [25], [30] are useful for system performance diagnosis. TRANSPLAY [45] uses partial checkpointing and deterministic record-replaying to capture minimum amount of data necessary to reexecute just the last few moments of the application before it encountered a failure. Triage [48] leverages checkpoint-replay to perform just-in-time problem diagnosis by repeatedly replaying a failure moment on the production site. X-ray [11] leverages record and replay technique to offload the heavy analysis on the replay computer. However, production site application recording and replay faces deployment challenges (e.g., requiring sensitive inputs) and cannot support online diagnosis. In comparison, PerfCompass does not require application record and replay to achieve easy deployment for production systems.

6 Discussion and Future Work

PerfCompass assumes that internal faults will only affect a subset of threads when they are triggered. Although this assumption is true for most cases, it might not hold for the memory leak bug because the system will not be affected by the bug until thrashing is triggered. Once thrashing occurs, the memory leak can cause a global impact to the system. Fortunately, memory leak bugs can be easily detected by monitoring system level metrics (e.g., memory consumption, CPU usage) using existing online black-box detection and diagnosis tools [18], [46].

Although PerfCompass can hint at the possible root cause of different problems based on the top affected system calls, we do not claim our approach to be free of false positives or intend to obviate the need for developers. Instead, we believe that those hints provided by PerfCompass should be combined with developer knowledge to make production run anomaly debugging easier.

Each external environment issue or internal software bug we have used to evaluate PerfCompass was triggered in isolation. It will be interesting to study the impact of concurrent faults, which however, is beyond the scope of this paper and

will be part of our future work. We have conducted a set of initial experiments to understand the behavior of PerfCompass under concurrent faults. First, we ran an experiment to examine the combined effect of an external CPU cap fault and an internal infinite loop bug. We found that 100% of the threads were affected, causing PerfCompass to first localize the issue to be external and thereby trigger external fault handling mechanisms (e.g., live VM migration). Additionally, we found that CPU related system calls were ranked high which gives guidance on where to migrate to. After migration is complete and the external issue is no longer present, the infinite loop bug is then correctly localized as internal. Second, we ran an experiment to evaluate PerfCompass using concurrent CPU and memory contention faults. Specifically, we triggered an overly low CPU fault and an overly low memory fault concurrently. PerfCompass calculates a fault impact factor of 100%, correctly indicating an external issue. Also, we found system calls related to memory were ranked the highest because the memory cap fault has the dominating impact in this case.

7 Conclusions

In this paper, we have presented PerfCompass, a novel online performance anomaly fault localization and inference tool for production cloud infrastructures. PerfCompass can efficiently distinguish between external and internal faults without requiring source code or runtime instrumentation. PerfCompass extracts fine-grained fault features from kernel-level system call traces and uses those fault features to perform online fault localization. PerfCompass can also provide useful hints about both external and internal faults by identifying top affected system calls. By focusing on kernel-level system call events, PerfCompass can be applied to different programs written in C/C++ or Java. We have implemented PerfCompass and evaluated it using a variety of commonly used open source server systems including Apache, MySQL, Tomcat, Cassandra, and Hadoop. We tested PerfCompass using 24 real cloud faults consisting of common environment issues in the shared cloud infrastructure and real performance bugs. Our results show that our fault differentiation scheme can achieve 100% fault localization accuracy and provide useful fault hints. PerfCompass is light-weight, which makes it practical for use in production cloud infrastructures.

References

[1] Funkload. http://funkload.nuxeo.org/.

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

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

[4] 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.

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

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

[7] Apache. http://httpd.apache.org/.

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

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

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

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

[12] 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.

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

[14] P. Barham, B. Dragovic, K. Fraser, S. H, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

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

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

[17] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

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

[19] Daniel J. Dean, Hiep Nguyen, Peipei Wang, and Xiaohui Gu. PerfCompass: Toward runtime performance anomaly fault localization for infrastructure-as-a-service clouds. In *HotCloud*, 2014.

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

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

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

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

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

[25] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global comprehension for distributed replay. In *NSDI*, 2007.

[26] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. What bugs live in the cloud?: A study of 3000+ issues in cloud systems. In *SoCC*, 2014.

[27] 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.

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

[29] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, 2007.

[30] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *SIGMETRICS*, 2010.

[31] Zhiling Lan, Ziming Zheng, and Yawei Li. Toward automated anomaly identification in large-scale systems. *Parallel and Distributed Systems, IEEE Transactions on*, 2010.

[32] Kaituo Li, Pallavi Joshi, Aarti Gupta, and Malay K Ganai. Reprolite: A lightweight tool to quickly reproduce hard system bugs. In *SoCC*. ACM, 2014.

[33] Haibo Mi, Huaimin Wang, Yangfan Zhou, M.R. Lyu, and Hua Cai. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 2013.

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

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

[36] D. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *USENIX ATC*, 2013.

[37] 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.

[38] 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.

[39] 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.

[40] 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.
[41] K. Shen, C. Stewart, C. Li, and X. Li. Reference-driven performance anomaly identification. In *SIGMETRICS*, 2009.
[42] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flashback: A light-weight rollback and deterministic replay extension for software debugging. In *USENIX ATC*, 2004.
[43] Christopher Stewart, Kai Shen, Arun Iyengar, and Jian Yin. Entomomodel: Understanding and avoiding performance anomaly manifestations. In *MASCOTS*, 2010.
[44] Y. Y. Su, M. Attariyan, and J. Flinn. Autobash: improving configuration management with operating system causality analysis. In *SOSP*, 2007.
[45] D. Subhraveti and J. Nieh. Record and transplay: partial checkpointing for replay debugging across heterogeneous systems. In *SIGMETRICS*, 2011.
[46] 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.
[47] A. Traeger, I. Deras, and E. Zadok. DARC: Dynamic analysis of root causes of latency distributions. In *SIGMETRICS*, 2008.
[48] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: diagnosing production run failures at the user's site. In *SOSP*, 2007.
[49] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y. M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *OSDI*, 2004.
[50] Y. M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. Strider: A black-box, state-based approach to change and configuration management and support. In *LISA*, 2003.
[51] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *OSDI*, 2004.
[52] 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.
[53] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *OSDI*, 2014.
[54] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *EuroSys*, 2013.

Daniel J. Dean is a PhD student in the Department of Computer Science at North Carolina State University. He received a BS and MS in computer science from Stony Brook University, New York in 2007 and 2009 respectively. He has interned with NEC Labs America in the summer of 2012 and IBM Research in the summer of 2013. Daniel is a student member of the IEEE.

Hiep Nguyen is a software engineer at Microsoft. He received his PhD from North Carolina State University in 2014 and a BS from Hanoi University of Technology, Vietnam in 2008. He has interned with NEC Labs America, Microsoft.

Peipei Wang is a first year Computer Science Phd student at North Carolina State University. Peipei got her master's degree in Computer Science and bachelor's degree in Software Engineering from Xi'an Jiaotong University, China. Peipei was a summer intern at Morgan Stanley IT Department in 2012. Her research interest is solving system diagnosis problems in cloud computing and big data.

Xiaohui Gu is an associate professor in the Department of Computer Science at the North Carolina State University. She received her PhD degree in 2004 and MS degree in 2001 from the Department of Computer Science, University of Illinois at Urbana-Champaign. She received her BS degree in computer science from Peking University, Beijing, China in 1999. She was a research staff member at IBM T. J. Watson Research Center, Hawthorne, New York, between 2004 and 2007. She received ILLIAC fellowship, David J. Kuck Best Master Thesis Award, and Saburo Muroga Fellowship from University of Illinois at Urbana-Champaign. She also received the IBM Invention Achievement Awards in 2004, 2006, and 2007. She has filed nine patents, and has published more than 60 research papers in international journals and major peer-reviewed conference proceedings. She is a recipient of NSF Career Award, four IBM Faculty Awards 2008, 2009, 2010, 2011, and two Google Research Awards 2009, 2011, best paper awards from ICDCS 2012 and CNSM 2010, and NCSU Faculty Research and Professional Development Award. She is a Senior Member of IEEE.

Anca Sailer Dr. Sailer received her Ph.D. in Computer Science from UPMC Sorbonne Universits, France, in 2000. She was a Research Member at Bell Labs from 2001 to 2003 where she specialized in Internet services traffic engineering and monitoring. In 2003 Dr. Sailer joined IBM where she is currently a Research Staff Member and IBM Master Inventor in the Service Research Group. She was the architect lead for the automation of the IBM Smart Cloud Enterprise Business Support Services which earned her in 2013 the IBM Outstanding Technical Achievement Award. Dr. Sailer holds over two dozen patents, has co-authored numerous publications in IEEE and ACM refereed journals and conferences, and co-edited two books, one on multicast and one on problem determination technologies. Her research interests include cloud computing business and operation management, self-healing technologies, and data mining. She is a Senior Member of IEEE.

Andrzej Kochut is a Research Staff Member and Manager at the IBM T. J. Watson Research Center. His research interests are in computer systems, networking, stochastic modeling and performance evaluation with recent focus on virtualization and cloud computing. Dr. Kochut is an active author and member of program committees of IEEE and ACM conferences and was also recognized with two Best Paper Awards. Dr. Kochut received his Ph.D. in Computer Science from the University of Maryland, College Park in 2005.