IASO: A Fail-Slow Detection and Mitigation Framework for Distributed Storage Services

Biswaranjan Panda, Deepthi Srinivasan, Huan Ke∗, Karan Gupta, Vinayak Khot, and Haryadi S. Gunawi∗

Nutanix Inc. University of Chicago∗

Abstract

We address the problem of “fail-slow” fault, a fault where a hardware or software component can still function (does not fail-stop) but in much lower performance than expected. To address this, we built IASO, a peer-based, non-intrusive fail-slow detection framework that has been deployed for more than 1.5 years across 39,000 nodes in our customer sites and helped our customers reduce major outages due to fail-slow incidents. IASO primarily works based on timeout signals (a negligible overhead of monitoring) and converts them into a stable and accurate fail-slow metric. IASO can quickly and accurately isolate a slow node within minutes. Within a 7-month period, IASO managed to catch 232 fail-slow incidents in our large deployment field. In this paper, we have also assembled a large dataset of 232 fail-slow incidents along with our analysis. We found that the fail-slow annual failure rate in our field is 1.02%.

1 Introduction

Maintaining high availability of distributed storage services in real deployment fields is challenging due to the various types of faults that can occur. In the last few years, there has been an emphasis on “fail-slow” fault mode [28, 32]. This means that a hardware or software component can still function (does not fail-stop) but in much lower performance than expected. Such faults have been studied under different names such “gray failure” [32], “limping” [24, 37], and “partial failures” [29]. We chose the term “fail-slow” for simplicity and reflecting a recent term [28].

The urgency here is that many distributed systems are still designed based on a binary model of no failure and fail-stop failures. Recent works shows that many distributed systems cannot gracefully tolerate fail-slow mode, i.e. the system cannot isolate and hide a fail-slow component, causing latency spikes or throughput degradation to users [24, 28, 31, 32, 56]. Worse, it has been reported that a fail-slow component can cause cascade of performance failures across the cluster, bringing down services for hours [24, 28]. This calls for the importance of designing systems that tolerate not just absolute failure of sub-components but can also gracefully handle the occurrence of performance faults.

In this context, our work in this paper makes the two following contributions:

(1) Design and implementation of a fail-slow mitigation framework. The first contribution of the paper is IASO, our peer-based, non-intrusive fail-slow detection framework that has been deployed for more than 1.5 years across 39,000 nodes in our customer sites. Before the integration with IASO we had more than 25 full outages (IOPS went to zero) due to cascading impacts of fail-slow incidents, not to mention many other occurrences of partial slowdowns. Since the integration with IASO, we had only 2 major outages (false negative cases) caused by fail slow.

Motivation: IASO is motivated by the following reasons.

First, we found that fail-slow faults can be caused by many root causes. Sole dependence on low-level detection tools [38, 40, 15, 4] at various levels of the software and hardware stack might not be sufficient. Thus, we need a fail-slow detection system that works at the service (distributed system) level. Most existing work focuses on hardware level outlier detection or software performance bugs but they might not cover all of the detailed root causes occurring in the field (§4.2.3).

Second, most existing efforts focus only on detection but not mitigation. We are only aware of a handful of works that perform mitigation in real deployments (more in §5). Yet, our findings suggest that if fail-slow incidents are not quickly and automatically isolated, it can cascade and directly affect users for hours or days. For this reason, it is paramount that deployed systems are equipped with fail-slow mitigation.

Third, although some computing frameworks such as MapReduce [1, 23] are equipped with fail-slow mitigation (e.g., via speculative execution [38] or cloning [10]), the tail tolerance is built in their abstractions (e.g., “jobs”, “tasks”) and not directly generalizable to many other distributed systems. Recent works revealed that many other distributed systems are still not fail-slow tolerant [24, Figure 1][56, Figure 12]. Hence, we need a more general way of addressing fail-slow faults in many distributed storage services.
Challenges and solutions: A fail-slow detection framework must be non-intrusive (negligible overhead), stable and accurate, and not accidentally make wrong decisions (e.g., quarantine healthy nodes). To achieve this, we make IASO peer-based, i.e., a slow service instance should be compared against its peers of the same service (e.g., the performance of Cassandra instance should not be compared to ZooKeeper’s). We also make IASO load aware, i.e., the relative performance of a service instance must not be improved or worsen just because the load on the node on which the instance is running on is different.

To achieve all of these, we created an algorithm (§2.2) that can work solely based on timeout signals. Our algorithm can convert timeout and successful-response statistics into a stable and accurate fail-slow detector. Our framework does not need to monitor every request latency, hence achieving a negligible overhead. IASO can quickly and accurately isolate a slow node within minutes. Within a 7-month period, IASO managed to catch 232 fail-slow incidents in our large deployment field. IASO also automatically quarantined the slow nodes and restored the clusters back to a healthier performance. We only encountered 9 confirmed false positives. Other false positives are because the fail-slowness disappeared when our engineers started diagnosing them (e.g., perhaps caused by unknown external conditions).

(2) A dataset and analysis of fail-slow incidents With IASO integration, we were able to capture many fail-slow incidents in the field. We have assembled a large dataset of fail-slow incidents along with our analysis [7]. To the best of our knowledge, this is the largest dataset of fail-slow cases publicly reported from within a company. Furthermore, existing accounts of fail-slow accidents are anecdotal [12, 28, 32], while our contribution includes some quantitative analysis (e.g., AFR, age correlation).

The dataset: The dataset contains 232 validated cases collected from the deployment of 39,000 nodes throughout a period of 7 months. This data pertains to a type of fully hyperconverged system [9] that we deploy in customer sites.

Findings: Our rich dataset allows us to make some statistical findings. First, given 232 independent cases across 39,000 nodes over 7 months, we can derive that the annual failure rate is 1.02% (232 × 12 / 7 / 39,000), which is relatively significant compared to rates of other types of faults (§4.2.1). Second, we uncovered a wide range of root causes (and the low-level sub-causes), which again accentuates the need for detection at the service level, not just at the individual hardware level. Third, we also observed the “infant mortality” pattern where younger machines exhibit more fail-slow incidents. Fourth, we show that if not mitigated properly, fail-slow cases can take hours or days to fully resolve, which again highlights the importance of automatically quarantining slow nodes.

The following sections present the design and implementation of IASO (§2), experimental results (§3), our dataset and findings (§4), related work and conclusion.

2 IASO

This section presents IASO, our framework for detecting the presence of an unhealthy node and enabling self healing of the cluster. We name our system after “Iaso”, the Greek goddess of recuperation from illness [8]. IASO is comprised of three stages:

1. Detection (§2.1-2.2): This step reduces the time to detect fail-slow incidents from hours to minutes while keeping false positives low.

2. Mitigation (§2.3): This step quarantines the faulty node and brings the cluster back to operation.

3. Resolution (§2.4): IASO automatically pages site reliability engineers (SREs) to identify the failed component and help support to do breakfix and assimilate the fixed component back into operation.

When building IASO, we adhere to the following design principles.

- Non intrusive: We attempt to reach a near 0% overhead, hence we use raw metrics that the deployed services already collect (e.g., number of timeouts and successful responses).
- Peer based: A slow service instance should be compared against its peers of the same service, e.g., the performance of Cassandra instance should be compared to other Cassandra instances, not ZooKeeper instances, as different types of services observe different types of workload. For this reason, we monitor at service-level requests, not at OS or hardware level.
- Load aware: The slowdown detection system must be aware of the service load. The relative performance of a peer must not be improved or worsen just because the load on the node the peer is running on is different. This means that the performance of a node must be normalized based on the capacity of the node; in our deployment, a cluster can have different machine capacities with different loads.
- Stable and accurate: As a degraded node will be quarantined, it is important to have a stable and accurate algorithm that does not accidentally make wrong decisions (false positives).

---

1 For this publication we only have analyzed the dataset for a 7 month period in 2017. Data from 2018 is still being perused and cleaned.
The next section (we can reach 20,000 IOPS [5]). Sampling can be a solution, but vice versa is not amenable (every request. However, with today’s high-throughput service itself (a nearly 0% overhead as counting timeouts and monitoring system is not intrusive to the performance of the time). Another advantage of using timeouts is that our responses were successful as well as the failed ones due to Sandra already have a built-in metric that collects how many responses is a simple increment operation).

### Figure 1: IASO components. The figure is described in the last paragraph of page 2 and also in Section 2.1. “Scr” denotes the hook that sends score table to ScoreDB.

The following are the terms we use in this paper. As shown in Figure 1, our system $S$ is a cluster of high-end machines (gray shades) running VMs wherein services are running (boxes). For example, $S$ comprises a ZooKeeper (ZK) service for cluster configuration manager, a Cassandra service (CS) for storing metadata, and our own blob-store service for storing data. Each VM runs an instance of each of the services (e.g., a VM runs three service instances, Cassandra, ZooKeeper and blob-store instances). These VMs are also known as controller VMs.

### 2.1 Detection

Our first goal is to detect which service instance is experiencing a slowdown. Currently we only address persistent fault, i.e., the instance is not being slowed down due to an intermittent condition such as a one-off high GC time. This section describes the main components of IASO as shown in Figure 1. The next section (§2.2) presents the detailed algorithm.

**RAW METRICS (LATENCY VS. TIMEOUTS):** One naive method to measure degradation is to measure the latency of every request. However, with today’s high-throughput services it is not amenable (e.g., per-node Cassandra throughput can reach 20,000 IOPS [5]). Sampling can be a solution, but we explored a different method.

In this work, we try a much cheaper method to detect degradation: counting timeouts. Many services such as Cassandra already have a built-in metric that collects how many responses were successful as well as the failed ones due to timeouts. Another advantage of using timeouts is that our monitoring system is not intrusive to the performance of the service itself (a nearly 0% overhead as counting timeouts and successful responses is a simple increment operation).

**SCORES:** We found that using raw timeout counts as a direct metric to measure outlier is not a stable and accurate way. Thus, we need to introduce the concept of “score”. Given a cluster of $N$ nodes with $N$ instances of a service, every instance can observe the performance of its $N−1$ peers and maintain a “score table” (as shown in Figure 1).

**STABLE SCORES:** The primary challenge we address in this work is how to convert timeout and success statistics into a stable and accurate degradation detector. Noisy scores can lead to more false positives where healthy nodes might be accidentally removed and vice versa. Later, the experiment section shows other unsuccessful algorithms that we tried (§3) which then led us to the current algorithm (§2.2).

One key to prevent scores from fluctuating along with the number of timeouts is by incorporating additive increase and multiplicative decrease (AIMD) [18] such as used in TCP congestion avoidance. Thus, our custom algorithm employs a technique similar to AIMD.

**SCOREDB server:** The scores collected from the service instances are stored in a database server called ScoreDB (Figure 1). For every peer, every instance keeps a score, hence in total ScoreDB maintains $N×(N−1)$ score variables (per every service monitored) including their historical values. Given these scores, ScoreDB runs an outlier detection part of our algorithm and quarantines the outlier. ScoreDB is also a replicated system (to anticipate degradation within itself).

### 2.2 Detection Algorithm

We now describe how IASO calculates the score metric and detects an outlier. The challenge is to convert timeout and success statistics into a stable and accurate degradation detector. For every equation listed below, the explanation is in the paragraph preceding the equation. Symbols $⇌$ and $⇒$ are used for backward references.

#### 2.2.1 Peer Scores

Given a cluster of $N$ instances within a service (e.g., Cassandra), every instance observes the performance of its peers and puts the corresponding scores in a score table containing $N−1$ peer scores. In our scoring system below, a score ranges from 1 to 100 where a higher value implies more severe degradation. For example, in the score table in Figure 1, Cassandra instance on node2 believes that Cassandra instance on node1 is slow (a score of 98).

As score continues to change, below we use $\text{prev}$ and $\text{score}$ to represent the scores in the last and current epoch respectively. An epoch is the interval at which every service instance runs the equations below (i.e., calculates a new score). The epoch is set to be 5 seconds and $\text{prev}$ to 1 in the beginning.

Next, we introduce $\text{ToRespRatio}$, the ratio of the number of timeouts and total responses between two peers within an epoch. This is essential to the load-awareness part of our algorithm, that timeout counts should be relative to the number.
number of total responses as the number of responses will vary across peers.

\[ \text{ToRespRatio} = \frac{\#\text{timeouts}}{\#\text{responses}} \]

We then set \( \text{ratioThresh} \), a timeout-response ratio threshold, with a constant value of 0.1 (e.g., 10 timeouts for every 100 responses). In our experience, 10% timeouts from a peer can cause a whole-cluster degradation. A higher value may make IASO react too late, while a lower may lead to more false positives (i.e., too sensitive). If \( \text{ToRespRatio} \) is larger than the \( \text{ratioThresh} \), it is likely a heavy degradation. Otherwise, it is likely caused by a temporary high load or a benign cause.

\[ \text{ratioThresh} = 0.1 \]

Next, we introduce \( \text{minTTR} \) as the minimum time to observe zero timeout from a peer before the score assigned to it decreases from 100 to 1 (slow to healthy). We set the time to be 2 minutes. The idea is that when a peer exhibits a zero timeout, it might mean that this peer is temporarily healthy but might suffer degradation again soon. The 2-minute mark is the time window in which a peer must “prove” itself that it is really healthy. A smaller window increases the risk that we may start assigning good scores to a temporarily good peer and thereby creating an unstable score pattern. A larger window has the disadvantage that we may mark a peer as fail-slow even if it has just completely recovered but the 2-minute window hasn’t passed. However, the latter scenario should be infrequent.

\[ \text{minTTR} = 2 \text{ mins} \]

With all of the values above, now we can stitch them into the score calculation. In every epoch, if \( \text{ToRespRatio} \) is 0 (no timeout), then the score will be calculated as shown below. This is the “additive decrease” part of our algorithm – the score will be slowly decreasing back to zero to show that the peer is really healthy.

\[
\begin{align*}
[ \text{if ToRespRatio is 0} ] \\
\text{score} &= \text{prev} - \left( \frac{100 \times \text{epoch}}{\text{minTTR}} \right)
\end{align*}
\]

Now, we discuss the case where some timeouts are observed (\( \text{ToRespRatio} \) is not zero). First, we introduce \( \text{minRatio} \) as a higher bound of the timeout-response ratio and threshold values. The idea here is that \( \text{ToRespRatio} \) can be very high (e.g., 90%, when a peer is highly unresponsive). This high value will make our algorithm below unstable. Thus, we cap it to the \( \text{ratioThresh} \) value (0.1), i.e., 10% already represents enough degradation.

\[ \text{minRatio} = \min(\text{ToRespRatio}, \text{ratioThresh}) \]

Finally, the last variable we introduce is \( \text{nearThresh} \) to measure how far the timeout-response ratio to the threshold (how far from the 10% timeouts). This threshold nearness ranges from 0 to 1.0.

\[ \text{nearThresh} = \min\text{Ratio} / \text{ratioThresh} \]

With all the new variables above, we now can calculate the score when timeout-response ratio is higher than zero. The equation below represents the “multiplicative increase” part of our algorithm where the score is increased by the threshold nearness. We put more examples below.

\[
\begin{align*}
[ \text{if ToRespRatio is not 0} ] \\
\text{score} &= \text{prev} + (\text{prev} \times \text{nearThresh})
\end{align*}
\]

Let’s use an example where an instance gave a score of 32 for a peer instance in the last epoch. Now, the current epoch sees too many timeouts beyond the threshold such that \( \text{nearThresh} \) is 1.0. Thus, the current score will jump from 32 to 32+32 (i.e., the score increases multiplicatively).

\[ \text{score} = 32 + (32 \times 1.0) = 64 \]

Let’s imagine another scenario where the \( \text{ToRespRatio} \) is as small as 0.01 (1% timeouts). Here, the \( \text{minRatio} \) will be 0.01 (see equation †) and the \( \text{nearThresh} \) be 0.1 (see ‡). Thus, the next score will only increment fractionally:

\[ \text{score} = 32 + (32 \times 0.1) = 35.2 \]

To sum up, our algorithm prevents scores from fluctuating along with the number of timeouts. That is, we linearly decrement the score when we do not observe any timeouts from a peer, but multiplicatively increase the score when we observe timeouts from the peer.

### 2.2.2 Scores Set

Every instance \( X \) then sends the scores of its peers \( (A, B, ...) \) to the ScoreDB server, which will then maintain a history of the scores. For example for a given peer \( A \), there are \( N-1 \) scores for \( A \) collected in every 5-second epoch.

For every peer, all the scores given for that peer are collected within a 10-minute sliding window, where ScoreDB then picks the 30\(^{th}\)-percentile value to be the representative score for that peer, such as for instance \( A \). The 30\(^{th}\)-percentile value implies that the peer instance must have 70% high score values within a 10-minute interval such that we do not inadvertently quarantine instances with mere transient faults. In our deployments, we have observed that a 10-minute window is wide enough to detect persistent faults. It may not be the absolute minimum but it does put an upper bound on the time to isolate a fail-slow peer.

At this point, ScoreDB has \( N \) representative scores for all the instances in the cluster and it submits these scores to the DBSCAN algorithm [6]. ScoreDB performs this every minute, but using the data from the past 10 minutes (a sliding window). DBSCAN [6] is an algorithm that takes a set of

\[ \text{ToRespRatio} = \#\text{timeouts} / \#\text{responses} \]

2To make the score multiplication increases faster/slower (i.e., more configurable), we can introduce a score multiplier with a usage such as: e.g., 32 + \( \text{scoreMultiplier} \times 32 \). We use \( \text{scoreMultiplier} = 1 \).
points and groups them such that points that are spatially close are grouped together while points which do not have enough close neighbors are classified as outliers. Thus, we configure DBSCAN to output a binary decision (whether an instance is “fast” or “slow”). We also only mark at most one outlier at a time to make sure we do not remove too many service instances (explained later in §2.3).

Finally, we emphasize that we only compare instances (scores) of the same service. We do not compare instance scores of Cassandra with those of ZooKeeper, thus the algorithm above runs for every service deployed. For example in Figure 1, the ScoreDB server maintains history of Cassandra and ZooKeeper peer scores separately.

2.3 Mitigation

After a service instance is marked as an outlier, IASO starts the mitigation process. Below are the three options that our customers can set in IASO configuration. The first one (service instance reboot) is the default configuration. The philosophy of our mitigation is that it is better to remove a highly degraded instance than allowing it to induce a cascading problem to the entire cluster. Other works [24, 56] already show how running with one less instance \((N-1)\) can give a better performance than running a full cluster \((N)\) with a degraded instance. IASO only quarantines at most one instance to prevent the cluster drops below its fault-tolerant level.

(1) SERVICE INSTANCE REBOOT AND LEADERSHIP REMOVAL: Here, IASO will restart the slow instance and remove leadership leases (if any) from the service instance running on that node. We emphasize here that we only remove the service instance (e.g., Cassandra/ZooKeeper slow instance), but not the underlying VM or the machine. As a reason, imagine a machine where an instance of service \(X\) uses the underlying slow disk, but another instance of service \(Y\) only utilizes the memory (still fast). Here, we want \(X\) to be rebooted and its leadership removed, but let \(Y\) continue to run normally as it is not affected by the slow disk.

Regarding the removal of leadership, in ZooKeeper, if the instance is a leader, rebooting the instance will automatically make ZooKeeper choose a new leader. This way the old slow leader is no longer the single point of performance failure. The only cost associated with this action is the rebuilding of leader state on some other healthy peer.

In Cassandra, every instance is responsible for a key range (our deployment does not use Cassandra’s virtual node feature). Here, we have two opposing options for mitigation. The expensive option is to remove the instance from the ring and trigger a whole-cluster key-range rebalancing, which might be a premature action as the instance perhaps can be fixed soon. The cheaper option is to let the slow instance be in the ring but not allow it to be part of the data transfer.

We chose the latter option and modified Cassandra slightly to achieve this. In this mode, the slow instance is no longer the primary owner of its key range, but rather one of the other replicas becomes the primary owner. The upside is that we postpone the need for whole-cluster key-range rebalancing. The downside is that the fault tolerance of newly added data will be down by one (e.g., we can only write to two replica nodes as the instance on the slow node is being isolated) and read throughput may be degraded due to the loss of one instance. We note that the fault tolerance of old data does not go down as the data is still there in the slow instance.

Regardless of the limitations of this default option, customers who have smaller clusters tend to choose this option as they do not have options to migrate the instance or VMs to another healthy machine. Below we discuss other options for customers with larger clusters.

(2) VM SHUTDOWN: This is a more severe action than the default option above. In this mode, the controller VM of the slow service instance is shut down and no services are started on the VM. The difference between this action and the default one above is that when VM is shut down, the services above will automatically run their recovery protocols (e.g., whole ring rebalancing). Thus, the fault tolerance of the data stays the same (e.g., 3-way replication is still maintained). The similarity is that there may also be a performance drop to the loss of a VM. When the problem is fixed, the VM is added backed and full performance can be restored.

(3) HOST MACHINE SHUTDOWN: This option is similar to VM shutdown. The difference is that our system will automatically migrate the entire VM from this host to another, which is a process transparent to the services running on the VM. There may be a potential VM rebalancing issue (e.g., a machine has too many VMs). For VM balancing, we employ our own proprietary VM rebalancing that is outside the scope of the paper. We also emphasize that in our deployment, these machines are running the services that we deploy. The machines are not shared with other tenants, hence we have a full control of when to shut down the machine.

2.4 Resolution

The last stage, resolution, is the manual part of the whole IASO operational procedure, which we describe here for completeness.

When detecting a fail-slow node, IASO generates a user alert on the customer monitoring UI. IASO also pages our site-reliability engineers (SREs) such that they can work with the affected customer to fix the problem. If there had been a cluster outage (i.e., cluster IOPS went to almost zero) before the mitigation, IASO helps the customer and our SREs in identifying the faulty node and service.

It is also possible that before the SREs perform the full
diagnosis, the problem already went away by itself from re-booting the slow node. We see this happens in cases such as CPU locks-ups or high heap usage levels. In such cases, IASO will no longer mark the node as a degraded node. In overall, when the problem is fixed, IASO immediately rolls back the fail-slow node actions executed before, and service instances on the newly recovered node regain their leadership responsibilities.

Temporary fail-slow faults can be recurrent (e.g., high heap usage level). To prevent such recurrent faults, the root cause must be fixed. For example, we could apply some custom optimizations to our services to prevent it from entering such a state again.

### 2.5 Other Implementation Details

#### INTEGRATION:
So far we have integrated IASO with Cassandra and ZooKeeper. The implementation complexity is shown in Table 1. The changes to the target services are non-intrusive (less than 600 LOC). The service instances use IASO library to measure local scores and send them to the ScoreDB server where the rest of the complexity lies. The total score data size of ScoreDB server is only 0.27 MB per day per cluster on average as it only needs to keep the score history of the last 10 minutes. The CPU overhead is near 0%.

<table>
<thead>
<tr>
<th>Components</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassandra modification</td>
<td>585</td>
</tr>
<tr>
<td>ZooKeeper modification</td>
<td>199</td>
</tr>
<tr>
<td>IASO node-level library</td>
<td>547</td>
</tr>
<tr>
<td>ScoreDB server</td>
<td>3377</td>
</tr>
</tbody>
</table>

Table 1: Implementation complexity (§2.5). The table shows our IASO integration effort.

In our deployment, the Cassandra-side IASO so far has been very effective. But as we deal with deployments of tens of thousands of nodes, we can potentially cover a wider set of failure types if we can integrate IASO with another service as well. Hence, we attempted to integrate IASO to ZooKeeper, but ZooKeeper employs a pure leader-follower architecture where followers do not transfer data with each other (i.e., 3-way writes flow from the leader to three followers, unlike in HDFS or Cassandra). The leader is a single point of performance failure [24]; if the leader’s NIC is slow, the writes to all the followers will slow down, hence no outlier.

For this, we add a simple, lightweight background ping-pong thread between ZooKeeper peers (only <200 LOC). Every 10 seconds, every instance picks a maximum of 7 random peers and makes an RPC that includes a synchronous disk write. Checking the disk latency this way is also beneficial since most data operations in Cassandra hit the cache, hence disk monitoring is a bit lacking. Besides these small changes, we emphasize that the rest of the algorithm is the same – the instances send the median latencies of their peers (median of 1 minute window) to ScoreDB and the DBSCAN algorithm will compute the outlier.

#### THRESHOLDS:
We would like to emphasize that the threshold values we use in our algorithms (§3.2) are based on our specific deployment experiences. It is possible that the values might not work in other cases.

### 3 Results

This section presents our experimental results, starting with unsuccessful experiences (§3.1) and then the successful ones (§3.2) and the false positive rates (§3.3).

#### 3.1 Unsuccessful Attempts

The first strawman approach we tried was to use the raw timeout count as a metric to sense service instance level performance degradation. Figure 2 shows the number of timeouts observed in three samples of real degraded instances (in different time periods and clusters). As shown, the timeouts observed occur in bursts although the fault is severe throughout the time interval. Thus, without saving the ratio of timeouts and responses for every peer over a given period, there is no way to detect whether these high scores were merely transient or if they were truly persistent and possibly catastrophic faults.

For this reason, we next attempted to create a more stable algorithm by defining a score to be the percentage of timeouts over the total responses in every epoch. The first line below is the same as the first equation in §2.2, and in the second line, a peer score is essentially the $\text{ToRespRatio}$.

\[
\text{ToRespRatio} = \#\text{timeouts} / \#\text{responses}
\]

\[
\text{score} = \text{ToRespRatio}
\]
Figure 2: **Timeout fluctuations (§3.1).** The figures show the number of timeouts observed over time in three samples of degraded nodes (different time period).

Figure 3a shows the result. Ideally the score should stay high throughout the degraded period, but instead we see one big spike and one small spike. We then modified the scoring algorithm slightly by using the median of the last 3-minute window:

\[
\text{score} = \text{median} (\text{ToRespRatio in last 3 mins})
\]

The result, as shown in Figure 3b, still shows the same behavior (a dip between the two spikes). We tried replacing the median using average and weighted average and the result is similar (Figures 3c-d).

### 3.2 Successful Results

The previous section provides the reason we invented our custom outlier detection. Figure 4a shows the resulting scores from our custom algorithm, as detailed in Section 2.2. We can see that the metadata service (Cassandra/MS) instance on the degraded node has high scores assigned to it from 11:30 to 13:15 hours. Note that this is the case where we have not enabled the mitigation procedure, i.e., the customer was experiencing degradation for almost 2 hours!

Correspondingly, to check that the scores are accurate, we checked the standard network performance graphs and we found that there had been a network issue at the exact time interval. Figure 4b shows the TCP SENDQ size on the network connection between another node with this unhealthy node. Furthermore, Figure 4c shows the ping latencies to the degraded machine.

From these graphs, we can see that bad network performance on the slow machine correlated perfectly with the bad scores assigned to the nodes running on it. As a side note, we can see that the two metrics in Figures 4b-c cannot be used as raw scores as they also fluctuate.

Next, Figure 5 shows what is happening in the ScoreDB server side for a different fail-slow incident. The picture shows the representative scores by instance X measured for its \(N-1\) peers on other nodes. For simplicity, the data here is from a cluster of 4 nodes. Figure 5a shows that node3 has a high score compared to other peers. But at this point node3 has not been marked as a definite outlier because its 30\(^{th}\) percentile score is not high yet. However, two minutes later, as shown in Figure 5b, we have sufficient scores for the 30\(^{th}\) percentile score to be high. When we plug this score into the DBSCAN algorithm, node3 was marked as a definite outlier.

IA SO automatically quarantines an outlier to prevent it slowing down the entire cluster. Figure 6 shows another case after we deploy IASO. Here, the figure shows that the cluster-level IOPS drops to almost zero with the presence of one degraded machine, essentially showing how a degraded node can impact the entire cluster, as also shown by other works [24, 56]. Packet losses and the cluster-level degradation started occurring at around 09:15am but just after 10 minutes, IASO’s mitigatory actions kicked in and the performance of the cluster was completely restored. Thus, with IASO, the time taken to quarantine a degraded node has now been brought down to the order of minutes. Note that the IOPS returns to “normal” although we lost a node, which is because in this scenario the 100K IOPS were far from the maximum throughput of the cluster.

### 3.3 True and False Positives

Figures 7a and 7b show the number of true and false positives we encountered every month across the 7 months, re-
respectively. For Figure 7b, the figure combines the number of “confirmed” and “probable” false positives as explained below.

Over a 7-month period, we encountered 9 confirmed false positives over the 232 true positives (confirmed fail-slow incidents), which brings our false positive rate to 3.7%. One major reason for our false positive is in our earlier versions of IASO where the cluster still sends data to a dead service instance and a healthy instance already becomes affected and “looks” slow as well. Here IASO incorrectly marks the healthy instance as an outlier. Due to space constraints, we put more false positive stories in our anonymized supplemental material [7].

We also encountered 41 probable false positives. We label these cases as “probable” because they do not necessarily suggest that IASO is imprecise. In these cases, by the time our SREs started debugging, the issue was no longer present and the service instances, VMs, and machines were healthy. Existing works gave some hints on the reasons behind this, for example, fail-slow incidents can be triggered by temporary environmental causes such as high temperature [28].

While we managed to record the false positives, we were not able to collect many false-negative reports (i.e., undetected fail-slow incidents). This is because the reality of a large company and our SREs have their own priorities and might not contact us when they found cases that were not but should have been detected by IASO. The false negatives we were aware of came from two 2 outages that happened after the deployment of IASO, which can be found in our supplemental material [7]. Other false negatives we noticed include low workloads as fail-slow faults with low workloads might not necessarily result in timeouts. We did not fix this problem as almost all our customers heavily utilize their clusters.

From our perspective, we prefer false positives over false negatives as in our system IASO pages site-reliability engineers whenever it detects a fail-slow failure. This gives
us a way to easily track and investigate such issues and improve our system over time. As for the worst case impact, a false positive can cause a cluster to temporarily operate in a reduced fault tolerance state as IASO’s extreme mitigation strategy can bring down a node. However, in case of a false negative, there can be an entire cluster outage which can stay undetected for hours.

4 Fail-Slow Dataset and Analysis

The deployment of IASO allows us to analyze fail-slow incidents in our vast field of clusters, which then enables us to perform new statistical studies. This section first describes our dataset (§4.1) followed with our findings (§4.2).

4.1 Dataset

We first describe our deployment settings. Our field consists of 39,000 nodes spread across many clusters. A cluster size ranges from 3 to 56 nodes. Our various cluster models and configurations (RAM size, storage, etc.) can be found in our supplemental material [7]. A cluster can contain heterogeneous nodes as we support heterogeneous applications and a broken hardware can be replaced with a higher-end one. Each node in a cluster runs a special VM called a controller VM where our data and control path services run. Among these services, Cassandra and Zookeeper run with IASO integration.

4.2 Findings

4.2.1 Frequency

With a large dataset, we are able to measure the annual failure rate (AFR) of fail-slow incidents. Given 232 independent cases across 39,000 nodes over 7 months, we can derive that fail-slow AFR is 1.02% (232 × 7 / 39,000).

Table 2 compares fail-slow AFR with the rates of other types of failures. As shown, fail-slow fault frequency is rel-
4.2.2 Root Causes

Next, we analyze the root causes of fail-slow incidents. To compare the frequencies of various different root causes of fail-slow incidents with those of fail-stop failures [51], we group the causes into six categories: Hardware, Software, Network, Environment, Human and Unknown. For example, all issues that had a tag of “memory” or “disk” in our support tickets are grouped under Hardware.

Figure 9 shows the breakdown of fail-slow root causes (and the comparison to fail-stop causes) respectively significant and cannot be ignored. Figure 8 breaks down the number of fail-slow incidents observed per day in our field over the 7 months. We see that barring a couple of days in between, there is at least one failure per day. These statistics accentuate the importance of fail-slow detection and mitigation frameworks such as IASO.

The Unknown count is quite significant because of a couple of reasons. One common reason is when a customer becomes unresponsive during the support case or does not want the issue to be investigated further without providing a clear reason. We believe this can be either because the customer did not notice any issue around the time the fail-slow alert was generated (thereby a false positive) or fixed the issue themselves without our help. The other reason is when the SREs could not find a specific root cause for the issue or did not tag the support case with a clear cause.

4.2.3 Root Sub-Causes

Table 3 shows further the breakdown of the sub-causes within each of the five root categories in the previous section. The numbers in the parentheses are the count of tickets.

Table 3: Root sub-causes (§4.2.3). The table shows the sub-causes within each of the five categories of known root causes. The dataset will be released publicly.

4.2.4 Age and Model

As our ticketing system automatically collects machine age data, we are able to correlate fail-slow failures with machine ages, as shown in Figure 10, bucketed into months ranging from 1 to 48.

For example, for hardware-induced slowdown, it can be because of faulty dimm, ECC/CRC errors, low memory, etc., while network-induced slowdown can be because of faulty NICs/switches, bad cables, packet drops, and network contention.

Our goal here is to show that fail-slow root causes vary widely. We believe this is a strong motivation why fail-slow detection and mitigation should be also deployed at the service level (not just low-level hardware level). Our findings are also consistent with those reported in a recent paper [28]; we observed in our field how fault conversions take place and how different failure types such as fail-stop (e.g., disk/SSD failure), fail-transient (e.g., GC), and fail-partial (ECC errors) can transform into fail-slow failures at the service level [28, §3.2].
from 1 to 48. We can see the “infant mortality” trend where younger machines exhibit more issues, but older (perhaps more stable) machines exhibit fewer issues. This follows the same failure trend in fail-stop failures [51, Figure 4]. This also supports a continuous paradigm where when the rate of fail-stop errors drops so does the fail-slow ones.

We also attempted to correlate fail-slow failures with the node model family and found no significant correlation, that every node model family suffers from faults across a majority of component types (see [7] for more).

### 4.2.5 Tickets TTR

Finally, Figure 11 shows the distribution of time to resolve the tickets (in hours) across different root causes. Table 4 shows the median, mean, and max values of the data in Figure 11. We emphasize that this metric does not represent the time for IASO to mitigate the issues (which is in the order of 10 minutes), but rather how long it takes to close a ticket. When a ticket is closed, the customer’s cluster is guaranteed to be back fully healthy.

The reason we show this data is to point out that a fail-slow root cause can take days to be fully resolved. This is consistent with anecdotal experiences shared by large-scale operators from various institutions [28, §3.5]. Hence, it is important to quickly quarantine the fail-slow component before the performance problem cascades to the entire cluster.

<table>
<thead>
<tr>
<th>Net</th>
<th>Unk</th>
<th>HW</th>
<th>SW</th>
<th>Human</th>
<th>Env</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median</td>
<td>79</td>
<td>145</td>
<td>126</td>
<td>234</td>
<td>108</td>
</tr>
<tr>
<td>Mean</td>
<td>149</td>
<td>220</td>
<td>244</td>
<td>323</td>
<td>165</td>
</tr>
<tr>
<td>Max</td>
<td>721</td>
<td>1033</td>
<td>1705</td>
<td>1238</td>
<td>625</td>
</tr>
</tbody>
</table>

Table 4: TTR tickets (§4.2.5). The table shows the median, mean, and maximum values of the data in Figure 11.

Table 5: Related work (IASO). The table categorizes works that relate to fail-slow detection, diagnosis, and mitigation across hardware-, software-, and service levels.

<table>
<thead>
<tr>
<th>Bug finding</th>
<th>HW</th>
<th>SW</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPMI[2],</td>
<td>SymDrive[47],</td>
<td>MacePC[38],</td>
<td>Orca[15]</td>
</tr>
<tr>
<td>SMART[4],</td>
<td>DDT[39],</td>
<td>PCatch[40],</td>
<td></td>
</tr>
<tr>
<td>Ganglia[42],</td>
<td>SPV[55]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Detection</th>
<th>IPMI[2],</th>
<th>UBL[20],</th>
<th>PeerReview[30],</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNMP[3],</td>
<td></td>
<td>Toddler[43],</td>
<td></td>
</tr>
<tr>
<td>SMART[4],</td>
<td></td>
<td></td>
<td>AFD[45]</td>
</tr>
<tr>
<td>Ganglia[42],</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Diagnosis</th>
<th>Roy[49],</th>
<th>Xray[13],</th>
<th>Canopy[36],</th>
</tr>
</thead>
<tbody>
<tr>
<td>PerfBlower[25],</td>
<td>Hytrace[19],</td>
<td>Pip[48],</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PerfScope[21],</td>
<td>Panorama[31]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PerfCompass[22]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Deepview[59],</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stitch[60],</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FaultLocalize[50]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mitigation</th>
<th>Carburizer[35],</th>
<th>Mantri[11],</th>
</tr>
</thead>
<tbody>
<tr>
<td>DisturbMLC[14],</td>
<td>DeepDive[44],</td>
<td></td>
</tr>
<tr>
<td>VibrateSSD[17],</td>
<td>PBSE[56]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PREPARE[57],</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IASO</td>
<td></td>
</tr>
</tbody>
</table>

5 Related Work

We now discuss related work beyond the papers that we already cited earlier. In particular, we break the discussion here to two categories: (1) works related to fail-slow detection and mitigation systems and (2) publications that release information about fail-slow incidents.

Table 5 shows that there are many tools, frameworks, and approaches that have been introduced or deployed for different levels of the hardware, software, and service stack. First, there are many bug-finding tools such as MacePC [38], PCatch [40], and Orca [15], but they are offline approaches. Second, there are online fail-slow detection tools across the hardware/software stack. For example, SMART [4] is a monitoring tool that can be used to detect hardware degradation but does not include diagnosis capability. Third, Pip [48], PivotTracing [41] and many others provide diagnosis approaches that work at the service level (not just one particular software) but they do not make quarantine decisions. Finally, IASO is in a category that performs detection and automated mitigation. In this space, we are not aware of many published works. The limitation of IASO is that it does not come with diagnosis tools. Thus, the diagnosis approaches in the 3rd row of Table 5 are orthogonal to our work.

Table 6 shows publications that release datasets on performance problems. The table shows the year span (Yr), number of fail-slow failures/bugs reported (#F), deployment size/number of nodes (#N), the number of systems/services the data is collected from (#S) and the scope of the root-cause analysis (A). The top part of the table represents incidents that appear in live deployments while the bottom of the
Table 6: Related work (fail-slow dataset). For each related work, the columns show the year span (Yr), number of fail-slow failures/bugs reported (#F), deployment size/number of nodes (#N), the number of systems/services the data is collected from (#S) and the scope of the root-cause analysis (A). In the last column (analysis), “h” represents hardware, “s” software, “n” network, “e” environment, and “m” human. Papers with “s”–only label implies bug-study papers.

<table>
<thead>
<tr>
<th>Related Work</th>
<th>Yr</th>
<th>#F</th>
<th>#N</th>
<th>#S</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>IASO</td>
<td>’16-17</td>
<td>232</td>
<td>39k</td>
<td>1</td>
<td>ehmnsu</td>
</tr>
<tr>
<td>Fail-slow[28]</td>
<td>’00-17</td>
<td>101</td>
<td>≥10k</td>
<td>12</td>
<td>hn</td>
</tr>
<tr>
<td>GrayFailure[32]</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Panorama[31]</td>
<td>’17-18</td>
<td>15</td>
<td>20</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>COS[27]</td>
<td>’09-15</td>
<td>126</td>
<td>-</td>
<td>32</td>
<td>ehmnsu</td>
</tr>
<tr>
<td>CBS[26]</td>
<td>’11-14</td>
<td>860</td>
<td>-</td>
<td>6</td>
<td>s</td>
</tr>
<tr>
<td>PerfBugs[34]</td>
<td>’00-11</td>
<td>109</td>
<td>-</td>
<td>5</td>
<td>s</td>
</tr>
<tr>
<td>Limplock[24]</td>
<td>’13</td>
<td>28</td>
<td>≥30</td>
<td>5</td>
<td>s</td>
</tr>
</tbody>
</table>

6 Conclusion

We have described our successful 1.5-year deployment of IASO. We found fail-slow detection and automated mitigation schemes are crucial in preventing fail-slow induced outages in our large deployment field. We would like to emphasize again that automatic fail-slow mitigation/quarantine schemes (beyond detection only) are relatively a new area of research. We hope our paper can provide insights to the development of better frameworks in the future.

As future work, we look forward to building a more aggressive algorithm that can quarantine a slow node shorter than our current 10-minute interval (and do so with low false positives) as well as automatically marking fail-slow faults that are resolved by themselves without depending on our customers or SREs (more in [7]). Furthermore, as we continue to collect peer scores reported in the field, we hope to learn more detailed characteristics.

7 Acknowledgments

We thank Ric Wheeler, our shepherd, and the anonymous reviewers for their tremendous feedback and comments. We also would like to thank Roger Liao and Anshul Purohit for their significant contributions during the development of IASO. We learnt a lot from the actual customer cases where IASO was effective and also from a few scenarios where we hit false positives. Thanks to Rob Savino and Mark Czarnecki for their effort to make this information available to us and help us with quite some root cause analysis. University of Chicago authors were supported by funding from NSF grant No. CNS-1350499.
References


[28] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Gollifher, Swaminathan Sundararaman, Xing Lin, Tim


