PBSE: A Robust Path-Based Speculative Execution for Degraded-Network Tail Tolerance in Data-Parallel Frameworks

Riza O. Suminto, Cesar A. Stuardo, Alexandra Clark
University of Chicago

Huan Ke, Tanakorn Leesatapornwongsa, Bo Fu[1]
University of Chicago

Daniar H. Kurniawan
Bandung Institute of Technology

Vincentius Martin[2]
Surya University

Uma Maheswara Rao G.
Intel Corp.

Haryadi S. Gunawi
University of Chicago

ABSTRACT

We reveal loopholes of Speculative Execution (SE) implementations under a unique fault model: node-level network throughput degradation. This problem appears in many data-parallel frameworks such as Hadoop MapReduce and Spark. To address this, we present PBSE, a robust, path-based speculative execution that employs three key ingredients: path progress, path diversity, and path-straggler detection and speculation. We show how PBSE is superior to other approaches such as cloning and aggressive speculation under the aforementioned fault model. PBSE is a general solution, applicable to many data-parallel frameworks such as Hadoop/HDFS+QFS, Spark and Flume.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; *Distributed architectures*; *Dependable and fault-tolerant systems and networks*;

KEYWORDS

Dependability, distributed systems, tail tolerance, speculative execution, Hadoop MapReduce, HDFS, Apache Spark

ACM Reference Format:
Riza O. Suminto, Cesar A. Stuardo, Alexandra Clark, Huan Ke, Tanakorn Leesatapornwongsa, Bo Fu[1], Daniar H. Kurniawan, Vincentius Martin[2], Uma Maheswara Rao G., and Haryadi S. Gunawi. 2017. PBSE: A Robust Path-Based Speculative Execution for Degraded-Network Tail Tolerance in Data-Parallel Frameworks. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 14 pages.
https://doi.org/10.1145/3127479.3131622

1 INTRODUCTION

Data-parallel frameworks have become a necessity good in large-scale computing. To finish jobs on time, such parallel frameworks must address the "*tail latency*" problem. One popular solution is *speculative execution (SE)*; with SE, if a task runs slower than other tasks in the same job (a "straggler"), the straggling task will be speculated (via a "backup task"). With a rich literature of SE algorithms (§3.4, §7), existing SE implementations such as in Hadoop and Spark are considered quite robust. Hadoop's SE for example has architecturally remained the same for the last six years, based on the LATE algorithm [83]; it can effectively handle stragglers caused by common sources of tail latencies such as resource contentions and heterogeneous resources.

However, we found an important source of tail latencies that current SE implementations cannot handle graciously: *node-level network throughput degradation* (*e.g.*, a 1Gbps NIC bandwidth drops to tens of Mbps or even below 1 Mbps). Such a fault model can be caused by a severe network contention (*e.g.*, from VM overpacking) or due to degraded network devices (*e.g.*, NICs and network switches whose bandwidth drops by orders of magnitude to Mbps/Kbps level in production systems; more in §2.2). The uniqueness of this fault model is that only the network device performs poorly, but other hardware components such as CPU and storage are working normally; it is significantly different than typical fault models for heterogeneous resources or CPU/storage contentions.

To understand the impact of this fault model, we tested Hadoop [11] as well as other systems including Spark [82], Flume [9], and S4 [12], on a cluster of machines with one slow-NIC node. We discovered that many tasks transfer data through the slow-NIC node but *cannot escape* from it, resulting in long tail latencies. Our analysis then uncovered many surprising *loopholes* in existing SE implementations (§3), which we bucket into two categories: the *"no" straggler* problem, where *all* tasks of a job involve the slow-NIC node (since all tasks are slow, there is "no" straggler detected) and the *straggling backup* problem, where the backup task involves the slow-NIC node again (hence, *both* of the original and the backup tasks are straggling at the same time).

[1]Now a PhD student at Purdue University
[2]Now a PhD student at Duke University

Overall, we found that *a network-degraded node is worse than a dead node*, as the node can create a *cascading* performance problem. One network-degraded node can make the performance of the entire cluster collapse (*e.g.*, after several hours the whole-cluster job throughput can drop from hundreds of jobs per hour to 1 job/hour). This cascading effect can happen as unspeculated slow tasks lock up the task slots for a long period of time.

Given the maturity of SE implementations (*e.g.*, Hadoop is 10 years old), we investigate further the underlying design flaws that lead to the loopholes. We believe there are *two flaws*. First, node-level network degradation (without CPU/storage contentions) is not considered a fault-model. Yet, this real fault model affects data-transfer *paths*, not just tasks per se. This leads us to the second flaw: existing SE approaches only report task progress but do not expose *path* progress. Sub-task progress*es* such as data transfer rates are simply lumped into *one* progress score, hiding slow paths from being detected by the SE algorithm.

In this paper, we present a robust solution to the problem, *path-based speculative execution* (PBSE), which contains three important ingredients: path progress, path diversity, and path-straggler detection and speculation.

First, in PBSE, *path progresses* are exposed by tasks to their job/application manager (AM). More specifically, tasks report the data transfer progresses of their Input→Map, Map→Reduce (shuffling), and Reduce→Output paths, allowing the AM to detect straggling paths. Unlike the simplified task progress score, our approach does *not* hide the complex dataflows and their progresses, which are needed for a more accurate SE. Paths were not typically exposed due to limited transparency between the computing (*e.g.*, Hadoop) and storage (*e.g.*, HDFS) layers; previously, only data locality is exposed. PBSE advocates the need for a more cross-layer collaboration, in a non-intrusive way.

Second, before straggling paths can be observed, PBSE must enforce *path diversity*. Let's consider an initial job (data-transfer) topology X→B and X→C, where node X can potentially experiences a degraded NIC. In such a topology, X is a *single point of tail-latency failure* (*"tail-SPOF"* for short). Path diversity ensures no potential tail-SPOF will happen in the initial job topology. Each MapReduce stage will now involve multiple distinct paths, enough for revealing straggling paths.

Finally, we develop *path-straggler detection and speculation*, which can accurately pinpoint slow-NIC nodes and create speculative backup tasks that avoid the problematic nodes. When a path A→B is straggling, the culprit can be A or B. For a more effective speculation, our techniques employ the concept of failure groups and a combination of heuristics such as greedy, deduction, and dynamic-retry approaches, which we uniquely personalize for MapReduce stages.

We have implemented PBSE in the Hadoop/HDFS stack in total of 6003 LOC. PBSE runs side by side with the base SE; it does not replace but rather enhances the current SE algorithm. Beyond Hadoop/HDFS, we also show that other data-parallel frameworks suffer from the same problem. To show PBSE generality, we also have successfully performed initial integrations to Hadoop/QFS stack [67], Spark [82], and Flume [9].

Symbols	Descriptions
AM	Application/Job Manager
I_i	Node for an HDFS input block to task i
I'_i, I''_i	2nd and 3rd replica nodes of an input block
M_i	Node for map task i
M'_i	Node for *speculated* (′) map i
R_i	Node for reduce task i
O_i	Node for output block
O'_i, O''_i	2nd and 3rd replica nodes of an output block

A sample of a job topology (2 maps, 2 reduces):	
Map phase	$I_1 \rightarrow M_1$, $I_2 \rightarrow M_2$
Shuffle phase	$M_1 \rightarrow R_1$, $M_1 \rightarrow R_2$, $M_2 \rightarrow R_1$, $M_2 \rightarrow R_2$
Reduce phase	$R_1 \rightarrow O_1 \rightarrow O'_1 \rightarrow O''_1$, $R_2 \rightarrow O_2 \rightarrow O'_2 \rightarrow O''_2$
Speculated M_1	$I'_1 \rightarrow M'_1$
Speculated R_2	$M_1 \rightarrow R'_2$, $M_2 \rightarrow R'_2$

Table 1: Symbols. ***The table above describes the symbols that we use to represent a job topology, as discussed in Section 2.1 and illustrated in Figures 1, 3, and 5.***

We performed an extensive evaluation of PBSE with a variety of NIC bandwidth degradations (60 to 0.1 Mbps), real-world production traces (Facebook and Cloudera), cluster sizes (15 to 60 nodes), scheduling policies (Capacity, FIFO, and Fair), and other tail-tolerance strategies (aggressive SE, cloning, and hedge read). Overall, we show that under our fault model, PBSE is superior to other strategies (between 2–70× speed-ups above the 90th-percentile under various severities of NIC degradation), which is possible because PBSE is able to escape from the network-degraded nodes (as it fixes the limitation of existing SE strategies).

In summary, PBSE is *robust*; it handles many loopholes (tail-SPOF topologies) caused by node-level network degradation, ensuring no single job is "locked" in degraded mode. Under this fault model, PBSE performs *faster* than other strategies such as advanced schedulers, aggressive SE, and cloning. PBSE is *accurate* in detecting nodes with a degraded network; a prime example is accurately detecting a map's slow NIC within the reduce/shuffling stage, while Hadoop always blames the reducer side. PBSE is *informed*; slow paths that used to be a silent failure in Hadoop SE are now exposed and avoided in speculative paths. PBSE is *simple*; it runs side-by-side with the base SE and its integration does not require major architectural changes. Finally, PBSE is *general*; it can be integrated to many data-parallel frameworks.

In the following sections, we present an extended motivation (§2), Hadoop SE loopholes (§3), PBSE design and evaluation (§4-5), further integrations (§6), related work and conclusion (§7-8).

2 BACKGROUND AND MOTIVATION

In this section, we first describe some background materials (§2.1) and present real cases of degraded network devices (§2.2) which motivates our unique fault model (§2.3). We then highlight the impact of this fault model to Hadoop cluster performance (§2.4) and discuss why existing monitoring tools are not sufficient and Hadoop SE must be modified to handle the fault model (§2.5).

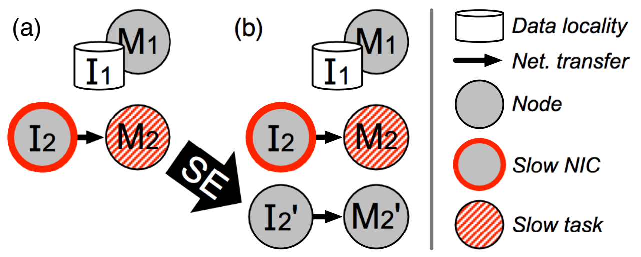

Figure 1: A Hadoop job and a successful SE. ***Figures (a) and (b) are explained in the "Symbols" and "Successful SE" discussions in Section 2.1, respectively.***

2.1 Background

Yarn/Hadoop 2.0: A Hadoop node contains task *containers/slots*. When a job is scheduled, Hadoop creates an *Application Manager (AM)* and deploys the job's parallel tasks on allocated containers. Each *task* sends a periodic *progress score* to AM (via heartbeat). When a task reads/writes a file, it asks HDFS *namenode* to retrieve the file's *datanode* locations. Hadoop and HDFS nodes are colocated, thus a task can access data *remotely* (via NIC) or *locally* (via disk; aka. "data locality"). A file is composed of 64MB *blocks*. Each is 3-way replicated.

Symbols: Table 1 describes the symbols we use througout the paper to represent a job topology. For example, Figure 1a illustrates a Hadoop job reading two input blocks (I_1 and I_2); each input block can have 3 replicas (*e.g.*, I_2, I'_2, I''_2). The job runs 2 map tasks (M_1, M_2); reduce tasks (R_1, R_2) are not shown yet. The first map achieves data locality ($I_1 \rightarrow M_1$ is local) while the second map reads data remotely ($I_2 \rightarrow M_2$ is via NICs). A complete job will have three stages: Input→Map (*e.g.*, $I_1 \rightarrow M_1$), Map→Reduce shuffle (*e.g.*, $M_1 \rightarrow R_1$, $M_1 \rightarrow R_2$), and Reduce→Output 3-node write pipeline (*e.g.*, $R_2 \rightarrow O_2 \rightarrow O'_2 \rightarrow O''_2$).

Successful SE: The Hadoop SE algorithm (or *"base SE"* for short), which is based on LATE [83, §4], runs in the AM of every job. Figure 1b depicts a successful SE: I_2's node has a degraded NIC (bold circle), thus M_2 runs slower than M_1 and is marked as a straggler, then the AM spawns a new *speculative/backup task* (M'_2) on a new node that coincidentally reads from another fast input replica ($I'_2 \rightarrow M'_2$). For every task, the AM by default limits to only one backup task.

2.2 Degraded Network Devices

Beyond fail-stop, network devices can exhibit "unexpected" forms of failures. Below, we re-tell the real cases of limping network devices in the field [1–8, 44, 48, 49, 57].

In many cases, NIC cards exhibit a high-degree of packet loss (from 10% up to 40%), which then causes spikes of TCP retries, dropping throughput by orders of magnitude. An unexpected autonegotiation between a NIC and a TOR switch reduced the bandwidth between them (an auto-configuration issue). A clogged air filter in a switch fan caused overheating, and subsequently heavy re-transmission (*e.g.*, 10% packet loss). Some optical transceivers collapsed from Gbps to Kbps rate (but only in one direction). A non-deterministic Linux driver bug degraded a Gbps NIC's performance to Kbps rate. Worn-out cables reportedly can also drop network performance. A worn-out Fibre Channel Pass-through module in a high-end server blade added 200-3000 ms delay.

As an additional note, we also attempted to find (or perform) large-scale statistical studies on this problem but to no avail. As alluded elsewhere, stories of "unexpected" failures are unfortunately "only passed by operators over beers" [36]. For performance-degraded devices, one of the issues is that, most hardware vendors do not log performance faults at such a low level (unlike hard errors [37]). Some companies log low-level performance metrics but aggregate the results (*e.g.*, hourly disk average latency [51]), preventing a detailed study. Thus, the problem of performance-degraded devices is still under studied and requires further investigation.

2.3 Fault Model

Given the cases above, our fault model is a *severe network bandwidth degradation* experienced by one or more machines. For example, the bandwidth of a NIC can drop to low Mbps or Kbps level, which can be caused by many hardware and software faults such as bit errors, extreme packet loss, overheating, clogged air filters, defects, buggy auto-negotiations, and buggy firmware and drivers, as discussed above.

A severe node-level bandwidth degradation can also happen in public multi-tenant clouds where extreme outliers are occasionally observed [66, Fig. 1]. For instance, if all the tenants of a 32-core machine run network intensive processes, each process might only observe ~30 Mbps, given a 1GBps NIC. With a higher-bandwidth 10-100GBps NIC and future 1000-core processors [26, 79], the same problem will apply. Furthemore, over-allocation of VMs more than the available CPUs can reduce the obtained bandwidth by each VM due to heavy context switching [76]. Such problem of "uneven congestion" across datanodes is relatively common [16].

2.4 Impacts

Slow tasks are not speculated: Under the fault model above, Hadoop SE fails to speculate slow tasks. Figure 2a shows the CDF of job duration times of a Facebook workload running on a 15-node Hadoop cluster without and with one 1-Mbps slow node (`With-0-Slow` vs. `With-1-Slow` nodes). The 1-Mbps slow node represents a degraded NIC. As shown, in a healthy cluster, all jobs finish in less than 3 minutes. But with a slow-NIC node, many tasks are not speculated and cannot escape the degraded NIC, resulting in long job tail latencies, with 10% of the jobs (y=0.9) finishing more than 1 hour.

"One degraded device to slow them all:" Figure 2b shows the impact of a slow NIC to the entire cluster over time. Without a slow NIC, the cluster's throughput (#jobs finished) increases steadily (around 172 jobs/hour). But with a slow NIC, after about 4 hours (x=250min) the cluster throughput collapses to 1 job/hour.

The two figures show that existing speculative execution fails to cut tail latencies induced by our fault model. Later in Section 3, we dissect the root causes.

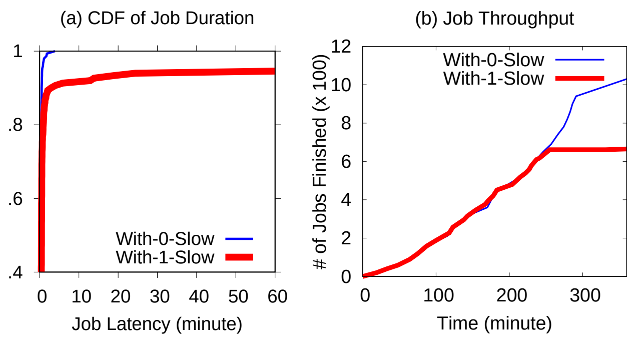

Figure 2: Impact of a degraded NIC. ***Figure (a) shows the CDF of job duration times of a Facebook workload on a 15-node Hadoop cluster without and with a slow node (*`With-0-Slow`* vs. *`With-1-Slow`* lines). The slow node has a 1-Mbps degraded NIC. Figure (b) is a replica of Figure 2 in our prior work [44], showing that after several hours, the problem cascades to entire cluster, making cluster throughput drops to 1 job/hour.***

2.5 The Need for a More Robust SE and Why Monitoring Tools Do Not Help

We initially believed that the problem can be easily solved with some cluster monitoring tools (hence why we did not invent PBSE directly after our earlier work [44]). However, the stories above point to the fact that monitoring tools do not always help operators detect and remove the problem quickly. The problems above took days to weeks to be fully resolved, and meanwhile, the problems continuously affected the users and sometimes cascaded to the entire cluster. One engineer called this a "costly debugging tail." That is, while more-frequent failures can be solved quickly, less-frequent but complex failures (that cannot be mitigated by the systems) can significantly cost the engineers' time. In one story, an entire group of developers were pulled to debug the problem, costing the company thousands of dollar.

Monitoring tools not sufficient for the following reasons: **(1)** Monitoring tools are important but they are "passive/offline" solutions; they help signal slow components but shutting down a slow machine is still the job of human operators (an automated shutdown algorithm can have false positives that incorrectly shutdown healthy machines). **(2)** To perform diagnoses, (data) nodes cannot be abruptly taken offline as it can cause excessive re-replication load [68]. Operator's decision to decommission degraded nodes must be well-grounded; many diagnoses must be performed before a node is taken offline. **(3)** Degraded modes can be complex (asymmetrical and/or transient), making diagnosis harder. In one worst-case story, several months were needed to pinpoint worn-out cables as the root cause. **(4)** As a result, there is a significant window of time between when the degradation started and when the faulty device is fully diagnosed. Worse, due to economies of scale, 100-1000 servers are managed by a single operator [34], hence longer diagnosis time.

In summary, the discussion above makes the case for a more robust speculative execution as a proper "online" solution to the problem. In fact, we believe that path straggler detection in PBSE can be used as an alarm or a debugging aid to existing monitoring tools.

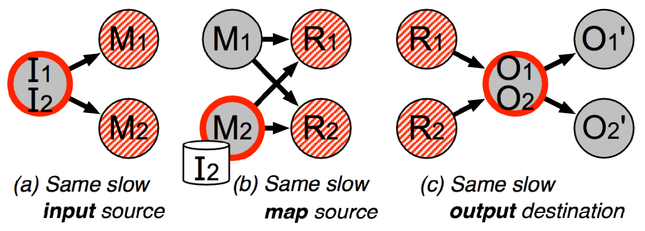

Figure 3: Tail-SPOF and "No" Stragglers. ***Figures (a)-(c) are described in Sections 3.1a-c, respectively. I_1/I_2 in Figure (a), M_2 in (b), and O_1/O_2 in (c) are a tail-SPOF that makes all affected tasks slow at the same time, hence "no" straggler. Please see Figure 1 for legend description.***

3 SE LOOPHOLES

This section presents the many cases of failed speculative execution (*i.e.*, "SE loopholes"). For simplicity of discussion, we only inject one degraded NIC.

Benchmarking: To test Hadoop SE robustness to the fault model above, we ran real-world production traces on 15-60 nodes with one slow NIC (more details in §5). To uncover SE loopholes, we collect all task topologies where the job latencies are significantly longer than the expected latency (as if the jobs run on a healthy cluster without any slow NIC). We ran long experiments, more than 850 hours, because every run leads to non-deterministic task placements that could reveal new loopholes.

SE Loopholes: An *SE loophole* is a unique topology where a job *cannot escape* from the slow NIC. That is, the job's latency follows the rate of the degraded bandwidth. In such a topology, the slow NIC becomes a *single point of tail-latency failure (a "**tail-SPOF**").* By showing tail-SPOF, we can reveal not just the straggling tasks, but also the *straggling paths*. Below we describe some of the representative loopholes we found (all have been confirmed by Hadoop developers). For each, we use a minimum topology for simplicity of illustration. The loopholes are categorized into *no-straggler-detected* (§3.1) and *straggling-backup* (§3.2) problems.

We note that our prior work only reported three "limplock" topologies [44, §5.1.2]) from only four simple microbenchmarks. In this subsequent work, hundreds of hours of deployment allow us to debug more job topologies and uncover more loopholes.

3.1 No Straggler Detected

Hadoop SE is only triggered when *at least* one task is straggling. We discovered several topologies where *all* tasks (of a job) are slow, hence "no" straggler.

(a) Same slow input source: Figure 3a shows two map tasks (M_1, M_2) reading data remotely. Coincidentally (due to HDFS's selection randomness when locality is not met), both tasks retrieve their input blocks (I_1, I_2) from the *same* slow-NIC node. Because *all* the tasks are slow, there is "no" straggler. Ideally, a notion of "path diversity" should be enforced to ensure that the tail-SPOF (I_1/I_2's node) is detected. For example, M_2 should choose another input source (I'_2 or I''_2).

(b) Same slow map source: Figure 3b shows a similar problem, but now during shuffle. Here, the map tasks (M_1, M_2) already complete normally; note that M_2 is also fast due to data locality ($I_2 \rightarrow M_2$

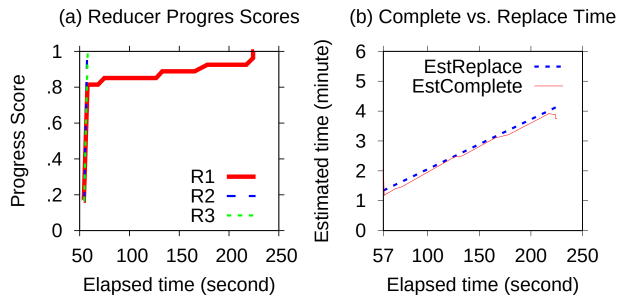

Figure 4: SE Algorithm "Bug". ***The two figures above are explained in Section 3.1d.***

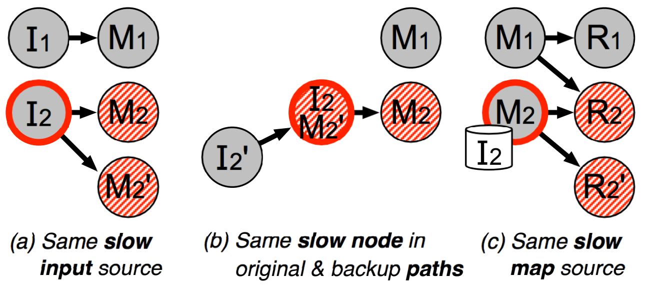
I1
M1
I2
M2
M2'
(a) Same slow input source
M1
I2 M2'
M2
I2'
(b) Same slow node in original & backup paths
M1
R1
M2
R2
I2
R2'
(c) Same slow map source

Figure 5: Tail-SPOF and Straggling Backups. ***Figures (a)-(c) are discussed in Sections 3.2a-c, respectively. I_2 in Figure (a), I_2/M_2' in (b), and M_2 in (c) are a tail-SPOF; the slow NIC is coincidentally involved again in the backup task. Please see Figure 1 for legend description.***

does not use the slow NIC). However, when shuffle starts, *all* the reducers (R_1, R_2) fetch M_2's intermediate data through the slow NIC, hence "no" straggling reducers. Ideally, if we monitor path progresses (the four arrows), the two straggling paths ($M_2 \rightarrow R_1$, $M_2 \rightarrow R_2$) and the culprit node (M_2's node) can be easily detected.

While case (a) above might only happen in small-parallel jobs, case (b) can easily happen in large-parallel jobs, as it only needs one slow map task with data locality to hit the slow NIC.

(c) Same slow output intersection: Figure 3c shows another case in write phase. Here, the reducers' write pipelines ($R_1 \rightarrow O_1 \rightarrow O_1'$, $R_2 \rightarrow O_2 \rightarrow O_2'$) intersect the same slow NIC (O_1/O_2's node), hence "no" straggling reducer.

(d) SE algorithm "bug": Figure 4a shows progress scores of three reducers, all observe a fast shuffle (the quick increase of progress scores to 0.8), but one reducer (R_1) gets a slow-NIC in its output pipeline (flat progress score). Ideally, R_1 (which is much slower than R_2&R_3) should be speculated, but SE is *never* triggered. Figure 4b reveals the root cause of why R_1 never get speculated. With a fast shuffle but a slow output, the estimated R_1 replacement time (`EstReplace`) is always slightly higher (0.1-0.3 minutes) than the estimated R_1 completion time (`EstComplete`). Hence, speculation is not deemed beneficial (incorrectly).

3.2 Straggling Backup Tasks

Let's suppose a straggler is detected, then the backup task is expected to finish faster. By default, Hadoop limits one backup per speculated task (*e.g.*, M_1' for M_1, but no M_1''). We found loopholes where this "one-shot" task speculation is also "unlucky" (involves the slow NIC again).

(a) Same slow input source: Figure 5a shows a map (M_2) reading from a slow remote node (I_2) and is correctly marked as a straggler (slower than M_1). However, when the backup task (M_2') started, HDFS gave the *same* input source (I_2). As a result, *both* the original and backup tasks (M_2, M_2') are slow.

(b) Same slow node in original and backup paths: Figure 5b reveals a similar case but with a slightly different topology. Here, the backup (M_2') chooses a different source (I_2'), but it is put in the slow node. As a result, both the original and backup paths ($I_2 \rightarrow M_2$ and $I_2' \rightarrow M_2'$) involve a tail-SPOF (M_2'/I_2's node).

(c) Same slow map source: Figure 5c depicts a similar shuffle topology as Figure 3b, but now the shuffle pattern is not all-to-all (job/data specific). Since M_2's NIC is slow, R_2 becomes a straggler. The backup R_2' however *cannot* choose another map other than reading through the slow M_2's NIC again. If the initial paths (the first three arrows) are exposed, a proper recovery can be done (*e.g.*, pinpoint M_2 as the culprit and run M_2').

3.3 The Cascading Impact

In Section 2.4 and Figure 2b, we show that the performance of the entire can eventually collapse. As explained in our previous work [44, §5.1.4], the reason for this is that the slow and unspeculated tasks are occupying the task containers/slots in healthy nodes for a long time. For example, in Figures 3 and 5, although only one node is degraded (bold edge), other nodes are affected (striped nodes). Newer jobs that arrive are possible to interact with the degraded node. Eventually, all tasks in all nodes are "locked up" by the degraded NIC and there are not enough free containers/slots for new jobs. In summary, failing to speculate slow tasks from a degraded network can be fatal.

3.4 The Flaws

Hadoop is a decade-old mature software and many SE algorithms are derived from Hadoop/MapReduce [43, 83]. Thus, we believe there are some fundamental flaws that lead to the existence of SE loopholes. We believe there are two flaws:

(1) Node-level network degradation is not incorporated as a fault model. Yet, such failures occur in production. This fault model is different than "node contentions" where CPU and local storage are also contended (in which cases, the base SE is sufficient). In our model, only the NIC is degraded, not CPU nor storage.

(2) Task ≠ Path. When the fault model above is not incorporated, the concept of path is not considered. Fatally, path progresse*s* of a task are lumped into *one* progress score, yet a task can observe differing path progresses. Due to lack of path information, slow paths are hidden. Worse, Hadoop can blame the straggling task even though the culprit is another node (*e.g.*, in Figure 5c, R_2 is blamed even though M_2 is the culprit).

In other works, task sub-progresses are also lumped into one progress score (*e.g.*, in Late [83, §4.4], Grass [32, §5.1], Mantri [33, §5.5], Wrangler [77, §4.2.1], Dolly [30, §2.1.1], ParaTimer [63, §2.5], Parallax [64, §3]). While these novel methods are superb in optimizing SE for other root causes (*e.g.*, node contentions, heterogeneous resources), they do not specifically tackle network-only degradation (at individual nodes). Some of these works also tries to monitor data transfer progresses [33], but they are still lumped into a single progress score.

4 PBSE

We now present the three important elements of PBSE: path progress (§4.1), path diversity (§4.2), and path-straggler detection and speculation (§4.3), and at the end conclude the advantages of PBSE (§8). This section details our design in the context of Hadoop/HDFS stack with 3-way replication and 1 slow NIC (F=1).[1]

4.1 Paths

The heart of PBSE is the exposure of path progresses to the SE algorithm. A path progress P is a tuple of $\{Src, Dst, Bytes, T, BW\}$, sent by tasks to the job's manager (AM); $Bytes$ denotes the amount of bytes transferred within the elapsed time T and BW denotes the path bandwidth (derived from $Bytes/T$) between the source-destination (Src, Dst) pair. Path progresses are piggybacked along with existing task heartbeats to the AM. In PBSE, tasks expose to AM the following paths:

• **Input→Map (I→M):** In Hadoop/ HDFS stack, this is typically a one-to-one path (*e.g.*, $I_2 \rightarrow M_2$) as a map task usually reads one 64/128-MB block. Inputs of multiple blocks are usually split to multiple map tasks.

• **Map→Reduce (M→R):** This is typically an all-to-all shuffling communication between a set of map and reduce tasks; many-to-many or one-to-one communication is possible, depending on the user-defined jobs and data content. The AM now can compare the path progress of *every* M→R path in the shuffle stage.

• **Reduce→Output (R→O):** Unlike earlier paths above, an output path is a pipeline of sub-paths (*e.g.*, $R_1 \rightarrow O_1 \rightarrow O_1' \rightarrow O_1''$). A single slow node in the pipeline will become a downstream bottleneck. To allow fine-grained detection, we expose the individual sub-path progresses. For example, if $R_1 \rightarrow O_1$ is fast, but $O_1 \rightarrow O_1'$ and $O_1' \rightarrow O_1''$ are slow, O_1' can be the culprit.

The key to our implementation is a more information exposure from the storage (HDFS) to compute (Hadoop) layers. Without more transparency, important information about paths is hidden. Fortunately, the concept of transparency in Hadoop/HDFS already exists (*e.g.*, data locality exposure), hence the feasibility of our extension. The core responsibility of HDFS does not change (*i.e.*, read/write files); it now simply exports more information to support more SE intelligence in the Hadoop layer.

4.2 Path Diversity

Straggler detection is only effective if independent progresses are comparable. However, patterns such as $X \rightarrow M_1$ and $X \rightarrow M_2$ with X as the tail-SPOF is possible, in which case potential stragglers are undetectable. To address this, *path diversity* prevents a potential tail-SPOF by enforcing independent, comparable paths. While the idea is simple, the challenge lies in efficiently removing potential input-SPOF, map-SPOF, reduce-SPOF, and output-SPOF in in every MapReduce stage:

(a) No input-SPOF in I→M paths: It is possible that map tasks on different nodes read inputs from the same node ($I_1 \rightarrow M_1$, $I_2 \rightarrow M_2$, and $I_1 = I_2$).[2] To enforce path diversity, map tasks must ask HDFS to diversify input nodes, at least to two (F+1) source nodes.

There are two possible designs, proactive and reactive. Proactive enforces all tasks of a job to synchronize with each other to verify the receipt of at least two input nodes. This early synchronization is not practical because tasks do not always start at the same time (depends on container availability). Furthermore, considering that in common cases not all jobs receive an input-SPOF, this approach imposes an unnecessary overhead.

We take the reactive approach. We let map tasks run independently in parallel, but when map tasks send their first heartbeats to the AM, they report their input nodes. If the AM detects a potential input-SPOF, it will reactively inform one (as F=1) of the tasks to ask HDFS namenode to re-pick another input node (*e.g.*, $I_2' \rightarrow M_2$ and $I_2' \neq I_1$).[3] After the switch (I_2 to I_2'), the task continues reading from the last read offset (no restart overhead).

(b) No map-SPOF in I→M and M→R paths: It is possible that map tasks are assigned to the same node ($I_1 \rightarrow M_1$, $I_2 \rightarrow M_2$, $M_1 = M_2$, and M_1/M_2's node is a potential tail-SPOF); note that Hadoop only disallows a backup and the original tasks to run in the same node (*e.g.*, $M_1 \neq M_1'$, $M_2 \neq M_2'$). Thus, to prevent one map-SPOF (F=1), we enforce at least two nodes (F+1) chosen for all the map tasks of a job. One caveat is when a job deploys only one map (reads only one input block), in which case we split it to two map tasks, each reading half of the input. This case however is very rare.

As of the implementation, when a job manager (AM) requests C containers from the resource manager (RM), the AM also supplies the rule. RM will then return $C-1$ containers to the AM first, which is important so that most tasks can start. For the last container, if the rule is not satisfied and no other node is currently available, RM must wait. To prevent starvation, if other tasks already finish half way, RM can break the rule.

(c) No reduce-SPOF in M→R and R→O paths: In a similar way, we enforce each job to have reducers at least in two different nodes. Since the number of reducers is defined by the user, not the runtime, the only way to prevent a potential reduce-SPOF is by cloning the single reducer. This is reasonable as a single reducer implies a small job and cloning small tasks is not costly [30].

(d) No output-SPOF in R→O paths: Output pipelines of *all* reducers can intersect the same node (*e.g.*, $R_1 \rightarrow O_1 \rightarrow O_1'$, $R_2 \rightarrow O_2 \rightarrow O_2'$, and $O_1 = O_2$). Handling this output-SPOF is similar to Rule (a). However, since write is different than read, the pipeline re-picking overhead can be significant if not designed carefully.

Through a few design iterations, we modify the reduce stage to *pre-allocate* write pipelines *during* shuffling and keep re-picking until all the pipelines are free from an output-SPOF. In vanilla Hadoop, write pipelines are created *after* shuffling (after reducers are ready

[1] F denotes the tolerable number of failures.

[2] A=B implies A and B are in the same node.

[3] A≠B implies A and B are *not* in the same node.

to write the output). Contrary, in our design, when shuffling finishes, the no-SPOF write pipelines are ready to use.

We now explain why pre-allocating pipelines removes a significant overhead. Unlike read switch in Rule (a), switching nodes in the middle of writes is not possible. In our strawman design, after pipeline creation (*e.g.*, $R_2 \rightarrow X \rightarrow O'_2 \rightarrow ...$), reducers report paths to AM and begin writing, similar to Rule (a). Imagine when an output-SPOF X is found but R_2 already wrote 5 MB. A simple switch (*e.g.*, $R_2 \rightarrow Y \rightarrow ...$) is *impossible* because R_2 no longer has the data (because R_2's HDFS client layer only buffers 4 MB of output). Filling Y with the already-transferred data from O'_2 will require complex changes in the storage layer (HDFS) and alter its append-only nature. Another way around is to create a *backup* reducer (R'_2) with a new no-SPOF pipeline (*e.g.*, $R'_2 \rightarrow Y \rightarrow ...$), which unfortunately incurs a high overhead as R'_2 must *repeat* the shuffle phase. For these reasons, we employ a background pre-allocation, which obviates pipeline switching in the middle of writes.

Another intricacy of output pipelines is that an output intersection does not always imply an output-SPOF. Let us consider $R_1 \rightarrow A \rightarrow X$ and $R_2 \rightarrow B \rightarrow X$. Although X is an output intersection, there is *enough* sub-path diversity needed to detect a tail-SPOF. Specifically, we can still compare the upper-stream $R_1 \rightarrow A$ and the lower-stream $A \rightarrow X$ to detect whether X is slow. Thus, as long as the intersection node is *not* the first node in *all* the write pipelines, pipeline re-picking is unnecessary. As an additional note, we collapse local-transfer edges; for example, if $R_1 \rightarrow A$ and $R_2 \rightarrow B$ are local disk writes, A and B are removed, resulting in $R_1 \rightarrow X$ and $R_2 \rightarrow X$, which will activate path diversity as X is a a potential tail-SPOF.

Finally, we would like to note that by default PBSE will follow the original task placement (including data locality) from the Hadoop scheduler and the original input source selection from HDFS. Only in rare conditions will PBSE break data locality. For example, let us suppose $I_1 \rightarrow M_1$ and $I_2 \rightarrow M_2$ achieve data locality and both data transfers happen in the same node. PBSE will try to move M_2 to another node (the "no map-SPOF" rule) ideally to one of the two other nodes that contain I_2's replicas (I'_2 or I''_2). But if the nodes of I'_2 and I''_2 do not have a free container, then M_2must be placed somewhere else and will read its input ($I_2/I'_2/I''_2$) remotely.

4.3 Detection and Speculation

As path diversity ensures no potential tail-SPOF, we then can compare paths, detect path-stragglers, and pinpoint the faulty node/NIC. Similar to base SE, PBSE detection algorithm is per-job (in AM) and runs for every MapReduce stage (input, shuffle, output). As an important note, PBSE runs side by side with the base SE; the latter handles task stragglers, while PBSE handles path stragglers. PBSE detection algorithm runs in three phases:

(1) Detecting path stragglers: In every MapReduce stage, AM collects a set of paths (S_P) and labels P as a potential straggling path if its BW (§4.1) is less than $\beta \times$ the average bandwidth, where $0 \leq \beta \leq 1.0$ (configurable). The straggling path will be speculated only if its estimated end time is longer than the estimated path replacement time (plus the standard deviation).[4] If a straggling path (*e.g.*, $A \rightarrow B$) is to be speculated, we execute the following phases below, in order to pinpoint which node (A or B) is the culprit.

[4]We omit our algorithm details because it is similar to task-level time-estimation and speculation [23]. The difference is that we run the speculation algorithm on path progresses, not just task-level progresses.

(2) Detecting the slow-NIC node with failure groups: We categorize every P in S_P into *failure/risk groups* [84]. A failure group G_N is created for every source/destination node N in S_P. If a path P involves a node N, P is put in G_N. For example, path $A \rightarrow B$ will be in G_A and G_B groups. In every group G, we take the total bandwidth. If there is a group whose bandwidth is smaller than $\beta \times$ the average of all group bandwidths, then a slow-NIC node is detected.

Let's consider the shuffling topology in Figure 3b with 1000Mbps normal links and a slow M_2's NIC at 5 Mbps. AM receives four paths ($M_1 \rightarrow R_1$, $M_1 \rightarrow R_2$, $M_2 \rightarrow R_1$, and $M_2 \rightarrow R_2$) along with their bandwidths (*e.g.*, 490, 450, 3, and 2 Mbps respectively). Four groups are created and the path bandwidths are grouped (M_1:{490, 450}, M_2:{3, 2}, R_1:{490, 3}, and R_2:{450, 2}). After the sums are computed, M_2's node (5 Mbps total) will be marked as the culprit, implying M_2 must be speculated, even though it is in the reduce stage (in PBSE, the stage does not define the straggler).

(3) Detecting the slow-NIC node with heuristics: Failure groups work effectively in cases with many paths (*e.g.*, many-to-many communications). In some cases, not enough paths exist to pinpoint the culprit. For example, given only a fast $A \rightarrow B$ and a straggling $C \rightarrow D$, we cannot pinpoint the faulty node (C or D). Fortunately, given three factors (path diversity rules, the nature of MapReduce stages, and existing rules in Hadoop SE), we can employ the following effective heuristics:

(a) Greedy approach: Let's consider a fast $I_1 \rightarrow M_1$ and a straggling $I_2 \rightarrow M_2$; the latter must be speculated, but the fault could be in I_2 or M_2. Fortunately, Hadoop SE by default prohibits M'_2 to run on the same node as M_2. Thus, we could speculate with $I_2 \rightarrow M'_2$. However, we take a greedy approach where we speculate a completely *new pair* $I'_2 \rightarrow M'_2$ (avoiding *both* I_2 and M_2 nodes). To implement this, when Hadoop spawns a task (M'_2), it can provide a blacklisted input source (I_2) to HDFS.

Arguably, node of I'_2 could be busier than I_2's node, and hence our greedy algorithm is sub-optimal. However, we find that HDFS does not employ a fine-grained load balancing policy; it only tries to achieve rack/node data locality (else, it uses a random selection). This simplicity is reasonable because Hadoop tasks are evenly spread out, hence a balanced cluster-wide read.

(b) Deduction approach: While the greedy approach works well in the Input→Map stage, other stages need to employ a deduction approach. Let's consider a one-to-one shuffling phase (a fast $M_1 \rightarrow R_1$ and a slow $M_2 \rightarrow R_2$). By deduction, since M_2 already "passes the check" in the $I \rightarrow M_2$ stage (it was not detected as a slow-NIC node), then the culprit is likely to be R_2. Thus, $M_2 \rightarrow R'_2$ backup path will start. Compared to deduction approach, employing a greedy approach in shuffling stage is more expensive (*e.g.*, speculating $M'_2 \rightarrow R'_2$ requires spawning M'_2).

(c) Dynamic retries: Using the same example above (slow $M_2 \rightarrow R_2$), caution must be taken if M_2 reads locally. That is, if $I \rightarrow M_2$ only involves a local transfer, M_2 is not yet proven to be fault-free. In this case, blaming M_2 or R_2 is only 50/50-chance correct. In such a case, we initially do not blame the map side because speculating

M_2 with M_2' is more expensive. We instead take the less expensive gamble; we first speculate the reducer (with R_2'), but if $M_2 \rightarrow R_2'$ path is also slow, we perform a *second* retry ($M_2' \rightarrow R_2'$). Put simply, sometimes it can take one or two retries to pinpoint one faulty node. We call this dynamic retries, which is different than the limited-retry in base SE (default of 1).

The above examples only cover I→M and M→R stages, but the techniques are also adopted for R→O stage.

5 EVALUATION

We implemented PBSE in Hadoop/HDFS v2.7.1 in 6003 LOC (3270 in AM, 1351 in Task Management, and 1382 in HDFS). We now evaluate our implementation.

Setup: We use Emulab nodes [15], each running a (dual-thread) 2×8-core Intel Xeon CPU E5-2630v3 @ 2.40GHz with 64GB DRAM and 1Gbps NIC. We use 15-60 nodes, 12 task slots per Hadoop node (the other 4 core for HDFS), and 64MB HDFS block size.[5] We set β=0.1 (§4.3), a non-aggressive path speculation.

Slowdown injection: We use Linux `tc` to delay *one* NIC to 60, 30, 10, 1, and 0.1 Mbps; 60-30 Mbps represent a contended NIC with 16-32 other network-intensive tenants and 10-0.1 Mbps represent a realistic degraded NIC; real cases of 10%-40% packet loss were observed in production systems (§2.2), which translate to 2 Mbps to 0.1 Mbps NIC (as throughput exhibits exponential-decaying pattern with respect to packet loss rate).

Workload: We use real-world production workloads from Facebook (FB2009 and FB2010) and Cloudera (CC-b and CC-e) [41]. In each, we pick a sequence of 150 jobs[6] with the lowest inter-arrival time (*i.e.*, a busy cluster). We use SWIM to replay and rescale the traces properly to our cluster sizes as instructed [24]. Figure 6 shows the distribution of job sizes and inter-arrival times.

Metrics: We use two primary metrics: job duration (T) and speed-up ($=T_{Base}/T_{PBSE}$).

5.1 PBSE vs. Hadoop (Base) SE

Figure 7 shows the CDF of latencies of 150 jobs from FB2010 on 15 nodes with five different setups from right (worse) to left (better): Base Hadoop SE with *one* 1Mbps slow NIC (`BaseSE-1Slow`), PBSE with the same slow NIC (`PBSE-1Slow`), PBSE without any bad NIC (`PBSE-0Slow`), Base SE without any bad NIC (`BaseSE-0Slow`), and Base SE with one dead node (`BaseSE-1Dead`).

We make the following observations from Figure 7. First, as alluded in §3, Hadoop SE cannot escape tail-SPOF caused by the degraded NIC, resulting in long job tail latencies with the longest job finishing after 6004 seconds (`BaseSE-1Slow` line). Second, PBSE is much more effective than Hadoop SE; it successfully cuts tail latencies induced by degraded NIC (`PBSE-1Slow` vs. `BaseSE-1Slow`). Third, PBSE cannot reach the "perfect" scenario (`BaseSE-0Slow`); we dissect this more later (§5.2). Fourth, with Hadoop SE, a slow NIC is worse than a dead node (`BaseSE-1Slow` vs. `BaseSE-1Dead`); put simply, Hadoop is robust against fail-stop failures but not degraded network. Finally, in the normal case, PBSE does not exhibit any overhead; the resulting job latencies in PBSE and Hadoop SE under no failure are similar (`PBSE-0Slow` vs. `Base-0Slow`).

We now perform further experiments by varying the degraded NIC bandwidth (Figure 8a), workload (8b), and cluster size (8c). To compress the resulting figures, we will only show the speedup of PBSE over Hadoop SE (a more readable metric), as explained in the figure caption.

Varying NIC degradation: Figure 8a shows PBSE speed-ups when we vary the NIC bandwidth of the slow node to 60, 30, 10, 1, and 0.1 Mbps (the FB2010 and 15-node setups are kept the same). We make two observations from this figure. First, PBSE has higher speed-ups at higher percentiles. In Hadoop SE, if a large job is "locked" by a tail-SPOF, the job's duration becomes extremely long. PBSE on the other hand can quickly detect and failover from the

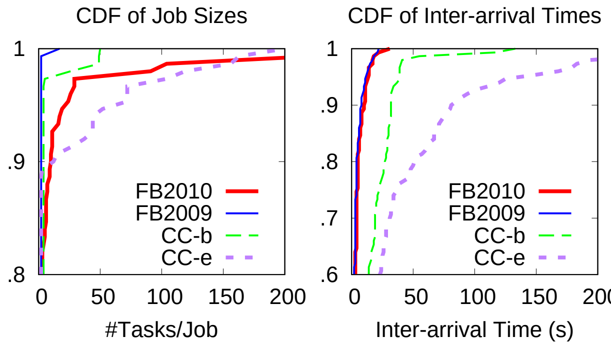

Figure 6: Distribution of job sizes and inter-arrival times. ***The left figure shows CDF of the number of (map) tasks per job within the chosen 150 jobs from each of the production traces. The number of reduce tasks is mostly 1 in all the jobs. The right figure shows the CDF of job inter-arrival times.***

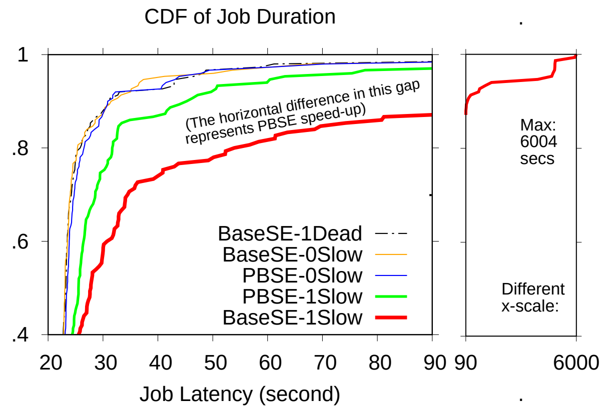

Figure 7: PBSE vs. Hadoop (Base) SE (§5.1). ***The figure above shows CDF of latencies of 150 FB2010 jobs running on 15 nodes with one 1-Mbps degraded NIC*** (`1Slow`)***, no degraded NIC*** (`0Slow`)***, and one dead node*** (`1Dead`)***.***

[5]Today, HDFS default block size is 128 MB, which actually will show better PBSE results because of the longer data transfer. We use 64 MB to be consistent with all of our initial experiments.

[6]150 jobs are chosen so that every normal run takes about 15 minutes; this is because the experiments with severe delay injections (*e.g.*, 1 Mbps) can run for hours for base Hadoop. Longer runs are possible but will prevent us from completing many experiments.

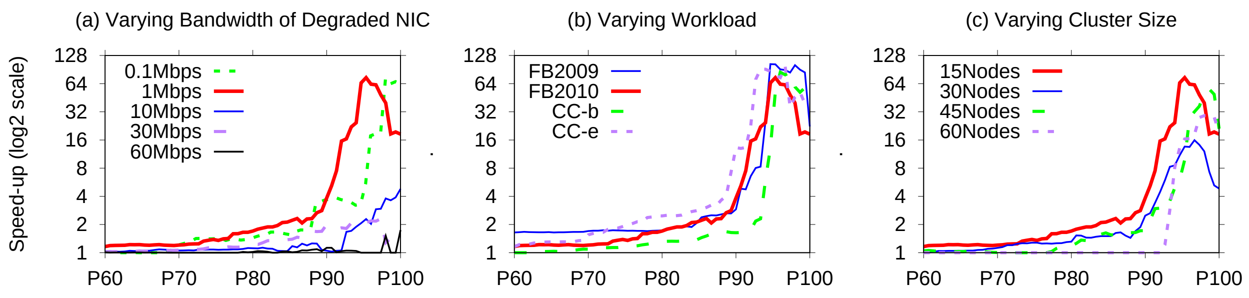

Figure 8: PBSE speed-ups (vs. Hadoop Base SE) with varying (a) degradation, (b) workload, and (c) cluster size (§5.1). ***The x-axis above represents the percentiles (y-axis) in Figure 7 (e.g., "P80" denotes 80th-percentile). For example, the bold (`1Mbps`) line in Figure 8a plots the PBSE speedup at every percentile from Figure 7 (i.e., the horizontal difference between the `PBSE-1Slow` and `Base-1Slow` lines). As an example point, in Figure 7, at 80th percentile (y=0.8), our speed-up is 1.7× (T_{Base}/T_{PBSE} = 54sec/32sec) but in Figure 8a, the axis is reversed for readability (at x=P80, PBSE speedup is y=1.7).***

Features	In stage:	#Tasks	#Jobs
Path	I→M	66	59
Diversity	M→R	125	125
(§4.2)	R→O	0	0
Path	I→M	62	54
Speculation	M→R	26	6
(§4.3)	R→O	28	12

Table 2: Activated PBSE features (§5.2). ***The table shows how many times each feature is activated, by task and job counts, in the*** `PBSE-1Slow` ***experiment in Figure 7.***

straggling paths. With a 60Mbps congested NIC, PBSE delivers some speed-ups (1.5-1.7×) above P98. With a more congested NIC (30 Mbps), PBSE benefits start to become apparent, showing 1.5-2× speed-ups above P90. Second, PBSE speed-up increases (2-70×) when the NIC degradation is more severe (*e.g.*, the speedups under 1 Mbps are relatively higher than 10 Mbps). However, under a very severe NIC degradation (0.1 Mbps), our speed-up is still positive but slightly reduced. The reason is that at 0.1 Mbps, the degraded node becomes highly congested, causing timeouts and triggering fail-stop failover. Again, in Hadoop SE, a dead node is better than a slow NIC (Figure 7). The dangerous point is when a degraded NIC slows down at a rate that does not trigger any timeout.

Varying workload and cluster size: Figure 8b shows PBSE speed-ups when we vary the workloads: FB2009, FB2010, CC-b, CC-e (1Mbps injection and 15-node setups are kept the same). As shown, PBSE works well in many different workloads. Finally, Figure 8c shows PBSE speed-ups when we vary the cluster size: 15 to 60 nodes (1Mbps injection and FB2010 setups are kept the same). The figure shows that regardless of the cluster size, a degraded NIC can affect many jobs. The larger the cluster size, tail-SPOF probability is reduced but still appear at a significant rate (> P90).

5.2 Detailed Analysis

We now dissect our first experiment's results (Figure 7).

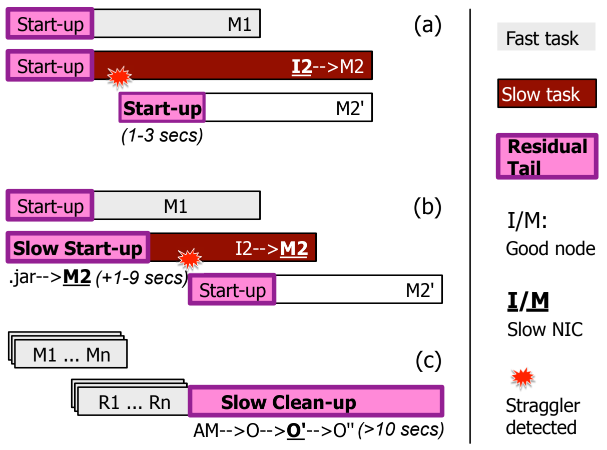

Figure 9: Residual sources of tail latencies (§5.2).

Activated features: Table 2 shows how many times PBSE features (§4.2-4.3) are activated in the `PBSE-1Slow` experiment in Figure 7. First, in terms of path diversity, 66 tasks (59 jobs) require I→M diversity. In the FB2010 workload, 125 jobs only have one reducer, thus requiring M→R diversity (reducer cloning). R→O diversity is rarely needed (0), mainly because of enough upper-stream paths to compare. Second, in terms of path speculation, we can see that all the I→M, M→R, and R→O speculations are needed (in 54, 6, and 12 jobs, respectively) to help the jobs escape all the many SE loopholes we discussed before (§3).

Residual overhead: We next discuss interesting findings on why PBSE cannot reach the "perfect" case (`Base-0Slow` vs. `PBSE-1Slow` lines shown in Figure 7). Below are the sources of residual overhead that we discovered:

Start-up overhead: Figure 9a shows that even when a straggling path (I_2→M_2) is detected early, running the backup task (M'_2) will require 1-3 seconds of start-up overhead, which includes JVM warm-up (class loading, interpretation) and "localization" [20] (including transferring application's `.jar` files from HDFS to the task's node).

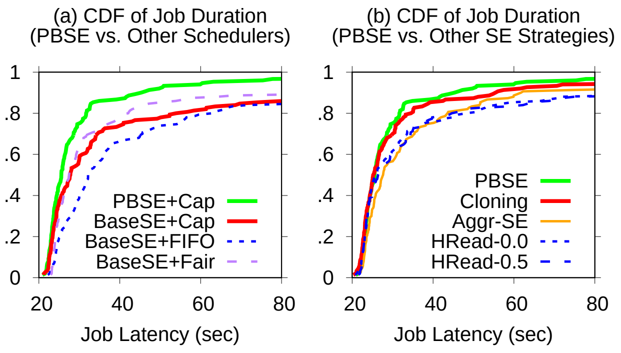

Figure 10: PBSE vs. other strategies (§5.3).

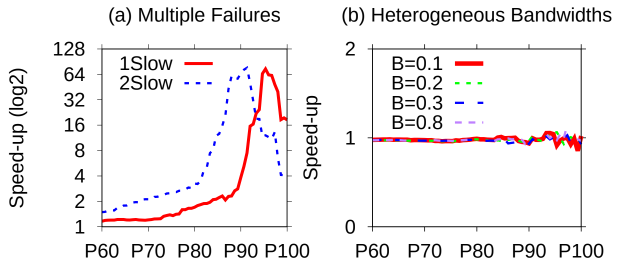

Figure 11: PBSE speed-ups with multiple failures and heterogeneous network (§5.4 and §5.5).

At this point, this overhead is not removable and still become an ongoing problem [60].

"Straggling" start-up/localization: Figure 9b shows two maps with one of them (M_2) running on a slow-NIC node. This causes the transferring of application's `.jar` files from HDFS to M_2's node to be slower than the one in M_1. In our experience, this "straggling" start-up can consume 1-9 seconds (depending on the job size). What we find interesting is that start-up time is *not* accounted in task progress and SE decision making. In other words, start-up durations of M_1 and M_2 are *not* compared, and hence no' straggling start-up is detected, delaying the task-straggler detection.

"Straggling" clean-up: Figure 9c shows that after a job finishes (but before returning to user), the AM must write a `JobHistory` file to HDFS (part of the clean-up operation). It is possible that one of the output nodes is slow (*e.g.*, AM$\rightarrow$O$\rightarrow\underline{O'}\rightarrow$O$''$), in which case AM will be stuck in this "straggling" clean-up, especially with a large `JobHistory` file from a large job ($M_1..M_n$, $R_1..R_n$, $n \gg 1$). This case is also *outside* the scope of SE.

We believe the last two findings reveal more flaws of existing tail-tolerance strategies: they only focus on "task progress," but do not include "operational progress" (start-up/clean-up) as part of SE decision making, which results in *irremovable* tail latencies. Again, these flaws surface when our unique fault model (§2.3) is considered. Fortunately, all the problems above are solvable; start-up overhead is being solved elsewhere [60] and straggling localization/clean-up can be unearthed by incorporating start-up/clean-up paths and their latencies as part of SE decision making.

5.3 PBSE vs. Other Strategies

In this section, we will compare PBSE against other scheduling and tail-tolerant approaches.

Figure 10a shows the same setup as in Figure 7, but now we vary the scheduling configurations: capacity (default), FIFO, and fair scheduling. The figure essentially confirms that the SE loopholes (§3) are *not* about scheduling problems; changing the scheduler does not eliminate tail latencies induced by the degraded NIC.

Figure 10b shows the same setup, but now we vary the tail-tolerant strategies: hedged read (0.5s), hedged read (0s), aggressive SE, and task cloning. The first one, hedged read (`HRead-0.5`), is a new HDFS feature [25] that enables a map task (*e.g.*, $I_2 \rightarrow M_2$) to automatically read from another replica ($I'_2 \rightarrow M_2$) if the first 64KB packet is not received after 0.5 sec. The map will use the data from the fastest read. The second one (`HRead-0.0`) does not wait at all. Hedged read can unnecessarily consume network resources. As shown, `HRead-0.0` does not eliminate all the tail-SPOF latencies (with a job latency maximum of 7708 seconds). This is because hedged read *only* solves the tail-SPOF scenarios in the *input* stage, but *not* across *all* the MapReduce stages.

The third one (`Aggr-SE`) is the base SE but with the most aggressive SE configuration[7] and the last one (`Cloning`) represents task-level cloning[8] [30] (a.k.a. hedged requests [42]). Aggressive SE speculates more intensively in all the MapReduce stages, but long tail latencies still appear (with a maximum of 6801 seconds). Even cloning also exhibits a long tail (3663 seconds at the end of the tail) as it still inherits the flaws of base SE. In this scenario, PBSE is the most effective (a maximum of only 251 seconds), as it solves the fundamental limitations of base SE.

In summary, all other approaches above only *reduce* but do *not eliminate* the possibility of tail-SPOF to happen. We did not empirically evaluate PBSE with other strategies in the literature [30, 31, 33, 63, 64, 77] because either they are either proprietary or integrated to non-Hadoop frameworks (*e.g.*, Dryad [54], SCOPE [39], or Spark [13]), but they were discussed earlier (§3.4).

5.4 PBSE with Multiple Failures

We also evaluate PBSE against two NIC failures (both 1 Mbps). This is done by changing the value of F (§4). To handle two slow-NIC nodes (F=2), at least three nodes (F+1) are required for path diversity. F=3 is currently not possible as the replication factor is only 3×. Figure 11a shows that PBSE performs effectively as well under the two-failure scenario.

5.5 PBSE on Heterogeneous Resources

To show that PBSE does not break the performance of Hadoop SE under heterogeneous resources [83], we run PBSE on a stable-state

[7]Aggressive SE configurations:
`speculative-cap-running-tasks`=1.0 (default=0.1);
`speculative-cap-total-tasks`=1.0 (default=0.01);
`minimum-allowed-tasks`=100 (default=10);
`retry-after-speculate`=1000ms (default=15000ms); and
`slowtaskthreshold`=0.1 (default=1.0).

[8]We slightly modified Hadoop to always speculate every task at the moment the task starts (Hadoop does not support cloning by default).

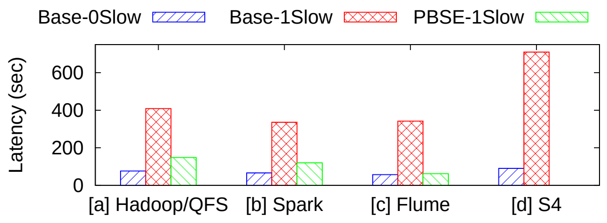

Figure 12: Beyond Hadoop/HDFS (§6). ***The figure shows latencies of microbenchmarks running on four different systems (Hadoop/QFS, Spark, Flume, and S4) with three different setups: baseline without degraded NIC (*** `Base-0Slow`***), baseline with one 1Mbps degraded NIC (*** `Base-1Slow`***), and with initial PBSE integration (*** `PBSE-1Slow`***). Baseline (*** `Base`***) implies the vanilla versions. The Hadoop/QFS microbenchmark and topology is shown in Figure 13c. The Spark microbenchmark is a 2-stage, 4-task, all-to-all communication as similarly depicted in Figure 3b. The Flume and S4 microbenchmarks have the same topology. We did not integrate PBSE to S4 as its development is discontinued.***

network throughput distribution in Amazon EC2 that is popularly cited (ranging from 200 to 920 Mbps; please see Figure 3 in [71] for the detailed distribution). Figure 11b shows that PBSE speed-up is constantly around one with small fluctuations at high percentiles from large jobs. The figure also shows the different path-straggler thresholds we use (β in §4.3). With a higher threshold, sensitivity is higher and more paths are speculated. However, because the heterogeneity is not severe (>100 Mbps), the original tasks always complete faster than the backup tasks.

5.6 Limitations

PBSE can fail in extreme corner cases: for example, if a file currently only has one surviving replica (path diversity is impossible); if a large batch of devices degrade simultaneously beyond the tolerable number of failures; or if there is not enough node availability. Note that the base Hadoop SE also fails in such cases. When these cases happen, PBSE can log warning messages to allow operators to query the log and correlate the warnings with slow jobs (if any).

Another limitation of PBSE is that in a virtualized environment (*e.g.*, EC2), if nodes (as VMs) are packed to the same machine, PBSE's path diversity will not work. PBSE works if the VMs are deployed across many machines and they expose the machine#.

6 BEYOND HADOOP AND HDFS

To show PBSE generality for many other data-parallel frameworks beyond Hadoop/HDFS, we analyzed Apache Spark [13, 82], Flume [9], S4 [12], and Quantcast File System (QFS) [67]. We found that all of them suffer from the tail-SPOF problem, as shown in Figure 12 (`Base-0Slow` vs. `Base-1Slow` bars). We have performed an initial integration of PBSE to Spark, Flume, and Hadoop/QFS stack and showed that it speculates effectively, avoids the degraded network, and cuts tail latencies (`PBSE-1Slow` bars). We did not integrate further to S4 as its development is discontinued.

We briefly describe the tail-tolerance flaws we found in these other systems. Spark (Figure 12b) has a built-in SE similar to Hadoop SE, hence it is prone to the same tail-SPOF issues (§3). Flume (Figure 12c) uses a static timeout to detect slow channels (no path comparisons). If a channel's throughput falls below 1000 events/sec, a failover is triggered. If a NIC degrades to slightly above the threshold, the timeout is not triggered. S4 (Figure 12d) has a similar static timeout to Flume. Worse, it has a shared queue design in the fan-out protocol. Thus, a slow recipient will cripple the sender in transferring data to the other recipients (a head-of-line blocking problem).

We also analyzed the Hadoop/QFS stack due to its differences from the Hadoop/HDFS (3-way replication) stack.[9] Hadoop/QFS represents computation on erasure-coded (EC) storage. Many EC storage systems [53, 67, 74] embed tail-tolerant mechanisms in their client layer. EC-level SE with *m* parities can tolerate up to *m* slow NICs. In addition to tolerating slow NICs, EC-level SE can also tolerate *rack-level* slowdown (which can be caused by a degraded TOR switch or a malfunctioning power in the rack). For example in Figure 13a, M_1 reads chunks of a file (I_a, I_b). As reading from I_b is slower than from I_a (due to the slow Rack-2), the EC client layer triggers its own EC-level SE, creating a backup speculative read from I_p to construct the late I_b.

Unfortunately, EC-level SE also has loopholes. Figure 13b shows a similar case but with slow Rack-1. Here, the EC-level SE is not triggered as *all* reads ($I_a \rightarrow M_1$, $I_b \rightarrow M_1$) are slow. Let's suppose another map (M_2) completes fast in Rack-3, as in Figure 13c. Here, Hadoop declares M_1 as a straggler, however it is possible that the backup M'_1 will run in Rack-1, which means it must *also* read through the slow rack. As a result, *both* original and backup tasks (M_1 and M'_1) are straggling.

To address this, with PBSE, the EC layer (QFS) exposes the individual read paths to Hadoop. In Figure 13c, if we expose $I_a \rightarrow M_1$ and $I_b \rightarrow M_1$ paths, Hadoop can try placing the backup M'_1 in another rack (Rack-2/3) and furthermore informs the EC layer to have M'_1 directly read from I_b and I_p (instead of I_a). Overall, the *key principle* is the same: when path progresses are exposed, the compute and storage layers can make a more informed decision. Figure 12a shows that in a topology like Figure 13c, without PBSE, the job follows the slow Rack-1 performance (`Base-1Slow` bar), but with PBSE, the job can escape from the slow rack (`PBSE-1Slow`).

In summary, we have performed successful initial integrations of PBSE to multiple systems, which we believe show its generality to any data-parallel frameworks that need robust tail tolerance against node-level network degradation. We leave full testing of these additional integrations as a future work.

7 RELATED WORK

We now discuss related work.

Paths: The concept of "paths" is prevalent in the context of modeling (*e.g.*, Magpie [38]), fine-grained tracing (*e.g.*, XTrace [45],

[9]Quantcast File System (QFS) [19, 67] is a Reed-Solomon (RS) erasure-coded (EC) distributed file system. Although HDFS-RAID supports RS [22], when we started the project (in 2015/2016), the RS is only executed in the background. In HDFS-RAID, files are still triple-replicated initially. Moreover, because the stripe size is the same as the block size (64 MB), only large files are erasure coded while small files are still triple replicated (*e.g.*, with RS(10,4), only files with 10×64MB size are erasure coded). HDFS with foreground EC (HDFS-EC) was still an ongoing non-stable development [17]. In contrast, QFS erasure-code data in the foreground with 64KB stripe size, hence our usage of QFS.

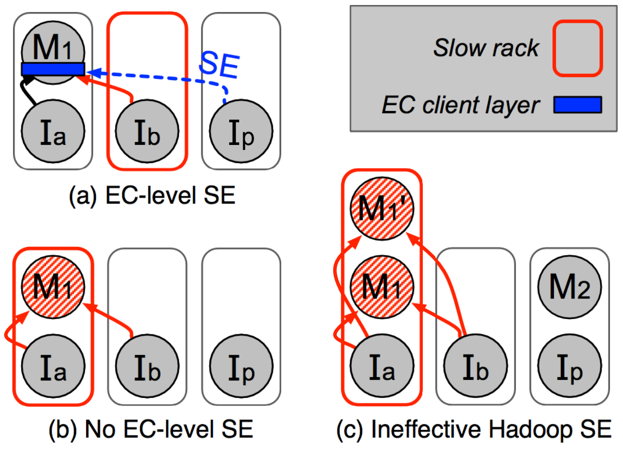

Figure 13: Rack slowdown and Erasure-Coded (EC) storage (§6). ***For simplicity, the figure shows RS(2,1), a Reed Solomon where an input file I is striped across two chunks (I_a, I_b) with one parity (I_p) with 64KB stripe size (see Figure 2 in [67] for more detail).***

Pivot Tracing [62]), diagnosis (*e.g.*, black-box debugging [28], path-based failure [40]), and availability auditing (*e.g.*, INDaaS [84]), among many others. This set of work is mainly about monitoring and diagnosing paths. In PBSE, we actively "control" paths, for a better online tail tolerance.

Tail tolerance: Earlier (§3.4), we have discussed a subtle limitation of existing SE implementations that hide path progresses [30, 32, 33, 63, 64, 83]. While SE is considered a reactive tail-tolerance, proactive ones have also been proposed, for example by cloning (*e.g.*, Dolly [30] and hedged requests [42]), launching few extra tasks (*e.g.*, KMN [70]), or placing tasks more intelligently (*e.g.*, Wrangler [77]). This novel set of work also uses the classical definition of progress score (§3.4). Probabilistically, cloning or launching extra tasks can reduce tail-SPOF, but fundamentally, as paths are not exposed and controlled, tail-SPOF is still *possible* (§5.3). Tail tolerance is also deployed in the storage layer (*e.g.*, RobuStore [74], CostTLO [73], C3 [69], Azure [53]). PBSE shows that storage and compute layers need to collaborate for a more robust tail tolerance.

Task placement/scheduling: There is a large body of work on task placement and scheduling (*e.g.*, Pacman [31], delay scheduling [81], Quincy [55], Retro [61]). These efforts attempt to cut tail latencies in the *initial* task placements by achieving better data locality, load balancing, and resource utilization. However, they do not modify SE algorithms, and thus SE (and PBSE) is orthogonal to this line of work.

Tail root causes: A large body of literature has discovered many root causes of tail latencies including resource contention of shared resources [33, 35], hardware performance variability [59], workload imbalance [46], data and compute skew [29], background processes [59], heterogeneous resources [83], degraded disks or SSDs [51, 78], and buggy machine configuration (*e.g.*, disabled process caches) [43]. For degraded disks and processors, as we mentioned before (§1), existing (task-based) speculations are sufficient to detect such problems. PBSE highlights that degraded network is an important fault model to address.

Iterative graph frameworks: Other types of data-parallel systems include iterative graph processing frameworks [10, 27, 65]. As reported, they do not employ speculative execution within the frameworks, but rather deal with stragglers within the running algorithms [52]. Our initial evaluation of Apache Giraph [10] shows that a slow NIC can hamper the entire graph computation, mainly because Giraph workers must occasionally checkpoint its states to HDFS (plus the use of barrier synchronization), thus experiencing a tail-SPOF (as in Figure 3c). This suggests that part of PBSE may be applicable to graph processing frameworks as well.

Distributed system bugs: Distributed systems are hard to get right. A plethora of related work combat a variety of bugs in distributed systems, including concurrency [50, 57], configuration [75], dependency [84], error/crash-handling [56, 80], performance [62], and scalability [58] bugs. In this paper, we highlight performance issues caused by speculative execution bugs/loopholes.

8 CONCLUSION

Performance-degraded mode is dangerous; software systems tend to continue using the device without explicit failure warnings (hence, no failover). Such intricate problems took hours or days until manually diagnosed, usually after whole-cluster performance is affected (a cascading failure) [2, 3, 8]. In this paper, we show that node-level network degradation combined with SE loopholes is dangerous. We believe it is the responsibility of software's tail-tolerant strategies, not just monitoring tools, to properly handle performance-degraded network devices. To this end, we have presented PBSE as a novel, online solution to the problem.

9 ACKNOWLEDGMENTS

We thank Walfredo Cirne, our shepherd, and the anonymous reviewers for their tremendous feedback and comments. We also would like to thank Andrew Wang and Aaron T. Myers for their help on Cloudera Hadoop traces, Achmad Imam Kistijantoro for his supervision on Martin's undergraduate thesis, Yohanes Surya and Nina Sugiarto for their support, and Bob Bartlett, Kevin Harms, Robert Ricci, Kirk Webb, Gary Grider, Parks Fields, H. Birali Runesha, Andree Jacobson, Xing Lin, and Dhruba Borthakur for their stories on degraded network devices. This material was supported by funding from NSF (grant Nos. CCF-1336580, CNS-1350499, CNS-1526304, CNS-1405959, and CNS-1563956) as well as generous donations from Dell EMC, Huawei, Google Faculty Research Award, NetApp Faculty Fellowship, and CERES Center for Unstoppable Computing. The experiments in this paper were performed in the Utah Emulab [15, 72], the NMC PRObE [18, 47], the University of Chicago RIVER [21], and Chameleon [14] clusters.

Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

REFERENCES

[1] Personal Communication from datacenter operators of University of Chicago IT Services.
[2] Personal Communication from Kevin Harms of Argonne National Laboratory.
[3] Personal Communication from Robert Ricci of University of Utah.
[4] Personal Communication from Gary Grider and Parks Fields of Los Alamos National Laboratory.
[5] Personal Communication from Xing Lin of NetApp.
[6] Personal Communication from H. Birali Runesha (Director of Research Computing Center, University of Chicago).
[7] Personal Communication from Andree Jacobson (Chief Information Officer at New Mexico Consortium).
[8] Personal Communication from Dhruba Borthakur of Facebook.
[9] Apache Flume. http://flume.apache.org/.
[10] Apache Giraph. http://giraph.apache.org/.
[11] Apache Hadoop. http://hadoop.apache.org.
[12] Apache S4. http://incubator.apache.org/s4/.
[13] Apache Spark. http://spark.apache.org/.
[14] Chameleon. https://www.chameleoncloud.org.
[15] Emulab Network Emulation Testbed. http://www.emulab.net.
[16] HDFS-8009: Signal congestion on the DataNode. https://issues.apache.org/jira/browse/HDFS-8009.
[17] Introduction to HDFS Erasure Coding in Apache Hadoop. http://blog.cloudera.com/blog/2015/09/introduction-to-hdfs-erasure-coding-in-apache-hadoop/.
[18] Parallel Reconfigurable Observational Environment (PRObE). http://www.nmc-probe.org.
[19] QFS. https://quantcast.github.io/qfs/.
[20] Resource Localization in Yarn: Deep dive. http://hortonworks.com/blog/resource-localization-in-yarn-deep-dive/.
[21] RIVER: A Research Infrastructure to Explore Volatility, Energy-Efficiency, and Resilience. http://river.cs.uchicago.edu.
[22] Saving capacity with HDFS RAID. https://code.facebook.com/posts/536638663113101/saving-capacity-with-hdfs-raid/.
[23] Speculative tasks in Hadoop. http://stackoverflow.com/questions/34342546/speculative-tasks-in-hadoop.
[24] Statistical Workload Injector for MapReduce (SWIM). https://github.com/SWIMProjectUCB/SWIM/wiki.
[25] Support 'hedged' reads in DFSClient. https://issues.apache.org/jira/browse/HDFS%2D5776.
[26] WorldâĂŹs First 1,000-Processor Chip. https://www.ucdavis.edu/news/worlds-first-1000-processor-chip/.
[27] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, , and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
[28] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.
[29] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters. In *Proceedings of the 2011 EuroSys Conference (EuroSys)*, 2011.
[30] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.
[31] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.
[32] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. GRASS: Trimming Stragglers in Approximation Analytics. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.
[33] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.
[34] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf.
[35] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, Dave Patterson, and Kathy Yelick. Cluster I/O with River: Making the Fast Case Common. In *The 1999 Workshop on Input/Output in Parallel and Distributed Systems (IOPADS)*, 1999.
[36] Peter Bailis and Kyle Kingsbury. The Network is Reliable. An informal survey of real-world communications failures. *ACM Queue*, 12(7), July 2014.
[37] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2007.
[38] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richar Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.
[39] Ronnie Chaiken, Bob Jenkins, Paul Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB)*, 2008.
[40] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Dave Patterson, Armando Fox, and Eric Brewer. Path-Based Failure and Evolution Management. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.
[41] Yanpei Chen, Sara Alspaugh, and Randy H. Katz. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. In *Proceedings of the 38th International Conference on Very Large Data Bases (VLDB)*, 2012.
[42] Jeffrey Dean and Luiz AndrÃľ Barroso. The Tail at Scale. *Communications of the ACM*, 56(2), February 2013.
[43] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.
[44] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.
[45] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-Trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.
[46] Rohan Gandhi, Di Xie, and Y. Charlie Hu. PIKACHU: How to Rebalance Load in Optimizing MapReduce On Heterogeneous Clusters. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, 2013.
[47] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. Probe: A thousand-node experimental cluster for computer systems research. *USENIX ;login:*, 38(3), June 2013.
[48] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.
[49] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, 2016.
[50] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.
[51] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.
[52] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Addressing the straggler problem for iterative convergent parallel ML. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, 2016.
[53] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure Coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, 2012.
[54] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 EuroSys Conference (EuroSys)*, 2007.
[55] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[56] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[57] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[58] Tanakorn Leesatapornwongsa, Cesar A. Stuardo, Riza O. Suminto, Huan Ke, Jeffrey F. Lukman, and Haryadi S. Gunawi. Scalability Bugs: When 100-Node Testing is Not Enough. In *The 16th Workshop on Hot Topics in Operating Systems (HotOS XVII)*, 2017.

[59] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[60] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. DonâĂŹt Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[61] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[62] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[63] Kristi Morton, Magdalena Balazinska, and Dan Grossman. ParaTimer: a progress indicator for MapReduce DAGs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2010.

[64] Kristi Morton, Abram L. Friesen, Magdalena Balazinska, and Dan Grossman. Estimating the progress of MapReduce pipelines. In *Proceedings of the 26th International Conference on Data Engineering (ICDE)*, 2010.

[65] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[66] Neda Nasiriani, Cheng Wang, George Kesidis, and Bhuvan Urgaonkar. Using Burstable Instances in the Public Cloud: When and How? 2016.

[67] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The Quantcast File System. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB)*, 2013.

[68] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB)*, 2013.

[69] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[70] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J. Franklin, and Ion Stoica. The Power of Choice in Data-Aware Cluster Scheduling. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[71] Guohui Wang and T. S. Eugene Ng. The impact of virtualization on network performance of amazon EC2 data center. In *The 29th IEEE International Conference on Computer Communications (INFOCOM)*, 2010.

[72] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[73] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[74] Huaxia Xia and Andrew A. Chien. RobuSTore: Robust Performance for Distributed Storage Systems. In *Proceedings of the 2007 Conference on High Performance Networking and Computing (SC)*, 2007.

[75] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do Not Blame Users for Misconfigurations. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[76] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[77] Neeraja J. Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. Wrangler: Predictable and Faster Jobs using Fewer Resources. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[78] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[79] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. In *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB)*, 2014.

[80] 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 *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[81] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 2010 EuroSys Conference (EuroSys)*, 2010.

[82] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[83] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[84] Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford. Heading Off Correlated Failures through Independence-as-a-Service. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.