Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems

Thanh Do[†], Mingzhe Hao, Tanakorn Leesatapornwongsa,
Tiratat Patana-anake, and Haryadi S. Gunawi

University of Chicago [†] University of Wisconsin-Madison

Abstract

We highlight one often-overlooked cause of performance failure: limpware – "limping" hardware whose performance degrades significantly compared to its specification. We report anecdotes of degraded disks and network components seen in large-scale production. To measure the system-level impact of limpware, we assembled limpbench, a set of benchmarks that combine data-intensive load and limpware injections. We benchmark five cloud systems (Hadoop, HDFS, ZooKeeper, Cassandra, and HBase) and find that limpware can severely impact distributed operations, nodes, and an entire cluster. From this, we introduce the concept of limplock, a situation where a system progresses slowly due to the presence of limpware and is not capable of failing over to healthy components. We show how each cloud system that we analyze can exhibit operation, node, and cluster limplock. We conclude that many cloud systems are not limpware tolerant.

1 Introduction

The success of cloud computing can be summarized with three supporting trends: the incredible growth of hardware performance and capacity ("big pipes"), the continuous success of software architects in building scalable distributed systems on thousands of big pipes, and the "Big Data" collected and analyzed at massive scale in a broad range of application areas. These success trends nevertheless bring a growing challenge: to ensure big data continuously flows in big pipes, cloud systems must deal with all kinds of failures, including hardware failures, software bugs, administrator mistakes, and many others. All of these lead to *performance failures*, which is considered a big "nuisance" in large-scale system management. Recent work has addressed many sources of performance failures such as heterogeneous systems [30, 52], unbalanced resource allocation [29, 42, 47], software bugs [33], configuration mistakes [15] and straggling tasks [13, 22].

In this paper, we highlight one often-overlooked cause of performance failures: *limpware* – "limping" hardware[1] whose performance degrades significantly compared to its specification. The growing complexity of technology scaling, manufacturing, design logic, usage, and operating environment increases the occurrence of limpware. We believe this trend will continue, and the concept of performance perfect hardware no longer holds. We have collected reports that show how disk and network performance can drop by orders of magnitude.

From these reports, we also find that unmanaged limpware can lead to cascades of performance failures across system components. For example, "there was a case of a 1-Gbps NIC card on a machine that suddenly was transmitting only at 1 Kbps, which then caused a chain reaction upstream in such a way that the performance of the entire workload of a 100-node cluster was crawling at a snail's pace, effectively making the system unavailable for all practical purposes" [8]. We name this condition as *limplock*[2], a situation where a system progresses extremely slow due to limpware and is not capable of failing over to healthy components (*i.e.*, the system enters and cannot exit from limping mode).

These stories led us to raise the following questions:

SOCC'13, October 01 - 03 2013, Santa Clara, CA, USA.

http://dx.doi.org/10.1145/2523616.2523627

[1] In automotive industry, the term "limp mode" is commonly used to describe a situation where vehicle computer receives sensor signals outside its programmed specifications. The same term is often used for software systems that exhibit performance faults [34]. We adopt the same term in the context of degraded hardware.

[2] Analogous to gridlock.

Are today's cloud systems susceptible to limplock? What are the system-level impacts of limpware on cloud systems, and how to quantify them? Why limpware in a machine can significantly degrade other nodes or even the entire cluster? Why does this happen in current system designs?

To address these questions, we assembled *limpbench*, a set of benchmarks that combine data-intensive load and limpware injections (*e.g.*, a degraded NIC or disk). We benchmark five popular and varied scale-out systems (Hadoop, HDFS, ZooKeeper, Cassandra, and HBase). With this, we unearth distributed protocols and system designs that are susceptible to limplock. We also show how limplock can cascade in these systems, for example, how a single slow NIC can make many map/reduce tasks enter limplock and eventually make a whole Hadoop cluster in limplock.

The limpbench results show that limpware can cripple not only the operations running on it, but also other healthy nodes, or even worse, a whole cluster. To classify such cascading failures, we introduce the concepts of *operation*, *node*, and *cluster limplock*. Operation limplock is the smallest measure of limplock where only the operations that involve limpware are experiencing slowdowns. Node limplock is a more severe condition where operations that must be served by a limplocked node will be affected although the operations do not involve limpware. Finally, cluster limplock is the most severe situation where limpware makes the performance of an entire cluster collapse.

We present how these three classes of limplock can occur in our target systems. We also pinpoint system designs that allow limplock to occur. For example, we find issues such as coarse-grained timeouts, single point of performance failure, resource exhaustion due to multi-purpose threads/queues, memoryless/revokeless retries, and backlogs in unbounded queues.

In conclusion, our findings show that although today's cloud systems utilize redundant resources, they are not capable of making limpware to "fail in place". Performance failures cascade, productivity is reduced, resources are underutilized, and energy is wasted. Therefore, we advocate that limpware should be considered as a "new" and important failure mode that future cloud systems should manage.

In the next section, we report again limpware occurrences that we have collected in our previous work [24]. Then, we present the new contributions of this paper:

- We present the concept of limplock and its three subclasses: operation, node and cluster limplock, along with system designs that allow them to happen (Section 3).
- We describe limpbench for Hadoop, HDFS, ZooKeeper, Cassandra, and HBase (Section 4). In total, we have run 56 experiments that benchmark 22 protocols with limpware, for a total of almost 8 hours under normal scenarios and 207 hours under limpware scenarios.
- We present in detail our findings (Section 5). For each system, we present the impacts of limpware on the system and the design deficiencies. Overall, we find 15 protocols that can exhibit limplock.

2 Cases of Limpware

To the best of our knowledge, there is no public large-scale data on limpware occurrences. Nevertheless, we have collected from practitioners many anecdotes of degraded disks and network components, along with the root causes and negative impacts [24]. These stories reaffirm the existence of limpware and the fact that hardware performance failures are not hidden at the device level but are exposed to applications. Below we present again our previous findings to motivate subsequent sections. We focus on I/O-related limpware (disks and network components) as they can slow down by orders of magnitude. For degraded processors, the worst scenario we found is only 26% slowdown [24].

Disks: Due to the complex mechanical nature of disk drives, disk components wear out and exhibit a performance failure. For example, a disk can have a *weak head* which could reduce read/write bandwidth to the affected platter by 80% or introduce more than 1 second latency on every I/O [6]. Mechanical spinning disks are not immune to *vibration* which can originate from bad disk drive packaging, missing screws, constant "nagging noise" of data centers, broken cooling fans, and earthquakes, potentially decreasing disk bandwidth by 10-66% [26, 32]. The disk stack also includes complex controller code that can contain *firmware bugs* that degrade performance over time [41]. Finally, as disks perform automatic *bad sector remapping*, a large number of sector errors will impose more seek cost. We also hear anecdotes from practitioners. For example, media failures can force disks to re-read each block multiple times before responding [9], and a set of disk volumes incurred a wait time as high as 103 seconds, uncorrected for 50 days, affecting the overall I/O performance [11]. We ourselves have experienced an impact of limpware; Emulab encountered an erratic RAID controller on a boss node that crippled the testbed[1] [43].

[1] Needless to say, the failure cascaded to user level; we (*i.e.*, the students) were ineffective that day.

Network: A *broken module/adapter* can increase I/O latency significantly. For example, a bad Fibre Channel passthrough module of a busy VM server can increase user-perceived network latency by ten times [7]. A broken adapter can lose or corrupt packets, forcing the firmware to perform *error correcting* which could slow down all connected machines. As a prime example, Intrepid Blue Gene/P administrators found a bad batch of optical transceivers that experienced a high error rate, collapsing throughput from 7 Gbps to just 2 Kbps; as the failure was not isolated, the affected cluster ceased to work [10]. A similar collapse was experienced at Facebook, but due to a different cause: the engineers found a *network driver bug* in Linux that degraded a NIC performance from 1 Gbps to 1 Kbps [8]. Finally, *power fluctuations* can also degrade switches and routers [25].

3 Limplock

To measure the system-level impacts of limpware, we assembled limpbench (§4). From our findings of running limpbench, we introduce a new of concept of *limplock*, a situation where a system progresses slowly due to limpware and is not capable of failing over to healthy components (*i.e.*, the system enters and cannot exit from limping mode). We observe that limpware does not just affect the operations running on it, but also other healthy nodes, or even worse, a whole cluster. To classify such cascading failures, we introduce three levels of limplock: *operation*, *node*, and *cluster*. Below, we describe each limplock level. In each level, we dissect system designs that allow limplock to occur and escalate, based on our analysis of our target systems. The complete results will be presented in Section 5.

3.1 Operation Limplock

The smallest measure of limplock is operation limplock. Let's consider a 3-node write pipeline where one of the nodes has a degraded NIC. In the absence of limpware detection and failover recovery, the data transfer will slow down and enter limplock. We uncover three system designs that allow operation limplock:

Coarse-grained timeout: Timeout is a form of performance failure detection, but a coarse-grained timeout does not help. For example, in HDFS, we observe that a large chunk of data is transferred in 64-KB packets, and a timeout is only thrown in the absence of a packet response for 60 seconds. This implies that limpware can limp to almost 1 KB/s without triggering a failover (§5.2). This read/write limplock brings negative implications to high-level software such as Hadoop and HBase that run on HDFS (§5.1 and §5.5).

Single point of failure (SPOF): Limpware can be failed over if there is another resource or data source (*i.e.*, the "No SPOF" principle). However, this principle is not always upheld. For example, because of performance reasons, Hadoop intermediate data is not replicated. Here, we find that a mapper with a degraded NIC can make all reducers of the job enter limplock, and surprisingly, the speculative execution does not work in this case (§5.1). Another example is SPOF due to indirection (*e.g.*, HBase on HDFS). To access a data region, a client must go through the one HBase server that manages the region (although the data is replicated in three nodes in HDFS). If this "gateway" server has limpware, then it becomes a performance SPOF (§5.5).

Memoryless/revokeless retry: A timeout is typically followed by a retry, and ideally a retry should not involve the same limpware. Yet, we find cases of prolonged limplock due to "memoryless" retry, a retry that does not use any information from a previously failing operation. We also find cases of revokeless retry. Here, a retry does not revoke previous operations. Under resource exhaustion, the retry cannot proceed.

3.2 Node Limplock

A node limplock is a situation where operations that must be served by this node experience a limplock, *although* the operations do *not* involve limpware. As an illustration, let's consider a node with a degraded disk. In-memory read operations served by this node should not experience a limplock. But, if the node is in limplock, the reads will also be affected. We also emphasize that a node limplock can happen on *other* nodes that do *not* contain limpware (*i.e.*, a cascading effect). For example, let's consider a node A communicating with a node B that has limpware. If all communicating threads in A are in limplock due to B, then A exhibits a node limplock, during which A cannot serve requests from/to other nodes. A node limplock leads to resource underutilization as the node cannot be used for other purposes. We uncover two system designs that can cascade operation limplock to node limplock:

Bounded multi-purpose thread/queue: Developers often use a bounded pool of multi-purpose threads where each thread can be used for different types of operation (*e.g.*, read and write). Here, if limplocked operations occupy all the threads, then the resource is exhausted, and the node enters limplock. For example, in HDFS, limplocked writes to a slow disk can occupy all the request threads at the master node, and thus in-memory reads are affected (§5.2). Similarly, a multi-purpose queue is often used; a node uses one queue to communicate to all other nodes, and hence a slow communication between a pair of nodes can make the single queue full, disabling communication with other nodes.

Unbounded thread/queue: Unbounded solutions can also lead to node limplock due to backlogs. For example, in the ZooKeeper quorum protocol, a slow follower can make the leader's in-memory queue grow as the follower cannot catch up with all the updates. Over time, this backlog will exhaust the leader's memory space and lead to a node limplock (§5.3).

3.3 Cluster Limplock

Cluster limplock is the most severe level of limplock. Here, a single limpware makes the whole cluster performance collapse. This condition is different from node limplock where only one or a subset of all the nodes are affected. Distributed systems are prone to cluster limplock as nodes communicate with each other via which limplock cascades. There are two scenarios that lead to cluster limplock.

All nodes in limplock: If all nodes in the system enter limplock, then the cluster is technically in limplock. This happens in protocols with a small maximum number of resources. For example, in Hadoop, by default a node can have two map and reduce tasks. We find that a wide fan-out (*e.g.*, many reducers reading from a slow mapper) can quickly cause cluster limplock (§5.1).

Master-slave architecture: This architecture is prone to cluster limplock. If a master exhibits a node limplock, then entire operations that are routed to the master will enter limplock. In cloud systems where the slave-to-master ratio is typically large (*e.g.*, HDFS), a master limplock can make many slave nodes underutilized.

4 Limpbench

We now present limpbench, a set of benchmarks that we assembled for two purposes: to quantify limplock in current cloud systems and to unearth system designs leading to limplock. In total, we have run 56 experiments to benchmark 22 protocols with limpware on five scale-out systems (Hadoop/HDFS-1.0.4, ZooKeeper-3.4.5, Cassandra-1.2.3, and HBase-0.94.2). Due to space constraints, we only report a subset of limpbench results in Table 1; excluded experiments lead to the same conclusions. Each row in Table 1 represents an *experiment*. In each experiment, we target a particular protocol, run a microbenchmark, and inject a slow NIC/disk. We run our experiments on the Emulab testbed [1]. Each experiment is repeated 3-5 times. Figure 1 shows the empirical results and will be described in Section 5. Limpbench is available on our group website [5].

4.1 Methodology

Each experiment includes four important components: data-intensive protocols, load stress, fault injections (limpware and crash), and white-box metrics.

• **Data-intensive protocols.** We evaluate data-intensive system protocols such as read, write, rebalancing, but not background protocols like gossipers. In some experiments, we mix protocols that require different resources (*e.g.*, read from cache, write to disk) to analyze cascades of limplock. Our target protocols are listed in the "Protocol" column of Table 1.

• **Load stress.** We construct microbenchmarks that stress request load (listed in the "Workload" column of Table 1). Performance failures often happen under system load. Each benchmark saturates 30-70% of the maximum throughput of the setup.

• **Fault Injection.** First, we perform *limpware injection* on a local network card (NIC) or disk. We focus on I/Os as they can slow down by orders of magnitude. The perfect network and disk throughputs are 100 Mbps and 80 MB/s respectively. In each experiment, we inject *three limpware scenarios* (slow down a NIC/disk by 10x, 100x, and 1000x). We only inject slow disk on experiments that involve synchronous writes to disk. Many protocols of our target systems only write data to buffer cache, and hence the majority of the experiments involve slow NIC. We also perform *node-aware* limpware injection; across different experiments, limpware is injected on different types of node (*e.g.*, master vs. datanode).

Second, we perform *crash injection*, mainly for two purposes: to show the duration of crash failover recovery (*e.g.*, write failover, block regeneration) and to analyze limpware impacts during fail-stop recovery (*e.g.*, limpware impact on a data regeneration process). The first purpose is to compare the speed of fail-stop recovery with limpware recovery (if any).

• **White-box metrics.** We monitor system-specific information such as the number of working threads and lengths of various request queues. We do not treat the systems as black boxes because limpware impact might be "hidden" behind external metrics such as response times. White-box monitoring helps us unearth hidden impacts and build better benchmarks. Monitoring requires modifications to our target systems. However, modifications are only needed for our detailed experiments; limpbench can run on the vanilla versions.

5 Results

We now present the results of running limpbench on Hadoop, HDFS, ZooKeeper, Cassandra, and HBase. The columns OL, NL, and CL in Table 1 label which experiments/protocols exhibit operation, node, and cluster limplock respectively. We will not discuss individual experiments but rather focus our discussion on how and why these protocols exhibit limplock. Figure 1 quanti-

ID	Protocol	Limp-ware	Injected Node	Workload	Base Latency	OL	NL	CL
F1	Logging	Disk	Master	Create 8000 empty files	12	√	√	√
F2	Write	Disk	Data	Create 30 64-MB files	182	.	.	.
F3	Read	Disk	Data	Read 30 64-MB files	120	.	.	.
F4	Metadata Read/Logging	Disk	Master	Stats 1000 files + heavy updates	9	√	√	√
F5	Checkpoint	Disk	Secondary	Checkpoint 60K transactions	39	√	.	.
F6	Write	Net	Data	Create 30 64-MB files	208	√	.	.
F7	Read	Net	Data	Read 30 64-MB files	104	√	.	.
F8	Regeneration	Net	Data	Regenerate 90 blocks	432	√	√	√
F9	Regeneration	Net	Data-S/Data-D	Scale replication factor from 2 to 4	11	√	.	.
F10	Balancing	Net	Data-O/Data-U	Move 3.47 GB of data	4105	√	.	.
F11	Decommission	Net	Data-L/Data-R	Decommission a node having 90 blocks	174	√	√	√
H1	Speculative execution	Net	Mapper	WordCount: 512 MB dataset	80	√	.	.
H2	Speculative execution	Net	Reducer	WordCount: 512 MB dataset	80	.	.	.
H3	Speculative execution	Net	Job Tracker	WordCount: 512 MB dataset	80	.	.	.
H4	Speculative execution	Net	Task Node	1000-task Facebook workload	4320	√	√	√
Z1	Get	Net	Leader	Get 7000 1-KB znodes	4	.	.	.
Z2	Get	Net	Follower	Get 7000 1-KB znodes	5	.	.	.
Z3	Set	Net	Leader	Set 7000 1-KB znodes	23	√	√	√
Z4	Set	Net	Follower	Set 7000 1-KB znodes	26	.	.	.
Z5	Set	Net	Follower	Set 20KB data 6000 times to 100 znodes	64	√	√	√
C1	Put (quorum)	Net	Data	Put 240K KeyValues	66	.	.	.
C2	Get (quorum)	Net	Data	Get 45K KeyValues	73	.	.	.
C3	Get (one) + Put (all)	Net	Data	Get 45K KeyValues + heavy puts	36	.	.	.
B1	Put	Net	Region Server	Put 300K KeyValues	61	√	.	.
B2	Get	Net	Region Server	Get 300K KeyValues	151	√	.	.
B3	Scan	Net	Region Server	Scan 300K KeyValues	17	√	.	.
B4	Cache Get/Put	Net	Data-H	Get 100 KeyValues + heavy puts	4	√	√	.
B5	Compaction	Net	Region Server	Compact 4 100-MB sstables	122	√	√	.

Table 1: Limpbench Experiments. *Each table entry represents an experiment in limpbench. F, H, Z, C, and B in the "ID" column represent HDFS, Hadoop, ZooKeeper, Cassandra, and HBase. The columns describe the experiment ID, the target protocol being tested, the limpware type (disk/network), the node type where the limpware is injected, the workload, and the base latency of the experiment under no limpware. A tick mark in OL, NL, and CL columns implies that the experiment leads to operation, node, and cluster limplock respectively. Data: datanode; Data-H: datanode storing HLog; Data-S, Data-D, Data-O, Data-U, Data-L and Data-R represent source, destination, over-utilized, under-utilized, leaving, and remaining datanodes, respectively. Due to space constraints, the table only reports a subset of limpbench results.*

fies the impact of limplock in each experiment. Uphill bars (*e.g.*, in F1, H1) imply the experiment observes a limplock. Flat bars (*e.g.*, in F2, H2) imply otherwise. Up-and-down bars (*e.g.*, in B1, Z5) imply that when congestion is severe (*e.g.*, 0.1 Mbps NIC), the connection to the affected node is "flapping" (connected and disconnected continuously), and the cluster performs better but not optimally. In the following sections, major findings are written in italic text. Section 5.6 summarizes our high-level findings and lessons learned.

5.1 Hadoop

In Hadoop, there is one job tracker which manages jobs running on slave nodes. Each job is divided into a set of map and reduce tasks. A mapper processes an input chunk from HDFS and outputs a list of key-value pairs. When all mappers have finished, each reducer fetches its portion of the map outputs, runs a reduce function, and writes the output to HDFS files. At the heart of Hadoop is speculative execution, a protocol that monitors task progress, detects stragglers, and runs backup tasks to minimize job execution time.

We evaluate the robustness of the default Hadoop speculation (LATE [52]) by injecting a degraded NIC on three kinds of nodes: job tracker, map and reduce nodes (to simplify diagnosis, we do not collocate mappers and reducers).

> *Definition:* In our discussion below, the term *"slow map/reduce node"* implies a map/reduce node that has a degraded NIC.

5.1.1 Limplock Free

We find that Hadoop speculation works as expected when a reduce node is slow (Figure 1, H2). Here, the affected reduce tasks are re-executed on other nodes. Slow job tracker also does not affect job execution time as it does not perform data-intensive tasks (H3).

5.1.2 Operation Limplock

We find three scenarios where Hadoop speculation is not immune to slow network: when (1) a map node is slow, (2) all reducers experience HDFS write limplock, and (3) both original and backup mappers read from a

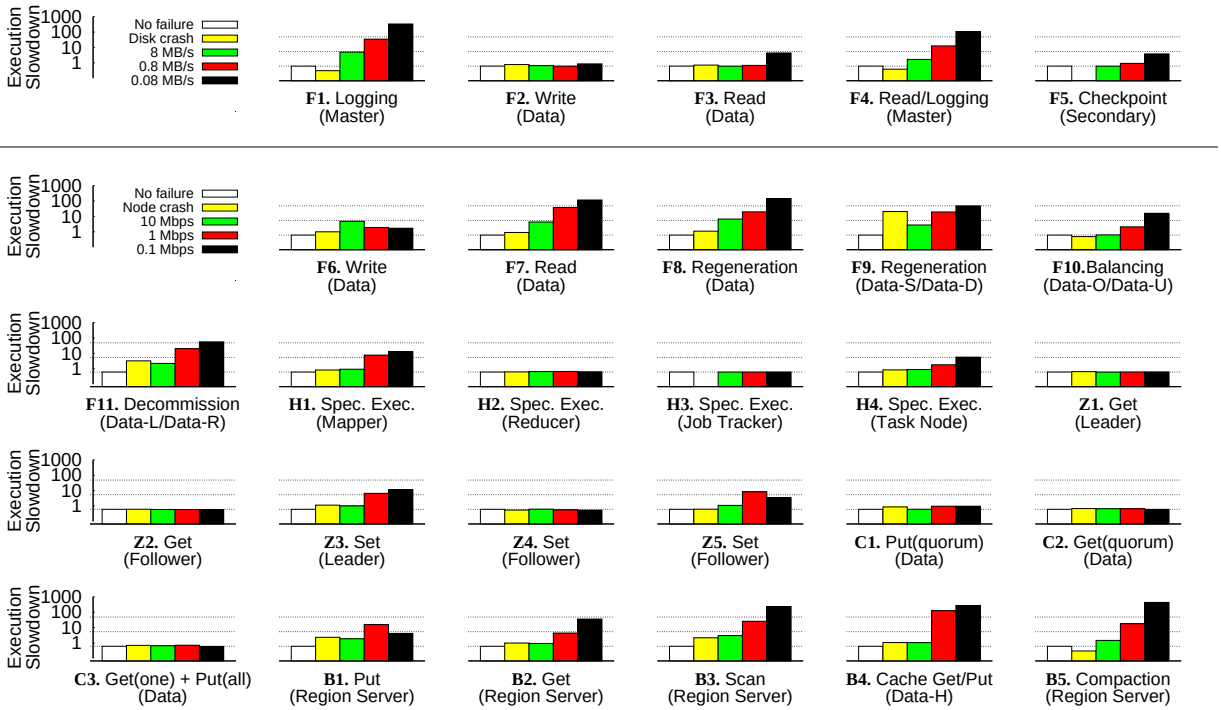

Figure 1: Limpbench Results. *Each graph represents the result of each experiment (e.g., F1) described in Table 1. The y-axis plots the slowdowns (in log scale) of an experiment under various limpware scenarios. In the first row, a slow disk is injected. In the rest, a slow NIC is injected. The graphs show that cloud systems are crash tolerant, but not limpware tolerant.*

remote slow node. These cases lead to job execution slowdowns by orders of magnitude (*e.g.*, H1). The root causes of the problem are imprecise straggler detection and intra-job speculation. We now discuss these three limplock scenarios.

First, *a slow map node can slow down all reducers of the same job, and hence does not trigger speculation*. We find that a mapper can run on a slow node without being marked as a straggler. This is because the input/output of a mapper typically involves only the local disk due to data locality; the slow NIC does not affect the mapper. However, during the reduce phase, the implication is severe: when all reducers of the same job fetch the mapper's output through the slow link, *all* of the reducers progress at the same slow rate. Speculation is not triggered because a reduce task is marked as a straggler only if it makes little progress relative to others of the *same* job.

Although it is the reducers that are affected, re-executing the reducers is not the solution. We constructed a synthetic job where only a subset of the reducers fetch data from a slow map node. The affected reducers are re-executed, however, the backup reducers still read from the same slow source again. The slow map node is a single point of performance failure, and the solution is to rerun the map task elsewhere.

Second, *all reducers of a job can exhibit HDFS write limplock, which does not trigger speculation*; HDFS write limplock is explained in §5.2. The essence is that *all* reducers write their outputs at the same slow rate, similar as the previous scenario. To illustrate these two scenarios, Figure 2a plots the progress scores of three reducers of a job. There are three significant regions: all scores initially progress slowly due to a slow input, then jump quickly in computation mode, and slow down again due to HDFS write limplock. These 40-second tasks finish after 1200 seconds due to limplock.

Finally, *both original and backup mappers can be in limplock when both read from a remote slow node via HDFS*. There are two underlying issues. First, to prevent thrashing, Hadoop limits the number of backup task (default is one); if a backup task exhibits limplock, so is the job. Second, although HDFS employs a 3-way replication, HDFS can pick the same slow node several times. This is a case of *memoryless retry*. That is, Hadoop does not inform HDFS that it wants a different source than the previous slow one.

5.1.3 Node Limplock

A node limplock occurs when its task slots are all occupied by limplocked tasks. A Hadoop node has a limited number of map and reduce slots. If all slots are occupied by limplocked map and reduce tasks, then the node will

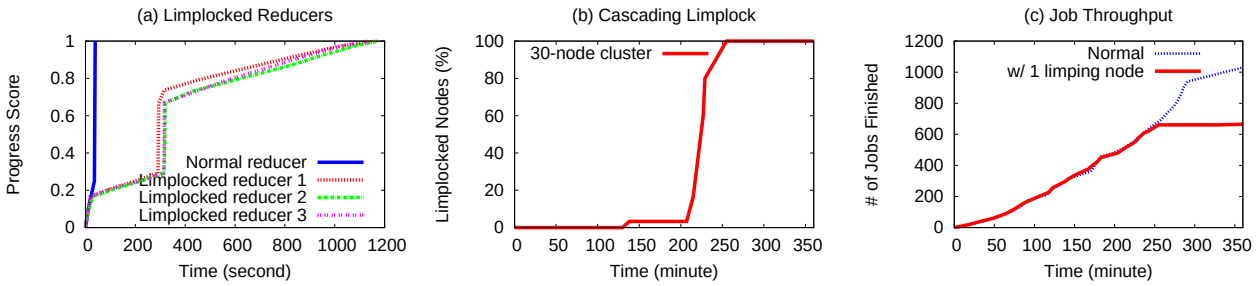

Figure 2: Hadoop Limplock. *The graphs show (a) the progress scores of limplocked reducers of a job in experiment H1 (a normal reducer is shown for comparison), (b) cascades of node limplock due to single limpware, and (c) a throughput collapse of a Hadoop cluster due to limpware. For Figures (b) and (c), we ran a Facebook workload [3] on a 30-node cluster.*

be in limplock. The node is underutilized as it cannot run other healthy jobs unaffected by the limpware.

5.1.4 Cluster Limplock

A single limpware can cripple an entire Hadoop cluster. This happens when limpware causes limplocked map/reduce tasks, which then lead to limplocked nodes and eventually a limplocked cluster when all nodes are in limplock. To illustrate this in real settings, we ran a Facebook workload [3, 17] on a 30-node cluster with a node that has a degraded NIC. Figure 2b shows how limpware can cause many nodes to enter limplock over time and eventually a cluster limplock. Figure 2c shows how the cluster is underutilized. In a normal scenario, the 30-node cluster finishes around 172 jobs/hour. However, under cluster limplock, the throughput collapses to almost 1 job/hour at t = 250 minutes.

5.2 HDFS

HDFS employs a dedicated master and multiple datanodes. The master serves metadata reads and writes with a fixed-size thread pool. All metadata is kept in memory for fast reads. A logging protocol writes metadata updates to an on-disk log that is replicated on three storage volumes. Datanodes serve data read and write requests. A data file is stored in 64-MB blocks. A new data block is written through a pipeline of three nodes by default. Dead datanodes will lead to under-replicated blocks, which then will trigger a block regeneration process. To reduce noise to foreground tasks, each datanode can only run two regeneration threads at a time with a throttled bandwidth. Block regeneration is also triggered when a datanode is decommissioned or users increase file replication factor on the fly. HDFS also provides a rebalancing process for balancing disk usage across datanodes.

We evaluate the robustness of HDFS protocols by injecting a degraded disk or NIC. To evaluate master protocols, we inject a degraded disk out of three available disks, but not NIC because there is only a single master.

5.2.1 Limplock Free

Datanode-related protocols are in general immune to slow disk. This is because data writes only flush updates to the OS buffer cache; on-disk flush happens in the background every 30 seconds. In our experiments, with 512 MB RAM, the write rate must be above 17 MB/s to reveal any impact. The network however is limited to only 12.5 MB/s.

5.2.2 Operation Limplock

HDFS is built for fail-stop tolerance but not limpware tolerance; we find numerous protocols that can exhibit limplock due to a degraded disk or NIC, as shown in Table 1. Below we frame our findings in terms of HDFS design deficiencies that lead to limplock-prone protocols: coarse-grained timeouts, memoryless and revokeless retries, and timeout-less protocols.

First, *HDFS data read and write protocols employ a coarse-grained timeout such that a NIC that limps above 1 KB/s will not trigger a failover* (Figure 1, F7). Specifically, these protocols transfer a large data block in 64-KB packets, and a timeout is only thrown in the absence of a packet response for 60 seconds. HDFS might expect that upper-level software employs a more fine-grained domain-specific timeout, however, such is not the case in Hadoop (§5.1) and HBase (§5.5). Timeouts based on relative performance [14] might be more appropriate than constant long timeouts.

Second, even in the presence of timeouts, *multiple retries can exhibit limplock due to memoryless retry*, similar to the Hadoop case. Here, limpware is involved again in recovery. Consider a file scale-up experiment from 2 to 4 replicas, and one of the source nodes is slow. Even with random selection, HDFS can choose the same slow source multiple times (F9).

Finally and surprisingly, *some protocols are timeout-less*. For example, the log protocol at the master writes to three storage volumes serially without any timeout. One degraded disk will slow down all log updates (F1). We believe this is because applications expect the OS to

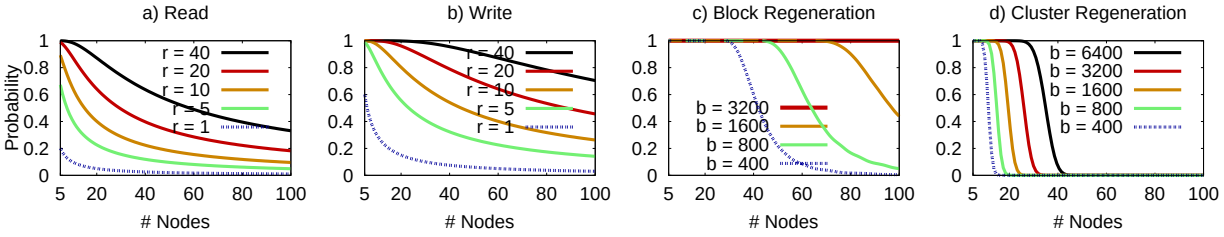

Figure 3: HDFS Limplock Probabilities. *The figures plot the probabilities of (a) read limplock/P_{rl}, (b) write limplock/P_{wl}, (c) block limplock/P_{bl}, (d) and cluster regeneration limplock/P_{cl}, as defined in Table 2. The x-axis plots cluster size.*

Symbol	Definition/Derivation
n	# nodes
b	# blocks per node / to regenerate
r	# user requests
P_{rl}	$1-(\frac{n-1}{n})^r$
P_{wl}	$1-(\frac{n-3}{n})^r$
P_{nl}	$1-(\frac{n-3}{n-2})^{\frac{b}{n-1}} - \frac{b}{(n-1)\times(n-2)} \times (\frac{n-3}{n-2})^{\frac{b-n+1}{n-1}}$
P_{cl}	${P_{nl}}^{n-2}$
$p_{nl}(i)$	$\binom{n-2}{i} \times {P_{nl}}^i \times (1-P_{nl})^{n-2-i}$
p_{bl}	$\sum_{i=2}^{n-2} p_{nl}(i) \times \frac{\binom{i}{2}}{\binom{n-1}{2}} + \sum_{i=1}^{n-2} p_{nl}(i) \times \frac{i}{\binom{n-1}{2}}$
P_{bl}	$1-(1-p_{bl})^b$

Table 2: HDFS Limplock Frequency. *The table shows the probabilities of a user experiences at least one read (P_{rl}) and write limplock (P_{wl}), a node is in regeneration limplock (P_{nl}), a cluster is in regeneration limplock (P_{cl}), exactly i nodes are in regeneration limplock ($p_{nl}(i)$), a block is in regeneration limplock (p_{bl}), and at least one block is in regeneration limplock (P_{bl}). Details can be found in [23].*

return some error code if hardware fails, but such is not the case for limpware.

To show how often operation limplock happens in HDFS, we model HDFS basic protocols (read, write, and regeneration), derive their limplock probabilities as shown in Table 2, and use simulations to confirm our results. The details are beyond the scope of this paper, but can be found in our technical report [23]. Figure 3a and 3b plot the probabilities of a user to experience at least one read and write limplock respectively as a function of cluster size and request count. The probabilities are relatively high for a small to medium cluster (*e.g.*, 30-node). This significantly affects HDFS users (*e.g.*, Hadoop operation limplock described in §5.1).

5.2.3 Node Limplock

We find two protocols that can lead to node limplock: regeneration and logging. The root cause is resource exhaustion by limplocked operations.

HDFS can exhibit a regeneration node limplock where all the node's regeneration threads are in limplock. Regeneration is run by the master for under-replicated blocks. For each block, the master chooses a source that has a surviving replica and a destination node. A source node can only run two regeneration threads at a time. Thus, a regeneration node limplock occurs if the source node has a slow NIC or when the master picks a slow destination for both threads. Here, the node's regeneration resources are all exhausted.

A regeneration node limplock cannot be unwound due to revokeless recovery. Interestingly, the master employs a timeout to "recover" a slow/failed regeneration process, however, it is revokeless; the recovery does not revoke the limplocked regeneration threads on the affected datanodes (it only implicitly revokes if the source/destination crashes). Therefore, as the master attempts to retry, the resources are still exhausted, and the retry fails silently.

A regeneration node limplock prolongs MTTR and potentially decreases MTTDL. Nodes that exhibit regeneration limplock can be harmful because the nodes cannot be used as sources for regenerating other under-replicated blocks. This essentially prolongs the data recovery time (MTTR). In one experiment, a stable state (zero under-replicated block) is reached after 37 hours, 309x slower than in a normal case (Figure 1, F8). If more nodes die during this long recovery, some blocks can be completely lost, essentially shortening the mean time to data loss (MTTDL). To generalize this problem, we introduce a new term, *block limplock*, which is a scenario where at least an under-replicated block B cannot be regenerated (possibly for a long time) because the source nodes are in limplock. We derive the probability of at least one block limplock (P_{bl} in Table 2) as a function of cluster size and number of blocks stored in every datanode (which also represents the number of under-replicated blocks). Figure 3c plots this probability. The number is alarmingly high; even in a 100-node cluster, a dead 200-GB node (3200 under-replicated blocks) will lead to at least one block limplock.

Other than regeneration, we do not find any datanode protocols that cause node limplock, mainly because a datanode does not have a bounded thread pool for other operations (*e.g.*, it creates a new thread with small mem-

ory footprint for each data read/write). We however find a node limplock case in the master logging protocol. In master-slave architecture, master limplock essentially leads to cluster limplock, which we describe next.

5.2.4 Cluster Limplock

The two HDFS protocols that can exhibit node limplock, regeneration and logging, eventually lead to cluster limplock.

First, *An HDFS cluster can experience a total regeneration limplock where all regeneration threads are in limplock.* As defined before, if all nodes are in limplock then the cluster is in limplock. In terms of regeneration, all regeneration threads progress slowly and affect the MTTR and MTTDL. Figure 3d plots the probability of cluster regeneration limplock (P_{cl} in Table 2). A small cluster (< 30 nodes) is prone to regeneration cluster limplock, as also confirmed in our simulation [23].

Second, *HDFS master is in limplock when all handlers are exhausted by limplocked log writes.* As discussed earlier, a slow disk at the master leads to limplocked log updates (F1). This leads to resource exhaustion under a high load of updates. This is because the master employs a fixed-size pool of multi-purpose threads for handling metadata read/write requests (default is 10). Therefore, as limplocked log writes occupy all the threads, incoming metadata read requests which only need to read in-memory metadata (do not involve the limpware) are blocked in a waiting queue. In one experiment, in-memory read throughput collapses by 233x (F4). Since the master is in node limplock, all operations that require metadata reads/writes essentially experience a cluster limplock.

5.3 ZooKeeper

ZooKeeper has a single leader and multiple follower nodes, and uses znodes as data abstraction. ZooKeeper basic APIs include create, set, get, delete, and sync. Znode get protocol is served by any node, but updates must be forwarded to the leader who executes a quorum-based atomic broadcast protocol to all the followers. If quorum is reached, ZooKeeper returns success.

In our evaluation, we inject a degraded NIC on two types of nodes: leader and follower. The client always connects to a healthy follower.

5.3.1 Limplock Free

As the client connects to a healthy node, get protocol is limplock free because the slow leader/follower is not involved (Figure 1, Z1, Z2). Similarly, based on one experiment (Z4), the quorum-based broadcast protocol is also limplock free. Here, a slow follower who has not yet committed updates does not affect the response time as the majority of the nodes are healthy.

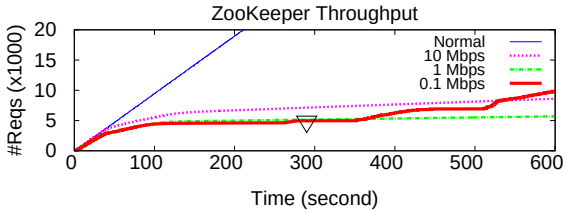

Figure 4: ZooKeeper Cluster Limplock. *The figure plots the number of requests served over time under different limpware scenarios.*

5.3.2 Operation Limplock

If the leader is slow, all updates are in limplock (Z3). Leader-follower architecture must ensure that the leader is the most robust node. We notice that the slow leader can be congested and its IPC timeout disconnects all connections, which then triggers a leader election process. However, as data connections were cut, congestion diminishes, and the slow leader can join the election. In fact, this previous slow leader is likely to be elected again as the election favors a node that has the latest epoch time and transaction ID; the only way a previous leader loses is if it is unavailable during the election.

5.3.3 Cluster Limplock

We find two scenarios that lead to cluster limplock. First, *a slow leader causes a cluster limplock with respect to update operations.* As described above, this is because all update operations involve the leader.

Second, *the presence of a slow follower in a quorum-based protocol can create a backlog at the leader which can cause a cluster limplock.* This is an interesting "hidden" backlog scenario. In our discussion above, in the presence of a slow follower, the quorum-based protocol "looks" limplock free (Z4). However, our white-box metrics hint an upcoming problem. Specifically, we monitor each queue that the leader maintains for each follower for forwarding updates, and we observe that the request queue for the slow follower keeps growing (a backlogged queue). In a short-running experiment (*e.g.*, 30 seconds in Z4), response time is not affected. However, in a larger and longer experiment, we start noticing cluster degradation (Z5).

A slow follower can cripple an entire ZooKeeper cluster. To illustrate this issue further, we plot the number of updates served over time in Figure 4. In a normal scenario, the throughput is constant at 90 requests/sec. With a slow follower (0.1 Mbps NIC), after 100 seconds, the cluster throughput collapses to 3 requests/sec. The root cause is resource exhaustion; the backlogged queue starts to exhaust the heap, and thus Java garbage collection (GC) works hard all the time to find free space. What happens next depends on the request and degradation rates. The leader can be in limplock for a long time

(dashed lines for 10 and 1 Mbps), or it can crash as it runs out of memory after a certain time ($\bigtriangledown$ on bold line). Even after a new leader is elected, the cluster throughput is never back to normal, only 13 requests/sec (bold line after $\bigtriangledown$). This is because the slow follower is still part of the ensemble, which means the new leader must send a big snapshot of backlog to the slow follower that can fail in the middle due to congestion and repeat continuously.

5.4 Cassandra

Cassandra is a distributed key-value store that partitions data across a ring of nodes using consistent hashing. Key gets/puts can be performed with consistency level one, quorum, and all. The common replication factor for a key is three. Given a key operation, a client directly connects to one of the three replica nodes (*i.e.*, the coordinator). Depending on the consistency level, a coordinator node may forward reads/writes to other replica nodes. Each Cassandra node monitors all the nodes in the cluster with dead/up labels. If a replica node is dead, a coordinator stores the updates to its local disk as "hints", which will be forwarded later when the node comes back up (*i.e.*, eventual consistency).

In our experiments, the client connects to a healthy coordinator, which then forwards requests to other replica nodes where one of them has a degraded NIC. At this point, we only analyze get and put protocols. Based on our initial results, Cassandra's architecture is in general limplock free, and only exhibits 2x slowdown. We are still in the process of crafting more benchmarks to unearth any possible limplock cases.

5.4.1 Limplock Free

For weak consistency operations ("quorum" and "one"), Cassandra's architecture is limplock free (Figure 1, C1). However, we observe that they are not completely unaffected; when a replica node limps at 1 and 0.1 Mbps, the client response time increases by almost 2x. We suspect some backlog/memory exhaustion similar to the ZooKeeper case, but our white-box monitoring finds none. This is because a coordinator writes outstanding requests as hints and discards them after no response for 10 seconds. We believe Cassandra employs this backlog prevention due to an incident in the past where a backlogged queue led to overflows in other nodes' queues, crippling nodes communication [4].

After further diagnosis, we find that the 2x slowdown is due to "flapping", a condition where peers see the slow node dead and up continuously as the node's gossip messages are buried in congestion. Due to flapping, the coordinator's write stage continuously stores and forwards hints. This flapping-induced background work leads to extra work by Java GC, which is the cause of 2x slowdown.

5.4.2 Operation Limplock

Gets and puts with full consistency are affected by a slow replica. This is expected as a direct implication of full consistency. However, in Cassandra, limplocked operations do not affect limplock-free operations, and thus Cassandra does not exhibit node limplock such as in Hadoop (§5.1) and HDFS (§5.2). This robustness comes from the staged event-driven architecture (SEDA) [49] (*i.e.*, there is no resource exhaustion due to multi-purpose threads). Specifically, Cassandra decouples read and write stages. Therefore, limplocked writes only exhaust the thread pool in the write stage, and limplock-free reads are not affected (C3), and vice versa. SEDA architecture proves to be robust in this particular case.

5.5 HBase

HBase is a distributed key-value store with a different architecture than Cassandra. While Cassandra directly manages data replication, HBase leverages HDFS for managing replicas (another level of indirection). HBase manages tables, partitioned into row ranges. Each row range is called a *region*, which is the unit for distribution and load balancing. HBase has two types of nodes: *region servers*, each serves one or more regions, and *master servers*, which assign regions to region servers. This mapping is stored in two special catalog tables, ROOT and META.

We evaluate HBase by injecting a degraded NIC on a region server. In addition, as HBase relies on HDFS, we also reproduce HDFS read/write limplock (§5.2) and analyze its impact on HBase.

5.5.1 Operation Limplock

HDFS read/write limplock directly affects HBase protocols. All HBase protocols that perform HDFS writes (such as commit-log updates, table compaction, table splitting) are directly affected (*e.g.*, Figure 1, B4). The impact of HDFS read limplock is only observed if the data is not in HBase caches.

5.5.2 Node Limplock

An HBase region server can exhibit node limplock due to resource exhaustion by limplocked HDFS writes. The issues of fixed resource pool and multi-purpose threads also occur in HBase. In particular, a region server exhibits a node limplock when its threads are all occupied by limplocked HDFS writes. As a result, incoming reads that could be served from in-memory are affected, 620x slower in one experiment (B4).

A slow region server is a performance SPOF. Indirection (*e.g.*, HBase on HDFS) simplifies system management, but could lead to a side effect, a performance

SPOF. That is, if a region server has a slow NIC, then all accesses to the regions that it manages will be in limplock (B1-3). Although a region is replicated three times in HDFS, access to any of the replicas must go through the region server. In contrast, Cassandra (without indirection) allows clients to connect directly to any replica. Regions will be migrated only if the managing server is dead, but not if it is slow.

A slow region server can lead to compaction backlog that requires manual handling. A periodic compaction job reads "sstables" of a table from HDFS, merges them, and writes a new sstable to HDFS (B5). A slow region server is unable to compact many sstables on time (a backlog). Even if the degraded NIC is replaced, the region server must perform major compactions of many sstables, which might not fit in memory (OOM), leading to a server outage. Compaction OOM is often reported and requires manual handling by administrators [2].

5.5.3 Cluster Limplock

A slow region server that manages catalog tables can introduce a cluster limplock. Although HBase is a decentralized system, a slow region server that manages catalog regions (*e.g.*, ROOT and META tables) can introduce a cluster limplock, which will impact new requests that have not cached catalog metadata.

5.6 Summary of Results

Impacts of limpware cascade. Operation limplock can spread to node and eventually cluster limplock. Almost all cloud systems we analyze are susceptible to limplock. Below we summarize our high-level findings and the lessons learned.

• **Hadoop:** Speculative execution, the heart of Hadoop's tail-tolerant strategy, has three loopholes that can lead to map/reduce operation limplock (§5.1). This combined with bounded map/reduce slots cause resource exhaustion that leads to node and cluster limplock where job throughput collapses by orders of magnitude. We find three design deficiencies in Hadoop. First, intra-job speculation has a flaw; if all tasks are slow due to limpware, then there is "no" straggler. Second, there is an imprecise accounting; a slow map node affects reducers' progress scores. Finally, a backup task does not always "cut the tail" as it can involve the same limpware (*e.g.*, due to memoryless retry).

• **HDFS:** Many HDFS protocols can exhibit limplock. Read/write limplock affects upper layers such as Hadoop and HBase. Block regeneration limplock could heavily degrade MTTR and MTTDL as recovery slows down by orders of magnitude. Master-slave architecture is highly prone to cluster limplock if the master exhibits node limplock. We conclude several deficiencies in HDFS system designs: coarse-grained timeouts, multi-purpose threads (lead to resource exhaustion), memoryless and revokeless retries, and timeoutless protocols. All of these must be fixed as upper layers expect performance reliability from HDFS.

• **ZooKeeper:** Our surprising finding here is that a quorum-based protocol is not always immune to performance failure. A slow follower can create a backlog of updates at the leader, which can trigger heavy GC process and eventually OOM. In this leader-follower architecture, a leader limplock becomes a cluster limplock.

• **Cassandra:** Limping failure does not heavily affect Cassandra. Weak consistency operations (*e.g.*, quorum) are not heavily affected because of the relaxed eventual consistency; long outstanding requests are converted as local hints. However, Cassandra is not completely immune to limpware; a slow node can lead to flapping which can introduce 2x slowdown. We also find that the SEDA architecture [49] in Cassandra can prevent node limplock as stages are isolated. We are still in the process of crafting more benchmarks to unearth any possible limplock cases.

• **HBase:** Operation, node, and cluster limplocks occur in HBase similar to other systems. A new finding here is an impact of indirection (HBase on HDFS). If a region server is in limplock, then all accesses to the regions that it manages will be in limplock. Indirection simplifies system management, but could lead to a performance SPOF.

6 Discussion

It is evident that the results we presented demand an era of limpware-tolerant cloud systems. Building such systems is our future agenda. In this section, we discuss several strategies. We categorize the discussion into three principles: limplock avoidance, detection and recovery. We do not claim that the solutions are final, in fact, we focus on open challenges and opportunities of implementing the solutions.

• **Limplock avoidance:** As today's cloud systems are built with fail-stop tolerance, one solution is to *convert limpware to a fail-stop failure*. However, a single machine can have multiple resources (multiple disks, NICs, and cores). Thus, crashing might not be a good option. Another reason is that hardware is managed by device drivers which can be buggy. Rebooting or fixing the buggy driver is a more appropriate solution in this case.

Another related solution is to automatically *quarantine* limpware to prevent cascading failures. The system then checks if the limpware is transient or permanent. A transient limpware can join the system again once it

is healthy. Quarantine however must be performed cautiously; an earthquake or a long power glitch can make many hardware pieces limp. Quarantining a large portion of a cluster is not desirable.

Limplock avoidance can also be achieved with *limplock-free design patterns*; every data structure and algorithm should take into account cascades of limplock. For example, developers can enforce the use of differentiated queues/threads where every queue/thread handles a different type of operation to prevent cascades of limplock (*e.g.*, similar to SEDA [49]). Developers can also explore destination-proportional queues to ensure that messages directed to a slow destination do not lead to resource exhaustion. We believe many more limplock-free design patterns can be explored.

"Scale can be your friend" [36]. This is also applicable here. Limplock probability is likely to decrease as scale increases. Unfortunately, this is not true if limplock cascades (*e.g.*, a Hadoop cluster limplock).

• **Limplock detection.** Detection must be accurate and efficient. Accurate detection is fundamental for proper recovery. Imagine a limpware-induced slowdown incorrectly detected as overload-induced. Here, a recovery might react (incorrectly) by throttling or reducing the workload as opposed to isolating the limpware. Thus, an end-to-end detection is needed. Unfortunately, high-level performance management does not always incorporate individual hardware performance. Hadoop's flaws is an example (§5.1). To reduce monitoring overhead, there is a need to explore methods that unearth implicit events from explicit events; fortunately implicit limpware behaviors can be attributed to certain explicit causes (*e.g.*, #remappings, #error corrections). Rather than having a full-blown monitoring, only explicit signals can be monitored. Beyond these strategies, limplock detection can leverage a rich body of literature in peer comparison [38], sampling [35], and root-cause analysis [15, 33, 44].

• **Limplock recovery.** A good recovery is one that allows limpware to "fail in place" (*i.e.*, still slow but not affecting other components). Within this principle, there are many strategies to consider: How to utilize nodes with degraded disks for in-memory computation only? How to distribute computations across racks connected by a degraded switch? How to differentiate recovery of transient vs. permanent limpware? Also, unlike the fail-stop principle where only two failure modes exist (fail or working), limpware introduces more complex failure modes; a hardware can slow down by just 1% or worse 50%. Different slowdowns might be handled differently; limpware might still be usable in different ways depending on the domain.

In summary, we believe the design space of limpware-tolerant cloud computing is vast. We will explore this in our future work. We also hope this paper provides a strong motivation for the cloud community to explore this space further.

7 Related Work

In our previous work [24], we advocate the concept of limpware-tolerant clouds based on our early findings; we set up three simple micro-benchmarks with limpware injections and found that limpware can make software systems limping; there is little limpware detection and recovery in these systems. In this paper, we present a more complete limpbench that covers more systems and delivers more results (56 experiments that cover 22 protocols). From in-depth findings, we formalize the problem with the concept of limplock and its taxonomy.

Recent work provides rich analysis of various hardware failures including machine failures, disk failures, memory corruption, and network failures [16, 27, 28, 39, 40, 45]. "Formal" studies of these failures were undertaken after anecdotes started to circulate. We argue that studies of limpware are needed.

Distributed jobs have to deal with performance variability originating from jitters and stragglers. Jitters are often transient and sporadic in nature [54]. Limpware on the other hand can be both transient and permanent and exhibit as much as 1000x slowdown, and hence should be treated differently. Stragglers are mostly detected at the task level and mitigated by speculative execution [22, 52], which can suffer from several pitfalls (§5.1). Another tail-tolerant approach is cloning requests [12, 21]. If designed carelessly, cloned requests might involve the same limpware, exhibiting the same pitfalls as in Hadoop. Cloning is also limited to small jobs with little resource consumption.

Many solutions have been proposed to enforce performance isolation and fairness at various levels (*e.g.*, disk [46], CPU [53], VM [29], and cloud tenants [42]) and to manage performance variability using runtime adaptation techniques [14, 30, 47]. In this paper, we argue an end-to-end approach is needed; high-level performance management policies must incorporate individual low-level hardware performance (§6).

Big data should flow in big pipes, but design flaws could introduce bottlenecks which lead to "small pipes" [31, 50], or in our case, limplock. To unearth design flaws, pinpoint implementation bugs, or diagnose misconfiguration, many approaches analyze system-specific information such as request flows [38], systems logs [51], and configuration snapshots [48]. We

similarly use white-box metrics for manual diagnosis of limpware-intolerant designs. Other work leverages black-box metrics for statistical performance diagnosis (*e.g.*, CPU usage) [18, 19, 34]. Here, code debugging is a non goal, and deep design flaws are hard to find.

Current cloud benchmarks (*e.g.*, YCSB [20], YCSB++ [37]) typically evaluate performance trade-offs among various cloud systems. Our work is complementary; limpbench evaluates the performance of cloud systems under limpware scenarios.

8 Conclusion

Limpware is a reality and a destructive failure mode. Yet, cloud systems are not immune to limpware. This leads to limplock at many levels (operation, node, and cluster). Decades of research portray how new failure modes always dramatically transform systems design and implementation. Likewise, we hope this paper provides a strong motivation and foundation that commence the transformation of today's cloud systems into limpware-tolerant systems.

9 Acknowledgments

We thank George Porter, our shepherd, and the anonymous reviewers for their tremendous feedback and comments. This material is based upon work supported by the NSF (grant Nos. CCF-1321958, CCF-1336580, and CCF-1017073). The experiments in this paper were performed in the Utah Emulab network testbed [1].

References

[1] Emulab Testbed. http://www.emulab.net.

[2] HBase User Mailing List. http://tinyurl.com/kwdsrwx.

[3] SWIM: Statistical Workload Injector for MapReduce. https://github.com/SWIMProjectUCB/SWIM/wiki.

[4] Tcp Manager only ever has one connection. http://tinyurl.com/n2vy5qw.

[5] UCARE LigHTS project: Limpware Tolerant Systems. http://ucare.cs.uchicago.edu/projects/lights.

[6] Weak Head. http://tinyurl.com/m26gx37.

[7] Message 1800544: High Latency. http://tinyurl.com/bdwkqsp, 2011.

[8] Personal Comm. from Dhruba Borthakur of Facebook, 2011.

[9] Personal Comm. from Andrew Baptist of Cleversafe, 2013.

[10] Personal Comm. from Kevin Harms of ANL, 2013.

[11] Personal Comm. from Mike Kasick of CMU, 2013.

[12] 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.

[13] 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.

[14] Remzi H. Arpaci-Dusseau. Run-Time Adaptation in River. *ACM Transactions on Computer Systems (TOCS)*, 21(1):36–86, February 2003.

[15] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[16] 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.

[17] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy H. Katz. The Case for Evaluating MapReduce Performance Using Workload Suites. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2011.

[18] Ludmila Cherkasova, Kivanc Ozonat, Ningfang Mi, Julie Symons, and Evgenia Smirni. Anomaly? Application Change? Or Workload Change? Towards Automated Detection of Application Performance Anomaly and Change. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2008.

[19] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeffrey S. Chase. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[20] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[21] Jeffrey Dean and Luiz Andre Barroso. Tail at Scale. *Communications of the ACM*, 2013.

[22] 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.

[23] Thanh Do and Haryadi S. Gunawi. Impact of Limpware on HDFS: A Probabilistic Estimation. Technical Report TR-2013-08, Department of Computer Science, University of Chicago, 2013.

[24] Thanh Do and Haryadi S. Gunawi. The Case for Limping-Hardware Tolerant Clouds. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2013.

[25] Erik Eckel. 10 tips for troubleshooting slowdowns in small business networks. http://www.techrepublic.com, 2007.

[26] Michael Feldman. Startup Takes Aim at Performance-Killing Vibration in Datacenter. http://www.hpcwire.com, 2010.

[27] Daniel Ford, Franis Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlna. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[28] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of SIGCOMM*, 2011.

[29] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST)*, 2009.

[30] Ajay Gulati, Ganesha Shanmuganathan, Irfan Ahmad, Carl A. Waldspurger, and Mustafa Uysal. Pesto: Online Storage Performance Management in Virtualized Datacenters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)*, 2011.

[31] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[32] Robin Harris. Bad, bad, bad vibrations. http://tinyurl.com/2c7ea6t, 2010.

[33] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and Detecting Real-World Performance Bugs. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[34] Michael P. Kasick, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Black-Box Problem Diagnosis in Parallel File Systems. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST)*, 2010.

[35] Emre Kiciman and Benjamin Livshits. Ajaxscope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[36] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[37] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio Lopez, Garth Gibson, Adam Fuchs, and Billie Rinaldi. YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)*, 2011.

[38] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing Performance Changes by Comparing Request Flows. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[39] Bianca Schroeder and Garth Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST)*, 2007.

[40] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A Large-Scale Field Study. In *Proceedings of the 2009 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2009.

[41] Anand Lal Shimpi. Intel Discovers Bug in 6-Series Chipset: Our Analysis. http://tinyurl.com/45twb2l, 2011.

[42] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[43] Utah Emulab Testbed. Disk controller problems on boss. http://www.emulab.net/news.php3, 2013.

[44] Avishay Traeger, Ivan Deras, and Erez Zadok. DARC: dynamic analysis of root causes of latency distributions. In *Proceedings of the 2008 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2008.

[45] Kashi Vishwanath and Nachi Nagappan. Characterizing Cloud Computing Hardware Reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[46] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: performance insulation for shared storage servers. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST)*, 2007.

[47] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: Enabling High-level SLOs on Shared Storage Systems. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*, 2012.

[48] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi min Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[49] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[50] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. ICTCP: Incast Congestion Control for TCP. In *The 6th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2010.

[51] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Detecting Large-Scale System Problems by Mining Console Logs. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[52] 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.

[53] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI2: CPU Performance Isolation for Shared Compute Clusters. In *Proceedings of the 2013 EuroSys Conference (EuroSys)*, 2013.

[54] Tao Zou, Guozhang Wang, Marcos Vaz Salles, David Bindel, Alan Demers, Johannes Gehrke, and Walker White. Making Time-stepped Applications Tick in the Cloud. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)*, 2011.