The Case for Limping-Hardware Tolerant Clouds

Thanh Do
University of Wisconsin–Madison

Haryadi S. Gunawi
University of Chicago

Abstract

With the advent of cloud computing, thousands of machines are connected and managed collectively. This era is confronted with a new challenge: performance variability, primarily caused by large-scale management issues such as hardware failures, software bugs, and configuration mistakes. In this paper, we highlight one overlooked cause: limping hardware – hardware whose performance degrades significantly compared to its specification. We present numerous cases of limping disks, network and processors seen in production, along with the negative impacts of such failures on existing large-scale distributed systems. From these findings, we advocate the concept of limping-hardware tolerant clouds.

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, distributed systems must deal with all kinds of failures, including hardware failures, software bugs, administrator mistakes, and many others. These failures lead to *performance variability*, which is considered a big "nuisance" in large-scale system management. Recent work has addressed many sources of performance variability such as heterogeneous systems [26, 41], unbalanced resource allocation [25, 36, 39], software bugs [29], configuration mistakes [20] and straggling tasks [18, 22].

In this paper, we highlight one overlooked cause of performance variability: *limping hardware* – hardware 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 limping hardware. We believe this trend will continue, and the concept of performance perfect hardware no longer holds. Therefore, we advocate that limping hardware should be considered as a new and important failure mode that future distributed systems should deal with. To corroborate this conjecture, we raise the following questions: What are the cases of limping hardware observed in production? What are the impacts on deployed systems? What are the design flaws? How should future generation systems manage limping hardware?

To address these questions, we first collected reports and anecdotes on cases of limping hardware, along with the root causes and negative impacts on applications and users (Section 2). We find that disk bandwidth can drop by 80%, network throughput by two orders of magnitude, and processor speed by 25%. Interestingly, such limping behavior is exhibited by both commodity as well as enterprise hardware. The anecdotes also indicate that unmanaged limping hardware can lead to cascades of performance failures across system components. For example, a limping network card can cause a chain reaction upstream and "cripple" the whole cluster [14, 16].

To further substantiate our conjecture, we performed a study of issues reported by developers of open-source scale-out systems such as Hadoop, HDFS, HBase, Cassandra, and ZooKeeper (Section 3). We manually reviewed 573 issues that pertain to failures in general, 53 out of which describe the systems' inability to deal with performance failures caused by increased load and degraded hardware. From this study, we are able to highlight some design flaws that underlie why these systems are limping-hardware intolerant. For example, a single thread/queue is often used to perform multiple operations of different types, and if one operation/queue element is affected by a limping hardware (*e.g.*, slow access to a limping disk), all other operations will be affected (*e.g.*, blocked heartbeats).

We hypothesized that many subcomponents of these systems are perhaps still limping-hardware intolerant. To confirm this, we built LATE (Limping-failure Analysis and TEsting framework) to rigorously analyze several scale-out systems against limping I/Os (Section 4). As expected, we unearthed more issues, among which is the grave absence of limping-hardware detection and failover recovery. For example, in HDFS, a data transfer can limp to 1 KB/s without triggering a failover. We also found that the Hadoop speculative execution does not work in dealing with limping hardware.

Overall, we conclude that although today's scale-out systems employ redundancies, they are not capable of making limping hardware "fail in place". Users, administrators, and developers often use laborious time-consuming ad-hoc methods to pinpoint and replace limping hardware. As a result, performance failures cascade, productivity is reduced, and energy is wasted. This leads us to introduce the concept of limping-hardware tolerant clouds in which scale-out systems can properly anticipate, detect, recover, and utilize various cases of limping hardware, isolating the negative implications from user applications (Section 5).

We note that the purpose of this paper is not to propose a specific solution, but simply to raise awareness of limping hardware, show the negative impacts on existing scale-out systems, and discuss open challenges and new research directions in building limping-hardware tolerant clouds. The following sections describe in detail the contributions of this paper that we have summarized above.

2 Cases of Limping Hardware

To the best of our knowledge, there is no public large-scale data on limping hardware. Nevertheless, we have collected many reports and anecdotes from practitioners, which include cases of limping disks, network, processors, and memory, along with the root causes and negative impacts on applications and users.

Limping Disks: Due to the complex mechanical nature of disk drives, disk components wear out and exhibit 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, while access speed to other healthy platters is not affected [1]. 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% [24, 28]. The disk stack also includes complex controller code that can contain *firmware bugs* that degrade performance over time [35]. Finally, as disks perform automatic *bad sector remapping*, a large number of sector errors will impose more seek cost.

Beyond the reports above, we also hear anecdotes from practitioners. For example, media failures can force disks to re-read each block multiple times before responding [15], 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 [17].

Limping 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 [13]. 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 [16]. A similar collapse was experienced at Facebook, but due to a different cause: the engineers found *network driver bugs* in the Linux stack that made a 1-Gbps NIC card transmit only at 1 Kbps, causing a chain reaction upstream that slowed down the entire 100-node cluster [14]. Finally, switches and routers can also experience *power fluctuations* that degrade performance [23].

Limping Processors and Memory: Changes in *silicon technology scaling* and scaling operating voltages to *near-threshold* for energy efficiency will produce hardware with high failure rates and variable performance [21]. Processor and memory degradation can also be attributed to *aging transistors*, and might require new solutions at the architecture or system level. Poorly designed thermals, fan failures, obstructions to airflow, and challenging workload mix can lead to *overheated* processors and memory, which can cause bandwidth throttling [32]. Finally, manufacturing variation in leakage and compensating power capping can produce as much as 26% variation in deployed systems [37].

Summary: These stories reaffirm the existence (perhaps non-rare) of limping hardware and the fact that hardware performance failures are not hidden at the device level but are exposed to applications. This motivates us to evaluate how today's systems deal with limping hardware. Do deployed systems have limping resiliency mechanisms? What is the system-level impact of a degraded hardware? Answers to these questions are paramount in building limping-hardware tolerant clouds.

3 Study of Bug/Issue Reports

To address the questions above, we performed a study of hundreds of issues reported by developers of scale-out systems such as Hadoop, HDFS, Cassandra, HBase, and ZooKeeper, and filtered 53 issues that describe the systems' inability to deal with performance variability caused by increased load and degraded hardware. Note that we include problems related to increased load because a normal load running on a limping hardware will lead to an increased load. We present some of our interesting findings in two forms: (1) the system-level impacts of limping hardware and (2) the lessons learned.

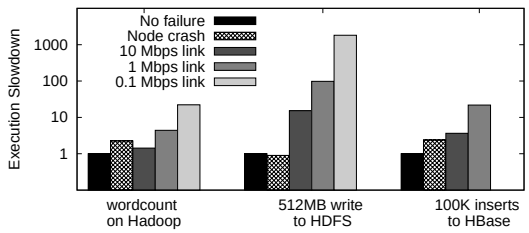

Figure 1: **Limping impacts.** *The x-axis shows our three tests. The y-axis shows the execution slowdown when a node's network link is degraded to 10, 1, and 0.1 Mbps. To illustrate the speed of fail-stop failover, we also crashed a node during the tests. Here, the latencies are only slightly affected (the "crash" bars). However, we observe no failover for limping cases; the slow node is not isolated and thus cripples the whole tests. The HBase test failed on 0.1 Mbps link.*

System-level impacts: Applications can *hang* or *crash*; a limping hardware can make application worker queues grow without bounds, run out of threads, or throw an out-of-memory exception [8, 12]. Nodes can be *incorrectly declared dead*; a limping storage or network link can delay heartbeat delivery from a slave node to the master node [7]. Worse, unhandled limping failures can lead to *cross-node cascading effects*; slow writes to on-disk transaction logs can cripple the node's ability to process incoming RPCs from clients and other nodes [11]; a full queue can overflow other nodes' queues, disabling communications across nodes [5]; and slow tasks can make bad long tails that slow down the entire cluster [6]. Cascading effects and bottlenecks in turn can cause *resource underutilization* [3]. Finally, limping servers can experience *severe overload*; clients or other nodes obliviously flood a limping node with retries [9, 10], and a temporary limping server that has recovered must suddenly serve many jobs that have been queued up during the slowdown [4]. In addition, heavy background jobs that are oblivious to limping hardware can also cause overload.

Lessons Learned/Design Flaws: We were able to highlight some design flaws that underlie why these systems are limping-hardware intolerant. Developers often use a *multi-purpose thread*; a single thread can perform multiple operations of different types, and if one operation is affected by a limping hardware (*e.g.*, volume scan), all other operations will be affected (*e.g.*, blocked heartbeats). Similarly, a *multi-purpose queue* is often used; a node uses one queue to communicate to all other nodes, and hence a limping communication between a pair of nodes can make the single queue full, disabling communication with other nodes. Limping *detection and failover recovery* are absent; tasks continue to run very slowly rather than being migrated, or a limping component is given the same amount of job as in the normal case. With no limping signals, clients or peers will *retry rather than throttle* failed operations, flooding the limping component with more load. Finally, there is *no back-pressure*; as requests flow across queues, one full queue backlogs other queues, and unbounded queues can grow infinitely, thereby overcommitting memory.

Summary: This study highlights several limping-intolerant designs. We find that new fixes are often applied to solve particular issues, and therefore we sense that many subcomponents of these systems are perhaps still limping intolerant. We confirm this hypothesis with a limping-failure analysis which we present next.

4 Limping-Failure Analysis

Our findings above motivate us to build LATE (Limping-failure Analysis and TEsting framework) which involves running real implementations, emulating limping hardware (*e.g.*, reducing network and disk throughputs), observing the system-wide impacts, and unearthing design flaws. We integrate LATE to our target systems (Hadoop, HDFS, and HBase) and perform macro and micro analysis. In macro analysis, we run benchmarks and inject coarse-grained limping failures (*e.g.*, a slow link). In micro analysis, we run specific protocols of the systems to exercise more code paths and inject unique limping I/Os observed in traversed paths. Due to space limitations, we do not explain the framework further, but rather present our findings of limping-intolerant designs (beyond what we have found manually from the bug study).

Macro analysis: We ran distributed workloads on several scale-out systems (*e.g.*, wordcount on Hadoop) running on five nodes. We emulate a limping local network card by introducing a slow link on one node. Figure 1 shows the severe implication: the execution time of all the workloads are increased by orders of magnitude. None of the systems are capable of detecting and recovering from the limping node. Although these systems are fail-stop tolerant, they are not limping-hardware tolerant.

We now turn our attention to the surprising result from the Hadoop test; we expected that the MapReduce speculative execution would rerun limping tasks on other healthy nodes. After in-depth analysis, we made two interesting observations. First, a mapper can run on a node with a slow link without being marked a straggler. This is because input and output files of a mapper are typically local files, and thus the slow link does not affect the mapper. However, when all reducers of the same job need to fetch the mapper's output through the slow link, *all* of the reducers progress at the same slow rate. The speculator is not triggered because a task is rerun only if it makes little progress relative to others of the same job.

Second, Hadoop depends on HDFS whose write protocol is not immune to limping I/Os. In HDFS, 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 a disk or network link can limp to almost 1 KB/s without triggering a failover in the write pipeline (as illustrated in the HDFS test in Figure 1). This has a negative implication to Hadoop. If all reducers of a job write their outputs to HDFS files that have a replica on a node with a slow link/disk, then *all* of the reducers will slow down without triggering the speculative execution.

Micro analysis: Our micro analysis reveals that existing *timeouts are for fail-stop*, but not limping failures. Typically, a timeout is triggered in the complete absence of a response for a long period. However, a limping hardware makes operations respond slowly (*e.g.*, the HDFS example above). We also found cases of *wrong detection* of limping node (*e.g.*, a normal node is marked dead in a slow multi-node pipeline), *unexploited parallelism* (*e.g.*, big data I/Os run in parallel, but small parallelizable transactional I/Os run sequentially), and *long lock contention* (*e.g.*, locks are sometimes held under limping I/Os, causing other operations to block for a long time).

Cases found from this analysis could happen in real deployment. Indeed, such is the case of the long lock contention problem. Intrepid administrators discovered that a limping optical transceiver caused GPFS operations on the affected node to hold tokens and locks for a long period of time. The failure cascaded, and after an hour, the whole GPFS cluster was in live-lock [16].

5 Limping-HW Tolerant Clouds

It is evident that the case studies presented throughout this paper demand a new era of limping-hardware tolerant clouds. In this section, we discuss open challenges and new research directions in building such systems. We categorize the discussion into four subprinciples: limping-hardware anticipation, detection, recovery, and utilization. We note that since this is an ongoing project, we do not propose a specific solution.

Anticipation: Limping-hardware anticipation requires limping-tolerant design patterns; every data structure and algorithm employed should take into account limping cases such that performance failures do not cascade. For example, developers can enforce the idea of *decoupled queues/threads* where every queue/thread handles a different type of operation (*e.g.*, client RPC, disk writes). One can also explore *destination-proportional queues* to ensure that messages directed to a limping destination do not fully occupy a bounded queue. We believe that when every data structure/algorithm is scrutinized, more limping-tolerant design patterns will emerge.

Developers may also want to quickly verify the absence of limping-intolerant design flaws in current and future versions. Our limping testing framework (§4) is not always a suitable option due to the time-consuming process of delaying I/Os. Therefore, we are building a *limping static analysis*, a new methodology that formally expresses "limping-tolerant rules" and performs rapid static analysis to find violations of the rules. Specifically, given a scale-out system, the tool abstracts out system-level constructs affected by limping failures (*e.g.*, queues, timeouts, threads, message handlers, locks) and then runs program analysis to search for rule violations such as missing timeouts, long lock contention, the use of multi-purpose queue/thread, and many other violations. While recent work advances static analysis by finding performance bugs [29], here we do so by focusing on limping-related bugs.

Detection: Limping-hardware detection must be accurate, fine-grained, and efficient. Accurate detection is fundamental for proper recovery. Imagine a limping-hardware induced slowdown that is incorrectly detected as overload-induced. In such case, an overload recovery might react (incorrectly) by throttling or reducing the workload, while a proper reaction is to identify and isolate the limping hardware. Fine-grained detection must be able to pinpoint the offending hardware (*e.g.*, pinpointing a limping disk as opposed to marking the node down). Finally, efficient detection should not impose excessive monitoring overhead.

To this end, we propose a distributed limping-hardware detection that synthesizes peer comparison, sampling, root-cause analysis; peer-comparison exposes a limping component from its peers, sampling reduces monitoring overhead, and root-cause analysis pinpoints the source of performance anomalies. We also propose methods that unearth implicit events from explicit events; fortunately many limping behaviors can be attributed to certain explicit causes. For example, a slow disk (implicit) can be caused by many sector remappings (explicit). Here, rather than having full-blown monitoring, one can only record explicit operations that can lead to limping behavior.

Recovery: While anticipation is about design patterns, recovery is about mechanisms and policies that manage limping hardware and allow them to "fail in place". For example, how to utilize nodes with limping disks for in-memory computation only? How to distribute computations across racks connected by a limping switch? How to differentiate recovery of transient vs. permanent limping hardware? How to move computation from a limping processor to a healthy one? In this context, distributed

systems might collaborate with operating systems in exchanging information about local hardware states. For non-local hardware such as network switches, the use of Software-Defined Networking (SDN) such as the OpenFlow framework should be explored.

Utilization: The fail-stop principle distinguishes only two failure modes (fail or working), and failed components cannot be used. In contrary, limping hardware introduces more complex failure modes; a hardware can slow down by just 1% or worse 50%. Different slowdowns might be handled differently, and hence limping hardware might still be usable in different ways depending on the domain. For example, rarely-accessed files can be put on a slow disk. Therefore limping-hardware utilization policies should also be developed.

6 Related Work

Recent work provides rich analysis of various hardware failures including machine crashes, disk failures, and memory corruption. "Formal" studies of these failures (*e.g.*, [33, 38]) were undertaken after anecdotes started to circulate. We argue that studies of limping hardware are greatly needed.

Distributed jobs have to deal with performance variability originating from stragglers, jitters, and heterogeneous hardware. Stragglers are mostly detected at the task level [22], which suffers from the limitations we described in Section 4. Jitters are often transient and sporadic in nature [42], while limping hardware could be both transient and permanent. Systems that deal with heterogeneous components typically track performance variations at levels above the hardware (*e.g.*, VM [26], request/RPC [34], or task level [41]). In other related work [31], hardware heterogeneity is expected and specified a priori, while limping hardware exhibits unexpected variability. Overall, most existing work *reacts* to performance variability but, to the best of our knowledge, none *manages* limping hardware (*e.g.*, tracking and isolating/utilizing).

Big data should flow in big pipes, but design flaws could introduce bottlenecks which lead to "small pipes" [27, 40]. In this context, our work unearths limping-intolerant designs in existing scale-out systems. Previous work on perturbation [30] only performed black-box analysis where code-debugging is a non goal, and hence did not unearth deep design flaws. In contrast, our ultimate goal is to learn from current design flaws and develop limping tolerant principles. Finally, the concept of hardware performance failure has been introduced before [19], however there was no in-depth study of the impact on existing systems.

7 Conclusion

New failure modes always dramatically transform systems design and implementation. Reports and anecdotes from large-scale system deployments have built the case that an important failure mode has emerged. Limping hardware without doubt is a destructive failure mode. Yet, current systems fail to properly manage limping hardware, and thus performance failures often cascade. It is our hope that this paper provides a strong motivation for the cloud community to discuss more challenges and new research directions in building future generation limping-hardware tolerant clouds.

8 Acknowledgments

We thank the anonymous reviewers for their tremendous feedback and comments. This material is based upon work supported by the NSF (grant Nos. CCF-1321958 and CCF-1017073). The experiments in this paper were performed in the Utah Emulab network testbed [2].

References

[1] http://forum.hddguru.com/search.php?keywords=weak-head.

[2] Emulab Network Emulation Testbed. http://www.emulab.net.

[3] Cassandra-1358: Clogged RRS/RMS stages.

[4] Cassandra-3853: Lower impact of slow nodes.

[5] Cassandra-488: Tcp Manager only has one connection.

[6] Hadoop-1984: Some reducer stuck at copy phase.

[7] Hadoop-4584: Slow generation of blockReport.

[8] HBase-4863: Make server thread pool bounded.

[9] HDFS-1203: Sleep before reentering after exception.

[10] HDFS-3032: Lease renewer tries forever.

[11] HDFS-599: Prioritize heartbeats over client requests.

[12] ZooKeeper-880: Send Worker grows without bounds.

[13] High Disk Latency. http://communities.vmware.com, 2011.

[14] Personal Communication from Dhruba Borthakur of Facebook, 2011.

[15] Personal Communication from Andrew Baptist of Cleversafe, 2013.

[16] Personal Communication from Kevin Harms of Argonne National Laboratory, 2013.

[17] Personal Communication from Mike Kasick of CMU, 2013.

[18] 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.

[19] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Fail-Stutter Fault Tolerance. In *The Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII)*, 2001.

[20] 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.

[21] Shekhar Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 2005.

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

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

[25] 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.

[26] 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.

[27] 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.

[28] Robin Harris. Bad, bad, bad vibrations. http://www.zdnet.com, 2010.

[29] 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.

[30] 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.

[31] David A. Koufaty, Dheeraj Reddy, and Scott Hahn. Bias scheudling in heterogeneous multi-core architectures. In *Proceedings of the 2010 EuroSys Conference (EuroSys)*, 2010.

[32] Jiang Lin, Hongzhong Zheng, Zhichun Zhu, Eugene Gorbatov, Howard David, and Zhao Zhang. Software Thermal Management of DRAM Memory for Multicore Systems. In *Proceedings of the 2008 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2008.

[33] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andre Barroso. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST)*, 2007.

[34] 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.

[35] Anand Lal Shimpi. Intel Discovers Bug in 6-Series Chipset: Our Analysis. http://www.anandtech.com, 2011.

[36] 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.

[37] Brian Van Straalen and Phil Collela. Resiliency and codesign. In *DOE Exascale Research Conference*, 2012. http://tinyurl.com/9wmq4cz.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.