THE UNIVERSITY OF CHICAGO

UNEARTHING CONCURRENCY AND SCALABILITY BUGS IN CLOUD-SCALE DISTRIBUTED SYSTEMS

A PROPOSAL SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
TANAKORN LEESATAPORNWONGSA

CHICAGO, ILLINOIS
2016

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Cloud services must be accessible anytime and anywhere and not lose or corrupt users' data (reliability), and scale as user base continues to grow (scalability). Unfortunately, cloud-scale distributed systems behind the services remain difficult to get right. Guaranteeing dependability has proven to be challenging in these systems. In this proposal, We are tackling this challenge. We try to unearth dependability bugs in cloud-scale distributed systems, in the aspects of reliability and scalability.

For reliability, we focus on one unsolved reliability problem in cloud systems, "*distributed concurrency (DC) bugs*". DC bugs are caused by non-deterministic order of distributed events such as message arrivals, faults, and reboots. Some interleavings of these events might not be handled properly, and lead to catastrophic failures such as data loss, data inconsistencies and downtimes.

The last seven years have seen a rise of implementation-level distributed system model checkers (dmck) for verifying the reliability of real distributed systems. Existing dmcks however rarely exercise multiple faults due to the state-space explosion problem, and thus do not address present reliability challenges of cloud systems in dealing with complex faults. To scale dmck, we introduce semantic-aware model checking (SAMC), a white-box principle that takes simple semantic information of the target system and incorporates that knowledge into state-space reduction policies.

For the second aspect, we focus on scalability. Scale surpasses the limit of a single machine in meeting users' increasing demands of compute and storage. On the negative side, scale creates new development and deployment issues. Developers must ensure that their algorithms and protocol designs to be scalable. However, until real deployment takes place, unexpected bugs in the actual implementations are unforeseen. This new era of cloud-scale distributed systems has given birth to "*scalability bugs*". They are latent bugs that are scale-dependent, and only surface in large scale.

We present SCK, a methodology that enables developers to scale-check distributed systems and find scalability bugs on one machine. To colocate a large number of nodes without sacrificing accuracy, we remove hardware contentions with four novel strategies. And with these techniques, we achieve high collocation factor.

CHAPTER 1

INTRODUCTION

As more data and computation move from local to cloud settings, cloud-scale distributed systems such as scale-out storage systems [14, 16, 20, 46], computing frameworks [15, 45], synchronization services [12, 31], and cluster management services [30, 52] have become a dominant backbone for many cloud services. Client-side software is getting thinner and more heavily relies on the capability, reliability, and availability of cloud systems. Users demand 24/7 dependability of cloud computing systems. They must be accessible anytime and anywhere and not lose or corrupt users' data, which means they must be reliable; they have to provision fast and stable response time, which means they need stable performance; and while user base continues growing, they must be scalable also.

Unfulfilled dependability is costly. Some researchers estimate that 568 hours of downtime at 13 well-known cloud services since 2007 to 2012 had an economic impact of more than $70 million [18]. Others predict worse: for every hour it is not up and running, a cloud service can take a hit between $1 to 5 million [43]. Unfortunately, such cloud-scale distributed systems remain difficult to get right. Cloud-scale distributed systems are getting more and more complex. New intricate bugs continue to create dependability issues that cause major economic loss. Guaranteeing dependability has proven to be challenging in these systems [22, 24, 53, 57].

In this proposal, we attempt to improve dependability of cloud-scale distributed systems. We are tackling this challenge by answering these 2 questions, (1) What bugs that harm the dependability?, and (2) how do we test the systems to unearth these bugs so developers can fix them? The first question is motivated by that we do not have comprehensive knowledge about the bugs in distributed systems. There are many bug studies on single-machine softwares [32, 44, 47, 48], yet there are few formal bug studies on distributed-systems softwares; they did not study in a great number and across multiple types of systems [42, 55]. We believe that we need comprehensive understanding about cloud bugs to combat them.

For the second question, we are motivated by the fact that in the past decade, systems community has developed many testing techniques [22, 24, 53, 57] to find bugs in distributed systems, but these techniques still have limitations. For example, FATE [22] tests reliability of systems by injecting faults, but it does not address concurrency in distributed systems. MODIST, which is a model checker, addresses concurrency, but it cannot work in reasonable time when injecting multiple faults. Or Exalt, which is a framework to test scalability, cannot be applied to CPU-intensive systems.

We choose to start dependability research on two aspects, reliability and scalability. For reliablity, we find that one unsolved reliability problem in cloud systems is "*distributed concurrency (DC) bugs*". DC bugs are caused by non-deterministic order of distributed events such as message arrivals, faults, and reboots. Cloud systems execute multiple complicated distributed protocols concurrently (*e.g.*, serving users' requests, operating some background tasks, and combined with untimely hardware failures). The possible interleavings of the distributed events are not completely envisioned by developers and some interleavings might not be handled properly. The buggy distributed interleavings can cause catastrophic failures such as data loss, data inconsistencies and downtimes. Our effort to tackle reliability issues will concentrate on DC bugs.

And for scalability, we see that most of the work [13, 37, 49, 53, 59, 60] focuses on the data path, mainly to validate the scalability of read/write operations (linear throughput or stable latency as the cluster scales). But scalability correctness however is not merely about the data path. Distributed systems are full of "control paths" such as bootstrapping, rebalancing, and adding/decommissioning nodes (scaling out/down). These management protocols must modify cluster-wide metadata that lives in each node in the system (*e.g.*, ring partition table) to decide how data flows in the cluster. Unfortunately, control path correctness is often overlooked, so we aim our attention to "*control-plane scalability bugs*" in this proposal.

We propose how to further the current testing techniques beyond the limitations in this proposal. The proposal is arranged in this order: chapter 2 explains the problem being solved in detail and discusses related work, chapter 3 shows our research in detail, and chapter 5 gives a conclusion. The proposal is a fusion of our previous work and our on-going work. It includes cloud bug studies [23, 41], semantic-aware model checking [40], and scale check methodology.

CHAPTER 2
BACKGROUND AND RELATED WORK

In this proposal, we aim to improve the dependability of the systems in two aspects, reliability and scalability. Our work focus on unearthing bugs that are related to these two issues. For reliability, we focus on *distributed concurrency (DC) bugs*, and for scalability, we focus on *control-plane scalability bugs*. This chapter discusses the background of these two types of bugs and related work to combat them.

2.1 Distributed Concurrency Bugs

One notorious type of software bugs is concurrency bugs. These timing-related bugs manifest non-deterministically, and hence are extremely difficult to detect, diagnose, and fix. A huge body of work exists in this space that focuses on "local" concurrency (LC) bugs in single-machine multi-threaded software, caused by incorrect interleaving of memory accesses.

Unfortunately, the reliability of datacenter distributed systems is severely threatened by non-deterministic concurrency bugs as well, which we refer as *distributed concurrency (DC) bugs*. Distributed systems execute many complicated distributed protocols on hundreds/thousands of machines with no common clocks. Moreover, cloud systems run on large clusters of unreliable commodity machines, an environment that produces a growing number and frequency of failures, including "surprising" failures [11, 29]. This combination makes distributed systems prone to DC bugs caused by non-deterministic timing of distributed events such as message arrivals, node crashes, node reboots, and timeouts. It is common to see complex fault-induced DC bugs such as the one in Figure 2.1.

We look at 104 DC bugs from widely-deployed cloud-scale datacenter distributed systems including Cassandra, Hadoop MapReduce, HBase, and ZooKeeper. Statistically, Figure 2.2a (FLT) shows that 63% of DC bugs must have at least one fault. In more detail, Figure 2.2b-d (TO,

ZooKeeper Bug #335:

1. Nodes A, B, C start with latest txid #10 and elect B as leader,
2. *B crashes*,
3. Leader election re-run; C becomes leader,
4. Client writes data; A and C commit new txid-value pair {#11:X},
5. *A crashes before* committing tx #11,
6. C loses quorum,
7. *C crashes*,
8. *A reboots* and *B reboots*,
9. A becomes leader,
10. Client updates data; A and B commit a new txid-value pair {#11:Y},
11. *C reboots after* A's new tx commit,
12. C synchronizes with A; C notifies A of {#11:X},
13. A replies to C the "diff" starting with tx 12 (excluding tx {#11:Y}!),
14. Violation: permanent data inconsistency as A and B have {#11:Y} and C has {#11:X}.

Figure 2.1: **Zookeeper bug #335.** *Timing steps for ZK-335 to manifest*

CR, RB) shows the percentage of issues that require timeouts, crashes and reboots respectively, including how many instances of such faults must be there; the rest is other faults such as disk errors (not shown).

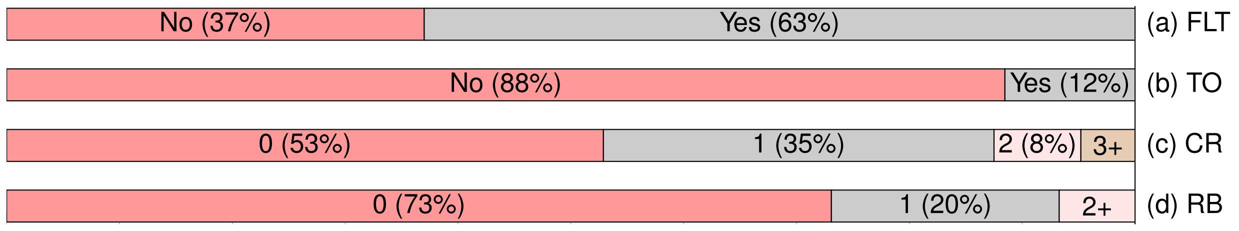

Figure 2.2: **Statistical overview of DC bugs.** *(a) the percentage of bugs that require faults; (b) the percentage of bugs that require timeouts; (c) the percentage of bugs that require the different number of crashes; (d) the percentages of bugs that require the different number of reboots.*

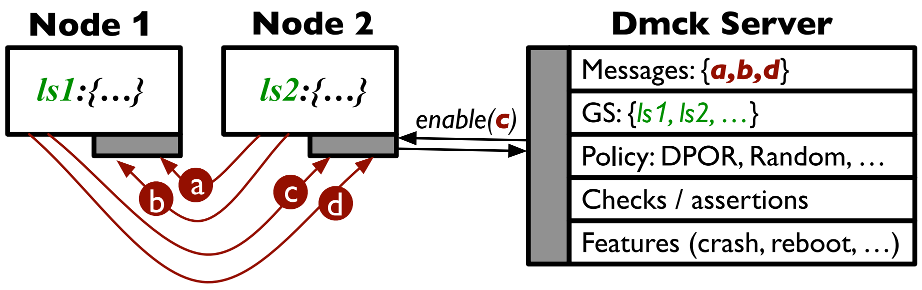

Figure 2.3: **DMCK.** *A typical framework of a distributed system model checker (dmck).*

2.1.1 Distributed Systems Model Checker (DMCK)

In order to unearth DC bugs the question we have to answer is: "*can we exercise necessary conditions (i.e.workloads and faults) and test different event re-ordering to hit the bugs?*". This is the job of distributed system model checkers (dmck), which are gaining popularity recently [24, 36, 50, 57]. Dmck works by intercepting distributed events and permuting their ordering, and hereby pushing the target system into corner-case situations and unearthing hard-to-find bugs. However, the more events included, the more scalability issues will arise due to state-space explosion.

The last seven years have seen a rise of software model checker that checks distributed systems directly at the implementation level. Figure 2.3 illustrates a dmck integration to a target distributed system, a simple representation of existing dmck frameworks [24, 36, 50, 57]. The dmck inserts an interposition layer in each node of the target system with the purpose of controlling all important events (*e.g.*, network messages, timeouts) and preventing the target system to process the events until the dmck enables them. A main dmck mechanism is the permutation of events; the goal is to push the target system into all possible ordering scenarios. For example, the dmck can enforce `abcd` ordering in one execution, `bcad` in another, and so on.

2.1.2 *State-of-the-Art DMCKs*

MODIST [57] is arguably one of the most powerful dmcks that comes with systematic reduction policies. MODIST has been integrated to real systems due to its exploration scalability. At the heart of MODIST is *dynamic partial order reduction (DPOR)* [19] which exploits the *independence* of events to reduce the state explosion. Independent events mean that it does not matter in what order the system execute the events, as their different orderings are considered equivalent.

To illustrate how MODIST adopts DPOR, let's use the example in Figure 2.3, which shows four concurrent outstanding messages `abcd` (`a` and `b` for `N1`, `c` and `d` for `N2`). A brute-force approach will try all possible combinations (`abcd`, `abdc`, `acbd`, `acdb`, `cabd`, and so on), for a total of 4! executions. Fortunately, the notion of event independence can be mapped to distributed system properties. For example, MODIST specifies this reduction policy: a message to be processed by a given node is independent of other concurrent messages destined to other nodes (based on vector clocks). Applying this policy to the example in Figure 2.3 implies that `a` and `b` are dependent[1] but they are independent of `c` and `d` (and vice versa). Since only dependent events need to be reordered, this reduction policy leads to only 4 executions (`ab-cd`, `ab-dc`, `ba-cd`, `ba-dc`), giving a 6x speed-up (4!/4).

Although MODIST's speed-up is significant, we find that one scalability limitation of its DPOR application is within its *black-box* approach; it only exploits general properties of distributed systems to define message independence. It does not exploit any semantic information from the target system to define more independent events.

Dynamic interface reduction (DIR) [24] is the next advancement to MODIST. This work suggests that a complete dmck must re-order not only messages (global events) but also thread interleavings (local events). The reduction intuition behind DIR is that different thread interleavings often lead to the same global events (*e.g.*, a node sends the same messages regardless of how

1. In model checking, "dependent" events mean that they must be re-ordered. "Dependent" does not mean "causally dependent".

threads are interleaved in that node). DIR records local exploration and replays future incoming messages without the need for global exploration. In our work, SAMC focuses only on global exploration (message and fault re-orderings). We believe DIR is orthogonal to SAMC, similar to the way DIR is orthogonal to MODIST.

MODIST and DIR are examples of dmcks that employ advanced systematic reduction policies. LMC [21] is similar to DIR; it also decouples local and global exploration. dBug [50] applies DPOR similarly to MODIST. There are other dmcks such as MACEMC [36] and CrystalBall [56] that use basic exploration methods such as depth first (DFS), weight-based, and random searches.

Other than the aforementioned methods, *symmetry* is another foundational reduction policy [17, 51]. Symmetry-based methods exploit the architectural symmetry present in the target system. For example, in a ring of nodes, one can rotate the ring without affecting the behavior of the system. Symmetry is powerful, but we find no existing dmcks that adopt symmetry.

Besides dmcks, there exists sophisticated testing frameworks for distributed systems (*e.g.*, FATE [22], PREFAIL [34], SETSUDO [33], OpenStack fault-injector [35]). This set of work emphasizes the importance of multiple faults, but their major limitation is that they are not a dmck. That is, they cannot systematically control and permute non-deterministic choices such as message and fault reorderings.

2.2 Control-Plane Scalability Bugs

Distributed systems are full of "control paths" such as bootstrapping, rebalancing, and adding/decommissioning nodes (scaling out/down). These management protocols must modify cluster-wide metadata that lives in each node in the system (*e.g.*, ring partition table) to decide how data flows in the cluster.

Our work emphasizes that scalability bugs also linger in control paths (*i.e.*, *control-plane scalability bugs*). These bugs are unfortunately often overlooked, but yet control path correctness is crucial in today's era of elastic cloud where cluster size is not constant all the time; rebooting,

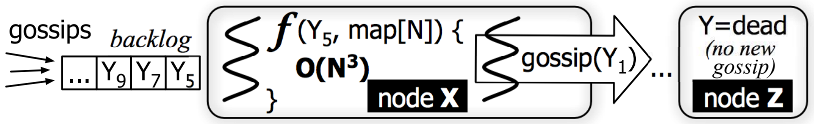

Figure 2.4: **Cassandra bug CA-6127.** *The problem of gossip-based failure detection in Cassandra*

scaling out/down, and rebalancing are common operations. We show an example of control-plane in §2.2.1, and our observations in §2.2.2.

2.2.1 A Sample Bug

We now describe in detail a control-plane scalability bug in Cassandra, CA-6127 [7]. The bug surfaced on a cluster with hundreds of nodes and led to "***flapping***" nodes, a condition where node up/down status continuously changes; tens of thousands of flaps[1] were observed.

To understand this bug, we need to understand the following protocols.

- **Bootstrap:** Each node first creates partition keys (*e.g.*, 32 random numbers) and gossips this information to peer nodes.

- **Gossip broadcast:** *Every second*, each node gossips to one random node about a list of nodes and partitions it knows (including itself) and their *version* numbers. Each node also increments its version number ("I'm still alive") before gossiping.

- **Gossip processing:** The receiving node then finds any state (metadata) differences between the two nodes to synchronize their views of the ring. Eventually, all nodes know about each other.

- **Failure detection:** *Every second*, a failure detection daemon runs [38]. Put simply, if a node X has not received a new gossip about Y *from anyone* (Y's version has not changed after

1. A "**flap**" is when a node X marks a peer node Y as down.

some period of time), X will declare Y dead (a flap). When X receives a new gossip about Y, it marks Y alive.

There are two factors that induce the bug. The first is the *long latency of scale-dependent state-update gossip processing during bootstrapping* ("f" in Figure 2.4). While gossip processing is usually fast in a stable cluster, it is expensive during bootstrapping as the gossips carry many new state changes about the ring; the state-update processing time is scale-dependent ($O(N^3)$); the larger the cluster (N), the larger the ring map, the longer the processing time is. This long latency is caused by **(1)** state-update checkpoint to on-disk database and **(2)** multi-map cloning and updates. The first one is needed for fast fault tolerance; after a node crashes, it can reboot fast as it knows the latest view of the ring. The second one is preferred for simplicity; Cassandra clones its `MultiMap` ring table and applies changes one by one to alleviate long write locks.

The second factor is the *single threaded* implementation of gossip processing. As shown in Figure 2.4, this inability to process multiple gossips/state updates concurrently (for the sake of preventing concurrency bugs) creates a *backlog* of new gossips. For example, in *every second*, Y tells someone it's alive with increasing version number (*e.g.*, Y_7), but the receiving nodes are still busy processing state changes and only forward Y's old version number (*e.g.*, Y_1). As Y's new gossip is not propagated on time, other nodes (*e.g.*, Z) will mark Y as dead. This happens to all nodes, not just Y.

2.2.2 Observations

From the bug above, we make several important observations regarding control-plane scalability bugs and distributed

- *Only appear at extreme scale:* CA-6127 does not surface in 30-node deployment. In 128-node cluster, the symptom appears mildly (tens of flaps). From 200-500 nodes, flapping skyrockets to thousands of flaps. Testing in small/medium scales is not sufficient.

- *Scalable in design, but not in practice.* Related to CA-6127, the accrual failure detector [28] in Cassandra is scalable in design [38]. However, the design proof does not account gossip processing time, which can be long. To understand the bug, the developers tried to "do the [simple] math" [7] but failed. In practice, new gossips are not propagated every second (due to the backlog). The actual implementations overload gossips with many other purposes (*e.g.*, announcing boot/rebalance changes) beyond their original design sketch.

- *Implementation specific and hard to predict.* The backlog-induced flapping in CA-6127 was caused specifically by Cassandra's implementation choice: metadata checkpoint, multi-map cloning, and its single-threaded implementation. State-update processing time is hard to predict (ranges from 0.001 to 4 seconds) as it depends on a 2-dimensional input: the receiving node's ring table size and the number of new state changes.

- *Cascading impacts of "not-so-independent" nodes.* In cluster-wide control protocols, distributed nodes are not necessarily independent; nodes must communicate with each other to synchronize their views of cluster metadata. As the cluster grows, the cluster metadata size increases. Thus, unpredictable processing time in individual nodes can create cascading impacts to the whole cluster.

- *Long and difficult large-scale debugging:* The bug report of CA-6127 generated over 40 back-and-forth discussion comments and took 2 months to fix. It is apparent [7] that there were many hurdles of deploying and debugging the buggy protocol at real scale. Important to note is that debugging is *not* a single iteration; developers must *repeatedly* instrument the system (add more logs) and re-run the system at scale to find and fix the bug, which is not trivial. The scalability bugs we studied took 6 to 157 days to fix (27 on average).

- *Not all developers have large test budgets:* Another factor of delayed fixes is the lack of budget for large test clusters. Such luxury tends to be accessible to developers in large companies, but not to open-source developers. When CA-6127 was submitted by a customer who had hundreds of nodes, the Cassandra developers did not have an instant access to a test cluster of the same scale.

- *Quick fixes and repeated bugs:* Bugs are often fixed with quick patches (development pressures), but the fix might not eradicate the problem completely [58]. The patch for CA-6127 simply disables failure detection during bootstrap. But the bug still appeared in another workload (*e.g.*, scaling out from 128 to 256 nodes). The simple fix has been removed later and the gossip protocol has been redesigned. And old fixes can become obsolete in protocol re-designs, which then can give birth to new scalability bugs such as the fix for CA-3831 became obsolete as "vnodes" was introduced, then led to a new scalability bug (CA-3881).

2.2.3 State of the Art

We now discuss popular approaches (simulation, extrapolation, and emulation) for unearthing scalability bugs. First, simulation approaches test system/application models in different scales [13, 37]. A model can look scalable but the actual implementation can contain unforeseen bugs. Our observations above accentuate the need for scale-checking distributed system *implementations* at *real scale*.

Second, extrapolation monitors system behaviors in "mini clusters" and extrapolates them to larger scales (§2.1 in [53]). However, mini clusters tend to be order(s) of magnitude smaller than real deployments. Most importantly, system behaviors do not always extrapolate linearly [53]; for the bugs in our work (§3.1.3), even an extrapolation based on a 100-node cluster will not reveal the bug symptoms.

Finally, real-scale emulation checks real implementations in an emulated environment (*e.g.*, DieCast and Exalt). DieCast [26], invented for network emulation, can colocate many processes/VMs on a single machine as if they run individually without contention. The trick is adding "time dilation factor" (TDF) support [27] into the VMM (*e.g.*, Xen). For example, TDF=5 implies that for every second of wall-clock time, each emulated VM on the VMM believes that time has advanced by only 200 ms. The most significant drawback of DieCast is that high colocation factor (*e.g.*, TDF=100) is likely not desirable, for two reasons: prolonged testing time (TDF=100 implies 100x longer run) and memory overcapacity. Many distributed systems today are implemented in managed languages (*e.g.*, Java, Erlang) whose runtimes consume non-negligible memory overhead. Java and Erlang VMs for example use around 70 and 64 MB of memory per process respectively. DieCast was only evaluated with TDF=10.

Exalt [53] targets I/O-intensive scalability bugs. With a custom data compression, users' data is compressed to zero byte on disk (but the size is recorded) while metadata is not compressed. With this, Exalt can co-locate 100 emulated HDFS datanodes on one machine. In its evaluation, most of the bugs reproduced are in the HDFS namenode which runs alone on one machine. As the authors stated, their approach "may not discover scalability problems that arise at the nodes that are being emulated [the datanodes]" (§4.1 in [53]). Thus, Exalt is not suitable for finding control-plane scalability bugs in P2P distributed systems.

In summary, we did not find a fast single-machine approach that can scale-check control-plane protocols in P2P systems. The scalability bugs here are typically caused by the scale-dependent processing time, not network or I/O bottlenecks. As DieCast targets *network* emulation via time dilation and Exalt targets *storage* space emulation via compression, our work uniquely targets *processing time* emulation, completing a missing piece.

CHAPTER 3
RESEARCH DETAIL

To unearth bugs in cloud-scale distributed systems, the first thing we need is comprehensive knowledge about them, so we can establish proper solutions. In our research plan, we start with general bug study on cloud systems to see the big picture of the current problems. Then we dissect DC bugs and scalability bugs to analyze. For DC bugs, we will advance the state of the art of dmcks. We are introducing more pruning policies to tackle state-space explosion problem. And for scalability bugs, we will fill the missing piece of control-plane protocol bugs focusing on P2P systems.

3.1 Bug Study

Bug or failure studies can significantly guide many aspects of dependability research. Many dependability researchers have recently employed formal studies on bugs and failures [25, 42]. These studies can identify opportunities for new research, build taxonomies of new problems, or test new tools. We start our work by doing formal bug study to gain foundations of bugs in cloud systems.

3.1.1 Cloud Bug Study (Previous Work)

As an initiative, our group have performed the largest bug study in six important Apache cloud infrastructures including Cassandra, Flume, Hadoop MapReduce, HBase, HDFS, and ZooKeeper [23]. We reviewed in total 21,399 submitted issues within a three-year period (2011-2014). We perform a deep analysis of 3,655 "vital" issues (*i.e.*, real issues affecting deployments) with a set of detailed classifications.

Methodology

- **Target Systems & Bug Repositories:** We select six popular cloud systems that represent a diverse set of system architectures: Hadoop MapReduce [4] (distributed computing frame-

Classification	**Labels**
Issue Type	Vital, miscellaneous.
Aspect	Reliability, performance, availability, security, consistency, scalability, topology, QoS.
Bug scope	Single machine, multiple machines, entire cluster.
Hardware	Core/processor, disk, memory, network, node.
HW Failure	Corrupt, limp, stop.
Software	Logic, error handling, optimization, config, race, hang, space, load.
Implication	Failed operation, performance, component downtime, data loss, data staleness, data corruption.

Table 3.1: **Issue Classifications.**

works), Hadoop File System (HDFS) [8] (scalable storage systems), HBase [5] and Cassandra [2] (distributed NoSQL systems), ZooKeeper [6] (synchronization services), and finally Flume [3] (streaming systems). All development projects of the target systems maintain highly organized issue repositories. Each repository contains development and deployment issues submitted mostly by the developers or a larger user community. The term “issue” is used here to represent both bugs and new features.

- **Issue Classifications:** We introduce issue classifications as displayed in Table 3.1. We carefully read each issue to decide whether the issue is vital. If an issue is vital we proceed with further classifications, otherwise it is labeled as miscellaneous and skipped in our study.

Study Result

The finding of this study was published in the CBS paper [23]. The product of our classifications is stored in CBSDB [1], a set of raw text files, data mining scripts and graph utilities, which enables us (and other CBSDB users) to perform both quantitative and qualitative analysis of cloud issues.

This study brings new insights on some of the most vexing problems in cloud systems. We show a wide range of intricate bugs, many of which are unique to distributed cloud systems (*e.g.*, scalability, topology, and killer bugs). And it is the main source of our DC-bug taxonomy and scalability-bug analysis.

3.1.2 DC Bug Taxonomy

While there have been many LC-bug studies, we are not aware of any large-scale study of DC bugs. To fill the void, we have created the largest and most comprehensive taxonomy of 104 real-world DC bugs (named TAXDC) from four distributed system: Cassandra, HBase, Hadoop MapReduce/Yarn, and ZooKeeper [41].

Methodology

- **Target Systems & Bug Repositories:** We examined bugs from four distributed systems that represent a diverse set of system architectures: Hadoop MapReduce [4] (distributed computing frameworks), HBase [5] and Cassandra [2] (distributed NoSQL systems), and ZooKeeper [6] (synchronization services). We started our study from CBSDB [1], which already labels issues related to concurrency bugs. However, beyond simple labeling, the CBS work does not differentiate DC from LC bugs and did not dissect DC bugs further. Thus, we first filtered out LC bugs, then exclude DC bugs that do not contain clear description, and finally randomly picked 104 samples from the remaining detailed DC bugs, specifically 19 Cassandra, 30 HBase, 36 Hadoop MapReduce, and 19 ZooKeeper DC bugs, reported in January 2011-2014 (the time range of CBS work).

- **Taxonomy:** We study the characteristics of DC bugs along three key stages: triggering, errors & failures, and fixing as shown in Table 3.2.

Study Result

We publish the result in TAXDC paper [41], and we also release TAXDC database online [1]. TAXDC contains in-depth characteristics of DC bugs, stored in the form of 2,083 classification labels and 4,528 lines of re-enumerated steps to the bugs that we manually added.

With TAXDC, we can answer important questions such as: What are the root causes of DC

Triggering
What is the triggering timing condition?
Message arrives unexpectedly late/early
Message arrives unexpectedly in the middle
Fault (component failures) at an unexpected state
Reboot at an unexpected state
What are the triggering inputs preconditions?
Fault, reboot, timeout, background protocols, and others
What is the triggering scope?
How many nodes/messages/protocols are involved?
Errors & Failures
What is the error symptom?
Local memory exceptions
Local semantic error messages & exceptions
Local hang
Local silent errors (inconsistent local states)
Global missing messages
Global unexpected messages
Global silent errors (inconsistent global states)
What is the failure symptom?
Node downtimes, data loss/corruption, operation failures, slowdowns
Fixing
What is the fix strategy?
Fix Timing: add global synchronization
Fix Timing: add local synchronization
Fix Handling: retry message handling at a later time
Fix Handling: ignore a message
Fix Handling: accepting a message without new computation logics
Fix Handling: others

Table 3.2: **Taxonomy of DC Bugs.**

bugs (out-of-order messages, failures, etc.)? Are existing LC-bug-detection tools applicable for DC bugs? How do developers fix DC bugs (by adding locks, states, etc.)? What are the inputs/triggering conditions? What are the minimum number of distributed events needed to trigger the bugs (how many messages to re-order, failures to inject, etc.)? What errors/effects (specification violations) are caused by DC bugs (deadlock, data loss, state inconsistency, performance problems, etc.)? How do propagation chains form from the root causes to errors? The answers to these questions guide our subsequent research projects.

3.1.3 Scalability Bug Study

For scalability bugs, the situation is even worse. We are not aware of any study on scalability bugs at all. We started a pilot study of scalability bugs to gain some insight about them. We studied 12 bugs in four key-value stores including Cassandra, Couchbase, Riak, and Voldemort.

Methodology

- **Target Systems & Bug Repositories** Focusing on control-plane protocol in P2P systems, we chose four popular P2P key-value stores to study: Cassandra, Couchbase, Riak, and Voldemort. Because CBS is a study aiming on different kinds of systems, the only P2P key-value store CBS addresses is Cassandra. We have to mine bug repositories for the others manually. And we found 12 control-plane scalability bugs. (This manual mining was arduous because there is no standard jargon for "scalability bugs"; we might have missed other related bugs.)
- **Detail Study** In each bug, we studied the problematic protocol in serveral aspects: protocol design, implementation, problem root cause, symptom, and fix.

Preliminary Result

From the study, we can gain some insight about scalability bugs in control-plane protocols. We show the observations in Section 2.2.2. These observations guide us how to create a methodology for checking scalability of the systems.

3.2 Distributed System Model Checking

One big challenge faced by a dmck is the state-space explosion problem (*i.e.*, there are too many distributed events to re-order). To address this, existing dmcks adopt a random walk or basic reduction techniques such as dynamic partial order reduction (DPOR). Despite these early successes, existing approaches cannot unearth many real-world DC bugs, so we are advancing the state of the art of dmck to combat DC bugs, which described next.

3.2.1 Semantic-Aware Model Checking

We started our work by specifically addressing two limitations of existing dmcks. First, existing dmcks treat every target system as a complete *black box*, and therefore perform unnecessary reorderings of distributed events that would lead to the same explored states (*i.e.*, redundant executions). Second, they do not incorporate complex multiple fault events (*e.g.*, crashes, reboots) into their exploration strategies, as such inclusion would exacerbate the state-space explosion problem.

To address these limitations, we introduced Semantic-Aware Model Checking (SAMC) [39, 40], a novel white-box model checking approach that takes *semantic knowledge* of how distributed events (specifically, messages, crashes, and reboots) are processed by the target system and incorporates that information in reduction policies. The policies are based on sound reduction techniques such as DPOR and symmetry. The policies tell SAMC not to re-order some pairs of events such as message-message pairs or crash-message pairs, yet preserves soundness, because those cut out re-orderings are redundant.

Semantic-Aware Reduction Policies

- **Local-Message Independence (LMI):** A black-box DPOR treats the message processing inside the node (local message) as a black box, and thus must declare the incoming messages to the same node as dependent. We propose LMI that if we have semantic knowledge about message processing, we can define independency relationship among local messages. LMI prune out state space by not reordering independent local messages.

- **Crash-Message Independence (CMI):** When crash happens, black-box DPOR expects that every node will react to the crash (recovery) and consider that the crash is independent to every current on-going messages. But CMI suggests that some crashes could be considered independent and avoid reordering those crashes.

- **Crash Recovery Symmetry (CRS):** While injecting crash, dmck needs to make sure that it covers every crash scenario by trying every possible crash (*i.e.* try crashing every possible node). However, at a particular time, the systems could recover from two different crashes in the same manner. CRS claims that dmck can explore only one crash recovery scenario, and prune out the others.

- **Reboot Synchronization Symmetry (RSS):** Same as CRS, dmck does not need to try reboot for every possibility, if the reboot does not lead to any different behavior.

Result

					#Executions				**Speed-up of SAMC vs.**		
Old Issue#	**Protocol**	**E**	**C**	**R**	**bDP**	**RND**	**rDP**	**SAMC**	**bDP**	**RND**	**rDP**
ZooKeeper-335	ZAB	120	3	3	↑5000	1057	↑5000	117	⇑43	9	⇑43
ZooKeeper-790	ZLE	21	1	1	14	225	82	7	2	32	12
ZooKeeper-975	ZLE	21	1	1	967	71	163	53	18	1	3
ZooKeeper-1075	ZLE	25	3	2	1081	86	250	16	68	5	16
ZooKeeper-1419	ZLE	25	3	2	924	2514	987	100	9	25	10
ZooKeeper-1492	ZLE	31	1	0	↑5000	↑5000	↑5000	576	⇑9	⇑9	⇑9
ZooKeeper-1653	ZAB	60	1	1	945	3756	3462	11	86	341	315
MapReduce-4748	SE	25	1	0	22	6	6	4	6	2	2
MapReduce-5489	CM	20	2	1	↑5000	↑5000	↑5000	53	⇑94	⇑94	⇑94
MapReduce-5505	CM	40	1	1	1212	↑5000	1210	40	30	⇑125	30
Cassandra-3395	RW+HH	25	1	1	2552	191	550	104	25	2	5
Cassandra-3626	GS	15	2	1	↑5000	↑5000	↑5000	96	⇑52	⇑52	⇑52

Table 3.3: **SAMC Speed in Finding Old Bugs.** *"E", "C" and "R" represent the number of events, crashes, and reboots necessary to hit the bug. The numbers in the middle four columns represent the number of executions to hit the bug across different policies. "bDP", "RND", and "rDP" stand for black-box DPOR (in* MODIST*), random, and random + black-box DPOR respectively. We stop at 5000 executions (around 2 days) if the bug cannot be found (labeled with "↑"). Thus, speed-up numbers marked with "⇑" are potentially much higher.*

SAMC is able to reproduce 12 old bugs in three cloud systems (Cassandra, Hadoop MapReduce, and ZooKeeper) involving 30-120 distributed events and multiple crashes and reboots. Some of these bugs cannot be unearthed by non-SAMC approaches, even after two days. SAMC can find the bugs up to 271x (33x on average) faster compared to state-of-the-art techniques as shown in Table 3.3. Additionally, we found two new bugs in Hadoop MapReduce and ZooKeeper.

3.3 Scale Emulation

Next we move on to scalability bugs. Our observations in Section 2.2.2 accentuate the need for scale checking distributed system implementations at real scale, not via simulation nor extrapolation. Our proposed solution is to colocate as many nodes as possible (*e.g.*, hundreds) on one machine without sacrificing accuracy by *emulate* hardware resources such that the individual nodes behave as if they run on independent machines. We are introducing SCK a scale-check methodology for control-plane protocols in P2P key-value systems.

We first summarize our target problem, challenges, and principles in tackling the problem.

- **Target problem/protocols:** We focus on scale-checking P2P control-plane protocols (*e.g.*, bootstrapping, rebalancing, scale out/down). The scalability bugs here tend to be caused by CPU-intensive computation on data structures that are dependent on cluster scale.

- **The challenge:** A major challenge here is "how can we *colocate hundreds of CPU-intensive and memory-hungry nodes* on one machine with limited resources and yet still achieve *high accuracy* and *unprolonged debugging time*?" High accuracy implies that the colocated nodes observe a similar behavior as if they run on independent machines. Unprolonged debugging time implies that if a scalability bug surfaces in T minutes in real deployment, it must also be observable within a similar time frame during the reproduction of the bug (unlike time dilation impacts in DieCast).

- **Principles:** Each distributed system has its own unique protocols, and implementation details. Thus, we introduce SCK as a *general methodology* that can be applied to various P2P systems in unique ways. We also show how distributed systems should be re-architected to make them *scale-checkable* ("SCK-able") on one machine. SCK is not designed to depend on custom operating systems or libraries (*e.g.*, time-dilated VMM or compression library).

3.3.1 Proposed Emulation Techniques

- **Processing Illusion (PIL):** To emulate CPU-intensive processing, we introduce *processing illusion* (PIL), an approach that *replaces an actual processing with* `sleep()`. For example, in CA-6127, we can replace the expensive ring-table update with `sleep(t)` where `t` is an accurate timing of how long the update takes.

  The intuition behind PIL is similar to the intuition behind other emulation techniques. For example, Exalt provides an illusion of storage space; their insight was "how data is processed is not affected by the content of the data being written, but only by its size" [53]. PIL provides an illusion of compute processing; our insight is that *"the key to computation is not the intermediate results, but rather the execution time and eventual output"*.

  To make PIL feasible, these are challenging questions that we are going to answer.

  1. How can a function be safely replaced with `sleep()`*without* changing the whole processing semantic?
  2. How to find specific functions that should be replaced with `sleep()`?
  3. How can we produce the output if the actual compute is skipped?
  4. How can we predict the actual compute time (`t`) accurately?

- **Single-Process Cluster (SPC):** The next challenge that we address is an overhead due to running multiple processes. For example, Voldemort [10] are implemented in Java, which each JVM consumes non-negligible memory overhead (70 MB). As we target 3-digit colocation factor, this memory overhead becomes an unnecessary limitation. Furthermore, a managed-language VM can contain advanced services. For example, Erlang VM contains a DNS service which sends heartbeat messages to other connected VMs. As hundreds of Erlang VMs (one for each Riak node) run on one machine, the heartbeat messages cause a "network" overflow that disconnects Erlang VMs.

To address this, we propose Single-Process Cluster (SPC) support wherein the whole cluster runs as threads in a single process. But SPC is not naive change. When we look at our target systems, this cannot be done without re-designing the code to support SPC. We introduce SCK-able architecture, in which systems can run in SPC mode. For example, systems should have arrays of per-node global data structures, and be rid of static-synchronized functions that lock the whole cluster when run in SPC mode.

Moreover, if we can make all nodes run in one process, user-kernel switching to send messages becomes unnecessary. Thus, we can create a shim layer in our target systems to bypass OS network calls to reduce this overhead.

- **Global Event Driven Architecture (GEDA):** Next, we can further SPC by reducing thread switching overhead. With SPC, a node still runs multiple daemon threads (gossiper, failure detector, etc.). With high colocation factor, there are more than one thousand threads that cause severe context switching and long queueing delays.

  We address this overhead by leveraging the staged event-driven architecture (SEDA) [54] common in distributed system implementations. With SEDA, each service/stage in each node exclusively has an event queue and a handler thread. In SCKmode, we convert SEDA to *global-event driven architecture* (GEDA). That is, for every stage, there is only *one* queue and one handler for the *whole* cluster.

  This architecture will reduce the number of threads in SCKrun, and mitigate thread context switching overhead.

- **Memory Footprint Reduction (MFR):** The last thing we can perform is memory footprint reduction (MFR). We propose that we can reduce memory overhead from system-specific root causes to prevent out-of-memory exceptions.

First, relevant services in the target protocol can "over-allocate" memory. For example, in Riak's bootstrap+rebalance protocol, each node creates $N \times P$ partition services although at the end only retain P partitions and never use (remove) the other $(N-1) \times P$ partitions (as reclaimed by other nodes). Worse, each partition service is an Erlang process (1.3 MB of memory overhead); colocating 30 nodes (N=30 with default P=64) will directly consume 75 GB of memory (30×30×64×1.3 MB) from the start. To reduce this memory usage, we must modify Riak to remove this unoptimized memory usage.

Second, some libraries can cause high memory footprints. For example, Voldemort nodes use Java NIO [9] which is fast but contains buffers and connection metadata that take up memory space. We should change this to network bypass from SPC.

We are working on these four proposed techniques. PIL is the most challenging technique to implement, and it is the most important technique to scale check CPU-intensive protocols. There are many questions we need to answer to make PIL possible. And as we mentioned, some techniques need to re-design the codebase to make the techniques possible, but we believe the modification is not complex and will be small.

CHAPTER 4
RESEARCH PROGRESS AND PLAN

4.1 Research Progress

Bug Study	
Cloud bug study	This is a previous work from our group. It was an equal effort work. We built CBSDB and published the CBS paper at SoCC '14 [23]. Tanakorn Leesatapornwongsa was responsible for ZooKeeper bug study in this work.
DC bug taxonomy	This is our complete work. It is done in 2015. We released our study and taxonomy (named TAXDC) online. This work was published at ASPLOS '16 [41]. Two students in this work did equal effort. Tanakorn Leesatapornwongsa was responsible for Cassandra and ZooKeeper DC bug study.
Scalability bug study	This is our preliminary work to understand scalability bugs in distributed systems. This is a foundation in our scale check project
Distributed-System Model Checking	
Semantic-aware model checking	This is our complete work. It is done in 2014. We released our model checker to the public and published the SAMC paper at OSDI '14
Scale Check	
SCK	This is our ongoing work. We are introducing scale check concepts to Cassandra, Riak, and Voldemort systems.

4.2 Research Plan

To fulfill the dissertation research, we are working on our last piece, *scale check*. This is a plan of action we proposed.

1. Develop processing illusion (PIL) on Cassandra, 1 month, 12/1/2016 - 12/31/2016
2. Develop single-process cluster (SPC) on Cassandra, 3 weeks, 1/1/2017 - 1/21/2017
3. Develop global event driven architecture (GEDA) on Cassandra, 1 weeks, 1/22/2017 - 1/28/2017
4. Develop memory footprint reduction (MFR) on Cassandra, 1 week, 1/29/2017 - 2/4/2017
5. Integrate all techniques and summarize the SCK methodology, 1 week, 2/5/2017 - 2/11/2017
6. Apply SCK to Riak and Voldemort, 3 weeks, 2/12/2017 - 2/25/2017
7. Evaluate SCK 2 weeks, 2/26/2017 - 3/11/2017
8. Write the dissertation, 1 month, 3/12/2017 - 4/12/2017
9. Defend the dissertation

CHAPTER 5
SUMMARY

In this proposal, we aim to unearth hidden bugs in cloud-scale distributed systems that weaken reliability and scalability of the systems. For reliability bugs, we focus in distributed concurrency (DC) bugs. For scalability bugs, we focus in control-plane protocol bug, specifically in P2P storage systems.

The key idea for unearthing DC bugs is advancing model checking techniques for dmcks to tackle state-space explosion. Because existing dmcks treat every target system as a complete *black box*, and therefore perform unnecessary reorderings of distributed events that would lead to the same explored states (*i.e.*, redundant executions). To tackle the problem, we introduced Semantic-Aware Model Checking (SAMC), a novel white-box model checking approach that takes *semantic knowledge* of how distributed events (specifically, messages, crashes, and reboots) are processed by the target system and incorporates that information in reduction policies. The policies are based on sound reduction techniques such as DPOR and symmetry.

And for scalability bugs, we believe from the study that we need to scale check distributed system implementation at real scale, not via simulation nor extraploation. Our proposed solution is to colocate as many nodes as possible (*e.g.*, hundreds) on one machine without sacrificing accuracy by *emulate* hardware resources such that the individual nodes behave as if they run on independent machines. We propose four emulation concepts to mitigate hardware contention problems. We are exploring these concepts to build scale check methodology for checking control-plane protocols in P2P key-value stores.

REFERENCES

[1] http://ucare.cs.uchicago.edu/projects/cbs/.

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

[3] Apache Flume. http://flume.apache.org/.

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

[5] Apache HBase. http://hbase.apache.org.

[6] Apache ZooKeeper. http://zookeeper.apache.org.

[7] BUG: CASSANDRA-6127: vnodes don't scale to hundreds of nodes. https://issues.apache.org/jira/browse/CASSANDRA-6127.

[8] HDFS Architecture. http://hadoop.apache.org/common/docs/current/hdfs_design.html.

[9] NIO in Voldemort: Non-heap memory usage. https://groups.google.com/forum/#!topic/project-voldemort/J7ADKefjR50.

[10] Project Voldemort. http://www.project-voldemort.com/voldemort/.

[11] Ken Birman, Gregory Chockler, and Robbert van Renesse. Towards a Cloud Computing Research Agenda. *ACM SIGACT News*, 40(2):68–80, June 2009.

[12] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[13] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[14] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[15] 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.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of 21st ACM SIGOPS symposium on Operating systems principles (SOSP)*, 2007.

[17] E. Allen Emerson, Somesh Jha, and Doron Peled. Combining Partial Order and Symmetry Reductions. In *The 3rd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 1997.

[18] Loek Essers. Cloud Failures Cost More Than $70 Million Since 2007, Researchers Estimate. http://www.pcworld.com, 2012.

[19] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2005.

[20] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[21] Rachid Guerraoui and Maysam Yabandeh. Model Checking a Networked System Without the Network. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[22] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[23] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[24] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[25] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. Failure Recovery: When the Cure Is Worse Than the Disease. In *The 14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*, 2013.

[26] Diwaker Gupta, Kashi Venkatesh Vishwanath, and Amin Vahdat. DieCast: Testing Distributed Systems with an Accurate Scale Model. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[27] Diwaker Gupta, Kenmeth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. To Infinity and Beyond: Time-Warped Network Emulation. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[28] Naohiro Hayashibara, Xavier Defago, Rami Yared, and Takuya Katayama. The Phi Accrual Failure Detector. In *The 23rd Symposium on Reliable Distributed Systems (SRDS)*, 2004.

[29] Alyssa Henry. Cloud Storage FUD: Failure and Uncertainty and Durability. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST)*, 2009.

[30] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[31] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*, 2010.

[32] 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.

[33] Pallavi Joshi, Malay Ganai, Gogul Balakrishnan, Aarti Gupta, and Nadia Papakonstantinou. SETSUDO : Perturbation-based Testing Framework for Scalable Distributed Systems. In *Conference on Timely Results in Operating Systems (TRIOS)*, 2013.

[34] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. PREFAIL: A Programmable Tool for Multiple-Failure Injection. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[35] Xiaoen Ju, Livio Soares, Kang G. Shin, Kyung Dong Ryu, and Dilma Da Silva. On Fault Resilience of OpenStack. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[36] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[37] Ignacio Laguna, Dong H. Ahn, Bronis R. de Supinski, Todd Gamblin, Gregory L. Lee, Martin Schulz, Saurabh Bagchi, Milind Kulkarni, Bowen Zhou, Zhezhe Chen, and Feng Qin. Debugging High-Performance Computing Applications at Massive Scales. *Communications of the ACM (CACM)*, 58(9), September 2015.

[38] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2009.

[39] Tanakorn Leesatapornwongsa and Haryadi S. Gunawi. SAMC: A Fast Model Checker for Finding Heisenbugs in Distributed Systems. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2015.

[40] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep

Bugs in Cloud Systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

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

[42] Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. A Characteristic Study on Failures of Production Distributed Data-Parallel Programs. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, 2013.

[43] David Linthicum. Calculating the true cost of cloud outages. `http://www.infoworld.com`, 2013.

[44] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

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

[46] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat Datacenter Storage. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[47] Nicolas Palix, Gal Thomas, Suman Saha, Christophe Calvs, Julia Lawall, and Gilles Muller. Faults in Linux: Ten Years Later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[48] Swarup Kumar Sahoo, John Criswell, and Vikram Adve. An Empirical Study of Reported Bugs in Server Software with Implications for Automated Bug Diagnosis. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, 2010.

[49] Sergei Shudler, Alexandru Calotoiu, Torsten Hoefler, Alexandre Strube, and Felix Wolf. Exascaling Your Library: Will Your Implementation Meet Your Expectations? In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

[50] Jiri Simsa, Randy Bryant, and Garth Gibson. dBug: Systematic Evaluation of Distributed Systems. In *5th International Workshop on Systems Software Verification (SSV)*, 2010.

[51] A. Prasad Sistla, Viktor Gyuris, and E. Allen Emerson. SMC: a symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering and Methodology*, 2010.

[52] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[53] Yang Wang, Manos Kapritsos, Lara Schmidt, Lorenzo Alvisi, and Mike Dahlin. Exalt: Empowering Researchers to Evaluate Large-Scale Storage Systems. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[54] 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.

[55] Tian Xiao, Jiaxing Zhang, Hucheng Zhou, Zhenyu Guo, Sean McDirmid, Wei Lin, Wenguang Chen, and Lidong Zhou. Nondeterminism in MapReduce Considered Harmful? An Empirical Study on Non-commutative Aggregators in MapReduce Programs. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014.

[56] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[57] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[58] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT international symposium on Foundations of software engineering (FSE)*, 2011.

[59] Bowen Zhou, Milind Kulkarni, and Saurabh Bagchi. Vrisha: Using Scaling Properties of Parallel Programs for Bug Detection and Localization. In *Proceedings of the 20th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2011.

[60] Bowen Zhou, Jonathan Too, Milind Kulkarni, and Saurabh Bagchi. WuKong: Automatically Detecting and Localizing Bugs that Manifest at Large System Scales. In *Proceedings of the 22nd IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2013.