DRAFT:
Scalability Bugs: When 100-Node Testing is Not Enough

Tanakorn Leesatapornwongsa, Cesar A. Stuardo, Riza O. Suminto,
Huan Ke, Jeffrey F. Lukman, and Haryadi S. Gunawi

University of Chicago

Abstract

We highlight the problem of scalability bugs, a new class of bugs that appear in "cloud-scale" distributed systems. Scalability bugs are latent bugs that are cluster-scale dependent, whose symptoms typically surface in large-scale deployments, but not in small or medium-scale deployments. The standard practice to test large distributed systems is to deploy them on a large number of machines ("real-scale testing"), which is difficult and expensive. New methods are needed to reduce developers' burdens in finding, reproducing, and debugging scalability bugs. We propose "scale check," an approach that helps developers find and replay scalability bugs at real scales, but do so only on one machine and still achieve a high accuracy (i.e., similar observed behaviors as if the nodes are deployed in real-scale testing).

1 Introduction

> "For Apache Hadoop, *testing at thousand-node scale* has been one of the most effective ways of finding bugs, but it's both *difficult* and *expensive*. It takes considerable expertise to deploy and operate a large-scale cluster, much less debug the issues. Running such a cluster also *costs thousands of dollars an hour*, making scale testing impossible for the solo contributor. As it stands, we are heavily *reliant on* test clusters operated by *large companies* to do scale testing. A way of finding scalability bugs without requiring running a large-scale cluster would be extremely useful." — Andrew Wang (Cloudera and Apache Hadoop PMC Member and Committer).

As Ousterhout stated, "scale has been the single most important force driving changes in system software over the last decade" [33]. Scale allows users to meet their increasing computing demands that cannot be met in a single machine and allows service providers to amass compute and storage resources from hundreds to thousands of machines. A tremendous deployment scale can be witnessed in the field. Tens of 500-node Cassandra clusters are running behind Netflix, a total of 100,000 Cassandra nodes are reportedly deployed in Apple, and 40,000 servers power up tens of Hadoop clusters at Yahoo! [1, 13, 14].

While scale delivers many benefits, it creates new development and deployment issues. Developers must ensure that their protocol designs are scalable, however until real-scale deployments take place, unexpected bugs deep in the actual implementations are hard to foresee. We believe that this new era of highly scalable distributed systems gives rise to a new type of bugs, *scalability bugs*, latent bugs that are scale dependent, whose symptoms surface in large-scale deployments, but not necessarily in small/medium-scale deployments.

As an example, let us consider a bug in Cassandra, a highly-scalable peer-to-peer key-value store. If a customer initially deploys a cluster of 50 nodes and later scales it out with 50 additional nodes, the operation can be done smoothly. However, if the customer deploys a 200-node cluster and then adds 200 more nodes, the protocol that rebalances the key-range partitions (which nodes should own which key ranges) becomes CPU intensive as the calculation has an $O(N^3)$ complexity where N is the number of nodes. This combined with the gossiping and failure detection logic leads to a scalability bug that makes the cluster unstable (many live nodes are declared as dead, making some data not reachable by the users).

We perform an in-depth study of 38 scalability bugs reported from the deployments of popular large-scale systems such as Hadoop, HBase, HDFS, Cassandra, Couchbase, Riak, and Voldemort. From this study, we observed many challenges in finding, reproducing, and debugging scalability bugs. As in the example above, bug symptoms sometimes surface only in large deployment scales (*e.g.*, N>100 nodes), hence small/medium-scale testing is not enough. Yet, not all developers have large test budgets, and even when they do, debugging on hundreds of nodes is time consuming and difficult, as also alluded in the developer's quote above. Furthermore, protocol algorithms can be scalable in the design

sketches, but not necessarily in the real deployments; there are specific implementation details whose implications at scale are hard to predict.

Existing testing practices however do not address the challenges above. For example, testing on "mini" clusters may not reveal all the scalability bugs. Extrapolation also does not work if the bug symptoms have not yet to surface in smaller scales. Simulation can verify large-scale models, but not the actual implementation code. Real-scale testing and debugging at the same scale as in customer-site deployments is expensive.

New solutions are needed to reduce developers' burdens in finding and debugging scalability bugs. We propose "*scale check*," an approach that helps developers find and replay scalability bugs at real scales, but do so on one machine and still achieve a high accuracy. A key challenge is to colocate as many nodes as possible (*e.g.*, hundreds) on one machine but still observe a similar behavior as real-scale testing. We will discuss the technical challenges that arise to achieve such a vision; for example, how to reduce colocation bottlenecks and emulate/replay CPU-intensive processing as if they run on independent machines without contention delays.

In subsequent sections, we present a narrative of scalability bugs that repeatedly appear in Cassandra development (§2), the lessons learned from our bug study (§3), the state of the art of large-scale testing (§4), our proposed solution (§5), other colocation challenges (§6), current state and future work (§7), and promising initial results (§8).

2 The Story of Cassandra

Our journey in understanding scalability bugs began when we observed repeated "flapping" problems in large-scale Cassandra deployments. Flapping is a cluster instability problem where node's up/down status continuously flaps. A "flap" is when a node X marks a peer node Y as down (and soon marks Y as alive again). We rigorously study a series of Cassandra bugs below that surfaced as the code evolved.

In Bug #C3831 [2], when a node D is decommissioned from a cluster ring, D initiates a gossip telling that all other nodes must rebalance the ring's key-ranges. This scale-dependent "pending key-range calculation" is CPU intensive with $O(MN^3log^3(N))$ complexity; M is the list of key-range changes in the gossip message. This in turn leaves many gossips not propagated on time, creating flapping symptoms that only appear at scale (at 200+ nodes; §8). The developers then optimized the code to $O(MN^2log^2(N))$ complexity.

Soon afterwards (Bug #C3881 [3]), Cassandra added the concept of virtual partitions/nodes (*e.g.*, P=256 per physical node). As an implication, the fix above did not scale as "N" becomes $N \times P$. The bug was fixed with a complete redesign of the pending key-range calculation, making it $O(MNPlog^2(NP))$.

About a year later (C5456 [4]), Cassandra code employs multi-threading between the pending key-range calculation and the gossip processing with a coarse-grained lock to protect sharing of the ring table. Unbeknownst to the developers, at scale, the key-range calculation can acquire the lock for a long time, causing flapping to reappear again. The fix clones the ring table for the key-range calculation, to release the lock early.

Later on (C6127 [5]), a similar bug reappeared. In the above cases, the problems appeared when the cluster grows/shrinks gradually. However, if customers bootstrap a large cluster (*e.g.*, 500+ nodes) from scratch (*i.e.*, all nodes do not know each other, with no established key ranges), the execution traverses a different code path that performs a fresh ring-table/key-range construction with $O(MN^2)$ complexity.

The story continues on (C6345, C6409, etc.). Fast forward today, Cassandra developers recently started a new umbrella ticket for discussing "Gossip 2.0," supposedly scalable to 1000+ nodes [7, 8]. Similar to Cassandra, other large-scale systems are prone to the same problem. So far, we have collected and analyzed 9 Cassandra, 5 Couchbase, 2 Hadoop, 9 HBase, 11 HDFS, 1 Riak, and 1 Voldemort scalability bugs, all caused user-visible impacts. This manual mining was arduous because there is no searchable jargon for "scalability bugs"; we might have missed other bugs.

3 Challenges

From all the bugs we studied, we observe many challenges in combating scalability bugs.

• *Not all developers have large test budgets:* Scalability bugs only surface at scale (§8). However, the luxury of using large test clusters tends to be accessible only to developers in large companies. When C6127 was submitted by a customer with 500+ nodes, the developers assigned to fix the bug did not have access to a cluster of the same scale, delaying the time to fix.

• *Long and difficult large-scale debugging:* Even when developers acquire large test clusters, we observe many hurdles of deploying and debugging the buggy protocol at real scale. Important to note, debugging is not a single iteration; developers often need to replay the whole process numerous times. The scalability bugs we studied took 1 month to fix on average (with a maximum of 5 months), and not to mention the tens of back-and-forth discussion comments among the developers.

• *Scalable in design, but not in implementation/practice.* One might wonder if the bugs can be avoided by simply verifying the high-level design. Unfortunately, the root causes are deep in the implementation details and hard to predict. For example, the gossip-related bugs in §2 involve the accrual failure detector/gossiper [27] which was actually adopted by Cassandra for its scalable property [29]. However, the design model and proof did not account gossip processing time during bootstrap/cluster-rescale, whose duration is hard to predict (ranges from 0.001 to 4 seconds in our test). For c6127, the developers tried to "do the math" [5] but failed. This is because in the actual implementation, the gossip protocol is overloaded with many other operational purposes (*e.g.*, announcing boot/rescale changes) beyond the original design. As code evolves, new scalability bugs reappear.

• *Diverse protocols.* While most works focus on the scalability of the data paths (read/write protocols), scalability correctness is not merely about the data paths. The bugs we studied lingered in diverse data and control paths, including bootstrap, scale-out, decommission, rebalance, and failover protocols, all must be tested at scale.

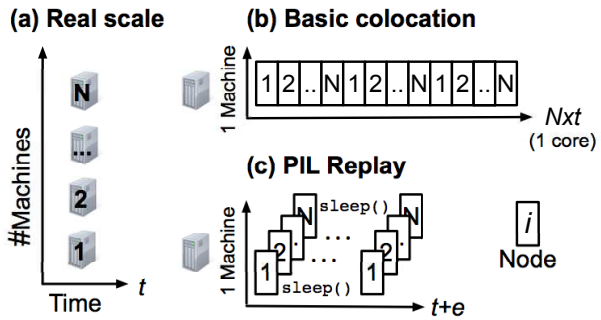

Figure 1: **Various scale-testing approaches.** *The left figure (a) illustrates a real-scale testing where the system/protocol under test is deployed on N machines, which illustratively takes t time to complete. The top figure (b) depicts a basic colocation where N nodes are packed into a single machine and exhibit CPU contention and context switching, which can take $N \times t$ time to complete (in one-processor scenario). The bottom figure (c) illustrates our processing illusion (PIL) as described in §5. Here, expensive functions are emulated with* `sleep()`*, thus the test time $t+e$ is similar to the real-scale testing.*

4 State of the Art

The last few years have seen a rise of research that addresses for scalability bugs in distributed systems and parallel applications, which can be categorized into four techniques.

Testing/benchmarking is arguably the developers' most popular choice. A standard practice is to test large-scale systems on "mini" clusters. If no bugs appear, the code "passes" the test. While this approach is straightforward, mini clusters tend to be order(s) of magnitude smaller than the real deployments [30, §2.2], hence bugs might not surface. Real-scale testing on the other hand is not economical, as illustrated in Figure 1a.

Simulation depends on the developers to model their code and then simulate the model in different scales [17, 28]. This approach is popular for modeling HPC applications, but is rarely used for server/infrastructure code which tends to be more complex to model. As alluded earlier, a design/model can look scalable but the actual implementation can still contain unforeseen bugs.

Extrapolation learns system behaviors in small scale (*e.g.*, 4-8 nodes) and then extrapolates them to larger scales [28, 41]. If the behavior in the real deployment is different than the extrapolated behavior, the code might be buggy. Again, bug symptoms might not appear in the small training scale, hence the behaviors are hard to extrapolate accurately.

Emulation runs implementation code at real scale but in an emulated (smaller) environment. For example, DieCast [24] can colocate many VMs on a single machine as if they run individually without contention. The trick is adding "time dilation factor" (TDF) support into the VMM [25]. For example, colocating 10 VMs (TDF=10) implies that for every second, each emulated VM believes that time has only advanced by 100 ms. With a higher colocation factor (TDF=N), each debugging iteration will imply a much longer run ($N \times t$), as illustrated in Figure 1b.

Another emulation technique, Exalt [34], targets I/O-intensive scalability tests. With Exalt, user data is compressed to zero byte on disk (but the size is recorded). With this, Exalt can colocate 100 HDFS datanodes on one machine without space contention. The evaluation mainly tests the scalability of the (non-emulated) HDFS master node and may not discover bugs in the emulated datanodes [34, §4.1]. While Exalt targets data paths and I/O emulation, 47%[1] of the scalability bugs that we studied involve complex scale-dependent CPU computations in data and control paths, which to the best of our knowledge are not addressed in existing literature.

5 Scale Check

We believe new methods are needed to help developers check their systems/protocols implementation at real scale but without the hurdles of running large test clus-

[1] The other 53% are unexpected serializations of $O(N)$ operations, which can be caught by slightly extending our program analysis (§5).

ters. In our work, we explore a new approach to find and replay scalability bugs in a "cheap" way such as on one machine, which we name *single-machine scale check* (or just "scale check" short).

The research question to address is: how to colocate a large number of CPU-intensive nodes on one machine with limited resources and yet still achieve high accuracy? High accuracy implies that the colocated nodes generate a similar behavior as if they run on independent machines. The reason for inaccuracy is illustrated in Figures 1a and 1b. With real-scale testing (Figure 1a), the protocol under test might finish in t seconds. However, with a basic colocation, the CPU-intensive nodes contend with each other in one machine. With only just 1 processor core for example, the protocol under test might finish in $N \times t$ seconds, hence the inaccuracy.

To address this, below we present the concept of processing illusion (PIL) and how to find PIL-safe functions and generate output of PIL-replaced functions.

• **Processing Illusion (PIL)** To achieve accuracy, we must address the CPU contention delays ($N \times t$) in basic colocation (Figure 1b). We propose emulating CPU-intensive processing with *processing illusion* (PIL), which replaces an actual processing with `sleep()`. With PIL, an expensive function will sleep and wake up in accurate time with the correct output. As illustrated in Figure 1c, if some computations can be emulated with `sleep()` and the output data is automatically generated given the input data, then the resulting time is more accurate ($t+e$) to the one in real-scale testing.

PIL extends the intuition behind data-space emulation [34], where the insight is: "how data is processed is not affected by the content of the data being written, but only by its size." For PIL, our insight is that *"the key to computation is not the intermediate results, but rather the execution time and eventual output."* In other words, what matters is the global cascading implication of the long execution time of the individual nodes.

• **Finding PIL-safe and offending functions:** One key question PIL method raises is: which functions can be safely replaced with `sleep()` *without* changing the whole processing semantic? We name them "PIL-safe functions/code blocks." We set a rule that a PIL-safe function must have a memoizable output (*i.e.*, a deterministic output on a given input) and not have any side effects such as disk I/Os, network messages, and blocking mechanisms such as locks. Many functions satisfy the rule above, but not all PIL-safe functions should "take the PIL"; that is, they might not be the "offending" functions that lead to scalability bugs. Thus, we raise another key question: which functions are offending?

We learned that many offending functions contain loops that are cluster-size dependent (*e.g.*, a `for`-loop that iterates a cluster-ring data structure). Some of the loops can also be a nested loop. Finding such code blocks are unfortunately not straightforward. Scale-dependent loops can span across multiple functions; in C6127, $O(N^3)$ loops span 1000+ LOC across 9 functions. Moreover, they can be inside some `if-else` branches reachable only from a certain path/workload; in C6127, the last $O(N^2)$ loop is only exercised if the cluster bootstraps from scratch. All of these suggest that finding PIL-safe and offending functions require an advanced program analysis (which we discuss later). Such a tool will guide the developers to decide which paths/protocols to test, to uncover potential scalability bugs.

• **Memoizing PIL-replaced functions:** PIL-safe and offending functions will become "PIL-replaced functions" where their actual processing will be skipped during replays with `sleep(t)`. Thus, two more questions to address are: how to produce the output if the actual computation is skipped and how to predict the actual compute time (`t`) accurately?

The answer to the first question is *pre-memoization*. That is, given a PIL-replaced code block, we need to first execute the code block and record the input/output around it. The only way to do this on a single machine is to run the protocol with basic colocation, which will consume some time due to the CPU contention delays. However, this will only be a *one-time* overhead, while the fast PIL-infused replay stage can be repeated numerous times without contention.

It is challenging to pre-memoize PIL-replaced functions with an offline input-sampling method without running the protocol at least once. The reason is that, in the context of large-scale, decentralized, non-deterministic distributed systems, covering all possible input/output pairs may require an "infinite" time and storage space. In other words, input/output pairs depend on the precise order of message arrivals, which can be random. In a ring rebalancing algorithm for example, with N nodes and P partitions/node, there are $(N^{NP})^2$ input/output pairs given all possible orderings. Thus, to cap the state space, the pre-memoization stage also records message ordering, which will be deterministically enforced during PIL-infused replay. With this "order determinism," we do not have to record all possible input/output pairs. We simply record pairs that are observed in one particular run of the protocol test.

The answer to the second question (predicting `t`) is *in-situ time recording*; in addition to storing input/output pairs we also store input/duration pairs (Figure 2c). It is almost impossible to predict compute time with a prediction/static-analysis approach. As mentioned above, nested loops can span across multiple functions with many `if-else` conditions. In a Cassandra bug,

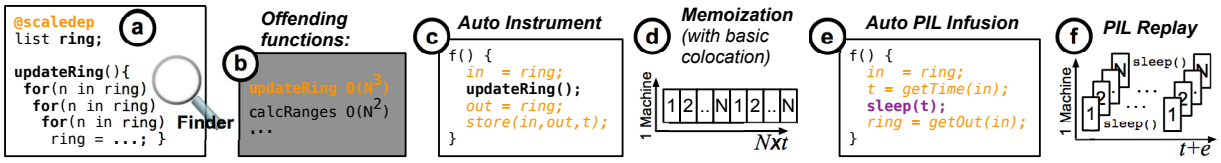

Figure 2: **The proposed flow of an automated scale-check process.** *The figure is described in Section §7.*

the duration of an offending code block can range from 0.001 to 4 seconds depending on multi-dimensional inputs. One might also wonder whether time recording is enough to hint the developers of the potential scalability bugs (*e.g.*, 4 seconds of compute should raise a red flag). As mentioned earlier, every implementation is unique (§3); for example, in C5456, if the lock is fine-grained, the long compute will not cause cascading impacts. Furthermore, patches of scalability bugs do not always remove the expensive computation. Put simply, scalability bugs are not merely about the expensive functions, but rather their global implications.

6 Other Colocation Challenges

PIL successfully reduces CPU contention, however as we colocate more nodes, before we hit 100% CPU utilization, we hit other colocation bottlenecks such as memory exhaustion and process/thread context-switching delays. We share some of the interesting stories below, which all point to the fact that current distributed systems are not built with scale-checkability in mind.

First, managed language runtimes consume non-negligible memory overhead (*e.g.*, 70MB/process in Java), which can prohibit colocation of hundreds of nodes. Second, as each node runs multiple daemon threads (gossiper, failure detector, etc.), with high colocation, thousands of threads cause severe context switching and long queuing delays. Finally, developers sometimes write simple, but inefficient and space-oblivious code; for example, in a rebalance protocol, each node over-allocates $(N-1)\times P\times$1.3MB partition services while only needing $P\times$1.3MB services eventually at the end.

Note that with N-node colocation, all the bottlenecks above are amplified by N times. This opens up a new research challenge: *how to re-design existing distributed systems to be scale-checkable?* Our current solution is to redesign the target systems and their unit tests, for example by running all nodes in one process (to reduce per-process runtime overhead) and redesign the code non-intrusively to a "global" event-driven architecture which mimics staged event-driven architecture (SEDA) [35], but with one queue and one multi-threaded handler for the whole cluster (to reduce context switching overhead).

7 Current State and Future Work

To reduce developers' burdens, the entire process above must be done automatically. Figure 2 depicts the whole scale-check process that we propose. ⓐ First, developers lightly annotate (*e.g.*, <30 LOC) data structures that are scale dependent. ⓑ The PIL-safe and offending function finder (a program analysis) will find loops that are scale dependent (that iterate on the scale-dependent data structures). This program analysis will provide reports of offending functions along with the paths (if-else branches) that would exercise them, so that the developers can set up the corresponding test workloads (*e.g.*, rebalancing, decommissioning). ⓒ The finder also automatically inserts input/output/time recording around the offending functions. ⓓ The target protocols are executed with basic colocation, which takes time and is inaccurate, but only a one-time overhead. ⓔ The deterministic replayer automatically replaces the expensive functions with `sleep(t)` and copy the output from the memoization database. ⓕ Finally, the fast and accurate PIL-infused replay can begin, and if needed, the developers can add more logs to debug the code at step ⓔ and replay again.

• **Current state:** We have built ⓓ the PIL memoizer and ⓕ replayer with promising results for Cassandra (§8). The PIL-replaced functions are currently picked and replaced manually.

• **Future work:** Our next major step is to automate the entire scale-check process (ⓐ-ⓒ and ⓔ in Figure 2), including the program analysis to find offending functions and automate PIL instrumentation, and integrate the process to other distributed systems beyond Cassandra.

8 Initial Results

This section presents some promising results in scale-checking Cassandra, geared towards answering the following two questions: (1) Can scale-check reproduce known scalability bugs? (2) Is scale-check accurate?

To measure accuracy, we compare scale-check results with real deployments of 32, 64, 128, and 256 nodes. Each (Nome [9]) machine has 16-core AMD Opteron(tm) 8454 processors with 32-GB DRAM. As it

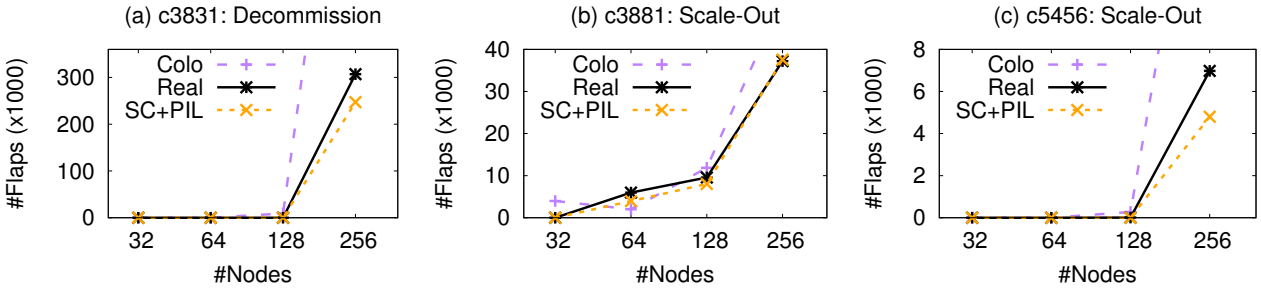

Figure 3: **Initial results.** *The figures show the results of reproducing three scalability bugs in Cassandra (C3831 [2], C3881 [3], and C5456 [4]). The y-axis shows the number of flaps (as explained in §2) when Cassandra is deployed on a number of nodes as plotted in the x-axis. The* `Real`*,* `Colo`*, and* `SC+PIL` *lines represent the number of flaps observed in real-scale testing, basic colocation, and PIL-infused replay respectively. The figures show that the results of* `SC+PIL` *are close to* `Real`*.*

is hard to acquire 256 machines (especially near deadlines), we pack a maximum of 8 nodes on one physical machine; for our target protocols, each node only uses at most 2 busy cores (*e.g.*, gossiper and gossip-processing threads). Scale-check only runs on one such machine.

Figure 3 show the three bugs we have reproduced (C3831 [2], C3881 [3], and C5456 [4]). The figures highlight that we can reproduce the same scalability bug symptoms on one machine. Specifically, the bug symptom shown in Figure 3 is the total number of flaps (alive-to-dead transitions; §2) observed in the whole cluster during the testing of the protocols. As shown, basic colocation ("`Colo`") leads to an inaccurate results that are far off from the real-scale testing ("`Real`"). However, our PIL-based scale-check process ("`SC+PIL`") mimics similar behaviors observed in real-scale testing.

The figures also show that significant *#flaps* only surface in larger deployment scales. For example, the flapping symptoms in C3831 and C5456 are not observable in 128-node deployments, again accentuating the need for real-scale emulation.

In terms of memoization and replay time, for 256-node colocation, the memoization time for the bugs we reproduced takes between 7 to 125 minutes while the replay time is only between 4 to 15 minutes, similar to the real deployments; the basic colocation does not take $N \times t$ duration because one node only consumes 2 cores (the machine has 16 cores) and also not every node is busy all the time. With PIL-infused replays, developers can have enough time budget to debug the buggy protocol numerous times as needed to discover the root cause.

Currently, on the 16-core 32-GB Nome machine, we can reach a maximum colocation factor of 512. When we tried colocating 600 nodes, we hit one of the following limitations: high CPU contention (>90% utilization), memory exhaustion (nodes receive out-of-memory exceptions and crash), or high event lateness (queuing delays from thread context switching).

9 Conclusion

Modern distributed systems are complex and prone to many types of bugs [21, 22], including concurrency [26, 32], configuration [37], service dependency [40], error handling [31, 38], performance [15, 18, 19], and security [39] bugs. In this paper, we argue that scalability bugs are new-generation bugs to combat in cloud-scale distributed systems, which we believe will raise many interesting research questions. More challenges lie ahead. Recent work reported a wide range of latent scalability bugs that depend on different axes of scale: cluster size, data/metadata size, load, and failure [16, 21, 23]. We hope our work will call for more innovations in this new area of research.

10 Acknowledgments

We thank the anonymous reviewers for their tremendous feedback and comments. This material was supported by funding from NSF (grant Nos. CCF-1336580, CNS-1350499, CNS-1526304, and CNS-1563956) as well as generous donations from Huawei, EMC, Google Faculty Research Award, NetApp Faculty Fellowship, and CERES Center for Unstoppable Computing. The experiments in this paper were performed mainly in the NMC PRObE [10, 20] testbed, and partially in the Utah Emulab [6, 36], Chameleon [12], and University of Chicago RIVER [11] testbeds, supported under NSF grants Nos. CNS-1042537, CNS-1042543, CNS-1419165 and CNS-1405959.

References

[1] Apache Cassandra. https://en.wikipedia.org/wiki/Apache_Cassandra.

[2] BUG: CASSANDRA-3831: scaling to large clusters in GossipStage impossible due to calculatePendingRanges. https://issues.apache.org/jira/browse/CASSANDRA-3831.

[3] BUG: CASSANDRA-3881: reduce computational complexity of processing topology changes. https://issues.apache.org/jira/browse/CASSANDRA-3881.

[4] BUG: CASSANDRA-5456: Large number of bootstrapping nodes cause gossip to stop working. https://issues.apache.org/jira/browse/CASSANDRA-5456.

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

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

[7] Gossip 2.0. https://issues.apache.org/jira/browse/CASSANDRA-12345.

[8] Gossip 2.0. http://mail-archives.apache.org/mod_mbox/cassandra-dev/201609.mbox/%3CCAHjqPuJMkfZwp9DDX45PNBNhkoGXsPW4TFT6Zxv%2BTTz_Pg3Y%2Bg%40mail.gmail.com%3E.

[9] NMC PRObE Nome Nodes. https://www.nmc-probe.org/wiki/Nome:Nodes.

[10] Parallel Reconfigurable Observational Environment (PRObE). http://www.nmc-probe.org.

[11] RIVER: A Research Infrastructure to Explore Volatility, Energy-Efficiency, and Resilience. http://river.cs.uchicago.edu.

[12] The Chameleon Cloud Project. https://www.chameleoncloud.org/.

[13] Running Netflix on Cassandra in the Cloud. https://www.youtube.com/watch?v=97VBdgIgcCU, 2013.

[14] Why the world's largest Hadoop installation may soon become the norm. http://www.techrepublic.com/article/why-the-worlds-largest-hadoop-installation-may-soon-become-the-norm/, 2014.

[15] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[16] Peter Bodik, Armando Fox, Michael Franklin, Michael Jordan, and David Patterson. Characterizing, Modeling, and Generating Workload Spikes for Stateful Services. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[17] 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.

[18] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[19] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[20] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. Probe: A thousand-node experimental cluster for computer systems research. *USENIX ;login:*, 38(3), June 2013.

[21] 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.

[22] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, 2016.

[23] 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.

[24] 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.

[25] Diwaker Gupta, Kenneth 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.

[26] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

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

[28] 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.

[29] 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.

[30] Tanakorn Leesatapornwongsa and Haryadi S. Gunawi. The Case for Drill-Ready Cloud Computing. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[31] 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.

[32] 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.

[33] John Ousterhout. Is scale your enemy, or is scale your friend?: technical perspective. *Communications of the ACM (CACM)*, 54(7), July 2011.

[34] 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.

[35] 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.

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

[37] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[38] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[39] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazieres. Securing Distributed Systems with Information Flow Control. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

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

[41] 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.