SLCK: A Single-Machine Approach for Finding Scalability Bugs in Cloud-Scale Distributed Systems

Double blind, Paper # $\mathbf{xx}_0$

Abstract

Abstract here. This is where we should put the abstract. Abstract should be written here. See Section **??**.

1 / 30

11 / 3

1 Introduction

(HSG: Decide the terms!!!!!!! idle wait? sleep-replacable functions? commit states?)

"is scale a friend or foe?" [**?**]. One one side it is a friend. As single machine hits its limit, and users demand of compute and storage, today has seen an explosive era of scalable systems. Friend. We are in the age of horizontal scalable systems. Bascially, horizontal scaling means when we add more machines to our distributed system, the system becomes more capable to yield more output. We have seeing a tremendeous amount of scale. For example Netflix runs 50 clusters of each 500-node of Cassandra. Apple is reported to run Cassandra clustesr with a total of 100,000 nodes. Hadoop and Yarn deployment also reach X nodes. This is the level that we say cloud-scale.

Scale also it is a foe. However, building scalable distributed systems is not trivial, we need scalable algorithms, scalable protocols, scalable design, and most of all the implementation must be scalable. More and more institutions are writing their own versions of distributed systems. (Personal communication with various engineers at VMware, Huawei, ...). and the must make sure their design works scalably.

Unfortunately, new types of bugs appear in this wworld of cloud-scale distributed systems [**?**]. One new type of bug is scalbility bugs. Scalability bugs do happen. The existing work has shown that there are many forms of scalability bugs, that happen in production. Scalability bugs are latent bugs that are scale-dependent. They don't arise in small-scale deployment but arise in large-scale production runs. Scalability bugs are hard to found. Developers might assume their protocols are scalable, but in certain deployment and environment, they do not. Thus, they depend on so many factors.

Unfortunately, many scalability bugs are only found in deployment, which is undesirable as it impacts users. In one case of scalability bugs in deployment, a user changes its system and users another system [**?**]. In recent work, Azure executives also create a big that we should find bugs in testing not in production.

Unfortunately today, the deployment practice has outpaced exsitign tools in checking their reliability, especially the scalability factor. There is not many approaches in detecting scalalabilit bugs prior to production or reproducing it in simple way. As an example, there is a Cassandra scalabilty bug, where a customer bootstrap a ring of 1000 nodes with 32 vnodes/machine and observe flapping, unstable cluster. Here, the developers don't have access to 1000 nodes and must change some configurations so that they can reproduce the problem in 100 nodes but with 1000 vnodes/machine. Luckily, they can reproduce it. In another case [**?**], developers believe that there should not be performance problems, but there is a customer who can only run the system on community machines with less compute power and thus exhibiting scalability bugs [**?**].

Put simply, scale testing is hard. Users sometime run larger scale than developers had tested. The most common approach is to use a "mini cluster" (*e.g.*, Facebook uses $\mathbf{xx}_1$ nodes, Skype uses $\mathbf{xx}_2$ nodes, more in §**??**) orders of magnitude smaller than real possible scale. But again, not all developers have access to large scale machines, as it is expensive. Start-up companies might not have the luxury of this "mini cluster". In our work, we attempt to break this barrier.

We present SLCK methodology, wherein we *enable developers to scale-check their system in a single machine*, thus revealing latent scalability bugs prior to deployment. Developers might have a question, "Does this algorithm/design/protocol really scale?". SLCK helps address this. We target actual implementation, not model. Just like fsck exists for every file system, we believe SLCK methodology should be integrated to every cloud-

scale distributed system.

The key challenge of SLCK is how many nodes can we colocate on a single machine without sacrificing accuracy. There are a few existing work that tackles this, but they focus on data-intensive tasks (*e.g.*, by compressing users data). With compression they can colocate 100 datanodes per machine and skips the I/O bottleneck, but doesn't fake out the compute time. Mostly focus on throughput. Other work also uses modeling and simulation to predict the throughput of their applications as they scale, but this line of work does not test the implementation. We provide more details later in Section **??**.

In short, most recent literature focuses on the throughput of data-plane. In this work, we develop SLCK methodology that focuses on control-plane scalability bugs. Control plane means **xx**$_3$. Control plane is overlooked, but they cause many scalability problems. They are rarely tested. But their operations become common in elastic world, scaling out and down.

The challenge of control-plane is that their scalability bugs can be exhibited by expensive metadata computation in every single node that in aggregate cause cluster instability. Thus, the question is: *how can we co-locate nodes in a single machine with limited CPU resources when each node is CPU intensive, but not sacrificing time?* For example, if we colocate 1000 nodes in a single machine, we don't want to see reduced accuracy and we do not want to see 1000x longer testing time. This question has never been addressed before.

Fortunately, from our careful study of **xx**$_4$ scalability bugs, we find that the individual expensive computation can be replaced with a "sleep" *without* changing the whole-cluster logic. This is what we call as *compute illusion*. This technique works as we address the following important questions in this work: Which functions can be replaced with with sleep()? and How is it possible that we can replace a function with sleep() without changing the cluster semantic/logic? How do we know accurately the compute time? How do we know the output if we skip the actual compute? Which functions should we replace with sleep()? With compute illusion, we can increase colocation factor to **xx**$_5$.

On top of this, we find that for SLCK, we find further colocation optimization opportunties that fit for offline SLCK: single process cluster (SPC), which **xx**$_6$ global even driven architecture (GEDA), which **xx**$_7$, network by pass, which **xx**$_8$, and lazy metadata allocations (LMA), which **xx**$_9$. Combined with everything, we can achieve 1000x colocation factor without sacrificing accuracy.

To show the generality of our methodoloyg, we have integrated SLCK to 3 systems: Cassandra, Riak and Voldemort. We have reproduced **xx**$_{10}$ old scalability bugs in the control plane. We are also working with the developers to integrate SLCK to find new bugs that they are still hunting.

In summary our contributions are: (1) in-depth study of control-plane scalability bugs in modern P2P systems and their characteristics, (2) develop a method, compute illusion, that allows nodes with CPU bottleneck to be colocated and does not affect testing time, (3) develop other SLCK optimization opportunites that can give colocation factor to 1000x, the highest among related work, (4) integration to 3 real systems.

2 Scalability Bugs

Scalability bugs are latent bugs that are scale-dependent. They don't arise in small-scale deployment but arise in large-scale production runs. Scalability bugs will show symptoms as illustrated in Figure **??**. A systems with good scalability factor either scale linearly as the system scales or stay at constant rate, while a scalability bug will break this behavior.

In this section, we give a background of scalability bugs (§**??**) a sample of control-plane scalability bug (§**??**), and its general characteristics (§**??**), and a summary of existing work that addresses scalability bugs (§**??**).

2.1 Taxonomy of Scalability Bugs

Cloud Bug Study (CBS) project lays out scalability bugs that were reported throughout the development of six cloud-scale distributed systems (§3.3 in [**?**]). The authors breakdown scalability issues into four axes of scale: cluster size, data size, load, and failure. For example, O(n3) computation that causes problems when the cluster size reaches 300 nodes (cluster-scale dependent); opening one big table with 100K regions takes an unexpectedly long time (data-scale dependent); a stampede of concurrent lease recoveries of 100,000 files cause HDFS heartbeast stop (load-scale dependent); and recovering 16,000 concurrent map failures take 7 hours (failure-scale dependent).

These examples show that finding potential scalability bugs prior to deployment is important. The different forms of scalability bugs above call for various innovations in this new untapped area.

In this work we focus on cluster-size dependent scalability bugs (or *cluster-scale bugs* in short). We analyze cluster-scale bugs in CBS data, and we further catego-

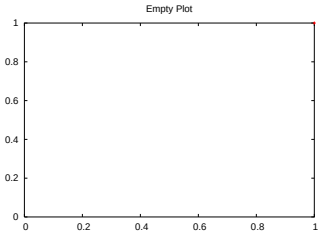

Figure 1: **Bug Cass1.** *This is a sample*

rize them into two types: data- and control-plane bugs. *Data-plane bugs* means the scalability bugs exists in the data management protocols. For example, bandwidth of HDFS append and write requests that plateau when datanode scale to thousands of nodes [**?**]. Data-plane bugs are typically depicted in Figure **??**a, that ideally the system throuthput should scale linearly with the cluster size; if it drops, then there is a data-plane related scalability bugs. *Control-plane bugs* linger in the cluster management operational protocols that manage the cluster metadata. In each distributed system, there are metadata that describes the cluster (P2P node key-range, ...). These metadata changes when cluster operational protocols are trigerred (*e.g.*, adding nodes, removing nodes, data rebalancing, ???). We will give a sample in the next section.

The reason why we distinguish data- and control-plane bugs are two-fold. First, we see that *all* existing work that we are aware of focuses on finding data-plane bugs (*e.g.*, ensure throughput linearly scales) (§**??**). However, *control-plane scalability bugs are overlooked*; we find from the CBS data, there are plenty of intricate control-plane bugs without clear solution on how to reveal them. Second, control-plane bugs are important in today's era of elastic cloud. Rebooting large cluster, scaling out/down cluster, and rebalancing, become common operations. These operations are common as cloud systems elastically scale. It has been highlighted that bugs live in under-tested operational protocols [**?**, **?**]. For these reasons, in this work, we focus on building a scale-check tool to reveal control-plane scalability bugs.

2.2 A Control-Plane Scalability Bug

In this section we describe a contro-plane scalability bug in Cassandra (ca6127). We will use this bug throughput the paper, hence the detailed description.

The bug was revealed when a customer bootstrapped 1024 Cassandra physical nodes (with 32 virtual nodes per physical node [**?**]) and notice that the cluster was unstable, specifically "flapping", a condition where nodes continuously switch from dead and live mode. This implies something wrong with the failure detection logic as healthy nodes are incorrectly considered dead (false positives). To understand this bug, we need to understand the boostrap, gossip, and failure detection protocols.

The protocols: (a) bootstrap: To enter Cassandra ring, a node runs the *bootstrap* protocol. Each node first creates virtual nodes (*e.g.*, 256 vnodes is the default) and assign to each vnode a random key the vnode is reponsible for (random number from 2^{64} keyspace). It then via gossipping informs itself to the *seed* node (*e.g.*, node1; as the booting node does not know anyone else). If the seed node has observed other booting nodes, the seed node will reply the gossip and tell the sender node about other nodes. **(b) gossip:** Every second, each node picks one random node that it already knows and send a gossip message. The gossip message contains a list of nodes it knows and their *versions* along with its own version number. Before sending the gossip, each node increases its *version* number by one, acting as a monotonically-increasing heartbeat number. Upon receiving a gossip, the receiving node checks if there is a new node in the gossip that it doesn't know, and if so, asks the sender for more information. The receiving node also compares the peer nodes' versions it receives with the ones it has, and if there are any differences both the sender and receiver synchronizes the metadata differences. In this protocol the most important element with respect to scalability bug we'll show is the *commit-state processing time*. In the gossip message, there is also peer nodes' commit state. The commit state will tell the receiver node that a particular node will take care certain key ranges in the key ring and the receiver needs to insert those key ranges into its current key ring. **(c) failure detection:** Every second, another background thread runs the Φ accrual failure detector [**?**]. Put simply, if a node X has not received a gossip about Y after a certain window period of time (note that the gossip can come from Z, it doesn't have to come from Y), X will declare node Y as dead (specifically, the Φ value for Y becomes greater than an accepted threshold). If Y is actually alive but declared dead, then it's a false positive.

The bug: The bug in this process is caused by the potentially long commit-state processing time. Again, this process happens when a receiving node hears about new nodes or when existing nodes it knows changes their states significantlly (*e.g.*, from boot to commit???). This processing time depends on the cluster size. The larger the cluster, during bootstrap, each node will hear more about more nodes and need to process them. There are two expensive parts in this commit-state processing. First, in addition of updateing the latest versions in memory, Cassandra node also writes that information to the

disk (Cassandra internal on-disk database using CQL). This is needed for fault-tolerant purposes, that when the node wakes up, it knows the latest states about the peer nodes. Second, give new peer states, it must update its ring table (a MultiMap data structure). *Interestingly*, the *implementation* of this update is expensive and its O(n-log-n) (??). That is, the node creates a new empty table and add the entry one by one. Thus, even when one entry changes, they copy the entire table and perform the update in the cloned table. The reason for this is that there are other user-facing protocols that use the MultiMap ring table, and the commit-state processing does not want to hold a read lock too long.

Given that the commit-processing time is long, we wonder why nodes are flapping, given the fact that the gossip thread runs independently every second and keep broadcasting new versions. Upon further inspection, we found out that gossip processing is single threaded and cannot run in parallel processing multiple gossips concurrently (for safety reasons, to prevent concurrency bugs on complex cluster metadata). This problem is illustrated in Figure **??**. All the new gossips with new versions are backlogged. For example, its possible say V tells X that Im alive for 5 seconds (Vs version already increases by 5 times), but X gossips to other nodes about V using an old version timestamp. Some other nodes eventually will declare V dead. In other words, the job of X is to forward Im alive gossip from other nodes (via version timestamp), but because X is so busy doing state-diff processing in the single-threaded and all the gossips are backlogged, X doesnt forward the new timestamps to other nodes. Other nodes eventually see this dead. Thus the busyness of the nodes prevent them to forward gossips with new versions.

As an implication, nodes see each others with having old versions without recent updates. It's like "I haven't heard new gossip about Z, and therefore I declare Z is dead". In 1024 nodes bootstrapping, there are **xx**$_{11}$ flapping scenarios (where a node is declared dead from alive). In this deployment, gossip processing time can take from **xx**$_{12}$-**xx**$_{13}$ seconds.

In summary, so when the cluster is big, we will see more state differences and hence longer state-diff processing time. This problem doesn't appear in smaller cluster because when the cluster is small, the metadata size is small and the processing time is fast.

2.3 Bug Characteristics

From the bug above (and other **xx**$_{14}$ bugs that we study, we take several important points.

Only appear at large scale. As mentioned before, the bug was revealed when a customer deploys it on 1000 nodes. Later we show that the bug does not significantly appear even in 100 node deployment.

Scalable in theory, but not in practice. The most interesting part is that accrual failure detector was adopted by Cassandra because it is theoretically "scalable" [**?**] as it assumes that "The time for propagation of a message is typically much shorter than the length of these intervals." [**?**]. However, it does not account the gossip processing time (*e.g.*, it assumes gossip processing time is fast) and hence ignore backlog cases. In practice, such as in Cassandra, gossip is not use for failure detection only, but also for announcing node state changes (startup, normal, etc.), hence gossip processing time can vary beyond the theoretical description.

Deeply hidden and hard to predict. Another way to look at the bug is that it is deeply hidden. As mentioned before, we were confused why new versions are not forwarded in time given the fact gossiper still send gossips every second. It turns out the bug was caused due to single-threaded design of gossiping processing time (to prevent concurrency bug) that eventually causes backlog. The gossip processing time itself is hard to predict (as we show later in evaluation) as it depends on the current node's partition table size and the number of new peer commit states (two-dimensional inputs). Gossip processing time varies from as low as x second to x seconds with unclear distribution.

Design and implementation specific. Continuing the above statement, the bug above was exhibited because of the decision to write state updates to disk (for fault tolerance) and perform the slow copy (for simplicity of implementation). Thus, different implementations of similar algorithms might show different results. Later, we show that in our experience in dealing with three systems (Cass, Riak, and Voldemort), all of them are essentially same class of systems but their scalability bugs appear in different forms. AGain this is the case of no-one size fits all system.

Control plane: not-so-independent nodes. Another lessons is that when we talked about control plane (especially in ring-based systems), the nodes in the cluster are not entirely independent with each other. They must communicate with each other in completing cluster-wide operational protocols, and this is where scalabilty bugs linger. As cluster grows, the cluster metadata size also grows. While data is disjoint, cluster metadata must be synchronized. This means that the individual per-node compute and I/O time can have cascading impact.

Major implications. Scalability bugs lead to undesirable behaviors. In this example above, we show a case of

flapping; there are many other cases that lead to flapping. Flapping in general is bad because if a node declared dead while it's alive, it will lead to hinted handoff as data for the "Dead" node is routed to somewhere else, and when the "dead" nodes are observed as live, it creates a flood of hinted handoff. In another case, for writes, if there is a dead nodes, and consitency level is high (e.g. quorum or all consitency), writes could be rejected, although it's actually possible to write.

Repeated bugs. We see similarly repeated bugs as the design of the systems evolve. The bugs involve operational protoclls and flapping. The patch for this bug was to disable failure detection of a peer node until 40 gossips for the peer node have been received. This patch is very *specific* to make sure we relax Phi during reboot time. Interestingly, this patch doesn't fix other scenarios where for example where we already have 1000 nodes and would like to scale-out by adding 1000 new more nodes. Flapping will be observed on the old 100 nodes. In fact, the fix does not appear in the latest version of Cassandra where they ??? (what kind of algorithm they do). A fix is obsolete as new design comes in. Another example is bug fix for ca3831 is obsolete as the vnode is introduced in subsequent version, causing another bug ca3881 to reveal. These suggests that SLCK availability will make sure a fix is applicable to different workloads and as the design of the system evolves, they can make sure the old bugs do not re-surface in a different way.

2.4 State of the Art

We now briefly discuss state-of-the-art tools to catch scalability bug. Detailed comparisons are provided in related work section (§**??**). We divide by extrapolation, simulation and emulation.

Extrapolation of scalability problems can be made by running a "mini cluster". For example, Facebook runs xx machines, Skype runs xx machines [**?**, **?**]. However, they are orders of magnitude than their real deployments. Second, start-ups might not have the luxury of having that mini cluster. Other work has shown deficiencies of exrapolation such as the fundemental assumption that resource consumption increases linearly with the load and the number of machines in the system does not always hold (§2.1 in [**?**]). (Milind stuffs are here too [**?**]). Hourly rental can also be expensive. 1000 instances in 1 hour cheapest can burn $100??

Simulation is based on model. This is common in the HPC community [**?**, **?**]. The drawback is that they do not reflect the real code, and as previous code suggests, real code can be much more complex with real concerns. Previous section also highlights why theoretical assumption does not hold in practice.

Emulation means, we can test the real code on large-scale environments but by changing the environments. We are only aware two major approaches. The first one is DieCast [**?**] that builds up on *time dilation* [**?**]. Time dilation factor (TDF) of 10 means that for every second of real time, all software in a dilated frame believes that time has advanced by only 100 ms. With TDF=10, DieCast can colocate 10 CPU-intensive processes on 1 physical machine as if they run individually without CPU contention. The downside is that if we subject a target system to an one-hour workload when scaling the system by a factor of 10, the test would take 10 hours of real time. Thus, high TDF (or colocation factor) is unreasonable. Time dilation was first created to ask what would happen to overall systems performance when we upgrade the network capacity (e.g. to from 100 Mbps to 1 GigE switch) without having the networking hardware. The approach thus slows down the application such that fooling the apps they see faster network. In this case, slowing down applications make sense, but not make sense in the context of scalability check.

The second one is Exalt which uses novel data compression. It's insight is that how data is processed is not affected by the content of the data being written, but only by its size. Thus users' data is compressed to nothing. Metadata is not compressed. Exalt is primarily designed to evalute I/O-intensive applications; the can co-locate 100 emulated datanodes on one machine. Most of the bugs that they find mainly in the HDFS namenode which runs exclusively on one machine. As the authors claim, a downside of this methodology is that "it may not discover salability problems that arise at the nodes that are being emulated" (the datanodes) (§4.1 in [**?**]). Thus, Exalt doesn't fit P2P systems such as Cassandra, Riak and Voldemort.

P2P systems [**?**].

In summary, we do not find a fast method to unearth scalability bugs in the control-plane of distributed systems. All the control-plane bugs that we study that lead to scalability problems are due to long individual computation time. DieCast+ModelNet targest network emulation (by sacrificing testing time). Exalt targets is best use for I/O emulation. We believe we need an approach to target CPU emulation. We believe we complete the missing piece.

3 SLCK

We now present SLCK. We first recap our goals and provide an overview. Then we present two core components

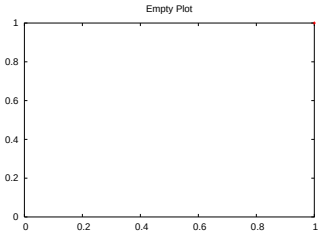

Figure 2: **Overview.** *This is a sample*

of SLCK: compute illusion (CI) and other supports for increasing colocation factor.

3.1 Overview

3.1.1 Goals and Benefits

We want to test the real code, not model. As other work focus on data-plane, our work focuses on the overlooked reliability of the control-plane. We want to do so in a cheap manner. We suspect access to large cluster is scarce and only given to a limited number of people. Cheap solution cheap, does not need to purchase or rent a large cluster. enable more developers to scale check their systems (just like how any developer can run unit test). Because it's a cheap solution, developers when trying new design or new patches, they can test that the new code does not break the scalability of other parts of the system. Thus similary scalability problems do not repeat. A single-machine approach also helps cluster debugging as scalability bugs can be observed globally, not individually (*e.g.*, all nodes observe flapping). We do not present a general framework, but rather a general methodology applicable to different systems. Later we see to tackle control-plane, the methodology must be applied in a specific manner to the target system. We want to achieve a high degree of colocation factory. While 100s of nodes deployment is the most popular. Thousands of nodes deployment also exist. To achieve all of the above, we essentially ask: how to re-architect distributed systems single-machine scale-checkable? That is, we do not shy away to radically change the systems design yet still ensuring the logic is not modified.

3.1.2 Design Overview

Figure 2 shows our SLCK methodology. Emulated node means it runs the actual code where some part of the environment (network message, compute, ..) has been emulated by another thing, but does *not* change the logic of the system that is being tested. An emulated node can be a process, thread, or essentially becomes a function. .... need to mention *emulated node*, but the most important thing is that the logic stays the same!!! Single-process cluster (SPC), Lazy metadata allocation (LMA), Network bypass (NB), and Global Event Driven Architecture (GEDA).

3.2 Compute Illussion (CI)

We introduced compute illusion with the following insights. First, we see that control-plane bugs are not data intensive (not caused by congestion). Heavy CPU computation is the one that creates cascading problem. From other work we learned that DieCast emulates future fast network by time dilation (using current network by slowing down processes) and Exalt emulates data-intensive operation with compressed data. Exalt's insiight is that how data is processed is not affected by the content of the data being written, but only by its size. In control-plane bugs, we find that how compute is processed does not depend on the intermediate computation, but only by its time and output. Thus, the only way to emulate CPU time with colocation without extending testing time is by *replacing compute with sleep()*. Thus, we emulate the compute time, but does *not* use the scarce single-machine CPU resources under colocation.

Compute illustion is the only way. Existing work does not solve CPU time. DieCast doesn't fit for control-plane scalability bugs because customers usually already complain that scalability bugs perceived symptoms cause something looks slow or unstable. If we need need to prolong testing time by orders of magnitude to see the slow things, it's undesirable. For example, in cassandra bug1, it takes $\mathbf{xx}_{15}$ minutes, with 1000 node colocation factor it will be 5000 minutes. This is doable, if the following questions are addressed.

This simple but outrageous idea raises further questions and challenges which we will address next: §**??**: Which functions can be replaced with with sleep()? and How is it possible that we can replace a function with sleep() without changing the cluster semantic/logic? §**??**: How do we know accurately the compute time? §**??**: How do we know the output if we skip the actual compute? §**??**: Which functions should we replace with sleep()?

3.2.1 Sleep-replacable Functions

The first question we ask is which function(s) should be replaced with sleep()? Definitely we cannot replace too many functions with sleep() because some logic must run for the cluster to operates. The goal of this step is to find if there's any offending function that we can replace with

sleep(). Again, our insights from studying **xx**$_{16}$ control-plane bugs is that there are culprit functions with long execution time that can be replaced with sleep() and yet still reveal the scalability bugs. We call this sleep-replacable functions. At the minimum, these functions should have the following characteristics: (a) the function should not perform external communications, (b) "stateless" – it has clear input and output and the intermediate state changes are irrelavant as the final output is the most important.

We find that there are three types of functions that are sleep-replacable but yet does not change the global semantic such that the scalability bugs are still revealable.

Local I/O execution: If the execution is long due to local I/O, then the local I/O time can be replaced with sleep(). For example, in ca6127, one of the long execution time is due to write to the disk for every state-diff item, which is important in live run for fault-tolerance (when the node reboots, it can know the latest metadata state), but it is not necessary executed to reveal the bugs. There is no computation output in this example. Thus, they can be easily replaced with sleep().

Skipped-safe execution: This is the type of compute whose output doesn't change the correctness of the protocol being tested. An example, is r??. Here, one of the long execution comes from an expensive (triple loop?) computation whose goal is to decide which part of the data to sent to other nodes for rebalancing purposes. In all the control-plane bugs that we study, data transfer (*e.g.*, from rebalancing) is *not* the bottleneck, which is expected as they run in the background and the transfers are independent of each other. Thus, this long execution can also be replaced with sleep() as the intermediate results of the computation (*e.g.*, which data should be transferred) are not needed to reveal the scalability bugs.

Memoizable output: There are functions whose output is needed for the cluster to progress correctly. The functions have memoizable outputs if it has one-on-one relation between the input and the output. If we already store the possible outputs from offline profiling (§**??**), we can change the computation with sleep() and then provides the output given the input.

Manufacturable converging output: If the output is needed and memoization is not possible (*e.g.*, due to insufficient memoization space; §**??**), then we check if the output can be manufactured with some domain specific knowledge. For example, in r??, ... **xx**$_{17}$ **(HSG: Is this still applicable??)**. We detail this example later int he evaluation (§**??**). This is very domain specific because the output must be make sure that it is converging (*e.g.*, node rebalance eventually is stable). The concept is similar to failure-oblivious computing (FOC) [**?**].

3.2.2 Time Profiling

Given a sleep-replacable function, how long is the sleep time? The answer depends on what kind of input the function observes in real run. Therefore we need to profile for each possible input the actual compute time. The developers must create profiling runs on different input size on a single machine.

The profiling runs depend on the input dependencies. The easiest is for simple functions whose content is a simple for loop. Here, typically we just need to construct input data structures that represent cluster size ranging from say 10 to 1000, and measure the function latency.

However, there are complex functions with multiple for loops and many if-else statements. Here, it is hard to predict the compute time as it depends on different types of inputs. The input in this case is a multi-dimensional space. For example, the input lists 800 nodes (gossip from a peer node) but it will be compared with 700 nodes that the receiving node has. In the Cassandra, case the compute time depends on two things: the number of new commits and the size of its current partition table. Thus, we need to profile the compute time for all possible input in this two-dimensional space (*e.g.*, what's the compute time if I hear about 124 new states and my current table is 87 partitions).

As we scale-check the system at a very large scale (*e.g.*, 1000 nodes), in the Cassandra case, profiling NxM input space will take **xx**$_{18}$ hours. In this case, sampling is needed. We use uniform sampling across the desired range (*e.g.*, 1 to 1000). With sampling every N items, we can reduce the profiling time N times.

Given this technique, there are two questions raised: First, is the compute time obvious by statically analyzing the function? The answer is not necessary. In our experience, compute time also depends on the machine CPU power (and storage latency). Although most cases because single-core frequency hits a wall limit, most compute time is the same. But we have observed cases where customers deploy their cluster on public machines (riak??) whose compute power is weak and scalability bugs appear even in a smaller scale. That's why the profiling time must be based on machine that is similar in deployed machine. Put simply, every deployment is different.

The second question: is it obvious from profiling that if a function is expensive (e.g. O(n2)), then it will lead to scalability problems? Not necessarily. As earlier section suggests, each implementation is unique. The scalability bug in Cassandra happens because the gossip processing is single-threaded and blocked. A different more complex implementation can make it multi-threaded that the

processing of multiple gossips run in parallel and in the background. Or, make the new versions of new gossips to be forwarded (a special code path). But Cassandra decides to keep the simplicity of other functions, and the fix is to reduce the O complexity of the function. In short, what we say is, the root cause of scalbility bugs is not merely because of the expensive function, but rather the global impact of the slow function as it runs on each node. It is about the global interaction and the potential cascading impacts of the expensive functions. Thus, testing the whole cluster is needed. In other cases, some developers might also say let's only support systems up to 100 nodes. so they just want to scalability up to 100 nodes (e.g. NetApp filers are built deep but not scale out), and if that's the case doesn't have to redo their algorithms.

We want to emphasize that profiling is not the same as extrapolation. In extrapolation, one stops at a certain scale (*e.g.*, 100) and extrapolate the behavior in larger scale (*e.g.*, 1000). Here, because we focus on potentially offending functions, we can profile that offending functions on a single machine using the fully desired range (*e.g.*, input size representing 1000 nodes).

3.2.3 Output Profiling (Memoization)

When the output of the computation is needed but we want to skip the computation, it means we must prepare the output given an input. Thus, for these functions, we perform output profiling (memoization). One challenge here is that, we cannot sample. For every possible input, we must know the output. Otherwise, in real runs, we cannot manufacture the output. The problem is the space of possible input at scale is big. In our experience, for output-needed functions, the input space is NxM, with each N is a different sorted list and M is a different sorted list, e.g. partition-to-node mapping. In total, we find that there are 2-to-the-???. Definitely DRAM space is not enough. We need SSD (how big???). And then it must support N IOPS (where N is the scale of the system) as these N nodes in real run will fetch the output concurrently on a single machine. Thus disk (max of 200 IOPS) is out of the picture. Thus, SLCK requires ??-TB SSD.

The time it takes to do output profiling (memoization) is $\mathbf{xx}_{19}$ (hours), but again is doable on a single machine and only a one-time overhead. We find that in Riak case that with memoization we can increase co-location factor by $\mathbf{xx}_{20}$. This is because we change a compute time ranging from $\mathbf{xx}_{21}$ to $\mathbf{xx}_{22}$ seconds with a memoized output table lookup which is just $\mathbf{xx}_{23}$ us with SSD.

a) #flaps = large Φ
b) Φ = f($T_{hbsilence}$, $T_{avghbperiod}$)
c) $T_{hbsilence}$ = g(hops, $T_{gossipexec}$)
d) $T_{avghbperiod}$ = h(all previous $T_{hbsilence}$)
hop = i(n) = log(n) on average
$T_{gossipexec}$ = k(partitionTable, commitStates, hardware)
partitionTable <= n
commitStates <= n

Figure 3: **Cass1.** *Φ The probability that observer thinks the test node dead $T_{hbsilence}$, Time period that observer has not receive hb containing test node info $T_{avghbperiod}$, The average $T_{hbsilence}$ in the past Hop, The number of hops heartbet got forwarded until it reached a ceratain node $T_{gossipexec}$, Gossip processing time in all nodes in $path_{propagation}$ See korn-sc.tex for descriptions*

3.2.4 Finding Offending Functions

The final question we ask is how to find such functions? Definitely we don't want to search all sleep-replacable functions and then profile them offline. Thus, we want to find only potentially offending functions that could impact scalability. These functions must have one characteristic: they are part of the global communication.

Thus, to find that function we perform reverse algorithm derivation (??). We begin with a high-level expecation of the cluster behavior (*i.e.*, the global correctness specification). In this example, we use Cassandra as an example, as summarized in Figure 3.

We start with the highest-level expectation that, a the cluster scales out (or bootstrap), there should not be any cluster-wide flapping. Or in fact, if there is no dead nodes, there is no flapping. Flapping means some nodes are declared dead. A node is declared down depending on the Φ value. The Φ value of a node Y depends on the the fact that node X has not received any new gossip about node Y (Y with new versions). We call this heartbeat silence. (Note again Y doesn't have to gossip X directly, Y can gossip via other nodes to X). Thus, Φ value depends on hbsilence and avghbperiod. The length of hbsilence (which is the distance between two different gossips regarding the same node Y depends on the number of hops and the length of each gossip processing time in each hop. The number of hop is shown by many papers for ring systems is log(n) on average, and this is a known factor that impacts the time and can be estimated algorithmitically. The gossip processing time on the other hand depends on the partitionTable, commitState, and the hardware capacity (CPU speed). Then partition table and commit state depends on the size of the cluster and their compute time cannot be estimated manually.

By performing this reverse analysis, we can figure out

that the time of gossip processing execution is a good target function. For other protocols and other systems, we will show similar reverse analysis in Section **??**. Other examples of offending functions that are sleep-replacable are: **xx**$_{24}$

We would like to note that such reverse analysis is different than failure detection algorithms that are typically presented in literature. Those algorithms are presented by saying *when* a node should be declared dead (*e.g.*, as Φ value is larger than). However, this algorithm reverse analysis forces us to think *why* a node can be declared dead accidentally. In literature, fialure detectio algoritms also typically claim the network delay as the culprit. This reverse analysis on the other hand shows that it can be caused by specific functions that delay new gossip to be sent.

As discussed before, everybody right their own formula. So although some papers exist [**?**], for every implementation developers must be able to breakdown the formula. Thus, this process must be a manual process.

3.3 Colocation Optimizations

Compute illusion provides the greatest benefit for colocating a large number of nodes in a single machine. Later we show that with compute illusion we can colocate up to **xx**$_{25}$ nodes per machine. However, we want to go more. thousands of nodes deployment is common today. We find that there are many other unnecessary resource utilization that can be removed but still doesn't break the logic. We now present four methodologies for optmizing colocation. Specifically, we shor further how we optimize resource usage, memory, network and CPU usage.

3.3.1 Single Process Cluster (SPC)

Our first obvious method is that we do not need to run each node in each of process. Thus, we change our target systems so that it can run all the nodes within a process, which we call as single process cluster (SPC). This is important because a process has context switch overhead and also high memory overhead. For example, Cassandra which uses Java has **xx**$_{26}$ KB/MB memory overhead for the JVM.

To support SPC, we need to redesign the per-node global data structure into array of global data structures. We also need to **xx**$_{27}$.

For Riak, this becomes interesting. Riak runs as an Erlang process. The challenge is **xx**$_{28}$.

With SPC, we can increase our colocation factor by **xx**$_{29}$.

3.3.2 Global Event Driven Architecture (GEDA)

After SPC, we still find that thread context switching is still the bottleneck. Again, with the goal of 1000 colocation factor, thousands of thread context switching is big per iteration. For example, a Cassandra node by default runs **xx**$_{30}$ threads such as send-gossip, receive-gossip, fault detector, and many others. At least these first three threads must exist, giving us a total of 3000 threads to run.

To address this, we take advantage of the SEDA (event-driven architecture) [**?**] of our target systems and convert it into global event driven architecture (GEDA). With SEDA (and SPC), each node exclusively has an event queue (*e.g.*, and its own handler threads). With GEDA, the *whole cluster* (all nodes) share *one* queue and the *same* handler threads. To give an example, let's imagine a gossip sending thread. With 1000 nodes, there are 1000 gossip threads initially, each sending a gossip every second. With GEDA, we only have a few (1-4, to exploit multicore) threads that will send out the gossips to other nodes on behalf of each node, ensuring each node only sends gossip every second. Thus, we remove context switching that should never existed on the first place. We reduce the number of threads by orders of magnitude *without* changing the processing logic. Note that in real deployment runs, our target systems still use SEDA. But in SLCK mode, it will run with GEDA. With GEDA, we can increase our colocation factor by **xx**$_{31}$ without sacrificing accuracy (there is no late gossip/event sent).

With GEDA, we cannot use sleep anymore. A compute event becomes an event that we put in the queue and picks up later.

3.3.3 Network Bypass (NB)

Finally, we see that certain event lateness is not unnecessary, as we go to the OS level to send the network and go back up to the application. This causes many unnecessary user-kernel context switching, and the kernel becomes the bottleneck. Thus, we create network bypass (NB) by creating a shim layer that skips network call to the OS whenever the system is run under SLCK mode. Although this not a major bottleneck for control-plane, we can increase the colocation factor slightly by **xx**$_{32}$.

What we find interesting is that we initially thoguht NB will speed up a little. But when every small optimization will have 1000x impact, then that small optimization is worth it.

3.3.4 Lazy Metadata Allocation (LMA)

The next bottleneck we see is that in the beginning each node must run the "init()" function which allocates all the metadata in memory. For example, in Cassandra it will allocate memtable, **xx**$_{33}$. However see that in testing scalabiltiy of a certain protocol, the code does not touch all the data structures (*e.g.*, gossip and failure detector only need **xx**$_{34}$ data structures).

Thus to prevent unnecessary data structure allocation of metadata that not being used for the protocols being scale-checked, the metadata will not be created until the service is called. We call this as lazy metadata allocation. For example, when the gossip function (function under test), if it only uses X, then SLCK will eventually create 100 of Xs to test 100-node deployment.

As an illustration, in Figure **??**, we represent these variaty of data structures with X, Y, Z. If we run the default initialization function ("init()"), then it will create all X, Y, Z, wasting memory space. ○, ●

To implement this, all data structures are encasuplated in the root data structure. When asking for a data structure it calls for rootDS.getGossipDS(), which will check if gossipDS already exists or not. In our experience, data structure init functiosn are independent to each other as they all depend on configuration files. Thus this laxy DS allocation is possible.

LMA increase colocotation factory by **xx**$_{35}$.

3.4 Discussions

We now discuss the pros and cons of our approach, with respect to efficiency, generality and orthogonality.

The primary time cost comes from profiling and memoizing time. Because the premise is that the developers only need to use one machine, profiling time can be long. However, profiling can be run once and just a one-time overhead, as long as the function stays the same. Thus the same profiling results can be used multiple times to test different workloads (cluster bootstrap, add-node, etc.).

Because also the premise of only using one machine, we can only test the scalability up to the maximum colocation factor and dependent on the power of the machine. If we can extend into a hybrid architecture that can work few multiple machines (*e.g.*, 10), we can increase the scalability test limit.

In terms of generality, again we are proposing a methodology, not a general framework. This is concious decision because we find it's almost impossible to scale-check control plane without system-specific integration (akin to fsck in checking metadata integrity), especially when they are built with different languages and environments (Erlang, Java, Scala, etc.). The way we show our generality is by integrating our approach to three systems, and so far we target P2P systems. Also, it is possible our SLCK methodologies not applicable to some control-plane scalability bugs. But, by experience of **xx**$_{36}$ bugs we study, that all the methods will work find. At least, we are the first one that study the control-plane bugs in detail.

In temrs of orthogonality, as we mentioned earlier, DieCast solves network emulation problem by combining hypervisor time dilation with network simulation (dummynet, ModelNet) to simulate fast network and not congesting the single host OS and device. Exalt solves the data space and transfer emulation problem by compressing users' data. SLCK solves the expensive compute problem with compute illusion. We believe our approach is orthgonal to the others. So far, we also emulate the data transfer time with delay. But with integration like in DieCast and Exalt, we can remove this emulation of data transfer.

4 Implementation

We apply our methodology to three systems. The resulting systems are called: SLCK^C, SLCK^R, SLCK^V.

The LOC are: **xx**$_{37}$, **xx**$_{38}$, and **xx**$_{39}$ LOC.

Note that compare to their unit test size which is **xx**$_{40}$, **xx**$_{41}$, **xx**$_{42}$, thse methodology only adds **xx**$_{43}$- **xx**$_{44}$ % code size and be able to find new types of bugs.

Our target systems can run in two modes, normal modes and SLCK mode.

Just like fsck that contains multiple phases [**?**, **?**], SLCK also contains a bunch of subtests. Each test is basically includes a description of the workload taht will exercise subpart of the services, and also some pre-initial state before running the workload. For example, for ca3831, we are decommissioning a node from stable cluster, we must bootstrap the cluster first. And it requires some keyspaces to exists, so we insert some keyspaces before running the decomission test. Currently we fake the data manually, but if SLCK is integrated with Exalt, faking data content can be done automatically.

We also add specifications of correctness behavior, and global monitoring hooked into each emulated node.

After all of these changes, the distributed systems still run normally without any overhead.

Also we need to specify the profiling tests and memoization test.

5 Evaluation

new bugs? new version!! deploy to new version and double check if there is new bug or not?

although there is no new bug. our methology is crucial. as changes to the gossip management.

how many times gossip file has been changed? how many patches? in the last one year?

5.1 Colocation Factor

We now measure the colocation factor. The colocation factor hits a limit when either one of the following conditions happen: CPU is constantly 100%, memory consumption reaches 100%, the accuracy of SLCK drops significantly. We run this on 16-core machine, ?? GB DRAM, ??

Figure 4 shows the improved colocation factor for every feature we add as described in Section **??**. The four figures show the mapping between correlation factor and four metrics: CPU, memory and two types of accuracy factors global status and event lateness.

For global status, we use number of flapping instances (a flap means the transition of a peer node being observed as alive to dead). We can see that overall SLCK is quite accurate, however the larger the cluster is SLCK condition deviates from the actual run. We believe because at scale, latency variance plays into a factor, and thus the actual run will see more delays and more flapping. However, we note the interesting part of unearthing scalability bugs is the pattern. Eventually a scalability bug will appear as we increase the cluster size.

We also use event lateness as the case of accuracy. This is primarily important as we use the GEDA architecture (§**??**). There are two types of event lateness we measure. First, note that each node must send a gossip once every second. However, if SLCK GEDA architecture send a gossip late (*e.g.*, after more than 1 second), then we measure the lateness. Second, each compute illusion also becomes an event as it is put in the queue and will proceed. If this event is late, we measure the lateness.

Figure 4 shows that our SLCK methodology allows us to incrase colocate factor signifaantly in a single machine without hitting CPU and memory bottlenecks and still maintain good accuracy. Overall, SLCK methodology can have a maximum colocation factor of $\mathbf{xx}_{45}$ for Cassandra, $\mathbf{xx}_{46}$ for Riak and $\mathbf{xx}_{47}$ for Voldemort. The numbers are different because their runtime is different (Java, Erlang, vs. ??) and the target code is different. We do not show the detailed figures for Riak and Voldemore due to space constraints.

5.2 Bug Reproducibility

In the rest of this evaluation we will show the $\mathbf{xx}_{48}$ bugs that we have reproduced with detailed evaluation. We have $\mathbf{xx}_{49}$, $\mathbf{xx}_{50}$, and $\mathbf{xx}_{51}$ Cassandra, Riak and Voldemort bugs respectively.

Due to space constraints, we will show the details only for the first Cassandra bug and show the subcomponent accuracy of SLCK in Cassandra. For the rest of the bugs, we will just show the high level results.

5.2.1 Cassandra Bug#1: ca6127

This bug has been described in Sectoin **??** in detail, where a customer complains about flapping when they bootstrap 1024 nodes. Figure 5 shows the breakdown of our measurement for each elements in Figure 3. Our goal is to show how SLCK is relatively accurate and show the same scalability bug pattern.

Figure 5a shows the global behavior which is the number of flapping. As we can see the bold solid and dashed lines show the same pattern. As the nodes scale, more flapping is observed.

Figure 3b shows that flapping is caused by Φ value that becomes big. Figure 5b shows that SLCK is also accurate compared to actual run in both buggy and fixed versions.

To reduce figure complexity, we now only show the buggy versions in SLCK and actual run. Φ is based on the heartbeat receiving period and the average. Figure 5c and 5d shows the whisker plots of this data. Altough SLCK is not exactly the same, again most importnatly is the pattern.

Heartbeat period is based on gossip processing time as shown in Figure 5d. It increases as the cluster increase.

Gossip processing time in actual run is the actual computation, but in SLCK, gossip processing time is replaced with idle wait. Figure 5e shows the comparison, and as discussed our idle wait time is as accurate as the actual computation time.

Gossip processing time depends on the number of commit states and number of partition table size the node sees. So we also measure the numbers and plot the distribution of what we observe in Figure 5f and 5g respectively.

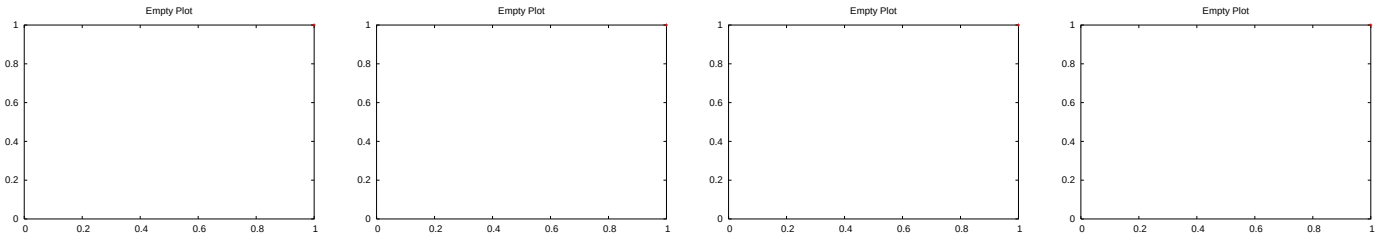

Figure 4: **Colocation Factor.** *Based on CPU, memory, and two types of accuracy, event lateness and global condition*

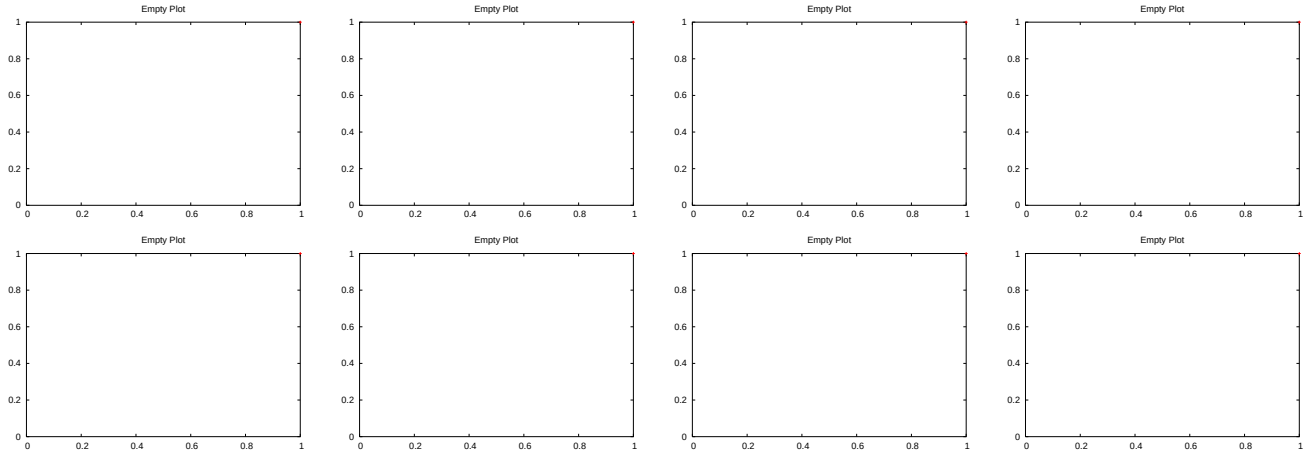

Figure 5: **Cass1.** *Figures: (a) flap, (b) max phi, (c) hb period, (d) avg. hb period, (e) gossip processing time, (f) commit state, (g) partition table size, (h) profiling*

Figure 5h shows our profiling time.

Overall the eight figures show how replacing the sleep-replacable functions with idle wait can lead to accurate patterns in every measured sub-metrics.

5.2.2 Other Bugs

We now describe the other bugs. Figure 6 show the accuracy of SLCK^C, SLCK^R, and SLCK^V in reproducing the rest of the bugs. Overall again they show the same accurate patterns.

The 2nd cassandra bug is about **xx**$_{52}$

The 3rd cassandra bug is about **xx**$_{53}$

The 4th cassandra bug is about **xx**$_{54}$

The Riak bug is about **xx**$_{55}$. We do not skip, but we could memoize and increase the colocation factor.

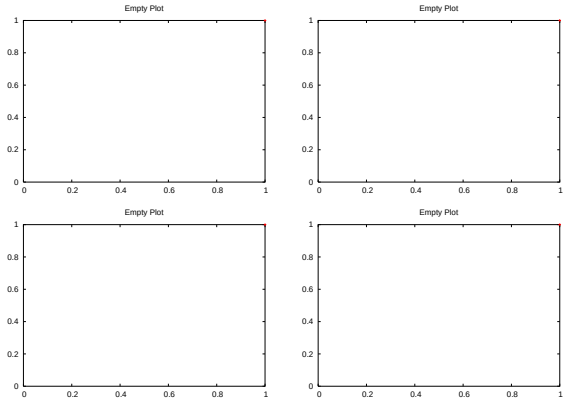

Figure 6: **Other Bugs.** *Four other bugs we show (or just show three is fine)*

5.2.3 New Bugs

(HSG: after content is ready in mid March, send it to Cassandra, Riak, and Voldemort developers, ask if they still have anything new bugs that they are still trying to reproduce.)

5.3 Profiling and Memoization

As mentioned before, our major cost comes from profiling time and memoization.

• **Profiling time:** Our profiling time for testing the gossip processing is $\mathbf{xx}_{56}$.

(Notes: don't mention we profile the time for bug1, we profile gossip processing)

• **Memoization table:** We memoizae for Riak case. The memoization table takes $\mathbf{xx}_{57}$ GB of space because of the range.

In SLCK mode the read time on average is only $\mathbf{xx}_{58}$ us and this replaces the original compute time which is.

Note again we want to mention that this profiling and memoization time is just an one-time overhead. Gossip processing is a function that is exercised for different kind of workload such as bootstrap, add-remove nodes.

Thus, when we run the expeirment by changing the cluster size, we can reuse this profiling and memoization time.

5.4 Other Comparisons

We now compare our results with other work. Although we cannot make an apple-to-apple comparisons, we want to show that our evaluation is at the same level as other related work.

Vrisha? Abhranta? Exalt? SKOPE? DieCast?

• **How big is the Actual cluster size?** What is the maximum number of nodes they run in actual experiments?

• **How big is the emulated cluster size?** Exalt runs 9600 datanodes on 96 physical machines, giving colocation factor of 100 nodes/machine. DieCast evaluation shows 40 node emulation with 10 nodes/machine although they claim they can run 100 nodes/machine, but nobody wants to have 100x longer testing time. Our case, because we remove the major bottleneck with compute illusion, we can reach 1000 nodes/machine.

Vrisha, 16 nodes??

• **How many systems integrated?** Exalt evaluates two systems (HBase and HDFS). DieCast integrates itself to $\mathbf{xx}_{59}$ systems (???). SLCK is integrated to three systems.

• **How many protocols/workloads run tested?**

We test three protocols: bootstrappling, add, remove nodes, ??

• **How many bugs reproduced?** Exalt discuss 6 distinct bugs in total (unclear if they're new or old bugs as they don't specify), most of the bugs are found in the NameNode (non-emulated). As mentioned earlier, Exalt "it may not discover salability problems that arise at the nodes that are being emulated". 5 out of the 6 bugs are in the NameNode. 1 bug in the Datanode (emulated one) but can be found another way by just using 1 datanode. We count the true actual bugs are only 2. The other 4 is about known implication about the design. For example, out of the 4, one is about configuration change (increasing RPC threads of NameNode from 10 to 256 to handle 1000s datanodes), one is about limiting number of files per directory as directory scalability is expected since it uses Array, and the other two are about and changing NameNode logging to disk into tmpfs without sync (which might not be desirable because NameNode logging will be lost when NameNode crashes and debugigng depends on log; a more proper way is to use very fast storage at the NameNode to support 9600 datanodes).

What we are trying to say is SLCK reveals a different types of bugs unrevealable in existing work.

DieCast does not reproduce any bugs.

Vrisha??

We reproduce $\mathbf{xx}_{60}$ old bugs and $\mathbf{xx}_{61}$ new bugs.

So we don't find false positives.

• **What kinds of bugs reproduced?**

Most work also focus on throughput and performance metrics [**?**, **?**, **?**, **?**]: does my throughput scale. Our work highlights that cluster management that involves elasticity is not just about throughput. It is about asking: is my cluster stable? is it flapping? can I scale out and scale down fast?

6 Related Work

Exalt [**?**] provides a compression library that target systems can use, focusing on checking data-intensive protocols. Exalt admits that the cost is CPU utilization (for compressing and decompressing) thus the technique is attractive for storage systems where the co-location bottleneck will be the storage capacity. Exalt only "fakes" the data but not metadata. In our case, to check control-plane bugs we also need to fake the metadata computation. Exalt do not solve the CPU bottleneck problem. It says when testing HBase "region servers are relative more CPU-intensive than DataNOde and therefore cannot benefit as much from colocating multiple nodes".

DieCast [**?**] using time-warped network emulation in added Xen hypervisor [**?**] (as explained in Section **??**). DieCast slows down each process by a constant factor. It compensates for this slow-down by multiplying the measured throughput by the same factor. The goal is to de-

velop a testing methodology and architecture that can accurately predict the behavior of modern network services while employing an order of magnitude less hardware resources. Time dilation was first created to ask what would happen to overall systems performance when we upgrade the network capacity (e.g. to from 100 Mbps to 1 GigE switch) without having the networking hardware. The approach thus slows down the application such that fooling the apps they see faster network. In this case, slowing down applications make sense. As co-location hits the I/O bottleneck they are dependent on disksim and network emulators such as dummynet or modelnet [**?**]. DieCast doesn't fit for control-plane scalability bugs because customers usually already complain that scalability bugs perceived symptoms cause something looks slow or unstable. If we need need to prolong testing time by orders of magnitude to see the slow things, it's undesirable. For example, in cassandra bug1, it takes $\mathbf{xx}_{62}$ minutes, with 1000 node colocation factor it will be 5000 minutes.

Another emulation for network is ModelNet [**?**]. It offer a more realistic alternative to simulation because they support running unmodified applications and operating systems. Unfortunately, such emulation is limited by the capacity of the available physical hardware and hence is often best suited to considering wide-area network conditions (with smaller bisection bandwidths) or smaller system configurations.

Exalt targets storage intensive operation. DieCast and ModelNet targets faster network emulation by slowing down processes. Nothing for CPU-induced scalability bugs. No single fast, suitable solution. Our solution will be orthogonal to Exalt library, and when combined can give powerful functionalities.

doesn't fit P2P systems such as Cassandra, Riak and Voldemort.

ns2 [18], disksim [24], exploring design tradeoffs for exascale system services through simulation [26].

see Vrisha. A popular approach to find bugs is statistical bug detection where abnormal behavior is compared with bug-free behavior. But if this are the bugs scale-dependent then there is no bug-free runs.

[**?**], [**?**]

a statistical approach to detect and localizing scale-dependent bugs. by build models of behavior based on bug-free behavior at small scales. models are constructed using kernel canonical correlation analysis (KCCA). explot scale-determined properties whose values are *predictably* dependent on application scale. But in scale-check, there are algorithms that are dependent on cluster size, but it doesn't mean it will create scalability problems.

Second, the model is built on some parameters such as communication-related parameters such as, volume of communication with each of its neighbors. The model inclures the parameters that should determine the behavior or each process. In cassandra, figuring out which parameters to collect will be challenging. Some bugs don't have linear relationship. Again bug cass1 shows that it doesn't appear much when it's 128 nodes.

Statistical debugging means we must profile online, and runtime profiling can be as high as 8%, especially if we don't know what to profile.

[**?**, **?**] Another reason is like P# where we have an observer node and a test node. But that framework does not test the real system code. For example, We tried that but ... ?? creating the stubs take time, cite P#. Stub also fails (see 2.2 of Exalt).

In HPC, applicaitons are partially responsibiltiy to scalability. In datacenter sofwtare infrastructure, scalability is the responsbility of the service infrastructure.

Thus, they must model the applications.

Other mdoeling: [**?**] [**?**]

check, check, check Distributed systems are hard and error prone. fsck is needed (HDFS fsck). dmck is needed ... [**?**, **?**, 1], and now it's time for more research on slck. fast fsck – need change the architecure. fast dmck – need to more semantic information. fast slck –

Our philosohpy of enabling more developers to scale check their systems is inline with the single-machine machine learning algorithms, which believes that their systems can encourage more users. large-scale graph on just a PC, [**?**]

Verification of distributed systems. But not cover the scalability. Proof. Our work hopefully initiates the verification of scalability bugs. distributed system correctness, proving. [**?**]

Rule cheking incurs high overheads due to state explosion.

scalable scheduling for cloud-scale computing. By design is obvious [**?**, **?**] scalable cluster management [**?**, **?**].

[**?**], [**?**] [**?**]

[**?**, **?**]

[**?**]

isaacs, baumann, peter [**?**], chase, kostic [**?**], chun [**?**], druschel [**?**, **?**], kaminsky [**?**], mickens [**?**],

[**?**]

7 Conclusion

(empty page)

"I hate Riak" "We don't have 1000 machines" ca6127

"Because of this issue, I've had to migrate from Riak to ElasticSearch." link

Is this expected behavior? Granted, these are EC2 boxes and Leveldb depends heavily on disk, but I can't imagine folks using Riak on production his this type of performance hit resulting from rebalancing. link

Distributed systems are hard and error prone. Building distributed systems are hard. We need to help developers with many "ck"s. fsck, dmck, osck [**?**] are common. fsck is needed (HDFS fsck). dmck is needed ... [**?**, **?**, 1], and now it's time for more research on "slck"..

Just like fsck is a must for every file system, we advocate the SLCK is a must for every system. As one of the first movers of scalability bug finder in this cloud space,

we solve control-plane as suggested by the quotes above. in fact as it is an unsolved problem in deployment, you can lose customers. or the third quote that developers don't expect

Scalability check is still an under research. We need more research in this space. scalability bugs are hard, scale of failure size .. [**?**]

we specifically target ring-based systems and their cluster metadata management (control plane). However we believe other types of systems (*e.g.*, streaming systems) will have their own interesting new challenges.

We hope that recent work like Exalt and ours will spur more exciting research in this space.

(empty page)

(empty page)

References

[1] 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 *OSDI '14*.

(empty page)

8 Bugs

These are the bugs description and probably this is will be under eval.

Aagain this is all scalability bugs that are in the control plane caused by cluster size.

8.1 Cassandra Bug1: ca6127

ca6127

About the protocol (gossip failure detector):

In Cassandra (v **xx**$_{63}$), when we bootstrap a large cluster at one, all the nodes must gossip to each other.

This is the standard setup for boostrapping a large cluster. We must apply seed nodes (the nodes that must be available when node bootstrap). Typically this can be one. Having multiple seed nodes dont matter, but if the seed nodes are not alive yet, this can prolong reboot nodes. When we add a new node, the new node only needs to know about the seed node, and the seed node will tell the new node about other nodes that the seed node already sees.

The goal of gossip is to collect pings and each node figures out which other nodes are alive.

In every 1 second, it picks 1 random node to send a gossip. This is the default setup to prevent heavy gossip message in the background.

A gossip is a list of nodes that the sending node has observed and the version (version when the state of the node changes e.g. when each key reposnbility, or every second it will change the version by 1).

Upon receiving a gossip, the receiver finds node that it doesnt know, and also find node that has different timestamps.

Then every second, another backgroudn process runs the accrual detector.

If a node X has not received a gossip about Y after a certain window period of time (note that the gossip can come from Z, it doesn't have to come from Y), X will declare node Y as dead. The accrual algorithm Phi, is described in **??**, which claims to be "scalable".

Gossip failure detector in Cassandra determine which node is dead by accrual failure detection. Observer calculate Φ (phi) of test node in every second. If the Φ is greater than the threshold, test node is declared dead.

In short, accrual failure detector works by: Cassandra determines node failures by calculating failure probabilities (Phi in accrual failure detection) of every node. The probability is a function of the silent time and average receiving interval. The silent time is duration between the last time the node received a gossip containing heartbeat of particular nodes and the current time. The average receiving interval is an average of duration between two gossips containing heartbeat of particular nodes that the node has received in the past.

If X doesnt know about node C, X will ask Y again more information about the new node (key, load of that node, state=bootstrapping/normal).

About the setup: We bootstrap N number of nodes concurrently with 1024 partitions/node. The submitter of the bug says that they try 10,000 nodes with 32 nodes.

Expectation: The expectation is that when nodes are added, the ring should be stable. There should be no nodes that are falsely marked as dead while they are alive.

In a stable cluster with no node crash, the average receiving interval is quite constant, and the silent time does not exceed a lot from the average receiving interval (approximately the same as the average). For example, in a cluster of 128 nodes that every node gossips every second and the gossip processing time is small, on average the nodes will receive a gossip containing heartbeat of other particular nodes every 7 seconds and the chance that the gossip will take up to 10 seconds is less than 5% (cite gossip paper).

Second, the gossip execution time is significantly large. Because Cassandra does not use gossip for crash detection only, but also uses for announce nodes' states, especially in the startup phase of the cluster. The more nodes' states in a gossip, the longer execution time is.

Bug: The bug that causes this problem is that when a node X receives a new gossip about a node N, X must perform a quite expensive operation: ???

Note that on every incoming gossip, it is possible that X hears about *multiple* new gossips about a set of nodes N_1...N_n, which lengthen the new gossip processing.

From this point, we know that the silent time could be as large as the time that cluster uses to gossip the heartbeats plus the total time the gossips get executed in each node throughout the gossip propagation. The time used to gossip to the whole cluster obviously depends on the cluster size (O(log n)). Now consider what affects the gossip processing time on each node. Cassandra piggybacks the gossip to carry nodes' states. A node will tell other node about all nodes' states that it knows to others, but not all states will take significant time to be pro-

cessed. Mostly, every states in the gossip is just variable assignment, except the commit state. The commit state will tell the receiver node that a particular node will take care certain key ranges in the key ring and the receiver needs to insert those key ranges into its current key ring.

What is the expensive??? If Z is already normal when X hears it, I put Z in ring table, then record Z to the disk. (we record to disk so that when X restarts X can restart fast). Putting Z in ring table, create a new empty table, and add entry one by one. Even one entry changes, they copy the entire table and perform the update in the cloned table, and later ???? (copy back or pointer switch). This is also interesting because Its a MultiMap of IP Address to all keys (the function is $\mathbf{xx}_{64}$ LOC with $\mathbf{xx}_{65}$ if() statements). There are potentially other protocols (and user-facing) that read MultiMap, and we dont want to hold the readlock to long. Thus this is the expensive part #1.

The expensive part #2, write Z to internal on-disk database. It has its own Cassandra database, and it writes itnernally with CQL.

So when the cluster is big, we will see more state differences and hence longer state-diff processing time. This state-diff processing time is single threaded because it touches many in-memory metadata. Thus, it cannot run concurrently. Thus, all other fresh gossips from other nodes with new versions are queued and cannot be processed. For example, its possible say V tells X that Im alive for 5 seconds (Vs version already increases by 5 times), but X gossips to other nodes about V using an old version timestamp. Some other nodes eventually will declare V dead. In other words, the job of X is to forward Im alive gossip from other nodes (via version timestamp), but because X is so busy doing state-diff processing in the single-threaded and all the gossips are backlogged, X doesnt forward the new timestamps to other nodes. Other nodes eventually see this dead.

This state-diff processing,

If a new node changes from bootstrap to normal,

The way Cassandra processes the gossip is that for each commit state, it copies its local view of key ring, determines which commit node that it has not seen committed yet, inserts the node's key ranges into the ring, and write the committed node's information to its internal database.

Most time is spent onto copying the key ring and writing to disk. For copying time, the time depends on the size of all key ranges that node has recognized. For the writing time, this is mostly constant depending on the machine hardware. Thus, the processing time depends on the number of commit state in the gossip and the size of key ranges in the receiver node. And if we consider the number of commit state and the key range size, they are bounded by the cluster size.

Now we can see how the cluster size affects real gossip implementation in Cassandra.

Interesting nature: This bug is interesting, because accrual algorithm Phi, is described in **??**, which claims to be "scalable", but the problem lies in the fact that new gossip processing must log something to the disk which can be long. This is because (why do we need to log something to the disk???). This again shows that when multiple concerns (fast reboot, ..) are put into consideration, the final implementation looks different than the formal descriptions of the algorithm.

This problem doesn't appear in smaller cluster because number of a physical node represents 256 vnodes in deployment. 256-1024 vnodes/node.

Fix: Wait until it sees 40 records. This is very specific to make sure we relax Phi during reboot time.

But interestingly this fix causes other problems, which we postpone until Section **??**.

The fix causes other problems, because if a node dies, it loses all the 40 records. It could also cause other problems when the node scale-out from an existing cluster.

8.2 Cassandra Bug2: ca3831

ca3831

About the protocol (gossip failure detector):

The gossip based failure detector here works like in ca6127, so I will not explain the failure detection part.

In Cassandra, gossip protocol is not for failure detection only, but it is used to announce information of one node to every other node too.

When we decomission node D from a large cluster, node D will gossip to others that it is leaving (set its state to be "leaving"). Every node that has got the gossip about the leaving will help announce this throughout the cluster (as usual in gossip protocol).

When a node receives a gossip containing leaving state of node D, it will calculate pending ranges for every keyspace. Pending range calculating means that for a certain table, which key ranges regard to this keyspace has to be moved.

Pending range calculating is done per keyspace because keyspace is the unit that we configure replication factor. Two keyspaces with different replication factor will have different ranges to be moved.

To configure replication factor of keyspace, we specify that how many replica we want for each data center. For example, for keyspace K, we could specify replication factor to be "datacenter1=2 and datacenter2=1" (put 2 replicas in datacenter1 and 1 replica in datacenter2).

Node D will wait until every node in the cluster has seen it is leaving, then it will change its state to be "left", and gossip this state again.

The pending ranges is the information telling other key range operations that will happen in the future if it can be executed. For example, we can not bootstrap a new node that has a key fall in a pending key range, because Cassandra is doing something with that key range.

Later on, after data in the pending key ranges has been migrated, Cassandra will clear all pending ranges. And we can do other key range operation that fall in those key ranges.

About the setup:

In a running cluster of N nodes without using vnode which every node has seen every other node already and has some keyspaces, we decommission one node.

Expectation:

The expectation is when we decommission a node, the decommission information (node's state changes to "leaving") should be announced throughout the cluster without affecting to failure detection.

Bug:

When we decommission node D from the cluster, it makes the flapping happens. Nodes in the cluster start detecting other nodes as down, and the flapping could continue for some amount of time after decommission finished.

The cause of the bug is in pending range calculation. This execution takes very long time to finish in the case of large cluster.

The false failure detections happen from the same reason as in ca6127. The gossip-based failure detection does not expect that the gossip processing will be slow and delay heartbeats of other nodes. And this make the nodes falsely declares other nodes as dead.

The expensive operation in this case is calculating key range ownership for each node. For example, for a keyspace with replication factor "datacenter1=2, datacenter2=1", if a node is in datacenter1, it will own two ranges, but if it is in datacenter2, it will own one range.

The logic of calculating key range ownership is that for each datacenter, construct the key ring from all nodes in that datacenter. For example, if datacenter1 has 5 nodes, it will construct a key ring that divided to 5 ranges regard to these 5 nodes.

To construct this key ring, Cassandra create empty ring and add nodes to this one by one. Every time a node is added to the ring, Cassandra will sort key list of the ring. Thus, if we have 5 nodes in the datacenter, Cassandra needs to sort this 5 times.

This is ineffective. Imagine we have 256 nodes. We want a ring of this 256 nodes, we have to sort key list 256 times

Although the computation esssentially only affect neighboring nodes, the *implementation* of the algorithm requires the code to copy the Map from scratch one by one. Again, this highlight that the problem is in the implementation.

Interesting nature/characteristics of the bug: Same as in ca6127, we do not expect gossip processing will be slow. The default threshold of accrual failure detection cannot adjust itself to a surge of heartbeat interval. (Accrual failure detection here is for adapt for the heartbeat interval due to longer hop of heartbeat announcement).

Interestingly, developers know that this pending range calculation is slow, but they still think it is manageble. Quote from developers' comment on top of calculatePendingRanges, "This is heavy and ineffective operation. This will be done only once when a node changes state in the cluster, so it should be manageable."

But it turns out to be false.

Fix: When constructing a key ring of each datacenter, developers added a method to add multiple nodes at the same time. (addKey(Key) -¿ addKeys(List¡Key¿)). So here, when we want to add construct 256-node ring, we will do key sorting only once

This fix is helpful, and it has been still adopted.

(the old bug is because the simplicity of keyrange pending calculation).

all cluster operations are piggybacked by gossip such as decommission, bootstrap, "I'm alive", load, keyrange, datamove,

8.3 Cassandra Bug3: ca3881

ca3881

About the protocol: Like in ca6127 (bug 1) and ca3831 (bug 2), this is about gossip-based failure detection. But in Cassandra, we use gossip protocol not only for failure detection, but also for announcing some information of nodes to every other nodes.

In a cluster with vnode used, when we add a new node B

to the cluster, B will pick its keys K1 - Kn randomly and tell every one it is bootstrapping and going to take care these keys K1 - Kn. When every node has known B and there is no conflict of B's keys with other nodes (there is no other nodes claiming a key in the same range as B, etc.), B will commit that it will take care these keys for sure, and B can serve requests (commit means B changes its state to be "normal").

When a node receives a gossip containing bootstrapping state of node B, it will calculate pending ranges for every keyspace. Pending range calculating means that for a certain table, which key ranges regard to this keyspace has to be moved. (As same as in ca3831).

The detail of failure detection is as same as ca6127 and pending range calculation detail is as same as ca3831.

But I suspect that in this bug, vnode feature was introduced in this version of Cassandra. (There was a mention from developers that vnodes make calculation longer.) Although, we have fixed the expensive operation in ca3831, vnode leads to more computation and make pending range calculation slow again.

The greater the number of bootstrapping nodes, the greater the probability of pending range calculations having to be done more often

About the setup:

In a running cluster of N nodes with vnode used which every node has seen every other node already and has some keyspaces, we bootstrap a new node (add a new node).

Bug:

When we add a new node B to the cluster, it makes the flapping happens. Nodes in the cluster start detecting other nodes as down.

The cause of the bug is as same as ca3831 which is in pending range calculation. This execution takes very long time to finish in the case of large cluster, especially, with vnodes used.

Interesting nature/characteristics of the bug:

Although, developers have made a fix in ca3831, the bug still happens here. Why? My guesses are that

1. The introduction of vnode feature. Quote from developers' comment "... calculateNaturalEndpoints will be called thousands of times with vnodes in some code paths." (calculateNaturalEndpoints is a method for calculation key range ownership.)

2. That fix was not good enough for some workload or some configurations. For example, adding a few nodes at the same time, or working in multiple datacenters.

Fix:

Unfortunately, I cannot download patches from the JIRA, because the links to the patches pointed out to external repository which is unavailable now.

But from my understanding regard to the description, developers did 2 things.

1. Add variables to track which nodes are in a specific datacenter. Before this, we need to iterate through all nodes to find out all nodes in the specific datacenter.

2. Reduce complexity of calculateNaturalEndpoints method to be O(log(N)). (Not sure how).

8.4 Cassandra Bug4: ca5456

ca5456

About the protocol: Like in ca6127 (bug 1) and ca3831 (bug 2), this is about gossip-based failure detection. But in Cassandra, we use gossip protocol not only for failure detection, but also for announcing some information of nodes to every other nodes.

In a running cluster, when we add a new node B to the cluster, B will tell every one it is bootstrapping.

When a node receives a gossip containing bootstrapping state of node B, it will calculate pending ranges for every keyspace.

The detail of failure detection is as same as ca6127 but for pending range calculation detail there are something more in this bug.

About the setup:

In a running cluster of N nodes (probably with vnode used) which every node has seen every other node already and has some keyspaces, we bootstrap a new node (add a new node).

Bug:

When calculating pending ranges, Cassandra needs to lock a set of bootstrapping nodes. And if this calculation is long, other thread needs to lock this has to wait for long time.

The report said that this make gossip stop. But I cannot see there is other thread uses this bootstrapping set.

My guess about this bug are,

1. Gossiper needs to acquire lock for bootstraping set, but it cannot, so it has to wait very long, and make gossip interval greater than 1 second. However, I do not think this is the case. I cannot find the code in gossiper using the bootstrapping set.

(HSG: Korn, if this is true, this could be interesting, that the blocking operation is not always on the gossip receiving end, but could be at the gossip sending end.)

2. Cassandra processes multiple gossip at the same time, but the second message still needs to wait for the first message to finish first.

Interesting nature/characteristics of the bug:

If the description in JIRA means gossiper delay gossip interval greater than 1 second, it makes the math model to calculate Phi in accrual failure detection cannot work.

Accrual failure detection in Cassandra expects that the gossip is sent every message.

Fix:

Cassandra locks the bootstrapping set, then copies the set, and unlock it. Then Cassandra works on the copied set.

8.5 Riak Bug1: r??

r??

About the protocol (gossip ring convergence):

Riak uses the gossip protocol to communicate the current state of partition ownership: which partitions are owned by which node at any given point of time. This is done sending asynchronous messages to a random node in the cluster every X milliseconds, where X is a random number between 1 and 60000. Everytime a node receives a new message, it compares the new received version with the one that it currently has: If the node determines that it should change its current version (based on a structure caled vector clock), it performs a set of operations that allows it to adapt its view of the partition ownership to the current set of nodes in the cluster, always looking to achieve the most balanced (where the number of partitions is evenly splitted among members and the replication value is respected) ownership.

To determine the current balance, the node X counts the number of partitions owned by each known node and checks if, for every known node N owning P_N partitions the perfect balance is met, by calculating:

$$\frac{P}{N} - P_N < -2 \vee \frac{P}{N} - P_N > 2 \qquad (1)$$

where P is the number of partitions (global for each node) and N is the number of known nodes. If 1 is not met, then the node proceeds with the rearrange procedure. The rearrange procedure consists on the following steps:

1. The node checks if the replication value (which is a parameter) is met or not. If the replication value of the cluster is T then it checks if for every subset of size T of the current arrangement all partitions on that set are owned by different nodes.
2. f the replication value was not met, a complete rebalance of the ring is performed. This operation consists on assigning partitions to all known nodes in a round robin fashion.
3. If the replication value was met and the current node has no claims, then the current node will claim partitions respecting the replication value T.
4. If the replication value is met and the current node has claims in the ring, the current node attempts to find the biggest hole in the ring. A hole is defined as the space between two partitions owned by the same node, and the size of this space is the number of partitions contained in it. When the biggest hole is located, the current node chooses the partition on the middle of that hole (integer division) and claims it.

This operations are performed in a recursive fashion until the node determines a balanced (closest to perfect balance) view of the ring. When it reaches this view, the node will adopt the result and communicate it to another (random) node in the cluster.

About the setup:

We bootstrap N number of nodes concurrently with 64 partitions per node. The submitters claimed that when the ring size is greater than the bug starts appearing when the ring size is greater than 256 partitions (in total).

Expectation:

The expectation is that when nodes are added, the ring should be stable, i.e, all nodes should have the same view of the partition ownership. The ring convergence time is dominated by the execution time of the operations that every node has to perform in order to build a balanced view of the ring.

Bug:

Every node performs cpu intensive operations to determine the perfect balance of the ring, based on the total number of partitions and the number of known nodes. While doing this calculations, it could receive other gossip messages that could lead to redo this process again, implying more cpu intensive operations. Since this operations are performed by every node in the cluster, the execution time needed to calculate the result and spread the message to the other nodes gets larger when the number of nodes and partitions grows since every node is

forced to perform cpu intensive operations potentially every time they gain knowledge about a new node in the cluster.

Interesting nature/characteristics of the bug:

This problem is interesting because it exposes how cpu intensive operations affect the behaviour of distributed protocols. In this case, the heavy operations are not related to the new ring creation, but related to the transferring of this new copy to our current view, which is centralized in another erlang module. It was designed that way to allow all the modules to have a common entry point to read the current view of the ring and to associate event listeners to different ring update events. The problem relies in that when we have larfge clusters, with large number of partitions per node the related operations (mostly $O(P^2)$) do not offer the required performance. So, this problem can also be interesting from the algorithm design and implementation point of view.

Fix:

The heavy rebalance operations are delegated to one node, which is called the claimant. This node prepares the changes (establishing data transfers and partition ownership) to finally commit changes only if requested (staged commit). When the changes are commited, this new ring is gossiped to other nodes, using an improved communication mechanism.

8.6 Voldemort Bug1: v??

vWhy voldemort rebalance so slow

About the protocol:

In Voldemort, rebalancing is actually a partition rebalancing. Existing nodes needs to move partitions to new nodes in order to achieve an even partition distribution.

Each Voldemort node holds exactly the same *cluster.xml* file to get the information about all nodes in the cluster. When we do rebalancing, it is actually doing rebalancing on a step-by-step transition from *cluster.xml* to *target-cluster.xml*, which is our goal of rebalancing.

Assume we add a new node and do rebalancing, *traget-cluster.xml* tells how many partitions and what partitions the new node will have. This determines how many partition transitions Voldemort will have. Each partition transition represents the migration of one primary partition along with all its side effect (i.e. migration of replicas + deletions). Here the default replication factor is 2.

When we add a new node X, without any partition, before doing rebalancing, we already get the incoming partitions p1, p2, p3 based on *target-cluster.xml*. When we do real rebalancing, the first step is to move partition p1. Voldemort will generate a rebalance plan for each step and assign a rebalance task to execute the partition transition. After that, repeat the same process to move partition p2, and then p3.

Each time when we move one partition, *all* nodes should change cluster metadata and rebalance state, then submit the corresponding rebalane task, which is responsible for completing the rebalancing plan involved in the partition transition. Normally, one partition transition involves a *stealer* node and a *donor* node. The donor node begins to copy the partition to the stealer node and then delete the partition on itself. The adminstrator exploits the rollback mechanism to inquire these two nodes about the completeness of the rebalance task. After it is done, this partition transition is done and begin the next partition transition.

About the setup:

I bootstrap 64 nodes with 20 partition per node.

- When I try to write 25000 k/v pairs (the size of each pair is 1K) to the cluster, we have rebalancing time of adding 16, 32, 64 nodes 2430s, 4452s, and 8429s, respectively.
- When I try to write 2500 k/v pairs (the size of each pair is 1K) to the cluster, we have rebalancing time of adding 16, 32, 64 nodes 1919s, 3538s, and 7112s, respectively.
- When I try to write 0 k/v pairs (the size of each pair is 1K) to the cluster, we have rebalancing time of adding 16, 32, 64 nodes 1932s, 3554s, and 7090s, respectively.

Bug:

The bug that causes this problem is that when we add multiple nodes, it must perform much more partition transitions.

Why does voldemort rebalance so slow?

If we move one partition, which just involves a donor node A and a stealer node B, all nodes should participate in cluster metadata change and rebalance state change. That means all nodes change from normal state to rebalance state and get ready to do rebalancing. Meanwhile, the admin uses rollback to inquire about the status of node A and node B. If it is not done in 1 second, the admin will inquire agin after 2 seconds, if it is still not done, the admin will continue inquiring after 4 seconds. In this way, just for one partition transition, the admin is possible to inquire more than 3 times, that's at least

1+2+4=7 seconds inquiring time. Imagine we add multiple nodes, it is equal to much more partition transistions. It is possible to be a disater in doing rebalancing.

(HSG: the problem is not in the admin inquiring, the problem is why does it take 7 seconds to just do one parittion move?)

Interesting nature/characteristics of the bug:

The rebalancing process contains cluster metadata change and reblance state change, which is scalabe. If we have more nodes, we need spend more time changing metadata and rebalance state. It is a time consuming process. In addition, adding more nodes means more partition transititons. As we mentioned before, rebalancing is to transist partition one by one. Therefore, it is further time consuming when we add multiple nodes and do rebalancing.

Fix:

For each partition transition, it involves two nodes, a donor and a stealer. Instead of moving partitions one by one, we can move partitions together based on one donor-stear pair. Each time when a stealeris connected with a donor, try to move all needed partitions between these two nodes at a time.

(HSG: Batching?????)

(empty page)

(empty page)

(empty page)

9 System Details

10 Cassandra

- Bootstrapping process of Cassandra with/without vnodes

  How does it choose its key?

  If it does not vnode, it waits for some specific of time to see the ring, then it chooses the mid between the largest interval.

  If it uses vnode, it chooses randomly.

- How does it know other nodes' keys? From gossip messages.

- Does a node gossip up/down status of the others via gossip?

  No!

- When does Cassandra node move from bootstrap to normal?

- **What is implication of false detection?**

  - False negative

  When coordinator node forwards the request to the dead node, it has to wait until timeout. Waste time!

  - False positive

  - If consistency level can meet, it must do hint-handoff, when a false-detected node comes back, there could be a flood of hinted-handoff that could make false detection again.

  - If consistency level cannot meet, the request is rejected, even though, it is possible to write.

  (Notes: good answers, we can use this later.)

- What is endpoint cache? What is endpoint cache invalidation? What events invalidate endpoint cache? (CA-6345)

  ???

- When false detection happens, it means gossip message is delayed, what delays the gossip message? When gossip protocol is run by separate thread, other computing should not affect gossip messages. ???

- How does coordinator re-route request to in-charge nodes? ???

(empty page)

11 Junk

11.1 Bug1

Fix: Wait until it sees 40 records. This is very specific to make sure we relax Phi during reboot time.

But interestingly this fix causes other problems, which we postpone until Section **??**.

The fix causes other problems, because if a node dies, it loses all the 40 records. It could also cause other problems when the node scale-out from an existing cluster.

11.2 Implication of Flapping

What is implication of false detection?

- False negative

When coordinator node forwards the request to the dead node, it has to wait until timeout. Waste time!

- False positive

- If consistency level can meet, it must do hint-handoff, when a false-detected node comes back, there could be a flood of hinted-handoff that could make false detection again.

- If consistency level cannot meet, the request is rejected, even though, it is possible to write.

11.3 On Cassandra Gossip

In a stable cluster with no node crash, the average receiving interval is quite constant, and the silent time does not exceed a lot from the average receiving interval (approximately the same as the average). For example, in a cluster of 128 nodes that every node gossips every second and the gossip processing time is small, on average the nodes will receive a gossip containing heartbeat of other particular nodes every 7 seconds and the chance that the gossip will take up to 10 seconds is less than 5% (cite gossip paper).

The aforementioned case is a case that the cluster has become stable. But from the beginning of Cassandra startup, the cluster needs time to reach the stable point due to 2 reasons. First, Cassandra assigns the first receiving interval to be calculated for the average to be 500 ms, which is quite small when cluster is big. Second, the gossip execution time is significantly large. Because Cassandra does not use gossip for crash detection only, but also uses for announce nodes' states, especially in the startup phase of the cluster. The more nodes' states in a gossip, the longer execution time is.

11.4 Phi

In short, accrual failure detector works by: Cassandra determines node failures by calculating failure probabilities (Phi in accrual failure detection) of every node. The probability is a function of the silent time and average receiving interval. The silent time is duration between the last time the node received a gossip containing heartbeat of particular nodes and the current time. The average receiving interval is an average of duration between two gossips containing heartbeat of particular nodes that the node has received in the past.

Scale of failure:

(empty page)

(empty page)

Bug	Scenario	Problem	Fix	Test setup
CA-6127 sc-clus (Korn)	Enable vnode - Bootstrap a large cluster at once (Not sure, but seem possible) - Add a big number of nodes to the running cluster	A node has not received gossip info of one certain node for long time. - T_{silent} is big because propagation path become longer. - Propagation path is long because of the size of the cluster. - Gossip message takes long time to be processed. - (Probably be one cause) The initial $T_{avghbperiod}$ is small (fixed at 500 ms) compare to the real T_{silent} in big cluster.	- Cassandra does not try to calculate Phi for a node that it has just known, but it will wait until it gets enough gossip to make sure that $T_{avghbperiod}$ is stable	- Bootstrap Cassandra cluster (Try this) - Add a big number of nodes to the running cluster (w/ and w/o tables)
CA-3831 sc-clus	(From bug report, but we cannot make it works) - Bootstrap a big cluster (Murphy has tried this) - Decommission one node from the big cluster, and add a new blank node to the cluster	- There is an expensive function gets called very often during bootstrapping This function takes a gossip message and key range as input. (Key range is big)	- Optimize the expensive function and reduce the number of function call	(Tested) - Start a big cluster, create some data, decommission a node, and then add a new node (Try this) - Bootstrap a big cluster - Add a big number of nodes to the running cluster (w/ and w/o tables)
CA-3881 sc-clus	Follow-up work from CA-3831 Everything should be the same as CA-3831			
CA-5456 sc-clus	- Bootstrap a big cluster	- Gossip processing function acquires lock too long, and this lock is needed by Gossiper	- Make the function release the lock as fast as possible	- Bootstrap a big cluster - Add a big number of nodes to the running cluster (w/ and w/o tables)
CA-6488 sc-clus sc-load?	In a cluster with a number of vnodes, send batch requests to the cluster (Not sure how large requests should be)	To process a request, Cassandra needs to do CPU-intensive computing (cloneTokenOnlyMap)	Cache the calculation result	Setup a cluster with a number of vnodes, and send batch requests
CA-6268 sc-clus	(Enable vnode in Cassandra) Hadoop is using Cassandra as its data storage	Cassandra generates a lot of splits (proportionately to the number of vnodes), so Hadoop needs to process a big number of splits	Merge consecutive ranges in a local node before generating Hadoop splits	Setup Hadoop + Cassandra, and find workload to trigger the computing. (Maybe just simple workload as word count)

Bug	Scenario	Problem	Fix	Test setup
CA-6345 sc-clus sc-load?	In a cluster with a number of vnodes, do some operation that alter the key range	The computing of key range altering takes high amount of CPU throughout the cluster	The fix caches a result of a function that is slow	Setup a cluster with a number of vnodes, and decommission a node / few nodes.
CA-5982 sc-data sc-load	Write big data that triggers memtable flushing.	MemoryMeter measures how big a CF in memory is, but it is slow. Although, the CF got flushed to disk, MemoryMeter still keeps it in memory. And the MemoryMeter queue is also unbounded (seems like Cassandra holding 2 copies of CF in memory).	Change some configs, change unbounded queue to size-configurable queue, make some code faster	Creating 2000 CFs, and then insert data to Cassandra
CA-5719 sc-load	In long running cluster that has many clients connect to it via Thrift, there will be memory leak.	Cassandra caches ClientState for each IP address but does not remove it	Add callback when client disconnects from the server, so the callback will remove the entry from the cache	Try running a number of client connections via Thrift interface
CA-4467 sc-data	Cassandra stores a big data on it, and the user wants to change the compaction strategy from SizeTieredCompactionStrategy to LeveledCompactionStrategy, but the free space is not enough, although the free space is half of the current usable space		No fix	Setup big data and change compaction strategy
CA-6496 sc-load sc-data	Do heavy workload to the Cassandra (mixed workload, read, write, delete)	The compaction during workload never finish	Change some hard-coded number that limit the output of compaction	Do heavy workload to the Cassandra (mixed workload, read, write, delete)
CA-5220 sc-clus	In a cluster with a number of vnodes, repair takes very long time to finish		No fix	Setup a cluster with a number of vnodes, and issue repair command
CA-4813 sc-data	When MapReduce is working with Cassandra, and it is reading a lot of data, the connection got disconnected	A bug in Thrift that miscalculates reading data size.	Fix logic bug	Setup a cluster with a number of vnodes, and run MapReduce jobs.

Φ = f($T_{hbsilence}$, $T_{avghbperiod}$)

$T_{hbsilence}$ = g(hops, $T_{gossipexec}$)

$T_{avghbperiod}$ = h(all previous $T_{hbsilence}$)

hop = i(n) = log(n) on average

$T_{gossipexec}$ = k(partitionTable, commitStates, hardware)

partitionTable <= n

commitStates <= n

Symbols	Description
Φ	The probability that observer thinks the test node dead
$T_{hbsilence}$	Time period that observer has not receive hb containing test node info
$T_{avghbperiod}$	The average $T_{hbsilence}$ in the past
Hop	The number of hops heartbet got forwarded until it reached a ceratain node
$T_{gossipexec}$	Gossip processing time in all nodes in $path_{propagation}$

I bootstraped (started from scratch) different Cassandra cluster size, and let they run for 5 minutes, then measure all parameters effecting the number of false detections.

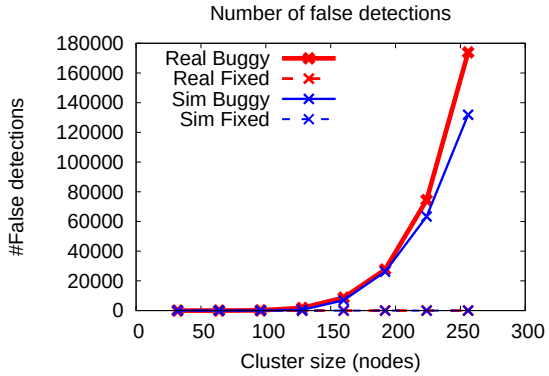

Figure 7: **The number of false detections occurs in the whole cluster.**

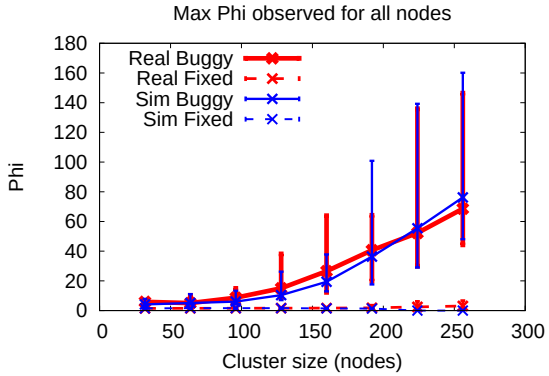

Figure 8: **Max Phi calculated for every node in the cluster.**

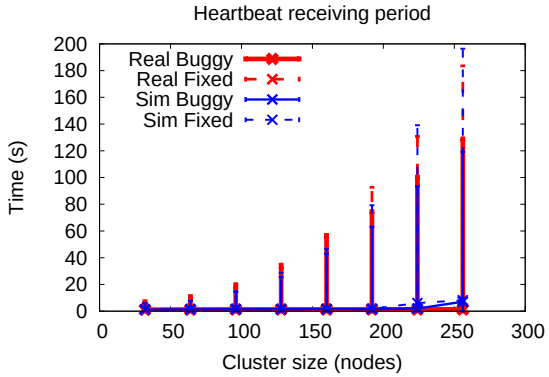

Figure 9: **Heartbeat receiving interval of every node.**

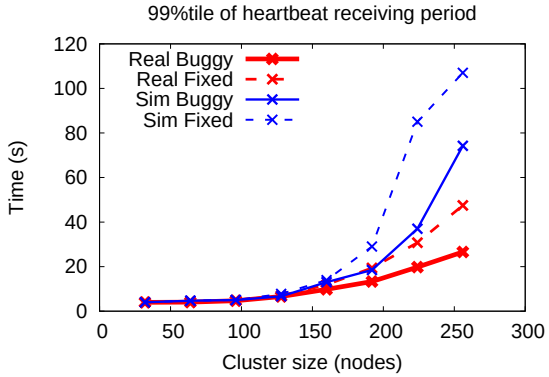

Figure 10: **99 percentile of heartbeat receiving interval of every node.**

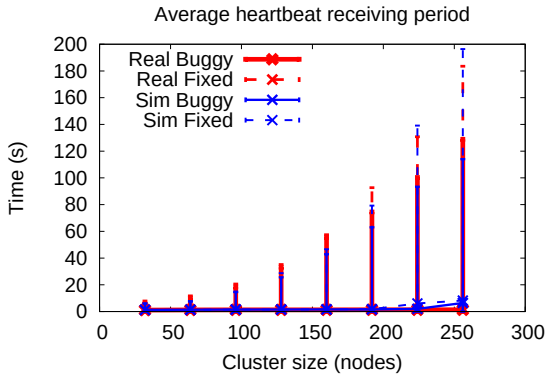

Figure 11: **Average of past heartbeat receiving interval.**

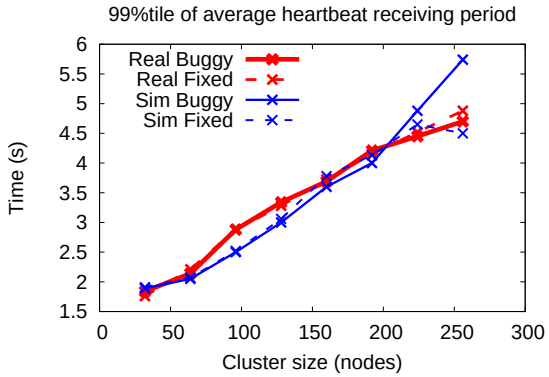

Figure 12: **Average of past heartbeat receiving intervals.**

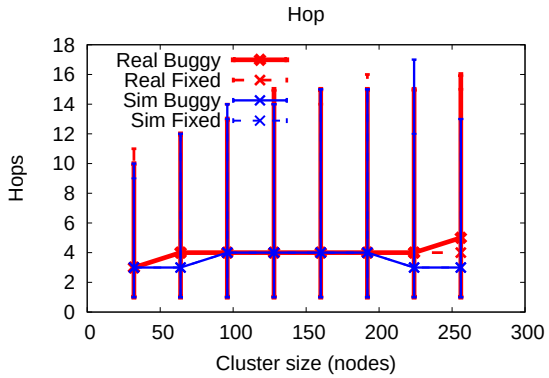

Figure 13: **The number of hops of heartbeat got gossiped.**

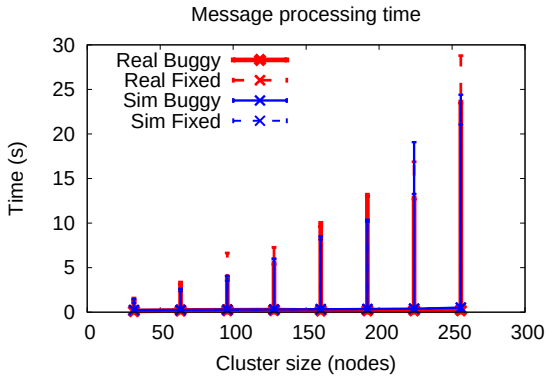

Figure 14: **Processing time of gossips containing commit states.**

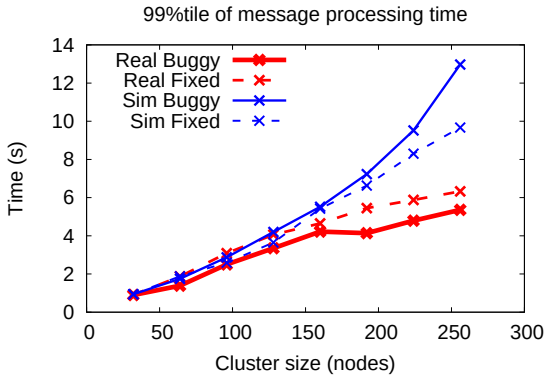

Figure 15: **99 percentile of processing time of gossips containing commit states.**

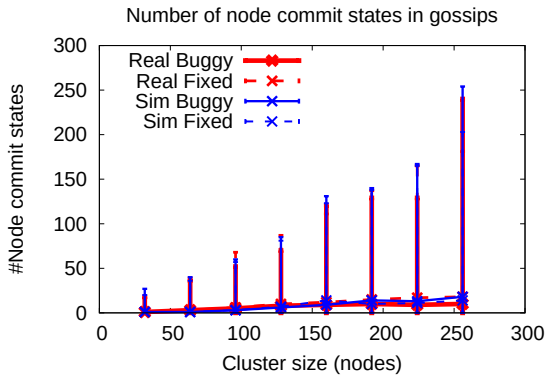

Figure 16: **The number of commit states in gossip messages.**

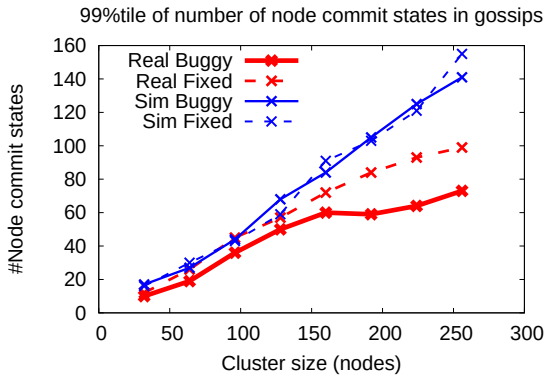

Figure 17: **99 percentile of the number of commit states in gossip messages.**

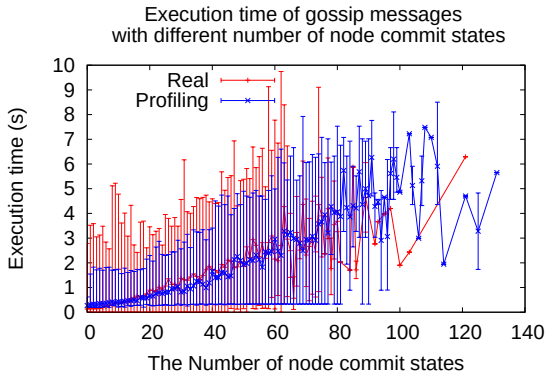

Figure 18: **The execution time of messages with different number of commit states in messages. The varying comes from the different states in the receiver.**

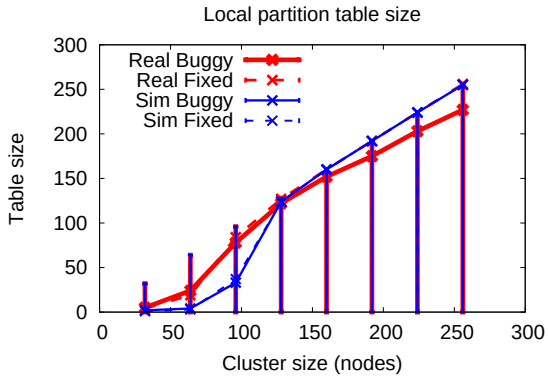

Figure 19: **The partition table size in nodes when processing a gossip.**

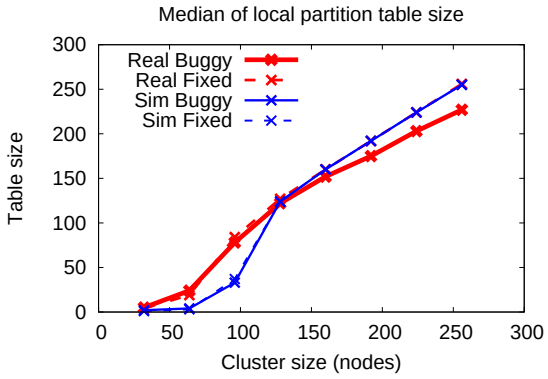

Figure 20: **The median of the partition table size in nodes when processing a gossip.**