THE UNIVERSITY OF CHICAGO

TOWARDS SCALE-CHECKABLE SYSTEMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER’S

DEPARTMENT OF COMPUTER SCIENCE

BY
CESAR A. STUARDO

CHICAGO, ILLINOIS
2019
To my family

Be not afraid of going slowly, be afraid only of standing still
To my son

The fox knows many things; the hedgehog one great thing
# TABLE OF CONTENTS

LIST OF FIGURES ................................................................. vii

LIST OF TABLES ..................................................................... viii

LIST OF ALGORITHMS ............................................................... ix

ACKNOWLEDGMENTS ................................................................. x

ABSTRACT ........................................................................... xi

1 INTRODUCTION ................................................................. 1

2 BACKGROUND AND MOTIVATION ........................................ 5

3 SCALECHECK ................................................................. 8
   3.1 SFIND ............................................................................ 8
   3.2 STEST ........................................................................ 11
      3.2.1 Black-Box Approaches ........................................ 11
      3.2.2 A White-Box Approach ....................................... 14
   3.3 Processing Illusion (PIL) ............................................. 16
      3.3.1 PIL-Safe Functions ............................................. 17
      3.3.2 Pre-Memoization (with Determinism) ................. 18
      3.3.3 Putting It All Together ....................................... 19

4 IMPLEMENTATION AND APPLICATION .................................. 21

5 EVALUATION ................................................................. 23
   5.1 Exposing Scalability Bugs .......................................... 23
   5.2 Discovering Unknown Bugs ....................................... 27
   5.3 Accuracy ..................................................................... 29
   5.4 Colocation Factor ....................................................... 30
   5.5 Pre-Memoization and Replay Time ............................... 33
   5.6 Test Coverage ........................................................... 33

6 DISCUSSION ................................................................. 34

7 RELATED WORK ............................................................... 38

8 CONCLUSION ................................................................. 40
**LIST OF FIGURES**

1.1 Definition of scalability bugs (§1) .................................................. 2

2.1 An example bug (§2a) ................................................................. 6

3.1 $O(N^3)$ scale-dependent loops (§3.1) ........................................ 9

3.2 Global Event Driven Architecture (§3.2.2) .................................. 14

3.3 SCALECHECK complete automated flow (§3.3.3) ......................... 16

5.1 SCALECHECK effectiveness in exposing scalability bugs (§5.1) .......... 24

5.2 Testing latest-stable protocols (§5.2) ......................................... 27

5.3 Cassandra internal metrics (§5.3) ............................................. 29

5.4 Accuracy in exposing c6127 (§5.3) ........................................... 31

5.5 Maximum colocation factor (§5.4) ........................................... 32
LIST OF TABLES

3.1 Colocation strategies and bottlenecks (§3.2) ........................................... 12
4.1 Integrations LOC (§4) .................................................................................. 22
5.1 Bug benchmark (§5.1) .................................................................................. 25
9.1 List of Scalability Bugs (§9.1) ................................................................. 41
# LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1  <code>sFind</code></td>
<td>44</td>
</tr>
<tr>
<td>9.2  <code>createCallGraph</code></td>
<td>45</td>
</tr>
<tr>
<td>9.3  <code>processScaleDependency</code></td>
<td>46</td>
</tr>
<tr>
<td>9.4  <code>collectLockAndIOMetadata</code></td>
<td>47</td>
</tr>
<tr>
<td>9.5  <code>maybeReportScaleDependentCandidate</code></td>
<td>48</td>
</tr>
<tr>
<td>9.6  <code>isPILCandidate</code></td>
<td>49</td>
</tr>
<tr>
<td>9.7  <code>insertPreMemoizationCode</code></td>
<td>50</td>
</tr>
<tr>
<td>9.8  <code>memoize block example</code></td>
<td>50</td>
</tr>
<tr>
<td>9.9  <code>replay block example</code></td>
<td>51</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

It is always difficult to write this type of sections, since is too easy to forget all the people that, with little pushes, helped us into achieving our goals. So let's first start by those who, sometimes inadvertently, gave me a word of wisdom, a pat in the back or an always welcome criticism. Your anonymous help will be always appreciated.

To my advisor, Haryadi Gunawi, who with his patience has helped me improve in many aspects. It seems that there is not a limit on the amount of things I can learn from you, and I sincerely appreciate all of your advises. I consider myself lucky on being part of your team.

To my wife Paulina and my son Cesar. Both of you have been with me during all these effort, which in reality is more our effort than my effort. With ups and downs, our little team has been able to overcome many difficulties, and that is something I cannot take for granted. The only sad part is that within the academic context, I cannot give you all the credit you deserve. But rest assured, you are both a true blessing to me.

To my parents, brothers, uncles, cousins, grandparents, family in law. You have all served as an inspiration to me, and it has been difficult to be so far away from you. You are always in my thoughts, and albeit none of you will probably ever read this, thanks for every prayer, every smile and every tought. You are all part of me.

And finally, to my old friend, long gone, but never forgotten. Thanks for everything, and stop by more often.
ABSTRACT

Is scale a friend or a foe? On the positive side, scale surpasses the limit of a single machine in meeting user’s increasing demands of computing and storage, leading to many developments of "cloud-scale" distributed systems. On the negative side, scale creates new development and deployment issues; First, developers must ensure that their algorithms and protocol designs are scalable, however, until real deployment takes place, unexpected bottlenecks in the actual implementations are often unforeseen. Second, the common practice of debugging scalability in large clusters is complex, costly, time consuming and is not ideal in times where continuous integration and deployment are priorities. We present ScaleCheck, an approach for discovering scalability bugs and for democratizing large-scale testing. ScaleCheck employs a program analysis technique, for finding potential causes of scalability bugs, and a series of colocation techniques, for testing implementation code at real scales but doing so on just a commodity PC. ScaleCheck has been integrated to several large-scale storage systems, Cassandra, HDFS, Riak, and Voldemort, and successfully exposed known and unknown scalability bugs, up to 512-node scale on a 16-core PC.
CHAPTER 1
INTRODUCTION

Being a critical backend of many today’s applications and services, distributed systems must be highly reliable. Decades of research combat a variety of distributed system bugs, including concurrency [45, 49], configuration [72, 73], dependency [77], error-handling [56, 75], performance [35, 40, 61, 74], and security [76] bugs. The challenge grows as distributed systems continue to scale, especially in the last couple of years where the field witnesses a phenomenal deployment scale; Netflix runs tens of 500-node Cassandra clusters [33], Apple deploys a total of 100,000 Cassandra nodes [1], Yahoo! revealed the largest Hadoop/HDFS cluster with 4500 nodes [34], and Cloudera’s customers deploy Spark on 1000 nodes [23, 26].

Is scale a friend or a foe [66]? On the positive side, scale surpasses the limit of a single machine in meeting increasing demands of compute and storage. On the negative side, this new era of “cloud-scale” distributed systems has given birth to a new class of bug, scalability bugs, as defined in Figure 1.1.

From our in-depth study of scalability bugs (§2), we identified two challenges. First, scalability bugs are not easy to discover; their symptoms only surface in large deployment scales (e.g., $N>100$ nodes). Protocol algorithms might seem scalable in design sketch, but until real deployment takes place, some bugs remain unforeseen (i.e., there are specific implementation choices whose impacts at scale are unpredictable). Last but not least, their root causes are often hidden in the rarely tested background and operations protocols.

Second, the common practice of debugging scalability bugs in large-scale systems is arduous, slow and expensive. For example, customers report scalability issues, but the developers might not have direct access to the same cluster scale and must wait for a “higher-level” budget approval for using large test clusters. As it stands today, many developers are heavily reliant on test clusters operated by large companies to do scale testing and only expert developers can do so [25].

These realities raise the following question: how to discover latent scalability bugs and democ-
scalability bugs: Latent bugs that are scale dependent, whose symptoms surface in large-scale deployments (e.g., $N>100$ nodes), but not necessarily in small/medium-scale (e.g., $N<100$) deployments.

Figure 1.1: Definition of scalability bugs (§1). Detailed examples are in §2a and §5.1.

ratize large-scale testing? To this end, we introduce SCALECHECK, a concept that emphasizes the need to scale-check distributed system implementations at real scales, but do so cheaply on just one machine, hence empowering more developers to perform large-scale testing and debugging.

We design SCALECHECK with two components (SFIND and STEST) to address the two challenges. First, to reveal hidden scalability bugs, we build SFIND, a program analysis support for finding “scale-dependent loops.” This strategy is based on our findings that the common root cause of scalability bugs is loops that iterate on data structures that grow as the system scales out (e.g., an $O(N^3)$ loop that iterates through lists of node descriptors). Such loops can span across multiple functions and classes and iterate a variety of data structures, hence the need for an automated approach. With SFIND output, developers can setup the necessary workloads that will exercise the loops and reveal any potential impacts to performance or availability.

Next, to democratize large-scale testing, we build STEST, a single-machine scale-testing framework. We target one machine because arguably the most popular testing practice is via unittests, which only requires a PC. Developers already invest a significant effort on unittests; their LOC can reach 20% of the system’s code itself. However, current distributed systems and their unittests are not built with single-machine scale-testing in mind. For example, naively packing nodes as processes/VMs onto one machine quickly hits a colocation limit of 50 nodes/machine and we found no way to achieve a high colocation factor with black-box methods (no target system modification). Thus, we introduce novel colocation techniques such as global-event driven architecture (GEDA) in single-process cluster and processing illusion (PIL) with non-intrusive modification.

To show the generality and effectiveness of SCALECHECK, we have integrated SCALECHECK
to a variety of large-scale storage systems, Cassandra [54], HDFS [17], Riak [29], and Voldemort [28], across a total of 15 earlier and newer releases. We scale-checked a total of 18 protocols (bootstrap, rebalance, add/decommission nodes, etc.), reproduced 10 known bugs and discovered 4 unknown critical scalability bugs (in Cassandra and HDFS). By only modifying the target systems in 179 to 918 LOC (and with a generic STEST library), we can colocate up to 512 nodes on a 16-core 32-GB commodity PC with high result accuracy (i.e., observe a similar behavior as in the real-scale deployment).

SCALECHECK is unique compared to related work. For example, scalability simulation [38, 53] only checks models, but SCALECHECK checks implementation code. Extrapolation from “mini clusters” [53, 78] does not work if the bug symptoms do not surface in small deployments, but SCALECHECK checks at real scales. Finally, emulation “tricks” run implementation code at real scale but in a smaller emulated environment [47, 70, 10] (the same category SCALECHECK can be put in), however existing techniques have limitations such as not addressing CPU contention and not finding potential causes automatically (more in §7). We also acknowledge many other works in improving storage scalability [41, 68], while our work emphasizes on scalability faults.

In summary, scalability bugs are new-generation bugs to combat in modern cloud-scale storage. Finding them without dependence of large clusters is a new research area to explore. In fact, this problem was discussed in a recent large meeting of Hadoop committee [25]. Currently, many new features in the alpha releases of Hadoop/HDFS still “sit on the shelf,” i.e., it is hard to test alpha (or even beta) releases at real scales as large production systems are not always accessible for testing. Some new features are still pushed and deployed but without much confidence. With this unideal reality, the committee agrees on the need for this new research, that it will increase their confidence on new releases [25]. Some companies began to invest in building scale-testing frameworks. For example, LinkedIn just released their scale-testing framework this year [10, 9] but it only emulates storage space specifically for HDFS.
In the following sections, we present an extended motivation (§2), SCALECHECK design, application and implementation, and evaluation (§3-5) discussion, related work, and conclusion (§6-8).
Scalability bugs are not a well-understood problem. To the best of our knowledge, we provide the first in-depth look at scalability bugs in scale-out systems.

(a) What is an example of scalability bugs? In Cassandra issue #c6127 in Figure 2.1 [6], the bug surfaced when bootstrapping a large cluster. Here, (a) Every second every node gossips to its peers its ring view and version number (e.g., Y gossiped up to version $Y_9$), (b) the receiving node (e.g., $X$) executes “$f()$” to synchronize the view, (c) when $N$ is large, this $O(N^3)$ scale-dependent process creates a backlog of new gossips, (d) thus $X$ keeps gossiping only the latest (old) versions (e.g., $Y_1$), (e) as $Y$’s recent gossips are not propagated on time, other nodes (e.g., $Z$) mark $Y$ as dead. This repeating process leads to a cluster instability with thousands of “flappings” as $N$ grows; a “flap” is when a node marks a peer as down and alive again. More detailed examples are presented in §5.1.

(b) Do they exist in many scalable systems? We have collected a total of 55 bugs in many modern distributed systems (13 in Cassandra, 5 in Couchbase, 6 in Hadoop, 13 in HBase, 16 in HDFS, 1 in Riak, and 1 in Voldemort). This is an arduous process due to the lack of searchable keywords for “scalability bugs”; we might have missed some other bugs. We post the full list in table 9.1.

All the bugs were reported from large deployments (100-1900 nodes). We emphasize again that all these bugs can only be reproduced at scale.

(c) What are the root causes? We study the buggy code, patches, and developer discussions and find that the majority (52) of the bugs are caused by scale-dependent loops, which iterate scale-dependent data structures (e.g., list of nodes); the rest is about logic bugs that can be caught with single-function testing. We break them down to three categories: (1) CPU-intensive loops (15 bugs); Figure 2.1 shows an example. (2) Disk IO loops (26 bugs); the pattern is similar to Figure 2.1 but the nested-loops contain disk IOs. (3) Locking-related loops (11 bugs); they can be in the form of locks inside the loops or vice versa. These patterns suggest that this problem lends
Figure 2.1: An example bug (§2a).  
(a) Every second every node gossips to its peers its ring view and version number (e.g., Y gossiped up to version $Y_9$), (b) the receiving node (e.g., X) executes “$f()$” to synchronize the view, (c) when $N$ is large, this $O(N^3)$ scale-dependent process creates a backlog of new gossips, (d) thus X keeps gossipping only the latest (old) versions (e.g., $Y_1$), (e) as Y’s recent gossips are not propagated on time, other nodes (e.g., Z) mark Y as dead.

(d) Where are they located? The bugs are within the user-facing read/write calls (12 bugs) and operational protocols (40 bugs) such as block report, bootstrap, consistency repair, decommission, de-replication, distributed fsck, heartbeat, job recovery, log cleaning, rebalance, and region assignment. This suggests that scalability correctness is not merely about the user-facing paths. Large systems are full of operational paths that must be scale-tested as well.

(e) When do they happen? User-facing read/write protocols run “all the time” in deployment, hence are continuously tested. Operational protocols, however, are not frequently exercised. In a stable-looking cluster, scalability bugs can linger silently until the buggy operational protocols are triggered (akin to buggy error handling). For the bugs in user-facing calls, most were triggered by unique workloads such as large deletions or writes after decommission.

(f) How do scalability bugs impact users? Scalability bugs can cause both performance and availability problems. Although many of the bugs are in the operational protocols, they can cascade to user-visible impacts. For example, when nodes are incorrectly declared dead, some data become unreachable; or scale-dependent operations in the master node (e.g., in HDFS) can cause global lock contention, hence longer time to process user read/write requests.

(g) Why were the bugs not found before? First, the workloads and the necessary scales to cover the buggy protocols are not captured in the unittests as creating a scalable test platform is not straightforward [25]. Second, protocols might be scalable in design, but not in practice. Related to c6127 (Figure 2.1), the failure detector/gossiper [50] was adopted for its “scalable” design [54].
However, the design does not account for the gossip processing time during bootstrap/cluster-changes, which can be long, and the subsequent backlogs. To debug, the developers tried to “do the [simple] math” but failed [6]. Specific implementation choices such as overloading gossips with many other purposes (e.g., announcing boot/rebalance changes) deviate from the original design sketch, hence the need for scale-testing the implementation code at real scales.

(h) **Are scalability bugs easy to debug and fix?** The bugs took 1 month to fix on average with tens of back-and-forth discussions. One big factor of delayed fixes is the lack of budget for large test clusters as such luxury tends to only be accessible in large companies, but not to open-source developers [25]. Another factor is that debugging and fixing are not a single-iteration task; developers must repeatedly instrument the system and re-run at scale to pinpoint the root cause and test the patch.
CHAPTER 3

SCALECHECK

We now present the design of SCALECHECK, which is composed of two parts to achieve two goals: SFIND (§3.1), a program analysis that exposes scale-dependent loops to developers, and STEST (§3.2), a set of colocation techniques that enable hundreds of nodes to be colocated on one machine for testing. While STEST produces accurate bug symptoms in most cases, it does not deliver accurate results when all nodes are CPU intensive. For this, we introduce PIL (§3.3), an emulation technique that provides processing illusion.

3.1 SFIND

The first challenge to address is: how to find scale-dependent loops? Unfortunately, it is not trivial as such loops can span multiple functions and iterate many scale-dependent collections (iterable data-structure instances such as list). In Figure 3.1, the \(O(N^3)\) loops span 1000+ LOC, 3 classes, and 10 functions and iterate 3 scale-dependent collections. This difficulty motivates SFIND, a generic program analysis that helps developers pinpoint scale-dependent loops. Below are the three main steps of SFIND.

(1) **Auto-tagging of scale-dependent collections**: SFIND first automatically tags scale-dependent collections. This is done by growing the cluster and data sizes (e.g., add nodes and add files/blocks) in steps. After each step, we record the size of each instantiated collection. When all the steps are done, we check each collection’s growth tendency and mark as scale dependent those whose size increases as the cluster/data size grows.

This, however, is insufficient due to two reasons. First, there are collections that only grow when background/operational tasks are triggered (§2d); thus, we must also run all non-foreground tasks. Second, there are “ephemeral” collections (e.g., messages) whose content are scale-dependent but might have been garbage collected by the runtime. Given that the measurements are taken in
The partial code segment above depicts the $O(N^3)$ loops in Figure 2.1. SFIND automatically tags epStateMap, affectedRanges, and map as scale-dependent collections.

steps, garbage collection can happen in between them so these collections will not be detected consistently, thus this phase must be iterated multiple times to remove such noise.

For Java systems, we track heap objects and map them to their instance names by writing around 1042 LOC of analysis on top of Java language supports such as JVMTI [65] and Reflection [21]. This phase also performs a dataflow analysis to taint all other variables derived from scale-dependent collections. In our experience, by scaling out to just 30 nodes (30 steps), which can be done easily on one machine, scale-dependent collections can be clearly observed (though not the symptoms). This phase found 32 scale-dependent collections in Cassandra (three in Figure 3.1) and 12 in HDFS.

(2) Finding scale-dependent loops: With the tagging, SFIND then automatically searches for scale-dependent loops, specifically by tainting loops (for, while) as well as recursive functions that iterate through the scale-dependent collections, performing a control-flow analysis to construct the nested Big $O$ complexity of each loop, and identifying the loop contents (CPU/instructions only, IOs, or locks). With these steps, in Figure 3.1 for example, SFIND can mark applyStateLocally as an $O(N^3)$ function. The full list of algorithms can be found in §9.2.

We also cover a special “implicit loop” – a synchronized (locking) function in a node that is being called by all the peer nodes. A common example is in the master-worker architecture where all the $N$ worker nodes RPC into a master’s lock-protected function. When $N$ grows, there is
a potential of lock contention (congestion) to the function (examples are in §5.1). SFIND also handles such scenarios by tagging RPC classes and searching for functions called by the peer nodes.

(3) **Reporting and triaging:** SFIND finds 131 scale-dependent loops in Cassandra and 92 in HDFS, hence the need for triaging. For example, if a function $g$ has lower complexity than $f$, and $g$ is within the call path of $f$, then testing $f$ can be prioritized. For every nested loop to test, SFIND reports the relevant control- and data-flows from the outer-most to inner-most loop, along with the entry points (either client/admin RPCs or background daemon threads). The entry points are finally ranked by counting the number of spanned scale-dependent lines of code, the theoretical complexity (in terms of scale-dependent data structures), the number of IO operations (including reads/writes) and the number of blocking operations (including locking and operations that block waiting for a future result) in that path. The theoretical complexity is not by itself a complete indicator of potential bottlenecks. For example, an entry point reported with high complexity, e.g. $O(N^3)$, but with no IO/Blocking operations on its code path might not be as bottleneck prone as one reported with less complexity, e.g. $O(N)$, but many IO/Blocking operations on its code path. This ranking helps developers prioritize and create the necessary test workloads. For example, in Figure 3.1, the $O(N^3)$ path is only exercised if the cluster bootstraps from scratch when peers do not know about each other (hinted from the “if(!localStateMap.get())”, “onChange()”, “state==STATUS” and “val==NORMAL”). SFIND reports that this entry point spans over 6700 scale-dependent lines of code and performs over $20N$ IO and $4N$ blocking operations, which implies that it is likely to become a bottleneck as the cluster size grows and should be prioritized.

Creating test workloads from SFIND report is a manual process. Automated test generation is possible for single-machine programs/libraries [37], however, we are not aware of any work that automates such process in the context of real-world, complex, large-scale distributed systems. We put our work in the context of DevOps culture [59] where developers are testers and vice versa, which (hopefully) simplifies test workload creation.
3.2 STest

The next challenge is: how to test scale-dependent loops at real scales (hundreds of nodes) on one machine? Many scale-dependent loops were unfortunately not subjected to testing because existing unitest frameworks do not scale. Below we describe the hurdles to achieve a high colocation factor. Starting in Section 3.2.1, we began with black-box methods (no/small target system modification).

Unfortunately, we found that existing systems are not built with single-machine scale-testing in mind (the theme of this section); we faced many colocation bottlenecks (memory/CPU contentions and context switching delays) that limit large colocation. In Section §3.2.2, we will describe our solutions to achieve single-machine scale-testable systems with minimal changes. All the methods we use are summarized in Table 3.1 using Cassandra as an example. Abbreviations of our methods (e.g., NP, SPC, GEDA) are added for ease of reference in the evaluation.

3.2.1 Black-Box Approaches

• Naive Packing (NP): The easiest setup is (naively) packing all nodes as processes on a single machine. However, we did not reach a large colocation factor, which is caused by the following reasons.

(a) Memory bottlenecks: Many distributed systems today are implemented in managed languages (e.g., Java, Erlang) whose runtimes consume non-negligible memory overhead. Java and Erlang VMs, for example, use around 70 and 64 MB of memory per process respectively. We also tried running nodes as Linux KVM VMs and using KSM (kernel samepage merging) tool. Interestingly, the tool does not find many duplicate pages even though the VMs/processes are supposed to be similar (as reported elsewhere [63]). Overall, including Cassandra’s memory usage, per-node memory consumption reaches 100 MB. Thus, a 32-GB machine can only colocate around 300 nodes.

(b) Process context switches: Before we hit the memory bottleneck (e.g., reach 300 nodes), we observed that the target systems’ “inaccuracy” is already high when we colocate just 50 nodes. For
measuring inaccuracy, we measure several application-level metrics; for example, in Cassandra, if gossips should be sent every 1 second, but are sent every 1.3 second, then the inaccuracy is 30%. We use 10% as the maximum acceptable inaccuracy/event lateness. We noticed high inaccuracies even before we hit the CPU bottlenecks (i.e., CPU has not reached 90% utilization). We suspected that the process context switches could be the reasons.

(c) **Managed-language VM limitations:** We also found that managed-language VMs are backed by advanced services. For example, Erlang VMM contains a DNS service that sends heartbeat messages among connected VMs. When hundreds of Erlang VMs (one for each Riak node) run on one Erlang VMM, the heartbeat messages cause a “network” overflow that undesirably disconnects Erlang VMs (also reported in [39]). Naive packing is infeasible.

**Single-Process Cluster (SPC) + Network Stub:** To address the bottlenecks above, we deployed all nodes as threads in a single process. Surprisingly, our target systems are not easy to run in this “single-process cluster.” For example, Cassandra developers bemoan the fact that their gossip/fault-detector protocols are not adequately scale-tested [14, 27] because Cassandra (and many other systems) uses “singleton” design pattern for simplicity (but bad for modularity) [31]. That is, most global states are static variables that cannot be modularized to per-node isolated variables.

Our strawman attempt was a redesign to a more modular one, which costs us almost 3000 LOC (and no longer a black-box method); Cassandra developers also attempted a similar method to no

<table>
<thead>
<tr>
<th>#Nodes per PC</th>
<th>LOC added</th>
<th>Colocation bottlenecks</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>(a) Naive (NP)</em></td>
<td>50</td>
<td>Memory, proc. switch</td>
</tr>
<tr>
<td><em>(b) SPC</em></td>
<td>70</td>
<td>User-kernel switch</td>
</tr>
<tr>
<td><em>(c) SPC+Stub</em></td>
<td>120</td>
<td>Context switch</td>
</tr>
<tr>
<td><em>(d) GEDA</em></td>
<td>130</td>
<td>CPU</td>
</tr>
<tr>
<td><em>(e) GEDA+PIL</em></td>
<td>512</td>
<td>CPU</td>
</tr>
</tbody>
</table>

Table 3.1: Colocation strategies and bottlenecks (§3.2).
avail [14, 27]. We found another way: leveraging class loader isolation support from the language runtime [22], which is rarely used but fits SPC purpose. In Java systems, we can manipulate the class loader hierarchy such that a node’s main thread (and all child threads) use an isolated set of Java class resources, not shared with those belonged to other nodes, hence no target system modification. We implemented these custom class loaders using a child-first model [8] (as opposed to parent-first model, default in Java), which allow us to selectively isolate jar files in a per-node (per class loader) basis. The node main function/class is spawned as a thread, but this thread is attached to the per-node class loader. As a consequence, each thread created by this main thread will use the per-node class loader as its default class loader. Thus, when the per-node main class runs and creates a static instance, it will create an instance of an isolated class, as if each node has its own Java class resources, not shared with other node threads. This is possible due to the fact that at runtime, a class is identified by its fully qualified name and the classLoader that loaded it. Thus, two classes with the same name but loaded by two different class loaders are considered different. Very recently, we found that Cassandra developers also begin to develop a similar method to address this problem [7].

By SPC-ing Cassandra, we now hit a colocation limit of 70 nodes (Table 3.1b), but still have not reached the memory or CPU bottlenecks. We suspected thread and/or user-kernel context switching as a root cause. We removed the latter by creating a generic network stub that (de)marshalls inter-node messages and skips the OS. This stub is also helpful in reducing network memory footprints under higher colocation. For example, in Voldemort, the nodes communicate via Java NIO [24] which is fast but contains buffers and connection metadata that take up memory space and prevent >200-node colocation (more in §5.4). For Cassandra, the network stub allows up to 120-node colocation (Table 3.1c).
Adding network stub is our last black-box approach as we found no other way to reduce thread context switching in a black-box way. In fact, we observed a massive thread context switching issue. In P2P systems such as Cassandra, *each* node spawns *a* thread to listen from *a* peer. Thus, just for messaging, there are $N^2$ threads to manage for the whole cluster. This can be solved by using `select()`-like system call [20], which would reduce the problem to $N$ threads. However, we still observed around $N \times 26$ active threads – each node still runs multiple service stages (gossiper, failure detector, etc.), each can be multi-threaded. A high colocation factor will spawn thousands of threads.

**Global Event Driven Arch. (GEDA):** To address the problem, we must redesign the target system, but with minimal changes. We leverage the staged event-driven architecture (SEDA) [71] (Figure 3.2a), common in server code, in which each service/stage (in each node) exclusively has an event queue and a thread pool. In STEST mode, we convert SEDA to a *global-event driven architecture* (GEDA; Figure 3.2b). That is, for every stage, there is only one queue and one thread pool for the *whole* cluster. As an example, let’s consider a periodic gossip service. With 500-node colocation, there are 500 threads in SPC, each sending a gossip every second. With GEDA, we only deploy a few threads (matched with the number of available cores) shared among all the nodes for sending gossips. As another example, for gossip processing stage, there is only one
global gossip-receiving queue shared among all the nodes.

GEDA works with a minimal code change to the target system. Logically, as events are about to be *enqueued* into the original per-node event queues (1 in Figure 3.2), we redirect them to GEDA-level event queues, to be later processed by GEDA worker threads. This only requires \( \sim 10 \) LOC change per stage (as we use aspect-oriented programming [2]). While simple, care must be taken for single-threaded/serialized stage. For example, Cassandra’s gossip processing is intentionally single-threaded to prevent concurrency issues. This is illustrated in case 2 in Figure 3.2 where the per-node stage is serialized (i.e., \( y \) must be processed after \( x \)). Here, *if* the events are forwarded down during *enqueue*, GEDA’s multiple threads will break the program semantic (e.g., \( x \) and \( y \) can be processed concurrently). Thus, for single-threaded/serialized stage, we must interpose at *dequeue* time (3 in Figure 3.2), which costs \( \sim 50 \) LOC change per stage. Serialized stages (or in general, thread-count restricted stages) require acknowledgment from the perspective of integration, otherwise processing semantics could be broken. For size restricted stages we distinguish two cases: (1) includes fixed-size executors (executors where the number of threads is not a parameter) and (2) single thread custom implementations (a thread that constantly pulls from a queue). For now, we have only found single threaded executors for (1). For (2), it is sometimes necessary to add conditional blocks on code to make sure this semantics are not broken, thus the increment on LOC’s.

Adding GEDA to Cassandra only costs us 581 LOC (Table 3.1d) and is simple; the same 10-50 LOC method above is simply repeated across all the stages. Overall, GEDA does not change the logic of the target systems, but successfully removes some delays that should have never existed in the first place, as if the nodes run exclusively on independent machines. For HDFS tests, GEDA enables 512-node colocation (§5.4) but for some Cassandra tests, it only enables around 130-node colocation (Table 3.1d), which we elaborate in the next section.
Figure 3.3: **SCALECHECK complete automated flow (§3.3.3).** "SCk" represents SCALECHECK. The left-most figure illustrates testing in real deployments, where testing time is fast (T) but requires N machines. Stages (a) to (d) reflect the automated SCALECHECK process as described in Section 3.3.3. STest mez in stage (c) runs on one machine but will take some time (>T). STest PIL in stage (d) still runs on one machine but only consumes a similar time as in deployment testing (T+ε) and can be replayed numerous times.

### 3.3 Processing Illusion (PIL)

Finally, the last challenge we address is: how to produce accurate results (*i.e.*, the same bug symptoms observed in real-scale deployment) when colocating hundreds of CPU-intensive nodes? We found that STest is sufficient for accurately revealing bug symptoms in scale-dependent lock-related loops or IO serializations, as these root causes do not contend for CPUs. For CPU-intensive loops, STest is also sufficient for master-worker architecture where only one node is CPU intensive (*e.g.*, HDFS master).

However, for CPU-intensive loops in P2P systems such as Cassandra, where *all* nodes are busy, the bug symptoms reported by STest are not accurate. For example, for Cassandra issue #c6127 (§2a), in 256-node real deployment, we observed around 2000 flappings (the bug symptom) but 21,000 flappings in STest. The inaccuracy gets worse as we scale; with N CPU-intensive nodes on a C-core machine, roughly N/C nodes contend on a given core.

To address this, we need to emulate CPU-intensive processing by supplementing STest with *processing illusion* (PIL), an approach that replaces an actual processing with `sleep(t)`. For example, for c6127, we can replace the expensive gossip-stage-changes processing (see Figures 2.1 and 3.1), with `sleep(t)` where t is an accurate timing of how long the processing takes.

The intuition behind PIL is similar to the intuition behind other emulation techniques. For example, Exalt provides an illusion of storage space; their insight was “how data is processed is not affected by the content of the data being written, but only by its size” [70]. Similarly, PIL
provides an illusion of compute processing; our insight is that “the key to computation is not the intermediate results, but rather the execution time and eventual output.” In other words, with PIL, we will still observe the overall timing behaviors and the corresponding impacts accurately.

PIL might sound outrageous, but it is feasible as we address the following concerns: how a function (or code block) can be safely replaced with sleep() without changing the whole processing semantic (§3.3.1) and how we can produce the output and predict the timing “τ” if the actual compute is skipped (§3.3.2)?

### 3.3.1 PIL-Safe Functions

Our first challenge is to ensure that functions (or code blocks) can be safely replaced with sleep(), but still retain the cluster-wide behavior and unearth the bug symptoms. We name such functions as “PIL-safe functions.” We identify two main characteristics of such functions: (1) Memoizable output: a PIL-safe function must have a memoizable (deterministic) output based on the input of the function. (2) Non-pertinent IOs: if a function performs local/remote disk IOs that are not pertinent to the correctness of the corresponding protocol, the function is PIL-safe. For example, in c6127, there is a ring-table checkpoint (not shown) needed for fault tolerance but is irrelevant (never read) during bootstrapping.

We extend SFIND to SFINDPIL, which includes a static analysis that finds code blocks in scale-dependent loops that can be safely PIL-ed. SFINDPIL analyzes the content of each loop in functions related to the relevant cluster state and checks for two cases: (1) The loop performs operations that affect the cluster state, so we need to insert pre-memoization and replay code to record/reconstruct the cluster state (more detail in §9.3). We consider all variables involved in the execution of a target protocol as relevant states. While our static analysis tool eases the identification of these variables, programmer intervention can help for additional verification. In (2), the loop performs non-pertinent operations only (such as IO). In this case, we can automatically replace the loop with a sleep call without affecting the behavior of the protocol.
3.3.2 Pre-Memoization (with Determinism)

As PIL-safe functions no longer perform the actual computation, the next question to address is: how do we manufacture the output such that the global behavior is not altered (e.g., rebalancing protocol should terminate successfully)? For functions with no pertinent outputs, we just need to do time profiling but not output recording. For functions with pertinent outputs, our solution is pre-memoization, which records input-output pairs and the processing time, specifically a tuple of three items (ByteString in, out, long nanoSec) indexed by hash(in)), which represent the to-be-modified variables before and after the function is executed and the processing time, respectively (Figure 3.3b).

Another challenge encountered is non-determinism: the state of each node (the input) depends on the order of arriving messages (which are typically random). Let’s consider Riak’s bootstrap+rebalance protocol where eventually all nodes own a similar number of partitions. A node initially has an unbalanced partition table, receives another partition table from a peer node, then inputs it to a rebalance function, and finally sends the output to a random node via gossiping. Every node repeats the same process until the cluster is balanced. In a Riak cluster with $N=256$ and $P=64$, there are in total 2489 rebalance iterations with a set of specific inputs in one run. Another run of the protocol will result in a different set of inputs due to gossip randomness. Our calculation shows that there are $(N^NP)^2$ possible inputs.

To address this, during pre-memoization, we also record non-determinism such as message orderings such that order determinism is enforced during replay. For example, across different runs, a Riak node now receives gossips from the same sequence of nodes. With order determinism, pre-memoization and SCALECHECK work as follow: (1) We first run the whole cluster on a real deployment and interpose sleep-safe functions. (2) When sleep-safe functions are executed, we record the inputs and corresponding outputs to a memoization database (SSD-backed files). (3) During this pre-memoization phase, we record message non-determinism (e.g., gossip send-receive pairs and their timings). (4) After pre-memoization completes, we can repeatedly run
SCALECHECK wherein order determinism is enforced (e.g., no randomness), sleep-safe functions replaced with PIL, and their outputs retrieved from the memoization database. Note that steps 1-3 are the only steps that require real deployment.

Other than this, similar to the theme in the previous section that existing systems are not amenable to single-machine testing, we found similar issues such as the use of wall-clock time which essentially incapacitates memoization and replay. Here, we convert wall-clock time to “cluster start time + elapse time” in 296 LOC (Table 3.1e).

### 3.3.3 Putting It All Together

Figure 3.3a-d summarizes the complete four stages of SCALECHECK: (a) SFIND searches for scale-dependent loops which helps developers create test workloads. (b) For test workloads that show CPU busyness in all nodes, SFIND\textsubscript{PIL} finds PIL-safe functions and inserts our pre-memoization library calls. Next, ST\textsubscript{EST} now works in two parts. (c) ST\textsubscript{EST}\textsubscript{mez} (without PIL) will run the test on a real cluster, but just one time, to pre-memoize PIL-safe functions and store the tuples to a SSD-backed database file. (d) ST\textsubscript{EST}\textsubscript{PIL} (with PIL) will then run by having SFIND\textsubscript{PIL} remove the pre-memoization library calls, replace the expensive PIL-safe function with \texttt{sleep(t)}, and insert our code that constructs the memoized output data. SCALECHECK also records message ordering during ST\textsubscript{EST}\textsubscript{mez} and replays the same order in ST\textsubscript{EST}\textsubscript{PIL} (not shown).

As another benefit, SCALECHECK can also ease real-scale debugging efforts. First, the only step that consumes more time is the no-PIL pre-memoization phase (Figure 3.3c), up to 6x longer time than real-deployment testing (§5.5). However, this is only a one-time overhead. Most importantly, developers can repeatedly re-run ST\textsubscript{EST}\textsubscript{PIL} (Figure 3.3d) as many times as needed (tens of iterations) until the bug behavior is completely understood. In ST\textsubscript{EST}\textsubscript{PIL}, the protocol under test runs in a similar duration as if all the nodes run on independent machines.

Second, some fixes can be tested by only re-running the last step; for example, fixes such as changing the failure detector \( \Phi \) algorithm (for c6127), caching slow methods (c3831), changing
lock management (c5456), and enabling parallel processing (v1212). However, if the fixes involve a complete redesign (e.g., optimized gossip processing in c3881, decentralized to centralized rebalancing in r3926), $\text{STEST}_{mez}$ must be repeated.
CHAPTER 4
IMPLEMENTATION AND APPLICATION

Table 4.1 quantifies the application of SCALECHECK techniques to a variety of distributed systems, Cassandra [54], HDFS [17], Riak [29], and Voldemort [28]. The major system-specific change is achieving “STEST-able systems” (i.e., supporting SPC and GEDA), which range between 179 to 918 LOC (less than 1% of the target code size). This is analogous to how file systems code are modified to make them “friendlier” to fsck [51, 60]. The rest is the generic SFIND and STEST library code (pre-memoization, auto PIL insertion, message order determinism support, AspectJ utilities). SFIND was built with Eclipse AST Parser [11] to support Java programs. We leave porting to Erlang’s parser [12, 13] as future work.

We emphasize that the integration effort was relatively easy for master-worker architectures (e.g., 179 LOC for HDFS) and for event-driven languages (e.g., 217 LOC for Riak). The integration costs to Java systems is higher compared to Erlang systems because Java runtime is not event-based by default (see Erlang discussion in §3.2). A more generic approach such as “making Java runtime support event driven without changing existing threads” (such as in Erlang) can help, but is another long research.

We repeated a new integration to a vanilla version of Cassandra entirely from scratch, which took less than a day. This is because our non-intrusive SPC and GEDA approach require only a small modification. As mentioned before, we expect SCALECHECK integration can be done seamlessly with today’s DevOps practice [59], where developers are testers and testers are developers.

We show the generality of SCALECHECK with two major efforts. First, we scale-checked a total of 18 protocols: 8 Cassandra (bootstrap, scale-out, decommission, drain, partial failure, snapshot, upgrade, and various administration statistic related protocols), 8 HDFS (write, decommission, full and incremental block reports, snapshot, volume failure, refresh, management and partial failures), 1 Riak (bootstrap+rebalance), and 1 Voldemort (rebalancing). A protocol can be built on top of other protocols (e.g., bootstrap on gossip and failure detection protocols).
Table 4.1: Integrations LOC (§4).

For these, we wrote new scale-test cases to exercise scale-dependent loops that are not covered in existing unit tests. For the new test cases, we only added 429-1498 LOC. Note that existing unittests focus on logical/functional correctness, but they do not perform scale-testing, hence our new test cases.

Second, for exposing known bugs, we applied SCALECHECK to a total of 10 earlier releases: 4 Cassandra, 4 HDFS, 1 Riak, and 1 Voldemort old releases. For finding unknown bugs, we also ran SCALECHECK on recent releases of the four systems.
CHAPTER 5
EVALUATION

We now evaluate SCALECHECK: Is SCALECHECK effective in exposing scalability bugs (§5.1-5.2), accurate (§5.3), scalable and efficient (§5.4-5.5)? We compare SCALECHECK with real deployments of 32 to 512 nodes, deployed on at most 128 machines (testbed group limit), each has 16-core AMD Opteron(tm) with 32-GB DRAM. Our target protocols only make at most 2 busy cores per node, which justifies why we pack 8 nodes per one 16-core machine for the real deployment.

5.1 Exposing Scalability Bugs

Table 5.1 lists the 10 real-world bugs we use for benchmarking SCALECHECK. We chose these 10 bugs (among the 55 bugs we studied) because the reports contain detailed descriptions of the bugs, which is important for us to create the “input” (i.e., the test cases). Figure 5.1 shows the accuracy of SCALECHECK in exposing the 10 bugs using the “bug-symptom” metrics in Table 5.1.

Results summary: First, SCALECHECK is effective and accurate in exposing scalability bugs, some of which only surface in 256+ nodes. As shown, for Cassandra and Riak bugs where all nodes are CPU intensive, PIL is needed for accuracy (SCk+PIL vs. Real lines in Figures 5.1a-d), but for the rest, STEST suffices (SCk vs. Real in 5.1e-f).

Second, SCALECHECK can help developers prevent recurring bugs; the series of Cassandra bugs (as described later below) involves the same protocols (gossip, rebalance, and failure detector) and create the same symptom (high #flaps). As code evolves, it can be continuously scale-checked with SCALECHECK.

Third, different systems of the same type (e.g., key-value stores, master-worker file systems) implement similar protocols. The effectiveness of SCALECHECK methods in scale-checking the different protocols above can be useful to many other distributed systems.
Figure 5.1: SCALECHECK effectiveness in exposing scalability bugs (§5.1). “SCk” represents SCALECHECK. The bugs are listed in Table 5.1. The x-axis represents the number of nodes (N). The figure title describes the y-axis, i.e., the bug symptom metrics as recorded in “Real” deployment vs. SCALECHECK. For Cassandra and Riak bugs (a-d), where all nodes are CPU-intensive, the bug symptoms are inaccurate without PIL (“SCk” lines). However, with PIL ("SCk+PIL" lines), the bug symptoms are relatively accurate as in the real deployment scenarios. For Voldemort and HDFS bugs (e-h), where there is no concurrent CPU busyness, PIL is not needed.

Bug descriptions: We now describe the 10 reproduced bugs in more detail.

(a) Figure 5.1a: In c3831 [5] protocol commission/decommission has complexity \(\sim O(n^4 \times (\log(n))^3)\) (n is the number of nodes) because for each entry in a message, a node needs to calculate and sort the current view of tokens. Moreover, the method StorageService#calculatePendingRanges (established as the culprit) is called multiple times for one gossip message. This causes CPU spikes, gossip backlog (gossip messages tend to accumulate, given that gossip processing is single threaded), and flapping (a node is declared “dead” incorrectly). The main collections iterated in
### Table 5.1: Bug benchmark (§5.1)

The table lists the scalability bugs we use for benchmarking SCALECHECK. “c” stands for Cassandra, “h” for HDFS, “r” for Riak, and “v” for Voldemort. The “N” column represents the #nodes for the bug symptoms to surface. The “Metric” column lists the quantifiable metrics of the bug symptoms; $T_{Comp}$, $T_{Lock}$, and $Q_{Size}$ denote computation time, lock time, and queue size, respectively. The “$T_m$” and “$T_{pil}$” columns quantify the duration of the pre-memoization (TEST$_{mez}$) and PIL replay (TEST$_{PIL}$) stages when $N \geq 256$, as discussed in §5.5. “–” implies PIL is unnecessary.

<table>
<thead>
<tr>
<th>Bug#</th>
<th>N</th>
<th>Protocol</th>
<th>Metric</th>
<th>$T_m$</th>
<th>$T_{pil}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>c6127 [6]</td>
<td>≥256</td>
<td>Bootstrap</td>
<td>#flaps</td>
<td>2h</td>
<td>15m</td>
</tr>
<tr>
<td>c3831 [5]</td>
<td>≥256</td>
<td>Decomm.</td>
<td>#flaps</td>
<td>17m</td>
<td>9m</td>
</tr>
<tr>
<td>c3881 [4]</td>
<td>≥64</td>
<td>Add nodes</td>
<td>#flaps</td>
<td>7m</td>
<td>5m</td>
</tr>
<tr>
<td>c5456 [3]</td>
<td>≥256</td>
<td>Add nodes</td>
<td>#flaps</td>
<td>16m</td>
<td>4m</td>
</tr>
<tr>
<td>r3926 [30]</td>
<td>≥128</td>
<td>Rebalance</td>
<td>$T_{Comp}$</td>
<td>6h</td>
<td>2h</td>
</tr>
<tr>
<td>v1212 [32]</td>
<td>≥128</td>
<td>Rebalance</td>
<td>$T_{Comp}$</td>
<td>22h</td>
<td>–</td>
</tr>
<tr>
<td>h9198 [18]</td>
<td>≥256</td>
<td>Blk. report</td>
<td>$Q_{Size}$</td>
<td>8m</td>
<td>–</td>
</tr>
<tr>
<td>h4061 [16]</td>
<td>≥256</td>
<td>Decomm.</td>
<td>$T_{Lock}$</td>
<td>6h</td>
<td>–</td>
</tr>
<tr>
<td>h1073 [15]</td>
<td>≥512</td>
<td>Pick nodes</td>
<td>$T_{Comp}$</td>
<td>1m</td>
<td>–</td>
</tr>
<tr>
<td>h395 [19]</td>
<td>≥512</td>
<td>Blk. report</td>
<td>$T_{Comp}$</td>
<td>5m</td>
<td>–</td>
</tr>
</tbody>
</table>

The main collections iterated in this scenario (as reported by our tools) are endpoint-state maps (maps that contain the metadata of each peer, fields of the classes `Gossiper`, `GossipDigestAck`, `GossipDigestAck2`), a map of unreachable endpoints (each unreachable peer ip address, field of the class `Gossiper`) and token (peer data) metadata (fields of the class `TokenMetadata`).

**b) Figure 5.1b:** c3881 [4] is similar to c3831. After patching c3831 developers realized that even if the maximum practical size of a cluster was improved the solution was not robust yet.

**c) Figure 5.1c:** In c5456 [3] protocol commission/decommission acquires a lock for computation with complexity $\sim O((np)^2 \times \log(np))$ (n is the number of nodes and p is number of vnodes per node) because for each vnode in a message, the algorithm keeps sorting tokens in nested loops. This lock blocks gossip processing causing flapping (a node is declared “dead” incorrectly). The main collections iterated in this scenario (as reported by our tools) are endpoint-state maps (maps that contain the metadata of each peer, fields of the classes `Gossiper`, `GossipDigestAck`, `GossipDigestAck2`), a map of unreachable endpoints (each unreachable peer ip address, field of the class `Gossiper`) and token (peer data) metadata (fields of the class `TokenMetadata`).

**d) Figure 5.1d:** In r3926 [30] the membership protocol requires that all nodes share the exact
same view of the partition table (a table that contains the relationship between partition and owner) and also that this table is “balanced”, meaning that each node should own a similar number of partitions. For this, whenever a message is received, a full “rebalance” algorithm is executed ($\sim O(n^3)$) and the resulting output is communicated to another node, which in time will perform a full “rebalance” operation. Given that this operation has a high asymptotic complexity and that is performed each time a node receives a new message, as the cluster grows convergence time tends to be long and while happening cpu usage tends to get higher. The main collection iterated in this scenario is a map (defined in module `riak_core_ring` of the `riak_core` package) that contains the mapping between each partition (key) and each owner (value). This data structure is both iterated explicitly (via for loop constructions) and recursively (common in Erlang).

**Figure 5.1e:** In v1212 [32] when new nodes join the cluster, the existing nodes will move part of key partitions to the new nodes. As per implementation, each gossip message moved partitions one by one, incrementing the number of messages and incurring in network overhead, severely affecting performance.

**Figure 5.1f:** In h9198 [18], each datanode will send incremental block report to namenode for every new block operation (e.g. creation or deletion). These reports are sent at the same time by each datanode, so the namenode could be processing $n*b$ (where $n$ is the number of datanodes and $b$ is the number of blocks) blocks, which severely degrades performance due to excessive (global) lock contention (`FSNamesystem` lock) from multiple IPC handler threads. This is a case of implicit scale dependency for the method `FSNameSystem#processIncrementalBlockReport` (is not executed within a loop, but is executed by every datanode call) and it involves datanode related collections (a map at the class `DatanodeManager`) and lists of blocks (in general part of the communication protocol classes, like `StorageBlockReportProto` or `LongDecoder`).

**Figure 5.1g:** In h4061 [16], when a datanode being decommissioned, the blocks that belong to it need to be moved/replicated. During this replication period, `DecommissionedMonitor` thread (namenode) will check the replication status of every block periodically and determine whether a
decommissioning datanode is now safe to terminate (all blocks have been replicated by a “live” datanode). This operation is blocking and holds FSNamesystem (global) lock, thus when the number of blocks grows the locking could severely degrades performance. The main collections involved in this case are datanode related collections (a map at the class DatanodeManager) and lists of blocks (located in classes PendingReplicationBlocks, UnderReplicatedBlocks or LongDecoder).

(h) Figure 5.1h: In h1073 [15], the namenode needs to choose a pipeline (of size $r$, where is the replication factor) for each file written. Each pipeline selection involves costly sorting, string comparison and grouping. In a scenario with multiple writes and a large number of datanodes the pipeline selection algorithm was causing connection timeouts (long processing time) and high CPU usage. The main collections involved in this case are datanode related collections (a map at the class DatanodeManager and a map of replicas at the class ReplicaMap).

5.2 Discovering Unknown Bugs

We also integrated SCALECHECK to recent stable versions of Cassandra, HDFS, Riak, and Voldemort, and found 1 unknown bug in Cassandra and 3 bugs in HDFS.

For the first bug (Cassandra), SFIND pointed us to the method Gossiper#applyStateLocally, which is invoked for every gossip message processed. Given that the complexity of this method is
$\sim O(n^3p)$ (where $n$ is the number of nodes and $p$ is the number of partitions), the processing time of each message tends to be long (e.g. for a 256-node cluster, the average processing time is 30 seconds). This expensive computation happens whenever cluster membership changes (e.g. when adding or removing nodes) and produces cluster instability (as shown in figure 5.2 (a)) in the form of flappings (a node is declared “dead” incorrectly).

In the second bug (HDFS) SFIND pointed us to a code path involving the methods (1) `DatanodeManager#refreshNodes` (holding a global lock) and (2) `DatanodeManager#refreshDatanodes` with complexity $\sim O(nb)$ (where $n$ is the number of datanodes and $b$ is the number of blocks per datanode). As reported, this could render the namenode unresponsive when several “fat” datanodes (containing many blocks) are recommissioned.

In the third bug (HDFS, figure 5.2 (b)) SFIND pointed us to a data structure (`AbstractINodeDiffList`) involved in snapshot diff reports (given two snapshots, get the difference between them). We found that the `snapshotDiff` operation has a complexity of $\sim O(nb)$ (where $n$ is the number of datanodes and $b$ is the number of blocks per datanode), involves a recursive operation (on the method `DirectorySnapshottableFeature#computeDiffRecursively`) and also holds a global lock.

In the fourth bug (HDFS, figure 5.2 (c)) SFIND pointed us to a code path involving the methods (1) `FSNameSystem#metaSave` (holding a global lock) and (2) `BlockManager#metaSave` with complexity $O(nb)$ where $n$ is the number of datanodes and $b$ is the number of blocks per datanode). As reported, this code path (executed as part of an administration command) could render the namenode unresponsive if not used with care. The impact of this operation is even more dangerous when there are many under replicated blocks (e.g. after a portion of the cluster fails or in the presence of network failures).

For Riak and Voldemort, we found that their latest-stable bootstrap/rebalance protocols do not exhibit any scalability bug up to 512 nodes.
Figure 5.3: **Cassandra internal metrics (§5.3).** Above are the metrics we measured within the Cassandra bootstrap protocol for measuring SCALECHECK accuracy (Figure 5.4). “f” represents “a function of” (i.e., an arbitrary function).

### 5.3 Accuracy

The goal of our next evaluation is to show that PIL-infused SCALECHECK mimics similar behaviors as in real-deployment testing and is accurate not only in the final bug-symptom metric but also in the detailed internal metrics. For this, we collected roughly 18 million values. For space, we only focus on c6127 [6] (see §2a).

Figure 5.3a-d shows the internal metrics that we measured within Cassandra failure detection protocol for *every pair* of nodes; the algorithm runs on every node A for every peer B. Figures 5.4a-d compare in detail the accuracy of STTEST without PIL (“SCk”) and STTESTPIL with PIL (“SCk+PIL”), respective to the real-deployment testing (“Real”).

(a) Figure 5.4a shows the total number of flaps (alive-to-dead transitions) observed in the whole cluster during bootstrapping. STTEST by itself will not be accurate if all nodes are CPU intensive (§3.3). However, with PIL, SCALECHECK closely mimics real deployment scenarios. Next, Figure 5.3a defines that \#flaps depends on \( \Phi \) [50]. Every node A maintains a \( \Phi \) for a peer B (a total of \( N \times (N-1) \) variables to monitor).

(b) Figure 5.4b shows the maximum \( \Phi \) values observed for every peer node; for graph clarity, from here on we only show with-PIL results. For example, for the 512-node setup, the whisker plots show the distribution of the maximum \( \Phi \) values observed for each of the 512 nodes. As shown, the larger the cluster, more \( \Phi \) values exceed the threshold value of 8, hence the flapping.
Figure 5.3b points that $\Phi$ depends on the average inter-arrival time of when new gossips about B arrives at A ($T_{avgGossip}$) and the time since A heard the last gossip about B ($T_{lastGossip}$). The point is that $T_{lastGossip}$ should not be much higher than $T_{avgGossip}$.

(c) Figure 5.4c shows the whisker plots of gossip inter-arrival times ($T_{lastGossip}$) that we collected for every A-B pair (millions of gossips as a gossip message contains $N$ gossips of the peer nodes). The figure shows that in larger clusters, new gossips do not arrive as fast as in smaller clusters, especially at high percentiles. Figure 5.3c shows that $T_{lastGossip}$ depends on how far B’s new gossips propagate through other nodes to A (#hops) and the gossip processing time in each hop ($T_{gossipExec}$). The latter ($T_{gossipExec}$) is essentially the state-update processing time ($T_{stateUpdate}$), triggered whenever there are state changes.

(d) Figure 5.4d (in log scale) shows the whisker plots of the state-update processing time ($T_{stateUpdate}$). In the 512-node setup, we measured around 25,000 state-update invocations. The figure shows that at high percentiles, $T_{stateUpdate}$ is scale dependent (the culprit). As shown in Figure 5.3d, $T_{stateUpdate}$ complicatedly depends on a scale-dependent 2-dimensional input ($Size_{ringTable}$ and $Size_{newStates}$). A node’s $Size_{ringTable}$ depends on how many nodes it knows, including the partition arrangement ($\leq N \times P$) and $Size_{newStates}$ ($\leq N$), which increases as cluster size grows.

### 5.4 Colocation Factor

This section shows the maximum colocation factor SCALECHECK can achieve as each technique is added one at a time on top of the other. To recap, the techniques are: single-process cluster (SPC), network stub (Stub), global event driven architecture (GEDA), and processing illusion (PIL). The results are based on a 16-core machine.

**Maximum colocation factor ("MaxCF"):** A maximum colocation factor is reached when the system behavior in SCALECHECK mode starts to “deviate” from the real deployment behavior. Deviation happens when one or more of the following bottlenecks are reached: (1) high average
Figure 5.4: **Accuracy in exposing c6127 (§5.3).** The figures represent the metrics presented in Figure 5.3, measured in real deployment (“Real”) and in SCALECHECK (“SCk”) with different cluster sizes (32, 64, 128, 256, and 512 in the x-axis). The y-axes (the metrics) are described in the figure titles.

CPU utilization (>90%), (2) memory exhaustion (nodes receive out-of-memory exceptions and crash), and (3) high event “lateness.”

Queuing delays from thread context switching can make events late to be processed, although the CPU utilization is not high. We instrument our target systems to measure event lateness of relevant events (as described in §3.2.2). We use 10% as the maximum acceptable event lateness. Note that the residual limiting bottlenecks come from the main logic of the target protocols, not removable with general methods.

**Results and observations:** Figure 5.5 shows different sequences of integration to our four target systems and the resulting maximum colocation factors. We make several important observations from this figure.

First, when multiple techniques are combined, they collectively achieve a high colocation factor (up to 512 nodes for the three systems respectively). For example, in Figure 5.5a, without using PIL in Cassandra, MaxCF only reaches 136. But with PIL, MaxCF significantly jumps to 512.
Figure 5.5: **Maximum colocation factor (§5.4).** The colocation factor reached as each technique is added.

When we increased the colocation factor (+100 nodes) beyond the maximum, we hit the residual bottlenecks mentioned before; at this point, we did not measure MaxCF with small increments (e.g., +1 node) due to time limitation.

Second, distributed systems are implemented in different ways. Thus, integrations to different systems face different sequences of bottlenecks. To show this, we tried different sequences of integration sequences. For example, in Cassandra (Figure 5.5a), our integration sequence is +SPC, +Stub, +GEDA, and +PIL (as we hit context switching overhead before CPU). For Riak (Figure 5.5b), we began with PIL as we hit CPU limitation first before hitting Erlang VMM network overflow which requires SPC (§3.2.1), and Riak does not require GEDA because Erlang, as an event-driven language, manages thread executions as events. Upon program start-up, Erlang (implicitly) starts a scheduler process per core. When “Erlang processes” (threads) run, their events are automatically scheduled; for example, when a process $X$ sends a message to $Y$, the message is automatically put to a runtime queue and scheduled. Thus, Erlang processes behave like event handlers, while the scheduler processes are synonymous to GEDA threads.

For Voldemort (Figure 5.5c), we began with SPC and then network stub to reduce Java VM and Java NIO memory overhead respectively, and PIL so far is not needed as the tested workload does not involve parallel CPU-intensive operations. For HDFS (Figure 5.5d), we only need SPC.
and GEDA but not PIL as only the master node that is CPU intensive (but not the datanodes).

Finally, it is the combination of all techniques that make SCALECHECK effective. For example, while in Figure 5.5a we apply the sequence of SPC+Stub+GEDA+PIL resulting in PIL as the dominant factor, in another experiment we applied a different sequence PIL+SPC+Stub and failed to hit 512 nodes, not until GEDA is added and becomes the dominant factor.

### 5.5 Pre-Memoization and Replay Time

The \( T_m \) and \( T_{pil} \) columns in Table 5.1 on page 25 quantifies the duration of the pre-memoization (\( T_{mez} \)) and PIL-based replay (\( T_{PIL} \)) stages when \( N \geq 256 \). For example, for CPU-intensive bugs such as c6127, the pre-memoization time takes 2 hours while the PIL-based replay is only 15 minutes (similar to the real-deployment test); for r3926, it is 6 vs. 2 hours. Pre-memoization does not necessarily take \( N \times \) longer time because one node only consumes 2 cores (while the machine has 16 cores) and also not every node is busy all the time.

### 5.6 Test Coverage

SFIND labeled 32 collections in Cassandra and 12 in HDFS as scale dependent. From these, SFIND identified 131 and 92 scale-dependent loops in Cassandra and HDFS (out of more than 1500 and 1900 total loops) respectively. So far, we have tested 57 (44%) and 64 (69%) of the loops in Cassandra and HDFS. The time-consuming factor is the manual creation of new test cases that will exercise the loops (see end of §3.1).

We emphasize that SFIND is not a bug-finding tool, hence the reason why we do not report false positives.
CHAPTER 6
DISCUSSION

• **What are the current limitations of SCALECHECK?** At the moment, our work focuses on scale-dependent CPU/processing time (§??a), and the “scale” here implies cluster size. However, there are other scaling problems that lead to I/O and memory contentions [44, 67, 69], usually caused by the scale of load [36, 46] or data size [62]. For emulating data size, we are only aware of one work, Exalt [70], which is orthogonal to SCALECHECK (more in §7). In our bug study, we learn that some load or data-size related bugs can be addressed with accurate modeling [46] (e.g., $d$ dead nodes will add $d/(N-d)$ load to every live node) and some others can already be reproduced with a single machine (e.g., loading as much file metadata to check the limit of HDFS memory bottleneck [69]). Nevertheless, we will continue our study of these other scaling dimensions, especially as scaling bugs in datacenter distributed systems is not a well-understood problem.

• **Can SCALECHECK reach \( >1000 \)-node colocation factor? Can SCALECHECK run on multiple machines?** So far, SCALECHECK is limited by the single machine’s resources. To increase colocation factor, a higher-end machine can be used. Another approach is to extend SCALECHECK to run on multiple machines. However, this means that we need to enable back the networking library, which originally already caused a colocation bottleneck. We leave this challenge for future work. However, a single-machine framework integrates well to the de-facto unit-test style.

• **Is SFIND sufficient (without STEST) to reveal scalability bugs?** Building a program analysis that covers all paths and understands the cascading impacts is challenging. Not all scale-dependent loops imply buggy code. For example, in c6127 [6], if Cassandra processes gossips in a multi-threaded manner, the long processing time might not cascade to failures. Nevertheless, for major types of bugs, developers tend to add new unit tests.

• **Can STEST\textsubscript{PIL} run without STEST\textsubscript{mez}?** Ideally, the fast STEST\textsubscript{PIL} (replay with PIL) should run without the long STEST\textsubscript{mez} (pre-memoization phase), possible only if we can construct all
input-output pairs. However, in the context of large-scale, non-deterministic systems, constructing all possible pairs requires an “infinite” time and storage space, because node states (the input) typically depend on the order in which messages arrive (mostly non-deterministic in real deployments). Let us consider Riak’s rebalance protocol, where each node gossips randomly its partition table, until a globally balanced state is reached. With $N=256$ and $P=64$ (#key-partitions/node), we counted 2489 rebalance iterations with unique input-output pairs in one complete run. In another run, a different set of input-output pairs is produced due to the non-determinism. Our calculation estimates $(NP)^2$ (“infinite”) possible pairs. In contrast, STest$_{mez}$ only needs to store around 2500 input-output pairs (1.3 GB of memoized data).

- **Can function-level performance profiling hints of scalability bugs?** It is possible, but in our experience, using average performance numbers do not help given that the processing time can vary wildly on some scenarios (e.g. can depend on if/else blocks).

- **What are other advantages of SCALECHECK?** Some bugs are dependent on the CPU specification of the machine. For example, the bug in c6127 produces more flappings in slower CPUs. With single-machine scale-testing, developers can easily pick a PC that has the closest specification to the machine that the customers run, hence observing a more similar bug symptom.

- **Can SCALECHECK be integrated with Exalt/Tardis [70]?** Conceptually SCALECHECK and Exalt/Tardis are orthogonal. That is, while SCALECHECK emulates processing time, Exalt/Tardis emulates data size. We attempted to integrate them together but Exalt’s code is no longer actively maintained (confirmed by the first author) so it will take a longer time to integrate these approaches. But again, we believe that the two concepts complement each other.

- **Why space of memory/disk is not a colocation bottleneck in our benchmarks?** There are multiple reasons for this.

  First, our benchmarks so far mainly test the control path logic which does not require big data input. The control paths of our system are typically not dependent on the data size, as each file IO can be submitted to the disk independently. However, as mentioned above, combining Exalt/Tardis
Second, we find that on startup, nodes do not take up much memory, which is as expected because the memory is mostly expected to be used for data caching. For example, HDFS only uses 25 MB and Cassandra 15 MB (excluding the runtime memory) after a startup. With 32 or 64 GB typical DRAM size, we can fit 1000 nodes on a machine. HDFS developers manually created the “TinyDataNode” and “PartialDataNode” classes where a datanode only takes 3-5 MB of memory, saving 20 MB per node. We suspect they do this to achieve faster unit tests, as the hundreds of tests must instantiate a datanode from scratch in every test.

One exception is a case of over-allocation of unnecessary memory, which will be deleted after the cluster achieves a “stable” state. For instance, in Riak’s rebalance protocol, each node creates \( N \times P \) partition services although at the end only retain \( P \) partitions and remove the other \((N-1) \times P\) partitions (as rebalanced to other nodes). Each partition service is an Erlang process (1.3 MB of memory overhead), thus colocating 30 nodes \((N=30\) with the default \( P=64\)) will directly consume 75 GB of memory \((30 \times 30 \times 64 \times 1.3 \text{ MB})\) from the start. We suspect Riak developers do it this way for the simplicity of writing the bootstrap protocol. To address this, we manually modified Riak to remove this unoptimized memory usage, which shows that one can rearrange the protocol to be memory efficient.

- **How does SCALECHECK address disk IO contention?** IO contention is a problem in single-machine scale-testing if we want to mimic deployments that use disks. That is, when colocating hundreds of nodes on a single machine, all the colocated IOs going to the single-spindle disk will create heavy contention that does not happen in deployment. This problem is automatically solved with PIL with its time memoization. During time pre-memoization, with heavy I/O contention, the timing being recorded is not accurate as it includes the queuing delay from the memoization. However, because during replay, we control the sleep timing, such timing can be altered to remove the unintended queuing delays that are supposed not to be there. So far, as we target control path (as mentioned above), the amount of data being read/written in a single IO is typically small and
does not exhibit random I/Os. For disks, deconstructing queuing delays in this environment is in general straightforward as the per-IO latency is stable (e.g., an IO typically takes 7 ms without any contention).

While we can mimic deployments using disks, during pre-memoization, we must use SSD as our back-end storage to store the memoized input/output pairs, exactly given the reason above. Using tmpdisk is also possible, but memory space is already reserved for the node states. Given our achieved colocation of 512 nodes, using SSD as the database storage does not cause contention issues. An SSD can have 32-128 parallel channels. A datacenter SSD can deliver 1M read IOPS and 200K write IOPS and a client-level SSD can deliver 400K read IOPS and 200K write IOPS.
CHAPTER 7
RELATED WORK

In Section 1, we briefly discussed related work in four categories: real-scale testing/benchmarking (direct, but not economical) [55, 25], large-scale simulation (easy to run, but rarely used for server infrastructure code) [38, 53, 52], extrapolation (easy to run, but missing bugs in small training scale) [53, 78, 58], and emulation. SCALECHECK falls in this category and below discuss three closely related works [70, 47, 10].

Exalt [70] targets IO-intensive (Big Data) scalability problems where storage capacity is the colocation bottleneck. Exalt’s library (Tardis) compresses users’ data to zero bytes on disk. With this, Exalt can co-locate 100 space-emulated HDFS datanodes per machine. As the authors stated, their approach “may not discover scalability problems that arise at the nodes that are being emulated” [70]. Thus, it cannot cover P2P systems where the scale-dependent code is in all the nodes. However, as Exalt targets storage space emulation and SCALECHECK addresses processing time emulation, we believe they complement each other. LinkedIn’s Dynamometer is similar to Exalt [10].

DieCast [47], invented for network emulation, can colocate processes/VMs on a single machine as if they run individually, by “dilating” time. The trick is adding a “time dilation factor” (TDF) support [48] into the VMM. For example, TDF=5 implies that for every second of wall-clock time, each emulated VM believes that time has advanced by only 200 ms ($1/TDF$ second). DieCast was only evaluated with a colocation factor (TDF) of 10 as the testing time significantly increases proportionally to the TDF; colocating 500 nodes will increase testing time by 500 times. DieCast was introduced for answering “what if the network is much faster?”, but not specifically for single-machine scale-testing. Another significant difference is that both Exalt and DieCast papers do not present an in-depth bug study.

In terms of related work in the static/program analysis space, Clarity [64] and Speed [43] use static analysis to look for potential performance bottlenecks by focusing on redundant traversals
and precise complexity bounding. Both approaches are evaluated in libraries. However, for distributed systems, real-scale testing can help reveal unintended complex component interactions, and not all scale-dependent loops cause problems.

Finally, a recent work also highlights the urgency of combating scalability bugs [57]. The work, however, does not employ methodical and incremental changes, only suggests a manual approach, and reproduces only 4 bugs in 1 system.
CHAPTER 8
CONCLUSION

Technical leaders of a large cloud provider emphasized that “the most critical problems today is how to improve testing coverage so that bugs can be uncovered during testing and not in production” [42]. It is now evident that scalability bugs are new-generation bugs to combat, that existing large-scale testing is arduous, expensive, and slow, and that today’s distributed systems are not single-machine scale-testable. Our work addresses these contemporary issues and will hopefully spur more solutions in this new area.
CHAPTER 9
APPENDIX

9.1 List of scalability bugs

Table 9.1 shows the complete list of 55 scalability bugs that we studied. These bugs are dependent to the scale of the cluster size.

Table 9.1: List of Scalability Bugs (§9.1)

<table>
<thead>
<tr>
<th>Systems</th>
<th>Issue</th>
<th>Issue Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassandra</td>
<td>2058</td>
<td>Load spikes due to MessagingService-generated garbage collection</td>
</tr>
<tr>
<td>Cassandra</td>
<td>3831</td>
<td>Scaling to large clusters in GossipStage impossible due to calculatePendingRanges</td>
</tr>
<tr>
<td>Cassandra</td>
<td>3881</td>
<td>Reduce computational complexity of processing topology changes</td>
</tr>
<tr>
<td>Cassandra</td>
<td>4288</td>
<td>Prevent thrift server from starting before gossip has settled</td>
</tr>
<tr>
<td>Cassandra</td>
<td>5220</td>
<td>Repair improvements when using vnodes</td>
</tr>
<tr>
<td>Cassandra</td>
<td>5456</td>
<td>Large number of bootstrapping nodes cause gossip to stop working</td>
</tr>
<tr>
<td>Cassandra</td>
<td>6127</td>
<td>vnodes don’t scale to hundreds of nodes</td>
</tr>
<tr>
<td>Cassandra</td>
<td>6268</td>
<td>Poor performance of Hadoop if any DC is using VNodes</td>
</tr>
<tr>
<td>Cassandra</td>
<td>6345</td>
<td>Endpoint cache invalidation causes CPU spike (on vnode rings?)</td>
</tr>
<tr>
<td>Cassandra</td>
<td>6409</td>
<td>Gossip performance improvement at node startup</td>
</tr>
<tr>
<td>Cassandra</td>
<td>6485</td>
<td>NPE in calculateNaturalEndpoints</td>
</tr>
<tr>
<td>Cassandra</td>
<td>6862</td>
<td>Poor performance of Hadoop if any DC is using VNodes</td>
</tr>
<tr>
<td>Cassandra</td>
<td>13968</td>
<td>Cannot replace a live node in large clusters</td>
</tr>
<tr>
<td>Couchbase</td>
<td>1040</td>
<td>Improve bootstrapping speed by creating/initializing all nodes in parallel</td>
</tr>
<tr>
<td>Couchbase</td>
<td>8640</td>
<td>Rightscale template :: 15 of 120 node sized array stranded in booting: exited with 2, expected 0.</td>
</tr>
<tr>
<td>Systems</td>
<td>Issue</td>
<td>Issue Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>Couchbase</td>
<td>13102</td>
<td>Empty and idle node runs Flusher frequently, problem for scaling past 10 buckets</td>
</tr>
<tr>
<td>Couchbase</td>
<td>15757</td>
<td>Graceful failover either fails or takes very long time, delta rebalance fails - with latest build 3470</td>
</tr>
<tr>
<td>Couchbase</td>
<td>16807</td>
<td>New UI slightly slow at least on 130 node cluster</td>
</tr>
<tr>
<td>Hadoop</td>
<td>3656</td>
<td>Sort job on 350 scale is consistently failing with latest MRV2 code</td>
</tr>
<tr>
<td>Hadoop</td>
<td>3711</td>
<td>AppMaster recovery for Medium to large jobs take long time</td>
</tr>
<tr>
<td>Hadoop</td>
<td>4478</td>
<td>TaskTracker’s heartbeat is out of control</td>
</tr>
<tr>
<td>Hadoop</td>
<td>4946</td>
<td>Type conversion of map completion events leads to performance problems with large jobs</td>
</tr>
<tr>
<td>Hadoop</td>
<td>5124</td>
<td>AM lacks flow control for task events</td>
</tr>
<tr>
<td>Hadoop</td>
<td>5508</td>
<td>JobTracker memory leak caused by unreleased FileSystem objects in JobInProgress#cleanupJob</td>
</tr>
<tr>
<td>HBase</td>
<td>3620</td>
<td>Make HBCK Faster</td>
</tr>
<tr>
<td>HBase</td>
<td>4742</td>
<td>Split dead servers log in parallel</td>
</tr>
<tr>
<td>HBase</td>
<td>5422</td>
<td>StartupBulkAssigner would cause a lot of timeout on RIT when assigning large numbers of regions (timeout = 3 mins)</td>
</tr>
<tr>
<td>HBase</td>
<td>6728</td>
<td>Prevent OOM possibility due to per connection responseQueue being unbounded</td>
</tr>
<tr>
<td>HBase</td>
<td>7060</td>
<td>Region load balancing by table does not handle the case where a table’s region count is lower than the number of the RS in the cluster</td>
</tr>
<tr>
<td>HBase</td>
<td>7190</td>
<td>Add an option to hbck to check only meta and assignment</td>
</tr>
<tr>
<td>HBase</td>
<td>8778</td>
<td>Region assignments scan table directory making them slow for huge tables</td>
</tr>
<tr>
<td>Systems</td>
<td>Issue</td>
<td>Issue Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>HBase</td>
<td>9208</td>
<td>ReplicationLogCleaner slow at large scale</td>
</tr>
<tr>
<td>HBase</td>
<td>9377</td>
<td>Backport HBASE- 9208 &quot;ReplicationLogCleaner slow at large scale”</td>
</tr>
<tr>
<td>HBase</td>
<td>9775</td>
<td>Client write path perf issues</td>
</tr>
<tr>
<td>HBase</td>
<td>10209</td>
<td>Speed region assign in failover</td>
</tr>
<tr>
<td>HBase</td>
<td>11290</td>
<td>Unlock RegionStates</td>
</tr>
<tr>
<td>HBase</td>
<td>12139</td>
<td>StochasticLoadBalancer doesn’t work on large lightly loaded clusters</td>
</tr>
<tr>
<td>HDFS</td>
<td>354</td>
<td>Data node process consumes 180% cpu</td>
</tr>
<tr>
<td>HDFS</td>
<td>395</td>
<td>DFS Scalability: Incremental block reports</td>
</tr>
<tr>
<td>HDFS</td>
<td>611</td>
<td>Heartbeats times from Datanodes increase when there are plenty of blocks to delete</td>
</tr>
<tr>
<td>HDFS</td>
<td>1073</td>
<td>DFS Scalability: high CPU usage in choosing replication targets and file open</td>
</tr>
<tr>
<td>HDFS</td>
<td>1851</td>
<td>HDFS-15 scalability improvements</td>
</tr>
<tr>
<td>HDFS</td>
<td>2495</td>
<td>Increase granularity of write operations in ReplicationMonitor thus reducing contention for write lock</td>
</tr>
<tr>
<td>HDFS</td>
<td>2938</td>
<td>Recursive delete of a large directory makes namenode unresponsive</td>
</tr>
<tr>
<td>HDFS</td>
<td>3990</td>
<td>NN’s health report has severe performance problems</td>
</tr>
<tr>
<td>HDFS</td>
<td>4061</td>
<td>Large number of decommission freezes the Namenode</td>
</tr>
<tr>
<td>HDFS</td>
<td>4075</td>
<td>Reduce recommissioning overhead</td>
</tr>
<tr>
<td>HDFS</td>
<td>4360</td>
<td>Multiple BlockFixer should be supported in order to improve scalability and reduce too much work on single BlockFixer</td>
</tr>
<tr>
<td>HDFS</td>
<td>4479</td>
<td>logSync() with the FSNamesystem lock held in commitBlockSynchronization</td>
</tr>
</tbody>
</table>
List of Scalability Bugs (continued)

<table>
<thead>
<tr>
<th>Systems</th>
<th>Issue</th>
<th>Issue Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS</td>
<td>4937</td>
<td>ReplicationMonitor can infinite-loop in BlockPlacementPolicyDefault#chooseRandom()</td>
</tr>
<tr>
<td>HDFS</td>
<td>9198</td>
<td>Coalesce IBR processing in the NN</td>
</tr>
<tr>
<td>HDFS</td>
<td>9287</td>
<td>Block placement completely fails if too many nodes are decommissioning</td>
</tr>
<tr>
<td>HDFS</td>
<td>10609</td>
<td>Uncaught InvalidEncryptionKeyException during pipeline recovery may abort downstream applications</td>
</tr>
<tr>
<td>Riak</td>
<td>3926</td>
<td>Large ring_creation_size</td>
</tr>
<tr>
<td>Voldemort</td>
<td>1212</td>
<td>Number of Partition</td>
</tr>
</tbody>
</table>

### 9.2 SFind algorithm Pseudocode

Algorithms 9.1 to 9.5 show the steps of SFind to find scale-dependent loops given a set of scale dependent data structures. The main idea is that for each method a call graph is created (algorithm 9.2) then each node is recursively “resolved” (loops are analyzed considering scale-dependent variables, collect IO/Lock metadata (algorithm 9.4), check for recursion) in order to assign a “scale dependency level” (algorithm 9.3) and finally report the methods considered potential candidates for scalability issues (algorithm 9.5).

#### Algorithm 9.1: sFind

```java
void sFind(List<Method> methods, int threshold, List<SDVariable> scaleDependentDataStructures) {
    for (Method method : methods) {
        // create a call graph
        CallGraphNode root = new CallGraphNode(method);
        CallGraph cGraph = new CallGraph(root);
```
createCallGraph(root, cGraph);
// process the method
processScaleDependency(root, scaleDependentDataStructures);
// report if possible candidate. Notice that the threshold its a parameter
maybeReportScaleDependentCandidate(method, threshold);
}

Algorithm 9.1 is the main algorithm. Starting from a list of methods (taken from source code) checks the scale dependency and IO/Lock metadata of each call graph. For each callgraph, it compares the scale dependency level (complexity of the method in terms of scale dependent data structures).

Algorithm 9.2: createCallGraph

void createCallGraph(CallGraphNode node, CallGraph callGraph){
    // set root method
    for(Method invoked : node.getMethod().getInvokedMethods()){
        if(invoked.compareTo(node.getMethod()) != 0){
            // if its a concrete method, we add it to the callGraph
            // and then we continue with its children
            CallGraphNode newNode = new CallGraphNode(invoked);
            callGraph.addNode(newNode);
            node.addChild(newNode);
            if(invoked.isConcrete()){
                createCallGraph(newNode, callGraph);
            }
        } else{
            // does not have a body to follow, lets look at concrete
            implementations
            for(Method concrete : getConcreteImplementations(invoked)){
                // notice that opening up the tree using concrete implementations
                might introduce

45
// many false positives by showing execution paths that are not feasible in reality.
CallGraphNode newNode = new CallGraphNode(concrete);
callGraph.addNode(newNode);
node.addChild(newNode);
createCallGraph(newNode, callGraph);
}
}
else{
    // just mark as recursive
    node.getMethod().markAsRecursive();
}
}

// ready

Algorithm 9.2 builds a call graph considering abstract method invocations and concrete implementors to resolve. We also look for recursion but we do not add recursive calls to the call graph (the same applies for methods called more than once in a call graph, but in that case, we do not mark it as recursive).

Algorithm 9.3: processScaleDependency

```java
void processScaleDependency(CallGraphNode node, List<SDVariable> scaleDependentDataStructures)
{
    if (node.isLeaf()){
        for (Loop loop : node.getLoops()){
            // this resolution also considers nesting
            if (loop.isScaleDependent(scaleDependentDataStructures)){
                node.incrementScaleDependencyLevel();
            }
        }
    }
}
```

46
Algorithm 9.3 recursively resolves the call graph (bottom up) in order to collect scale dependency and lock/IO metadata. For each node (a method) we analyze scale dependency (using the list of scale-dependent data structures) in order to compute a scale dependency degree. From the perspective of recursion, a method is established as scale-dependent if (1) it invokes a scale-dependent method and is marked as recursive, (2) it has a scale dependent loop and is marked as recursive or (3) the input parameters are considered scale dependent.

Algorithm 9.4: collectLockAndIOMetadata

```java
void collectLockAndIOMetadata(CallGraphNode node){
    // we first analyze the locking context of this node
    if (node.isInvokedInSynchronizedContext()){
        // mark as blocking considering a synchronized block
        // or explicit locking mechanisms (reentrant locks, etc...)
        node.markAsBlocking(node.getLockingContext());
    }
    if (node.performsIO()){
        // IO is detected by looking for java library invocations related
```
Algorithm 9.4 collects lock/IO metadata by looking at synchronized blocks, explicit lock mechanisms (e.g. ReentrantLocks) and specific IO library calls (e.g. FileReader#read). Notice that currently we don’t look for network related IO.

Algorithm 9.5: maybeReportScaleDependentCandidate

```java
void maybeReportScaleDependentCandidate(Method scaleDependentMethod, int threshold){
    // at this point, all conditions regarding lock/IO, recursivity
    // and scale dependency degree have been propagated. So in here, we
    // just report based on the threshold. In the actual implementation,
    // the report includes a graphical view of the callgraph and
    // indications of locking,
    // IO and recursivity.
    if(scaleDependentMethod.getScaleDependencyDegree() >= threshold){
        print(scaleDependentMethod);
    }
}
```

Algorithm 9.5 prints a possibly scale-dependent method by checking its scale dependency degree (previously calculated) and comparing it to a threshold (one of the parameters of our analysis).

### 9.3 PIL algorithm Pseudocode

Algorithms 9.6 to 9.9 show the pseudo-code of our PIL algorithms. The code basically attempts to find PIL-safe functions. Note that it focuses on finding scale-dependent loops that can be PIL-ed as opposed to finding all PIL-safe code blocks.
Algorithm 9.6 establishes if a code block (a scale dependent loop) is a candidate for PIL. The main idea is to analyze the contents of the loops in function of the relevant cluster state and the operations performed in that loop. In here, we need to distinguish between two cases: (1) the loop performs non-pertinent operations only (such as IO). In this case, we can safely replace the loop by a `sleep` call without affecting the behavior of the protocol. In (2), the loop performs operations that affect the cluster state, so we need to insert pre-memoization and replay code (see algorithms 9.7, 9.8 and 9.9) to record/reconstruct cluster state at some point of time.

```
Algorithm 9.6: isPILCandidate

```int```
```
isPILCandidate (Loop loop, Method loopContext)
{
    // in here, we are going to analyze a loop in its context
    boolean hasOnlyNonPertinentOperations = true;
    // we consider a loop that only performs IO operations as non pertinent
    // these are typically method calls
    for (Statement statement : loop.getStatements()){
        if (!statement.isIOOperation()) hasOnlyNonPertinentOperations = false;
    }
    // can be PIL-ed without memoization
    if (hasOnlyNonPertinentOperations) return 1;
    // now second case
    for (Statement statement : loop.getStatements()){
        // we perform static analysis to check what are the related variables
        // this is a time consuming operation. State is globally defined
        if (StaticAnalysisManager.touchesClusterState(statement)){
            // can be PIL-ed with memoization
            return 2;
        }
    }
    // cannot be PIL-ed
    return 0;
```
Algorithm 9.7 inserts the pre-memoization and replay code into a target method (using a code block as reference). The blocks that are inserted are shown at algorithms 9.8 and 9.9. We use the same algorithm to illustrate the two cases described above since the difference between them are minimal from an implementation perspective.

Algorithm 9.7: insertPreMemoizationCode

```java
void insertPreMemoizationCode(Block targetBlock, Method targetMethod, Block replayBlock, Block memoizeBlock, int pilType) {
    // we have identified the target block we want to modify, so we just insert code
    method.insertBefore(targetBlock, replayBlock);
    if (pilType == 2) method.insertBefore(targetBlock, memoizeBlock);
    // done
}
```

It is important to notice that in here we do require programmer defined cluster state. We consider as relevant state all variables involved in the execution of a target protocol and we discard non pertinent operations (such as IO). Our static analysis tools ease the identification of these variables, but programmer intervention is needed to discard/add possible false positives. Also, given that we use Java serialization to save/reconstruct the target state, we also require that all classes involved are marked as `Serializable`. In this context, programmers can modify the generated code in order to use custom (probably faster) serialization mechanisms by overriding the base classes provided by our API. Given that most of the code is generated automatically, we consider that the effort related to this integration task is minimal compared to manual identification/implementation. Finally, we reduce the overhead of reading/writing cluster state by using SSD’s.
Time time = now();
// execution of target block happens here
Time elapsed = now() - time;
if (isMemoizeEnabled()) {
    State state = StateManager.recordClustersState();
    StateSerializer.recordClusterState(state, elapsed);
}
// method continues

Algorithm 9.9: replay block example

if (isReplayEnabled()) {
    // the arguments to this call identify the current state
    State state = StateSerializer.getState(currentStateId);
    // sleep
    sleep(State.getProcessingTime());
    // now reconstruct, only for PIL type 2
    if (shouldReconstructState())
        StateManager.reconstructStateFromSnapshot(state);
}
else {
    // normal execution of target block happens here
}
REFERENCES


[25] Personal Communication from Andrew Wang and Wei-Chiu Chuang of Cloudera and Uma Maheswara Rao Gangumalla of Intel; they are also part of Apache Hadoop Project Management Committee (PMC) members.

[26] Personal Communication from Imran Rashid (Software Developer at Cloudera).

[27] Personal Communication from Jonathan Ellis, Joel Knighton, Josh McKenzie, and other Cassandra developers.


[32] Voldemort bug: Number of partitions. https://groups.google.com/forum/#!msg/project-voldemort/3vrZfZgQp2Y/Uqt8NgJHg4A.


55


