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
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
CESAR ANDRES STUARDO MORAGA

CHICAGO, ILLINOIS
2022

To my wife Paulina, and my son Cesar

I am the wisest man alive, for I know one thing, and that is that I know nothing.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

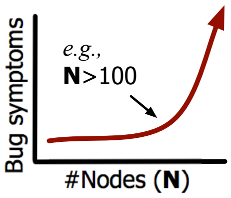
ABSTRACT

In this document, we present our approaches for understanding and discovering scalability faults, i.e. faults whose symptoms appear at larger scales but are not visible at smaller scales. First, we present a study of over 350 scalability faults collected from the repositories of 10 popular open-source distributed systems. We analyze the symptoms they produce, the scenarios in which they manifest, their root causes, the effectiveness of existing testing tools in detecting them and the solutions and effort involved in tackling them. Then, we present SCALECHECK, an emulation-based approach for discovering scalability faults in large-scale distributed storage systems. SCALECHECK employs a set of black and white box techniques to allow developers to "deploy" a cluster in a single-machine and accurately observe the behavior of their systems as if they were deployed in multiple machines. Moreover, SCALECHECK includes a collection-tracking mechanism that allows developers to discover potentially harmful code paths affected by the increase in the number of nodes in the cluster. We integrated this approach into 4 popular distributed storage systems and accurately reproduced the symptoms of 10 known scalability faults using a single machine. Finally, we present SVIEW, a framework for identifying and analyzing potential scalability faults in large-scale distributed systems. SVIEW combines instrumentation and statistical concepts to identify dimensional code fragments (DCFs), i.e. pieces of code whose number of executions (e.g., # loop iterations, # method executions) is positively correlated with the increase in the size of one or more system dimensions (e.g. # number of files, # clients, # requests), with static analysis modules that detect faulty code patterns involving the DCFs. SVIEW's lightweight approach does not require modifications in the system under test, it's portable without effort across different versions of the same system and focuses on the root cause of scalability faults rather than the symptoms they produce. We evaluate SVIEW in 15 different versions of 4 popular distributed systems and use our analysis modules to detect known and unknown scalability faults.

CHAPTER 1

INTRODUCTION

In light of the limits of Moore's Law, Dennard scaling and the ever increasing computing demand, the last decade has seen unprecedented deployment scales: tens of thousands Cassandra nodes are managing petabytes of data at Netflix, more than 50K Yarn nodes are performing Exabyte-size data analytics at Microsoft and over 150K Hadoop *jobs* are submitted at Twitter on a daily basis [62, 63, 102]. Scale surpasses the limits of a single machine in meeting increasing demands in compute and storage, allowing companies to cope with unexpected emergencies: in light of the (almost over?) covid pandemic, Zoom experienced an order of magnitude growth in its users, going from around 10M to 200M, and a similar trend in the amount of cloud instances used to satisfy this unprecedented increase in traffic [86, 110].



Scalability faults: Latent faults that are scale dependent, whose symptoms surface in large-scale deployments (*e.g.*, $N > 100$ nodes), but not necessarily in small/medium-scale deployments (*e.g.*, $N < 100$ nodes).

Figure 1.1: **Scalability fault definition.** *Detailed examples are in section 1.1.*

There is an undeniable benefit on the rise of large-scale cloud systems, but is scale a friend or a foe [156]?. As pointed out more than a decade ago, scalability is not an after-thought: It requires systems to be designed with it in mind, as many of their protocols could seem scalable when evaluated using *small inputs* (*e.g.*small amounts of data) or under low load but explode in terms of *cost* (*e.g.*, execution time, memory consumption) when the size of the input or load grows [106]. In this document, we refer to these types of explosive manifestations as **scalability faults**, *i.e.*faults that become visible at larger scales but are not visible at smaller scales, as illustrated in figure 1.1.

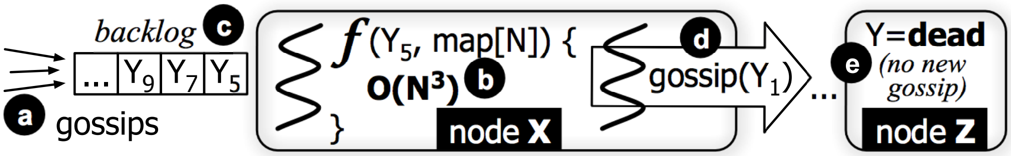


Figure 1.2: **Cassandra scalability fault.** *The figure represents the process related to ca-6127. This fault surfaced when bootstrapping hundreds of nodes and led to cluster instability with tens of thousands of flappings; a "flap" is when a node marks a peer node as down and then alive again.* ***(a)*** *During bootstrapping, every node gossips to peer nodes its view of the node-ring (the peers' version numbers it has received). Every second, each node sends a gossip to a random node and increases its own version number ("I'm still alive" flag); for example, node Y has gossiped up to version Y_9.* ***(b)*** *The receiving node (e.g., node X) then finds any view differences between the two nodes to synchronize their views of the ring. The root cause is that the gossip processing during bootstrapping is scale-dependent (the "$O(N^3)$ $f()$"), because the gossips carry many new view changes.* ***(c)*** *Because gossip processing is single-threaded (to alleviate concurrency issues), and when N is large, it creates a backlog of new gossips.* ***(d)*** *As a result, the processing node only sends the latest (old) versions it knows about peer nodes; for example, X only gossips version Y_1 to other nodes.* ***(e)*** *As Y's recent gossips are not propagated on time, other nodes (e.g., Z) will mark Y as dead.*

1.1 Motivation

As a motivational an example, let us consider the fault in Cassandra [1], a scalable peer-to-peer key-value store, described in figure 1.2. If a customer initially deploys a cluster of 50 nodes and later scales it out with 50 additional nodes, the operation can be done smoothly. However, if the customer deploys a 200-node cluster and then adds 200 more nodes, the protocol that rebalances the key-range partitions (which nodes should own which key ranges) becomes CPU intensive as the calculation has an $O(N^3)$ complexity where N is the number of nodes. This combined with the gossiping and failure detection logic leads to a scalability fault that makes the cluster unstable (many live nodes are declared as dead, making some data not reachable by the users).

But Cassandra is far from being the only system exhibiting scalability faults. As shown in figure 1.3, large scale deployments of HDFS, Riak and Voldemort are prone to this types of issues too. In figure 1.3(a)[57], when D HDFS's datanodes are decommissioned, the blocks must be replicated to the other $N-D$ nodes. Every 5 minutes, the DecommissionMonitor thread in HDFS's namenode iterates all the block descriptors to check if the D nodes can be safely decommissioned (when

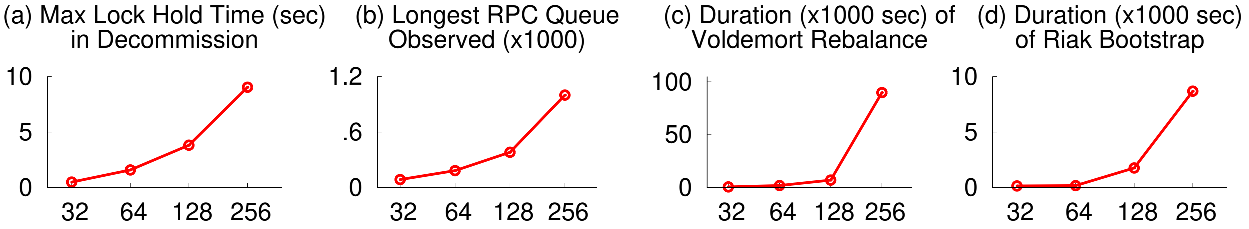


Figure 1.3: **Other scalability faults.** *In all figures, the x axis represents the number of nodes, while the y axis is related to an specific symptom. In (a), the symptom is the amount of time the HDFS's decommission lock is being held. In (b), the symptom is the size of the HDFS's RPC queue. In (c) the symptom is the duration of Voldemort's rebalance process. Finally, in (d) the symptom is the duration of Riak's bootstrap process.*

all data replications complete). This thread, unfortunately, must hold the global file system lock. When $N > 256$, this process can hold the lock (i.e., stall user requests) for more than 10 seconds ($y > 10$).

In figure 1.3(b)[58] incremental block reports (IBRs) from HDFS datanodes to the namenode acquire the global master lock. As N grows, more IBR calls acquire the lock. The IBR requests quickly backlog the namenode's IPC queue; with 256 nodes, the IPC queue hits the max of 1000 pending requests; $y = 1$ (×1000). When this happens, user requests are undesirably dropped by the namenode. The fix batches the IBR request processing. In figure 1.3(c)[105] Voldemort's rebalancing was not optimized for large clusters; it led to more stealer-donor partition transitions as the cluster size grows (128+ nodes). Finally, in 1.3(d)[95] Riak's rebalancing algorithm employed 3 complex stages (claim-target, claim-hole, full rebalance) to converge to a perfectly balanced ring. Each node runs this CPU-intensive algorithm on every bootstrap gossip received. The larger the cluster, the longer time the perfect balance is achieved (a high y value in 128+ nodes).

Motivated by these and other faults we have studied (discussed later at chapter 2 and summarized at tables A.1 and C.1), we have made the following observations about scalability faults:

- Scalability faults **only appear at extreme scale:** ca-6127 does not surface in 30-node deployment. In 128-node cluster, the symptom appears mildly (tens of flaps). From 200-500 nodes, flapping skyrockets from hundreds to thousands of flaps. Testing in small/medium scales is not sufficient, which is also true for other faults we studied.

- Protocols are **scalable in design, but not in practice:** related to ca-6127, the accrual failure detector/gossiper [129] was interestingly adopted by Cassandra as it is scalable in design [136]. However, the design proof does not account gossip processing time during bootstrap, which can be long. To understand the fault, the developers tried to "do the [simple] math" [15] but failed. In practice, the assumption that new gossips are propagated every second is not met (due to the backlog). The actual implementations overload gossips with many other purposes (*e.g.*, announcing boot/rebalance changes) beyond their original design sketch.

- Scalability faults are **implementation specific and hard to predict.** the backlog-induced flapping in ca-6127 was caused specifically by Cassandra's implementation choice: metadata checkpoint, multi-map cloning, and its single-threaded implementation. State-update processing time is hard to predict (ranges from 0.001 to 4 seconds) as it depends on a 2-dimensional input: the receiving node's ring table size and the number of new state changes [15].

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

- Large-scale debugging is **long and difficult:** ca-6127 generated over 40 back-and-forth discussion comments and took 2 months to fix. It is apparent that there were many hurdles of deploying and debugging the buggy protocol at real scale [15]. Important to note is that debugging is *not* a single iteration; developers must *repeatedly* instrument the system (add more logs) and re-run the system at scale to find and fix the bug, which is not trivial. The scalability faults we studied took from 6 to 157 days to fix (27 on average).

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

- **Developers perform quick fixes and face repeated faults.** Faults are often fixed with quick patches (development pressures), but the new fix might not eradicate the problem completely [178]. For example, for ca-6127, the patch simply disables failure detection during bootstrap. As the protocol was not redesigned, the fault still appeared in another workload (*e.g.*, scaling out from 128 to 256 nodes). In the next version of Cassandra, the simple fix has been removed and the gossip protocol has been redesigned. We also found that old fixes can become obsolete in protocol re-designs, which then can give birth to new scalability faults. For example, the fix for ca-3831 became obsolete as "vnodes" was introduced, which then gave rise to a new vnode-related scalability fault (ca-3881).

1.2 Research Questions

Motivated by our observations and findings, we spent the first two years of our work studying scalability faults, an effort that involved tens of students collecting, classifying and discussing those, aiming to address the following research questions:

- **RQ1: How do scalability faults manifest?** *What are the common symptoms observed by reporters?*

- **RQ2: What are their common root causes?** Some common patterns have been observed before [139], but *are there other root causes?*, and if so, *how common are those?*

- **RQ3: What are the common solutions proposed by developers?** and *how much effort is involved in fixing these faults?*

Understanding the intricate details of scalability faults, their root causes, related symptoms and manifestation scenarios was vital for understanding how current testing mechanisms fail or succeed to address them. After this study, we decided to tackle the lack of large-scale testing tools, in an effort that included months of developing deep knowledge of the architecture and internals of tens of distributed systems and reproducing tens, if not hundreds, of faulty scenarios, aiming to answer two research questions:

- **RQ4: How to discover latent scalability faults?** Scalability faults 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 faults remain unforeseen.

- **RQ5: How to democratize large-scale testing?** According to our study and industrial experience [91], 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, which heavily increases the cost of this practice.

Bu the approach we proposed for the latter research questions left many questions unanswered and had a few disadvantages [165]. It relied heavily on emulation and involved modifications to the target systems (on the order of hundreds of lines of code). The inherent limitations of emulation and the effort involved hinder its adoption. Moreover, our static analysis approach was focused on data structures, did not capture the relevant code patterns, produced false positives and more importantly did not provide a sufficient explanation on **how** the scalability of the system is affected by certain code patterns.

With the latter in mind, we invested the final two years of our work in creating a testing approach that is (a) focused on the root cause of the problem, not the related symptoms, (b) is not subject to the culprits of emulation, (c) is portable without effort between different versions of a system and (d) guides developers in understanding the relationship between the system and its

dimensionality (*e.g.* # nodes, # tables, # clients), *i.e.*, **how** the increase in size of such dimensions affects the system. The goal here is to address the following research questions:

- **RQ6: Which pieces of code are affected by dimensionality?** Mature distributed systems are usually comprised of hundreds of thousands of lines of code, but according to our observations, only some of those code fragments are **dimensional**, *i.e.*, as one or more system dimensions grow, the number of executions of these fragments also grows.

- **RQ7: How to detect dimensional code fragments?** *What are the necessary steps and tools we need to find such fragments?*. Moreover, *which techniques are useful in determining their correlation with certain dimensions and categorizing their growth trends?*.

- **RQ8: When do dimensional code fragments become problematic?** Dimensional code fragments are the building blocks of distributed systems, thus not all of them are inherently problematic. Then, *when do they become problematic?*.

These research questions are the main focus of this document. Over the next sections we detail our contributions, and provide a high-level overview of our intents to address those questions.

1.3 Towards Scale-Checkable Systems

1.3.1 Scalability Faults in Large-Scale Cloud Systems: A Comprehensive Study

The first part of this document is focused on understanding scalability faults and answering **RQ1**, **RQ2** and **RQ3**. To do so, in chapter 2 we present a comprehensive study of deployment and development fault reports from 10 popular large-scale cloud systems, including Hadoop, MapReduce, HDFS, HBase, Cassandra, Kafka, Ignite, Spark, Storm, and Yarn, reviewing over 110K issues from a 14-year period. From those, we selected 350 faults that manifest at larger scales (e.g. hundreds of nodes) but not at smaller scales (e.g. tens of nodes), *i.e.* scalability faults, as previously defined in figure 1.1. Our study includes the dimensions and system protocols involved, the symptoms observed and the mechanisms, if any, for detecting such faults. We further analyze the faulty code and identify the related root causes and categorize the proposed solutions.

1.3.2 ScaleCheck: *A Single-Machine Approach for Discovering Scalability Bugs in Large Distributed Systems*

Next, from the study presented at chapter 2 we took a 55 issue sample (13 in Cassandra, 5 in Couchbase, 6 in Hadoop, 13 in HBase, 16 in HDFS, 1 in Riak, and 1 in Voldemort, summarized at table A.1) and outline two main challenges: First, albeit systems usually include unit tests, benchmarks and API's that allow developers to build custom tests, no tool is focused on finding scalability faults. Second, the common practice of debugging such faults is arduous, slow and expensive. For example, when customers report scalability issues, 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. To overcome these challenges, chapter 3 presents ScaleCheck, a set of tools and practices for discovering scalability faults in distributed systems, our attempt to address **RQ4** and **RQ5**. First, to reveal hidden scalability faults, we build SFind, a program analysis support for finding "scale-dependent loops". Our strategy, based on the findings of chapter 2, focuses on loops that iterate on data structures that grow as the system scales out (e.g., the $O(N^3)$ described at figure 1.2). 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. For this, we introduce novel colocation techniques such as global-event driven architecture (GEDA) in single-process cluster and processing illusion (PIL) with non-intrusive modification.

1.3.3 SView: Identifying and Analyzing Potential Scalability Faults in Large-Scale Distributed Systems

Finally, in chapter 4 we present SView, a framework for identifying and analyzing potential scalability faults in large-scale distributed systems, our attempt to address **RQ6**, **RQ7** and **RQ8**. SView combines instrumentation, scaling workloads and mathematical concepts like correlation to detect **dimensional code fragments (DCFs)**, this is, pieces of code whose number of executions (i.e., # loop iterations, # method executions) is positively correlated with one or more system dimensions. Once detected, SView employs *similarity measures* to categorize DCFs into complexity categories (e.g., linear or superlinear), in an attempt to explain developers how impactful the DCFs are regarding their system's dimensionality. As DCFs or their complexity categories are not problematic by themselves, SView uses static analysis modules to look for problematic code patterns that cause performance degradation in critical paths, severe protocol contention or other more classic issues such as dimensional I/O. Combining these 3 factors, SView provides the necessary *guidance* and *depth* in order to find scalability faults in complex distributed systems.

1.4 Thesis Overview

The rest of this document is organized as follows: chapter 2 presents the study outlined in section 1.3.1, including its methodology (section 2.1), observed symptoms (section 2.2), categorization of fault-manifestation scenarios (section 2.3), protocols types (section 2.4), root causes (section 2.5), and observed solution patterns (section 2.6).

Chapter 3 presents the details of ScaleCheck [165], outlined in section 1.3.2, including a complete description of SFind, a static-analysis tool intended to detect scalability faults in source code (section 3.1), the techniques we deviced to create STest, a single-machine distributed systems runtime designed to reproduce scalability faults in a single machine (section 3.2), the implementation details and evaluation of the aforementioned components (sections 3.4 and 3.5 respectively), and a discussion on the limitations of this work (3.6).

Chapter 4 presents the details of SView [166], outlined in section 1.3.3, including the main motivation for that work, **dimensional code fragments (DCFs)** (section 4.1), and continuing with a design overview of SView (section 4.2). Then, in the following sections, we detail our contributions: study of DCFs (section 4.3), the design of SView (section 4.4) and the analysis modules built on top (section 4.5), and an in-depth evaluation of SView that covers 4 real distributed systems with a total of 15 versions (section 4.6). We then close the chapter with related work (section 4.7) and conclusions (section 4.8)

Finally, chapter 5 presents our conclusions and details on future work and chapter 6 discusses other (unrelated to the main topic of this document) contributions done during my Ph.D. program.

CHAPTER 2

SCALABILITY FAULTS IN LARGE-SCALE CLOUD SYSTEMS: A COMPREHENSIVE STUDY

As discussed in section 1.3.1, this chapter presents our scalability fault study, where the goal is to address the following research questions:

- **RQ1: How do scalability faults manifest?** *What are the common symptoms observed by reporters?*

- **RQ2: What are their common root causes?** Some common patterns have been observed before [139], but *are there other root causes?*, and if so, *how common are those?*

- **RQ3: What are the common solutions proposed by developers?** and *how much effort is involved in fixing these faults?*

The rest of this chapter is organized as follows: In section 2.1 we present our methodology and in the following sections we address our 3 research questions, including a discussion on observed symptoms at section 2.2, categorization of fault-manifestation scenarios at section 2.3, protocols types at section 2.4, root causes at section 2.5 and observed solutions at section 2.6. Finally, we summarize this chapter and show our contributions in section 2.7.

2.1 Methodology

In this section, we describe our methodology, in specific how we choose the target reports, the classifications we created and the resulting database.

• **Issue repositories**: The development projects of our target systems [1, 2, 3, 4, 5, 7, 9, 10, 11, 50] are all hosted under the Apache Software Foundation Project [8], where each of them maintains a highly organized issue repository [6]. Each repository contains development and deployment issues submitted mostly by developers and sometimes by a larger user community composed mainly

Dimension	**% reports**	**Examples**
Load	39	#requests, rpcs
Data	36	# or size of tables, files, partitions
Cluster	21	#peers, *datanodes*, *namenodes*
Fail	4	#processing errors

Table 2.1: **Dimensions, summary and examples.** *The second column shows the percentage of the total reports (350) that corresponds to dimensional each axis.*

of operators and other practitioners. We use the term "issue", "fault" and "report" interchangeably in this paper to represent both bugs and new features.

• **Issue selection**: We manually analyzed 110K issues, ranging from 2007 to 2020, and selected 350 instances where the root cause was related to an increase in the scale of **load**, the scale of **data**-size, the scale of **cluster**-size and/or the scale of **failure**. Examples of each dimension are shown at the third column of table 2.1, while the dimensional per-system break down is shown in figure 2.1.

• **Issue analysis**: We analyzed the description and comments of each reports, including the related pull-requests, if any, to understand the observed symptoms and the issue manifestation mechanism. For the symptoms, we identified 6 main categories and assigned one or more to each fault, as a report might include more than one type of reported symptom. We also documented the severity of the symptoms: issues that caused node crashes, hangs or when in general one or more nodes in the cluster *stopped working* were considered *severe*, while others were considered *mild*. In the case of manifestation mechanisms, we identified 2 main categories and divided one of those into 3 subcategories.

• **Source code analysis**: We analyzed the faulty code paths, which we refer in this paper as *protocols*, using the pointers provided by developers, if any. We created 3 protocol categories, including a generic one in which we classified reports that could affect the first 2 types, and assigned a single category to each report. We then studied the issue in depth to identify its root cause. We used the pointers provided by reporters and developers and sometimes compared the faulty code with the proposed solution in order to understand the possible contributing factors, selecting the one that

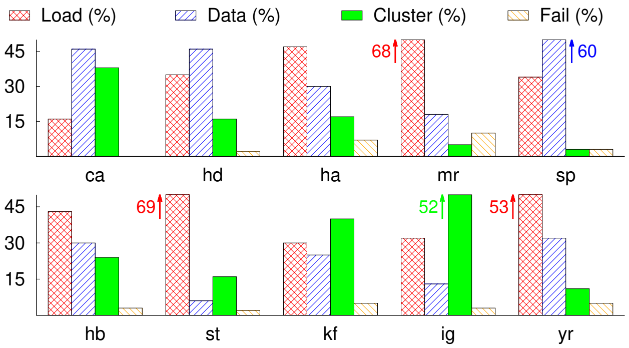


Figure 2.1: **Per-system dimension break down.** *From left to right, the systems in the top-most figure are Cassandra, HDFS, Hadoop, MapReduce and Spark, while the ones in the figure at the bottom are HBase, Storm, Kafka, Ignite, Yarn. The bar graphs show the percentage of reports associated with each dimension, as defined in table 2.1.*

contributes the most according to our analysis, as the *root cause*. We finally created 4 main root cause categories with 2-3 subcategories each, for a total of 11 types of root causes, and assigned one and only one to each report.

• **Patch analysis**: Finally, we analyzed the proposed patches and pull requests. For each, we collected data related to the effort involved in fixing the issue (the modified lines of code in the latest patch), the amount of and size of the related unit tests (number of new/modified classes and methods) and analyzed the solution patterns. For the later, we identified 28 common solution patterns, representing commonly known solutions (*e.g.*throttling or batching).

• **Scale Fault DB (SFDB)**: The product of our classifications is stored in SFDB [96], a set of raw text files that enables us (and future SFDB users) to perform both quantitative and qualitative analysis of scalability faults using the tagging system described in table 2.2.

• **Threats to validity**: To the best of our effort, we consider our collection process as complete as possible, albeit we might have missed relevant reports due to manual scanning. Each issue cited in this report was discussed by at least 3 people. If an ambiguity raised when tagging an issue, we discussed such ambiguity until we reached a unified conclusion. Finally, we report the solution and patches as they were informed by the end of February 2020. After that date, new fixes might have been added and new developments on the issue might have occurred, including an issue being

Classification	**Cardinality**
creation year	1
dimension(s) involved	+
patch count	1
latest patch LOC	1/0
new unit tests (classes and methods)	1
mod unit tests (classes and methods)	1
manifestation category	1
protocol category	1
root cause category and subcategory	1
observed symptoms	+
observed severity	+
solution mechanism	+
reported cluster size	*
reported data size	*
reported load	*
reported company	1/0

Table 2.2: **Per-issue SFDB classifications.** *The second column shows the cardinality of each tag, where "1" stands for "one and only one", "1/0" for "one or none", "+" for "at least one" and "*" for "zero or more". The actual tagging system is defined at the database [96] documentation.*

reopened due to the solution not being effective, which is part of the fluid nature of this type of repositories.

2.2 Observed Symptoms: *How* do scalability faults manifest?

We identified 6 main symptom categories related to *contention*, *crashes*, *performance* degradation, high machine-level *resource* usage, high *memory* usage and *operation failures*. Below we discuss each category, including the main dimensions involved, the severity of the fault and real world examples.

• In over **57%**of faults reporters observe some level of **performance (pe)** degradation, such as tail latency or long execution time, as shown in the third column of figure 2.2(a). The severity breakdown, also in figure 2.2(a), seems to indicate that most of these degradations did not cause severe implications. This is one of the most common reported symptoms and is, according to our observations, closely related to the data and cluster-size dimensions, as shown in the third column

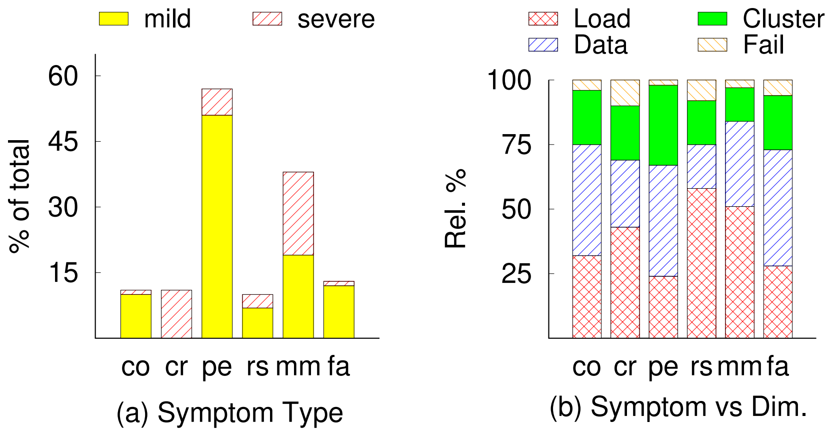


Figure 2.2: **Symptom types: severity and dimensional break down.** *In figure (a), the symptom categories are non-exclusive, thus the sum of the percentages is larger than 100%. In the aforementioned figure, the y-axis represents % of the total while in (b) the percentages are relative. In the x-axis of all figures, "co" stands for "contention", "cr" for "crash", "pe" for "performance", "rs" for "OS resources", "mm" for "memory" and "fa" for "operation failure", while in figure (b) title "Dim." stands for "Dimension" (section 2.1).*

of figure 2.2(b). Examples are very diverse and include st-2733, were heavy *garbage collection* [70] activity degraded storm's *worker* node performance, rendering it unable to cope with the incoming traffic.

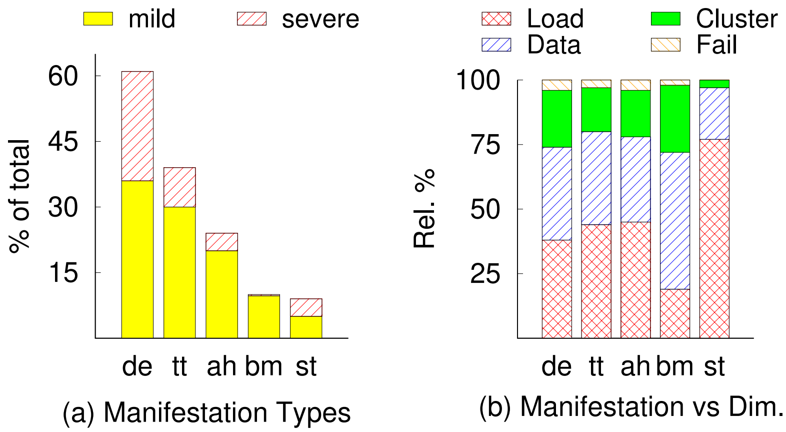
• In over **38%** of faults reporters observe some level of **memory (mm)** related issues, such as high memory usage, running out of memory, frequent object allocation and/or bloating data structures, as shown in the fifth column of figure 2.2(a). The severity breakdown indicates that in half of these cases the memory consumption caused severe impacts, *i.e.*one or more nodes in the cluster crashed due to *OOM*. This is the second largest category and seems to be heavily related to the data and load dimensions, as shown in the fifth column of figure 2.2(b). Examples are very diverse and include yr-9067, where improper cleaning of connections upon failure ends up building a large memory leak that in time ends up rendering the *Resource Manager* out of memory.

• In over **11%** of faults reporters observe some level of lock-related **contention (co)**, as shown in the first column of figure 2.2(a). We initially expected that most of the faults where contention was observed were related to the load dimension, but the first column of figure 2.2(b) proved us wrong. After further analysis, we realized that, as will be discussed in section 2.3, protocols sensitive to an increase in load, like user-facing protocols, are frequently performance-tested and contention bottlenecks are one of the first possible issues testers will look for. On the other hand, the larger the amount of data the more possible contention between user-facing and operational protocols (*e.g.*to maintain consistency), such as in hd-14854 where decommissioning a single *datanode D* uses a global lock that is held on the *namenode* while iterating over *D*'s files , thus the more files in *D* the more time the cluster is not responsive.

• In over **11%**of faults reporters observe single or multiple node **crashes (cr)**, as shown in the second column of figure 2.2(a). In this case, all reports indicate severe implications, for logical reasons. Crashes are one of the categories more strongly correlated to the fail dimension, as shown in the second column of figure 2.2(b). Notable examples include ca-15013, where memory related issues in a heavily loaded cluster ended up killing multiple nodes.

• In over **10%**of faults reporters observe some level of machine-related **resource (rs)** shortage, such as storage bloating, high CPU usage or exhaustion of file descriptors, as shown in the fourth column of figure 2.2(a). This type of faults are commonly related to the load and cluster-size dimensions, since typical system's designs involve per-node or per-request connections (sometimes organized in pools), as shown in the fourth column of figure 2.2(b). Examples in this category include ha-15813, where wrong per-request connection caching logic leads to machines running out of file descriptors under load.

• Finally, in over **13%**of faults reporters observe some level of **operation failures (fa)**, such as timeouts, as shown in the sixth column of figure 2.2(a). Operation failures are highly correlated to the data and cluster-size dimensions which we suspect is related to the reasons we described previously for the contention category. Examples reporting operation failures include ig-12042, where reporters claim that attempting to remove a *data page* from a fully populated *data region* ends up in allocating one more *data page*, causing the operation to fail.



(a) Manifestation Types

(b) Manifestation vs Dim.

Figure 2.3: **Manifestation types: severity and dimensional break down.** *In figure (a), the third, fourth and fifth columns are a break down of the second column. Also in figure (a), the y-axis represents % of the total, since the categories are exclusive, while in (b) the percentages are relative given that the categories are non exclusive. In the x-axis of all figures, "de" stands for "deployment", "tt" for "test total", "ah" for "ad-hoc", "bm" for "benchmark" and "st" for "stress", while in figure (b) title "Dim." stands for "Dimension" (section 2.1).*

2.3 Testing or Deployment: *When* do scalability faults manifest?

According to our observations, over **61%**faults were found when the system was already deployed while **39%**were detected before using *tests*, which we further subdivided based on the mechanism in *ad-hoc* testing, were testers create their own workload using provided testing APIs [51, 64] and use automated tools to detect possible bottlenecks [75, 78, 90], using known system *benchmarks* [52, 101] and using *stress-test* tools [23, 80]. Below we describe the major traits of our 2 main categories and 3 subcategories and present the related findings.

• Over **61%**of all faults manifest during **deployment (de)**, as shown in the first column of figure 2.3(a). The category not only concentrates the most issues, but also the most severe manifestations, where almost half of the total occurrences killed at least one instance in the cluster. Dimensionally, it concentrates almost all cluster and failure size related issues, as shown in the first column of figure 2.3(b).

• Over **39%**of all faults manifest during **testing (tt)**, as shown in the second column of figure 2.3(a), using APIs and tools as the ones shown in table 2.3. These mechanisms are usually de-

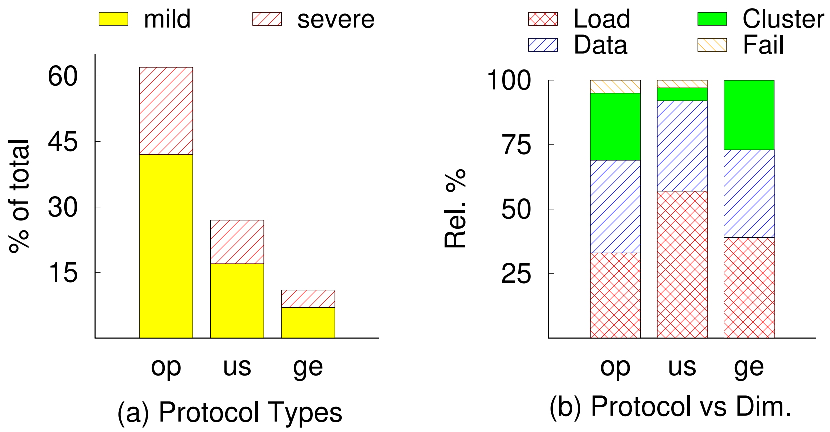
Type	**ha**	**mr**	**hd**	**hb**	**ca**	**kf**	**ig**	**sp**	**st**	**yr**
Test APIs	[51]	[87]	[51]	[55]	[19]	[82]	✗	[99]	✗	[108]
Stress	[53]	[53]	[101]	[54]	[23]	[80]	[67]	[98]	[100]	[107]
Benchmark	[52]	✗	[60]	✗	[20]	[81]	[66]	✗	✗	✗

Table 2.3: **Test APIs, stress-test tools and benchmark examples.** *The APIs, benchmarks and stress-test tools in this table might not be the only ones available. ✗ indicates that, to the best of our efforts, we could not locate any publicly available tool or framework that included reusable code in the related category.*

signed to catch faults in protocols that are related to data and load management, which is why proportionally most of the issues being reported using them are related to those dimensions, as shown in the second column of figure 2.3(b). We further subdivided this category in 3 subcategories related to the test mechanism that was employed by the reporter, which we describe below:

• Over **20%**of all faults and **52%**of the faults found during testing manifest while performing **ad-hoc (ah)** tests, as shown in the third column of figure 2.3(a). As stated before, ad-hoc testing often involves custom APIs that allow developers to create *single-node* clusters, *i.e.*clusters where multiple nodes are deployed in the same machine (but with different communication ports), distributed testing frameworks like *DTest* [64], or simply involve in-house test on production clusters running workloads at scales that closely resemble the scale customers have. Examples of faults found this way include ca-15364, where developers created custom distributed tests and used the aforementioned tools to detect a performance fault when starting a node with too many *unverified* transactions.

• Over **10%**of all faults and **27%**of all faults found during testing manifest while using known **benchmarks (bm)**, as shown in the fourth column of figure 2.3(a). As most of these benchmarks are focused on measuring performance, typically in terms of execution time, most of the faults found using these are related to the data and cluster-size dimensions, their most common target [141, 175], as shown in the fourth column at figure 2.3(b). Examples in this category include ig-12087, where reporters use custom benchmarks to trace down performance regressions related to the data dimension and kf-4444, where an operational protocol involves super-linear iteration over possibly large data structures (as discussed in section 2.5.1), causing an impact on shutdown-



(a) Protocol Types

(b) Protocol vs Dim.

Figure 2.4: **Protocol types: dimensional break down.** *In figure (a), the y-axis represents % of the total, since the categories are exclusive, while in (b) the percentages are relative given that the categories are non exclusive. In the x-axis of all figures, "op" stands for "operational", "us" for "user-facing" and "ge" for "general", while in figure (b) title "Dim." stands for "Dimension" (section 2.1).)*

command's performance.

• Finally, over **9%**of all faults and **21%**of all faults found during testing manifest while using **stress-test (st)** tools, as shown in the fifth column of figure 2.3(a). Being tools mostly focused on creating different levels of load with varied levels of customization, it is not surprising that over **77%**of these are related to the load-size dimension, as shown the fifth column of figure 2.3(b). Typical examples related to the load and data-size dimensions include reports of high contention under load, as in hd-14997, or reports of running out of memory while stress-testing using large-sized requests, as in ca-14747. Examples of cluster-size related issues discovered using these kind of tools include hb-10501, where faulty logic in region assignment tend to produce overloaded (too many regions) *Region Servers*.

2.4 Faulty Protocols: *Where* do scalability faults manifest?

Besides the main read/write *user-facing* protocols, we identified many *operational* protocols, which perform background and on-demand management operations. Examples of these are *compaction* and *scrub* operations in Cassandra [18, 22], were the former is an operation that can happen in the background when certain conditions are met or be triggered manually by operators using an administrative interface [21], and the later is a data-cleaning operation triggered by operators. In addition to these two types of protocols, we also identified *general* code paths, where the reported faults do not explicitly belong to any operational or user-facing path but are part of a more general component made to be reused across the whole system. Below we describe the major traits of our 3 categories and present the related findings.

• **Operational protocols (op)** are involved in **62%**of the total faults, as shown in the first column of figure 2.4(a). As operational protocols often deal with coordination among peers, membership and failures in those operations, they tend to concentrate cluster-size and failure-size related faults, as shown in the first column of figure 2.4(b), the later being extremely rare in other categories. Examples in this category are very diverse and include a case where a non-scalable membership protocol implementation lead to intermittent changes in membership in large scale deployments, as in ca-3881, and excessive cross-system communication during partition reassignment, as in kf-6134.

• **User-Facing protocols (us)** are involved in **27%**of the total faults, as shown in the second column of figure 2.4(a). As all user-facing operations deal with either adding or retrieving data from the system through a myriad of mechanisms expressed as different types of requests, it is not surprising that almost **92%**of these faults are related to the *data* and *load* dimensions, as shown in the second column of figure 2.4b). Examples of this type of faults are related to cluster-size related resource leaks as in kf-1567, load-size related resource leaks as in ha-16242 and failure storms caused by failed read requests as in mr-2947.

• **General protocols (ge)** are involved in **11%**of the total faults, as shown in the third column of figure 2.4(a), and are related to code paths that can belong to both operational and user-facing protocols. They involve code that is designed to be reused like shared data structures, common cross-cutting code paths implementing desirable system's properties, like consistency or durability and connection and other resource managers. Examples in this category include ca-5506, where the fault is related to non-scalable data structure design that uses too much memory when instantiated many times and ig-8681, where a super-linear implementation of a consistency-related operation triggered from both operational and user-facing paths degrades performance.

2.5 Root causes

In this section we present our findings related to the 4 main root cause categories and 11 subcategories we identified. In section 2.5.1 we discuss *compute* faults, our most common root cause typically related to performance and contention. In section 2.5.2 and section 2.5.3 we discuss *unbound* and *bloat* faults, generally related to resource consumption, especially memory. Finally, in section 2.5.4 we discuss *logic* issues that tend to cause faults at scale, including *leaks*, *races* and even *corner cases*.

2.5.1 Compute Faults

Compute faults are related to explicit iteration of *scale dependent data structures*, previously noticed at [139, 165] and defined there as data structures whose size grows as the size of a specific dimension (*e.g.*#files, #peers) grows. It is our largest category, representing **41%** of the total reports and includes 3 subcategories:

• **compute-cross** faults, accounting for **32%** of this category, where external API calls (*e.g.*methods from pluggable components or external clients) or IO operations (disk or network) are performed while iterating scale dependent data structures, creating an amplification effect that is correlated to one or more scale dimensions. These faults are based in known performance antipatterns [88] and have been observed in several types or architectures [119, 176, 180]. An example is shown in figure 2.5(a). There, each Kafka *topic* owns one or more *partition*. Updates to this ownership, invoking the method *updateOwner*, trigger synchronous communication with a cross-layer component (Zookeeper, at line 5). As the number of partitions grows, the number of synchronous messages grows, thus the execution time of the whole operation becomes dependent on the network

```
1  void updateOwner(Partition[] ps){
2    //for every partition
3    for(Partition p: ps){
4      //cross-system call
5      ZKClient.send(p.id, p.owner);
6    }
7  }
```

```
1  void updateTokens(Token[] ts){
2    writeLock();
3    for every token
4    for(Token t: ts){
5      if(!cachedTokens.contains(t)){
6        cachedTokens.add(t)
7      }
8    }
9    writeUnlock();
10 }
```

```
1  void unwindEvicts(Partition[] all){
2    for every partition
3    for(Partition o : all){
4      for every token
5      for(Partition i : o.versions()){
6        if(i.isExpired()) {
7          i.markForRemoval();
8        }
9      }
10   }
11 }
```

(a) compute-cross (b) compute-sync (c) compute-app

Figure 2.5: ***Compute* fault code samples.** *Based on real world faults, (a) is based on kf-5642 and shows a compute-cross fault when communicating with Zookeeper's client in a synchronous fashion in Kafka. (b) is based on ca-5456 and shows a compute-sync fault in Cassandra where frequent, long and blocking consistency management operations collide with membership updates, putting the later in distress. Finally, (c) is based on ig-8681 illustrates compute-app faults, where albeit no heavy or expensive (e.g.IO) operations are present, the loop nesting and its location (inside a performance-critical path) creates an unnecessary performance issue.*

speed and the load on the external component.

- **compute-sync** faults, representing **39%** of this category, where iteration of scale dependent data structures is protected by global *locks*, creating a synchronization bottleneck where the amount of time every other thread needs to wait for said lock is correlated to one or more scale dimensions. This type of faults have been observed frequently [165, 180] in centralized architectures such as HDFS. In this category, we included both cases where the locks wrap scale dependent iteration and cases where the locks are located *inside* the iteration, where the former is the most frequent. An example is shown in figure 2.5(b). There, the method *updateTokens* is invoked every time *range movements* are performed in order to maintain consistency when a Cassandra cluster's topology is changing. But since this operation is performed holding a global lock (line 2), other threads, in particular the thread that handles membership changes, have to wait until the whole operation finishes. As the number of data partitions grow (or the cluster becomes larger), the execution time of the code block between lines 4 and 8 grows too, causing a negative impact on elasticity, akin to what was observed at [139, 165].

- **compute-app**, accounting for **29%** of this category, where iteration of scale dependent data

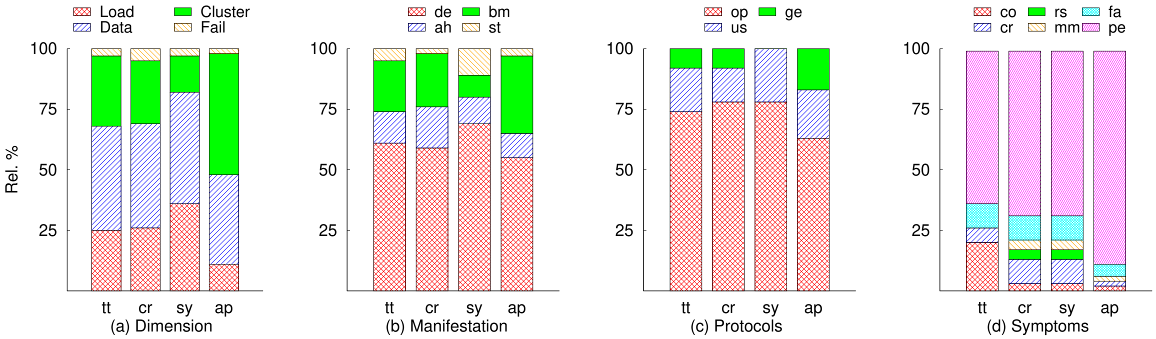


Figure 2.6: ***Compute* faults: dimensional, manifestation, protocol and symptom break down.** *In all figures the y-axis represents the relative percentage of the category. Also, "tt" stands for "total", "cr" for "compute-cross", "sy" for "compute-sync" and "ap" for "compute-app". Other abbreviations are discussed at section 2.1, section 2.2, section 2.4 and section 2.2 respectively.*

structures, typically in a nested fashion, is placed in performance critical paths, creating bottlenecks. We grouped in this category faults where albeit the computation(s) inside the iteration itself are not considered *costly*, the common nesting is $>= 2$, meaning that the correlation between one or more scale dimensions influences execution time in a quadratic fashion. An example is shown in figure 2.5(c). There, *unwindEvicts* is invoked every time a remote Ignite command is processed. This method iterates two scale dependent collections of $O(P = \#\text{partitions})$ size (lines 3 and 5) in a nested fashion, adding an $O(P^2)$ complexity factor to every remote command execution. Since the later are considered part of the critical path and the execution time of this method grows as the number of partitions grow, invoking this method was considered problematic.

• **Discussion**: Dimensionally, as shown at the first column of figure 2.6(a), *compute* faults seem to be highly correlated to the cluster-size and data-size dimensions, which sounds intuitive considering that iteration tends to happen over *collections* of elements such as *lists of peers*, *lists of blocks*, *lists of files*, and others. The subcategory breakdown shows that *compute-app*, *i.e.*faults related to nested loops, are the ones that concentrate the impact on the cluster size dimension, while this dimension seems to be least commonly involved in *compute-sync* faults, *i.e.*, lock-contention related faults. According to our observations, this is related to the fact that what is being protected by the aforementioned locks is usually data and, very uncommonly, cluster wide *bookkeeping* data structures, such as the ones identified as culprits in [139, 165].

```
void processRequest(Request r){
  // Execute
  Result re = r.execute();
  if(re.isDone()){
    // no admission control
    responseQueue.add(re);
  }
}
```

(a) unb-collection

```
void loadCache(Domain[] ds){
  for(Domain d : ds){
    // Bring all into memory
    Entity[] es = d.load();
    for(Entity e : es){
      // Accumulate in cache
      cache.put(e);
    }
  }
}
```

(b) unb-alloc

```
void processRequest(Request r){
  Result re = r.execute();
  // one connection per request
  Socket connection =
    new Socket(r.ip, r.port, 10);
  replyTo(connection, rs);
}
```

(c) unb-os

Figure 2.7: ***Unbound* fault code samples.** *Based on real world faults, (a) is based on ca-15013 and illustrates a case where large, in terms of size, responses accumulate in an Java Executor's processing queue (unbounded) until the node ends up running out of memory. (b) is based on yr-7147 and illustrates a case where a component tries to load large, in terms of size, domain entities, consuming all of the application's memory and failing to start. Finally, (c) is based on ha-15696 and shows a case where each request is processed using a single socket. When the system is flooded with requests, the machine runs out of file descriptors and consumes an excessive amount of memory.*

As shown in figure 2.6(a), several of these faults are found in deployment, which is related to the fact that most of these faults are located in operational protocols, as shown in 2.6(c), an observation that also supports the claims at [165] on those being typically under tested when compared to user-facing protocols. Figure 2.6(b) also shows that only around 25% of the faults can be caught using one of the known benchmarks or stress-test tools, an apparently poor effectivity. Finally, as iteration is commonly related to performance, it is not surprising that the later is the most common observed symptom, as shown in figure 2.6(d), others being failures and contention, intuitively related to the fact that slow performance might imply timeouts and the already discussed contention between operational and user-facing protocols.

2.5.2 Unbound faults

Unbound faults are related to data structures that grow without bounds, generally in terms of size, when one or more dimensions grow. The category is closely related to unbounded resource consumption and message processing protocols (*e.g.*RPC or intra-cluster communication), as the later can be considered *implicit* iteration over *implicit* scale dependent data structures whose size depends on the amount of load. This category represents **23%** of the total reports and includes 3 subcategories:

• **unb-collection** faults, accounting for **43%** of this category, where a long-lived data structure, such as inbound/outbound message processing queues or multi-purpose caches, grow without bounds in response to the growth of one or more dimensions, typically a combination of load and/or data size. According to our observations, these long-lived data structures tend to contain objects of variable size and have none, or faulty, access control mechanism. An example is shown in figure 2.7(a). There, Cassandra message response queue get filled up by pending responses (line 7) of variable and possibly large size. If the responding threads (which consume those objects) cannot keep up, the queue keeps growing without bounds and the node runs out of memory.

• **unb-alloc** faults, representing **35%** of this category, where temporary memory allocations, such as stack-level data structures (*e.g.list* of rows loaded from an on-disk table or *set* of peers used for speeding up a search algorithm), grow without bounds in response to the growth of one or more dimensions. This category can be intuitively related to the lack of *buffering* or *paging* when loading from a remote source, such as a file or even a database, as observed by others [89, 119]. An example is shown in figure 2.7(b). There, the method *loadCache* is invoked every time the *Timeline Server* [103] component of Yarn is started. The application's cache is loaded from on-disk contents (line 7), but to do so the whole file is materialized in memory in one method call (line 4), causing the component to run out of memory when the related files get too big (contain many records). Notice that in this case the component is killed and it can be restarted only if there is more memory available or the caching feature is disabled.

• **unb-os** faults, accounting for **22%** of this category, where OS resource allocation, such as threads and file descriptors, grow without bounds in response to the growth of one or more dimensions. These faults tend to be uncommon in newer versions since the unbounded increase of threads and/or sockets is a known issue (*e.g.*, thread-per-request is known to be a bad choice) and is typ-

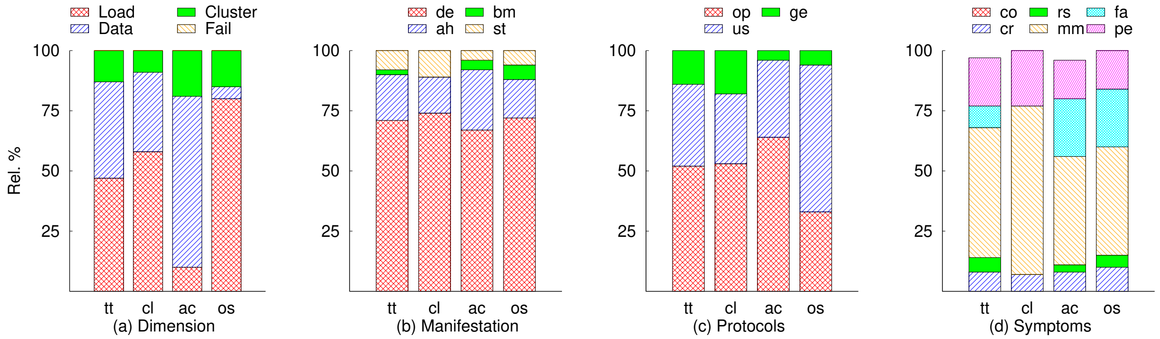


Figure 2.8: ***Unbound* faults: dimensional, manifestation, protocol and symptom break down.** *In all figures the y-axis represents the relative percentage of the category. Also, "tt" stands for "total", "cl" for "unb-collection", "ac" for "unb-alloc" and "os" for "unb-os". Other abbreviations are discussed at section 2.1, section 2.2, section 2.4 and section 2.2 respectively.*

ically targeted in early designs using frameworks like [69]. Notice that just as in *unb-collection*, this category is strongly related to RPC or message processing paths (or *implicit* iteration). An example is shown in figure 2.7(c). There, an encryption-related HDFS component creates a single *Socket* (with a corresponding file descriptor) every time a request is processed. Even if these connections are meant to be short-lived, as the default idle-timeout for it is set to 10 seconds (line 5), an explosive increase in the amount of requests (by calling method *processRequest*) ends up in failures (system run out of file descriptors) and performance impacts (due to the increase of memory consumption).

• **Discussion**: In terms of dimensions, as shown in figure 2.8(a), load-size is clearly the dominant category in every case except in *unb-alloc*, where what is being loaded into memory is not necessarily related with load-size, but more likely with data-size. According to what we have observed, the cluster-size dimension might play an amplifying role when, for example, a centralized component loads lists of *blocks* [61], where the size of that list is proportional to said dimension, a pattern that is common in HDFS (as in *blocks per datanode*).

Figure 2.8(b) shows how ineffective benchmarks and stress-tests are in locating these type of faults are. We expected benchmarks to be a little more useful for *unb-alloc*, where the same framework and code used to measure execution time could be used to check memory consumption, akin to how [76] is used in some cases. On the other hand, as *unb-collection* and *unb-os* are closely

```java
class StatsMetadata {
  // Heavy object
  byte[] min = new
    byte [1000000];
  byte[] max = new
    byte [1000000];
  // ...
}
```

(a) bloat-waste

```java
class IndexSummary {
  // long[] is preferable
  List<Long> pos =
    new ArrayList<>();
  // byte[][] is preferable
  Map<Byte, Byte> keys =
    new Map<>();
  // ...
}
```

(b) bloat-opt

Figure 2.9: ***Bloat* fault code samples.** *Here, (a) is based on ca-15400, illustrating a case in which a class is designed with a heavy and wasteful overhead, while (b) is based on ca-5506 and shows a class design that uses java collections, thus bloats memory as described in [134].*

related to message processing protocols, we expected stress-test to be useful in these cases, but according to our observations they are rarely used for this purposes (or are rarely useful for this type of efforts).

Finally, the reported symptoms are dominated by memory consumption, failures and performance, as shown in figure 2.8(d). The later sounds surprising but is mainly due to the fact that since all the target systems are Java based, frequent GC [70] is always one of the symptoms observed. As the later is an operation typically performed by many threads in a blocking fashion, performance tends to suffer. Unexpectedly, not many issues report suffering from machine-level resource shortage, which according to our observations tends to be related to the fact that typical connection-per-request models involve creating one thread to handle said connection. As each of those threads uses memory to, for example, allocate a stack [73], reporters tend to see severe memory issues *way before* file descriptors are exhausted.

2.5.3 Bloat faults

Bloat faults are generally related to data structures for which their design limits the amount of times they can be instantiated, thus becoming problematic when that amount is correlated with the size of one or more dimensions. Albeit these bloating or sometimes sub-optimal *scale dependent*

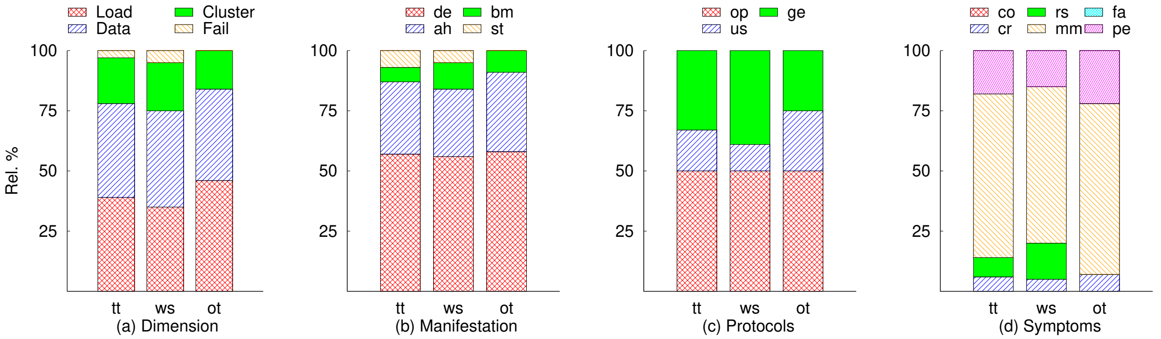


Figure 2.10: ***Bloat* faults: dimensional, manifestation, protocol and symptom break down.** *In all figures the y-axis represents the relative percentage of the category. Also, "tt" stands for "total", "ws" for "bloat-waste" and "ot" for "bloat-opt". Other abbreviations are discussed at section 2.1, section 2.2, section 2.4 and section 2.2 respectively.*

structures do not have specific purposes nor belong to specific domains, their design seems to follow some known antipatterns related to over-engineering or "just in case" programming [150]. This category represents **8%** of the total reports and includes 2 subcategories:

• **bloat-waste** faults, accounting for **72%** of this category, related to design issues where unnecessarily large fields such as fixed arrays or buffers are referenced, either directly or indirectly, by *scale dependent structures*, causing memory pressure. We consider this type of bloat to be a waste, but make the observation that the aforementioned large fields might be necessary for some instances (*i.e.*, they are used at their capacity) while in others they might be wasteful. An example is shown in figure 2.9(a). There, Cassandra creates one *StatsMetadata* instance per each *BigTableReader*, an internal data structure related to the load dimension. The former declares two fixed-size 1M buffers that, according to developers, are frequently not used in their entirety. When under high load, many live *BigTableReader* instances lead to high memory usage and eventually ends in out of memory issues, affecting not one but several nodes in the cluster.

• **bloat-opt** faults, representing **28%** of this category, related to design optimizations in which developers report that certain fields of a *scale dependent structure* could be represented using less space in order to decrease memory consumption when many of these instances are alive. According to our observations, these correspond mostly to optimizations and design improvements that allow operators to alleviate memory consumption without resorting to vertical scaling. An

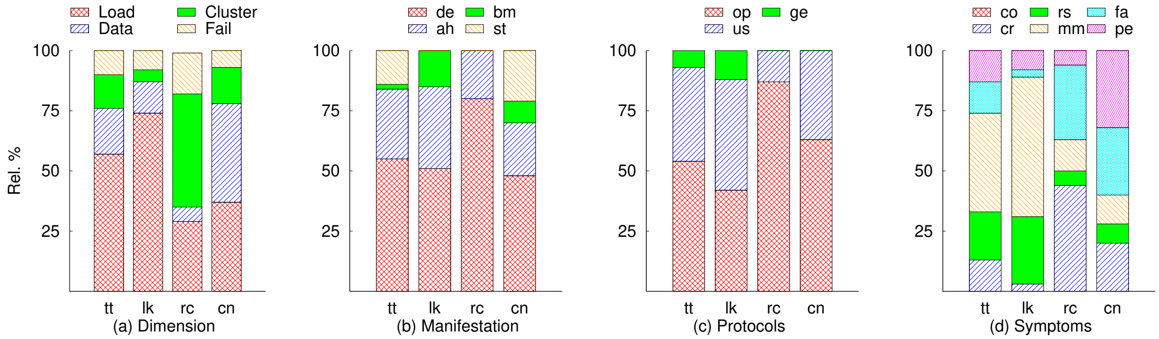


Figure 2.11: ***Logic* faults: dimensional, manifestation, protocol and symptom break down.** *In all figures the y-axis represents the relative percentage of the category. Also, "tt" stands for "total", "lk" for "logic-leak", "rc" for "logic-race" and "cn" for "logic-corner". Other abbreviations are discussed at section 2.1, section 2.2, section 2.4 and section 2.2 respectively.*

example is shown at figure 2.9(b). There, Cassandra nodes use a *scale dependent structure* called *IndexSummary* which reportedly uses the collections framework [68, 134, 150] to store the internal representation of said index. As the later uses objects references, a 2X per-element overhead in positions (line 3-4) and over 10X in keys (line 6-7) caused a 70% overhead in a cluster with billions of rows.

• **Discussion**: In terms of dimensions, as shown in figure 2.10(a), there is a dominant trend of both data-size and load-size, but along with *compute* faults, this category also concentrates several cluster-size related issues. This is mainly due to *state* related data structures, accumulating *per-peer* or *per-replica* elements with a bloating design. The intuition behind this claim is also reflected in 2.10(c), where *general* protocols are present in a fair share of these faults, indicating these *scale dependent structures* are likely cluster-wide bookkeeping-related classes.

According to 2.10(b), detecting these faults during testing is not as complex as *unbound* faults, which we attribute to the fact that this type of memory related issues are related to design thus are observable regardless of the workload, making them easier to observe than the aforementioned class using standard tools [75, 77, 78].

2.5.4 Logic faults

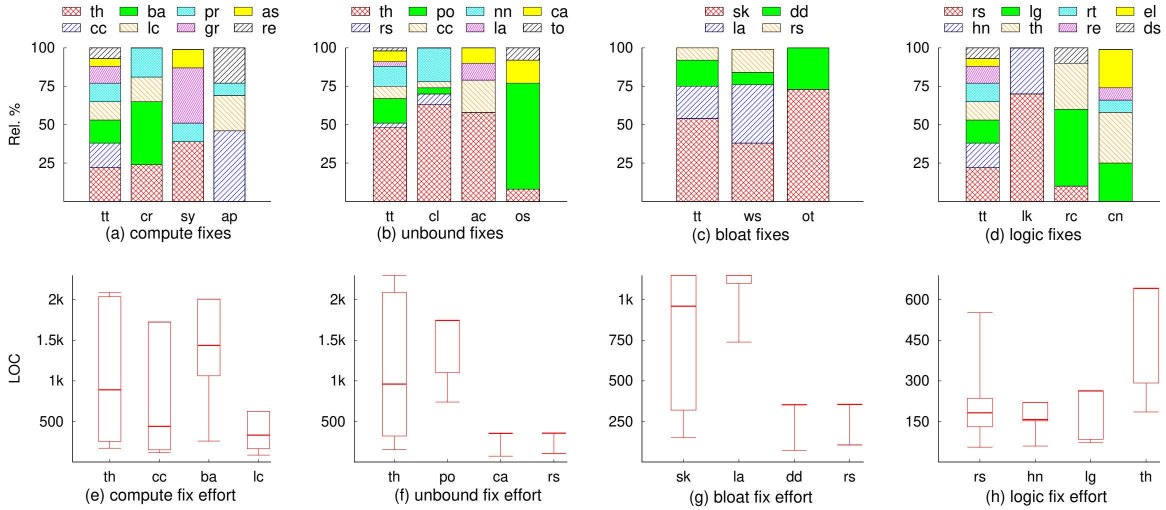
Logic faults are related to logic corner cases that become problematic only at large-scale, but not (or do not appear to be of concern) at smaller scales. This category represents a surprising **28%** of the total reports and includes memory and resource *leaks* (**60%** of the category), data *races* (**15%** of the category) and small *corner cases* in logic that become apparent at large-scale (**25%** of the category).

Examples of leaks and races correspond to the classical descriptions given at [123, 133, 137, 138, 144], and can be intuitively related to large-scale deployments. Fault report samples for those two categories can be seen at hd-13039 and ha-16385 respectively. Examples for *corner cases* wildly vary, ranging from limiting data type choices, as in st-3256 where choosing *short* over *int* for storing thread ids limits the amount of possible threads, a "disaster for large scale computing" according to the reporter, to *chicken-and-egg* like problems such as ig-12042, where removing one *page* from a fully populated *data region* requires allocating one more page in said region.

The dimensional, protocolar and symptomatic breakdown of this category is shown at figure 2.11, where the load-size dimension dominates on *leaks* while the cluster-size dimension plays an important role in *races*, the reason being most of the races we detected are between operational and user-facing protocols depending on each other. Regarding symptoms, *leaks* clearly concentrate resource-related ones while *races* concentrate failures and crashes. Finally, the diversity of *corner cases* is shown in the fourth column of figures 2.11(a)-(d), where no clear trend can be observed.

2.6 Solution patterns and engineering effort

According to our findings, around **87%** of the issues in our database have at least one publicly available patch, the median being 2 patches and the $90th$ percentile rounding 10 patches. The amount of changes in those, in terms of LOC (lines of code, including insertions and deletions) moves between 94 (median) and 625 ($90th$ percentile) LOC. In our analysis we identified a total of **28** different patterns and we describe the **21** most prominent below. To put them in context, we



(a) compute fixes (b) unbound fixes (c) bloat fixes (d) logic fixes

(e) compute fix effort (f) unbound fix effort (g) bloat fix effort (h) logic fix effort

Figure 2.12: **Common solution techniques and effort involved.** *Figures (a)-(d) show the most common solution patterns for each root cause. There, "tt" stands for "total", "lk" for "leak", "rc" for "race" and "cn" for "corner case". The abbreviations shown in the x-axis of figures (a), (b) and (c) are the same used in figures 2.6, 2.8 and 2.10 respectively. Figures (e)-(f) show the amount of effort, in terms of LOC, for the* 4 *most common solution techniques in relationship to the root cause.*

present them along with the related root causes they address.

2.6.1 Solving Compute faults

According to our observations, solution techniques for this type of faults fall in 8 known patterns, **throttling (th)**, **batching (ba)**, **parallelizing (pr)**, **asynchronous implementation (as)**, **caching (cc)**, **lazy-computing (lc)**, **granularity-modification (gr)** and **removing (re)**. The frequency of each technique is shown in figure 2.12(a), where the first column groups the total of this root cause category while the subsequent columns refer to each subcategory.

Compute-cross faults, related to explicit iteration containing cross-layer communication, have solution patterns related to *batching*, when the issue is related to too many roundtrips [88, 119, 145, 176], *parallelizing*, consisting in implementing a parallel version of an algorithm to avoid contention and in this case used when the cross-layer communication is slow and *lazy-computing*, *i.e.*avoid the computation at all and resort to perform it only in certain cases. Regarding *compute-sync* faults, the dominating fixes are related to *throttling* mechanisms, where the goal is to release the locks involved while iterating effectively *rate-limiting* contention and allowing more fairness, and *granularity-modification*, consisting in shrinking the critical section or even moving it outside the iteration, if possible at all.

Compute-app faults tend to involve expensive computation in critical paths and are typically solved by *caching*, *i.e.*avoid performing expensive computations too frequently and instead store the results, *lazy-computing* also avoiding the need to compute frequently by resorting to last-minute computations or simply by *removing* unnecessary computations from critical paths. Notice that the later is not always possible, thus the first two techniques decrease the frequency of the computation but do not avoid it in its entirety. Finally, figure 2.12(e) shows that the effort for the 4 most common techniques, *throttling*, *caching*, *batching* and *lazy-computing*, ranges between hundreds and 2000 LOC. This variation is mainly due to the fact that creating a throttling mechanism to solve *compute-sync* faults (*i.e.*, lock throttling) is much less expensive than creating a throttling mechanism from the ground to avoid cross-layer calls, where the later might involve new unit tests, new parameters and changes in other layers.

2.6.2 Solving Unbound faults

According to our observations, solution techniques for this type of faults fall in 8 known solution patterns, **throttling (th)**, **pooling (po)**, **capping (ca)**, eager **resource-release (rs)**, **caching (cc)**, **lazy-allocation (la)**, creating **timeout-mechanisms (to)** or simply **none (nn)**, since this root cause category concentrates most of the unsolved issues. The frequency of each technique is shown in figure 2.12(b), where the first column groups the total of this root cause category while the subsequent columns refer to each subcategory.

Unb-collection faults are complex to solve from the point of view of code, since they are related to memory pressure thus depend on the amount of memory available. Here, *throttling* dominates, due to its relationship with Little's Law [143], along with eager *resource-release*, as in releasing resources as fast as possible to make more room for concurrent memory-hungry tasks. In *unb-alloc* faults, the dominating fixes are related to *throttling* mechanisms (*e.g.*paging), *caching* to avoid too many or too large allocations, *lazy-allocation* to resort to allocate only when strictly necessary or simply *capping*, creating a hard-cap on the amount of elements that can be loaded and failing when that cap is exceeded, thus avoiding the issue.

In the case of *unb-os* faults, *pooling*, *i.e.*create pools of resources instead of unboundedly allocating them, is the dominating category and the most intuitive solution. In the same spirit as eager resource-release, a *timeout-mechanism* can be created to release resources and avoid faults when too many requests come at the same time. Finally, figure 2.12(f) shows that the effort for the 4 most common techniques, *throttling*, *pooling*, *capping* and *resource-release*, ranges between hundreds and 2000 LOC. Complex solutions like *pooling* or *throttling* clearly require more lines of code due to their complexity. On the other hand, *capping* and *releasing* tend to be easier and have less side effects, requiring less tests and less code in general. The former is typically designed as a *fail-fast* mechanism, *i.e.*fail when the cap is exceeded, making its implementation less complex and even requiring less testing.

2.6.3 Solving Bloat faults

According to our observations, solution techniques for this type of faults fall in 4 known solution patterns, **shrinking (sk)**, **deduplicating (dd)**, **lazy-allocation (la)** and **resource-release (rs)**. The frequency of each technique is shown in figure 2.12(c), where the first column groups the total of this root cause category while the subsequent columns refer to each subcategory.

Bloat-waste faults are generally solved by *shrinking* the bloating data structure (reduce the weight), creating *lazy-allocation* mechanisms to reduce memory pressure or simply by eager *resource-release*, where the target resource is memory and the mechanism is to allocate just before the first use. On the other hand, more colorful techniques appear for *bloat-opt* faults, where finding *deduplication* opportunities [131, 149] is an interesting optimization. Others include the ones described at [134], aiming at bloating *java collections* [68, 134, 150]. Finally, figure 2.12(g) shows that the effort for the aforementioned techniques ranges anywhere between hundreds and 1000 LOC, where *shrinking* and *lazy-allocation* require more changes (since they are related to architectural changes) and *deduplicating* and eager *resource-release* require the least effort.

2.6.4 Solving Logic faults

According to our observations, solution techniques for this type of faults fall in 8 known solution patterns, **resource-release (rs)**, creating proper **error handlers (hn)**, **throttling (th)**, small adjustments in **logic (lg)**, **disabling (ds)** a component, **enlarging (el)** a predefined cap, creating **retry (rt)** mechanism or simply **removing (re)** faulty code. The frequency of each technique is shown in figure 2.12(d), where the first column groups the total of this root cause category while the subsequent columns refer to each subcategory.

The first two mechanisms are the ones built to alleviate *logic-leak* faults, where most happen either because there is simply no cleanup logic or said logic does not properly handle error cases. Throttling is sometimes used for *logic-race* faults, as in to slow-down one component to avoid races with others. The rest of the mechanisms are used depending on the specific problem since, as discussed earlier, this category is diverse in terms of software defects. Finally, figure 2.12(g) shows that the effort for the aforementioned techniques is below 650 LOC. This is compliant with the fact that these are small corner cases in logic and according to our analysis most of the effort goes in creating new unit tests or modifying existing ones.

2.7 Conclusions

To the best of our knowledge this chapter presents the largest empirical study on the field, in the hopes on bringing new insights on these intricate faults. We show a wide range of examples and describe situations that are only visible at large-scale deployments. We studied their reported symptoms (section 2.2), related protocols (section 2.4), and detection mechanisms (section 2.3), if any. We further analyzed the faulty code and classified the root causes (section 2.5), or main contributing factor, of each fault. We finally identified the main solution techniques, analyzed the patches and inspected the newly incorporated or modified unit-tests (section 2.6). To summarize, our contributions are the following:

1. To the best of our knowledge, we conduct the largest and deepest study of scalability faults in cloud systems, covering a wide range of architectures and design patterns. Our findings can help understand the main characteristics of scalability faults and provide guidance to future research.

2. The publicly available product of our classifications, SFDB [96], a set of classification text files containing the aforementioned 350 reports classified using the tagging system described at table 2.2.

We hope and believe that the product of our study [96] will be beneficial for the research community and fuel future developments in the field.

CHAPTER 3

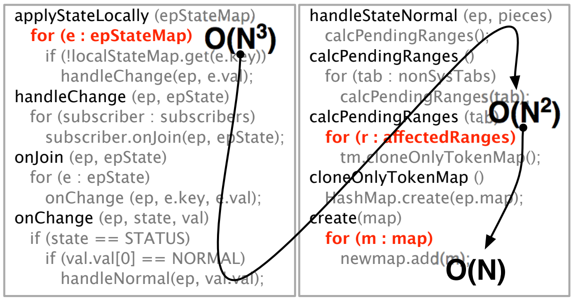
SCALECHECK: A SINGLE-MACHINE APPROACH FOR DISCOVERING SCALABILITY BUGS IN LARGE DISTRIBUTED SYSTEMS

As discussed in section 1.3.2, this chapter presents SCALECHECK [165], a single-machine approach for discovering scalability faults in large distributed systems. The goal of this chapter is to address the following research questions:

- **RQ4: How to discover latent scalability faults?** Scalability faults 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 faults remain unforeseen.

- **RQ5: How to democratize large-scale testing?** According to our study and industrial experience [91], 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, which heavily increases the cost of this practice.

The rest of this chapter is organized as follows: section section 3.1 presents SFIND, a static-analysis tool intended to address the first research question of this chapter. Then, section 3.2 presents STEST, a runtime environment intended to address the second research question of this chapter, discussing our black-box (section 3.2.1) and white-box (section 3.2.2 approaches. In section section 3.3 we show how SFIND and STEST collaborate as a primitive version of our target scale-test pipeline, while in sections 3.4 and 3.5 we discuss implementation details and evaluate the aforementioned components. Finally, sections 3.6 and 3.7 show the limitations of our approach and the conclusions of this chapter.



```
applyStateLocally (epStateMap)
   for (e : epStateMap)
      if (!localStateMap.get(e.key))
         handleChange(ep, e.val);
handleChange (ep, epState)
   for (subscriber : subscribers)
      subscriber.onJoin(ep, epState);
onJoin (ep, epState)
   for (e : epState)
      onChange (ep, e.key, e.val);
onChange (ep, state, val)
   if (state == STATUS)
      if (val.val[0] == NORMAL)
         handleNormal(ep, val.val);
```

```
handleStateNormal (ep, pieces)
   calcPendingRanges();
calcPendingRanges ()
   for (tab : nonSysTabs)
      calcPendingRanges(tab);
calcPendingRanges (tab)
   for (r : affectedRanges)
      tm.cloneOnlyTokenMap();
cloneOnlyTokenMap ()
   HashMap.create(ep.map);
create(map)
   for (m : map)
      newmap.add(m);
```

Figure 3.1: $O(N^3)$ **scale-depended loops.** *The partial code segment above depicts the $O(N^3)$ loops in ca-6127. Note that not all loops are scale-dependent loops. The "epStateMap", "affectedRanges", and "map" variables are not the annotated scale-dependent variables, however* SFIND *taints them with a dataflow analysis.*

3.1 SFIND

Based on the findings presented at section 2.5.1, around **41%** of scalability faults are caused by *scale-dependent iteration*, *i.e.* in-code loop statements (such as for or while loops) that become *scale-dependent* in terms of the number of iterations they need to perform. We further subdivided this category into 3 subcategories:

- **compute-cross** faults, accounting for **32%** of the category, where external API calls (*e.g.*methods from pluggable components or external clients) or IO operations (disk or network) are performed while iterating scale dependent data structures.

- **compute-sync** faults, representing **39%** of this category, where iteration of scale dependent data structures is protected by global *locks*, creating a synchronization bottleneck where the amount of time every other thread needs to wait for said lock is correlated to one or more scale dimensions

- **compute-app**, accounting for **29%** of this category, where iteration of scale dependent data structures, typically in a nested fashion, is placed in performance critical paths, creating bottlenecks.

The focus of this chapter are **compute-app** faults. Here, the first challenge to address is: how to find scale-dependent loops, *i.e.* loops that traverse the aforementioned scale-dependent data

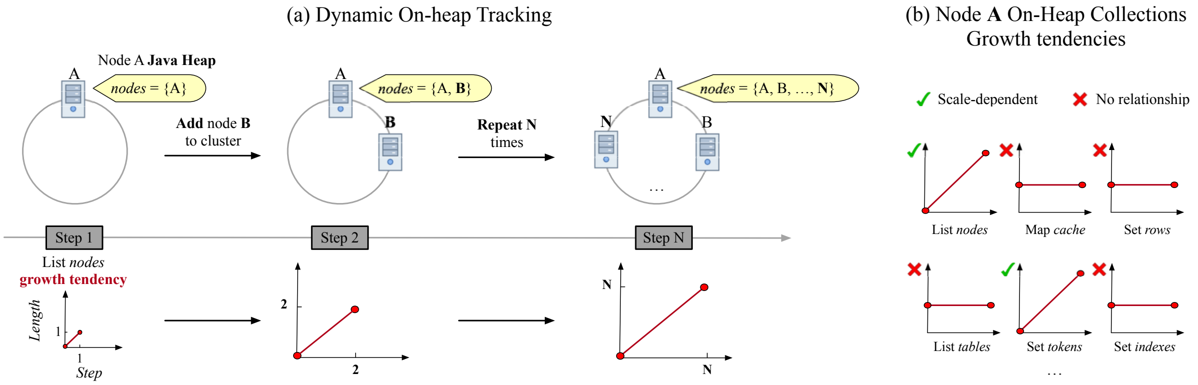


Figure 3.2: **SFind on-heap collection tracking and mapping.** *This figure illustrates the process of auto-tagging scale-dependent collections described in section 3.1. In (a), at every step the cluster is grown by one peer, which increases the size of the list "nodes", an on-heap object [104], by a single element. In (b), as only some of said on-heap collections grow when the cluster size grows, not all collections will be tagged as scale-dependent.*

structures? 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, 10 functions and iterate 3 scale-dependent collections. This difficulty motivates SFind, a generic program analysis tool that helps developers pinpoint scale-dependent loops. Below are the three main steps of SFind.

3.1.1 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, as shown in figure 3.2(a). 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, as shown in figure 3.2(b). This, however, is insufficient due to two reasons. First, there are collections that only grow when background/operational tasks are triggered; 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 [70]. Given that the measurements are taken in 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-based 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 [155] and Reflection [72]. 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 ($N = 30$ 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.

3.1.2 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. With these steps, in figure 3.1 for example, SFIND can mark `applyStateLocally` as an $O(N^3)$ function.

3.1.3 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 [114], 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 [142] 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 unittest 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.

- **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 [131]). Overall, including Cassandra's memory usage, per-node memory consumption reaches 100 MB. Thus, a 32-GB machine can only colocate around 300 nodes.

- **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.

- **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 [116]). Naive packing is infeasible.

Technique	**#Nodes per PC**	**LOC added**	**Colocation bottlenecks**
Black/gray-box approaches (section 3.2.1)			
(a) Naive (NP)	50	–	Memory, proc. switch
(b) SPC	70	–	User-kernel switch
(c) SPC+Stub	120	+91	Context switch
White-box approaches (section 3.2.2)			
(d) GEDA	130	+581	CPU
(e) GEDA+PIL	512	+246	CPU

Table 3.1: **Colocation strategies and bottlenecks.** *Here, "NP" stands for Naive Packing, "SPC" for single process cluster, "SPC+Stub" for single process cluster + network stub (section 3.2.1), "GEDA" for global event-driven architecture and PIL for "Processing Illusion" (section 3.2.2).*

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 [16, 92] because Cassandra (and many other systems) uses "singleton" design pattern for simplicity (but bad for modularity) [97]. 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 avail [16, 92]. We found another way: leveraging class loader isolation support from the language runtime [74], 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. In later (by the time of publication) versions, we found that Cassandra developers also begin to develop a similar method to address this problem [14].

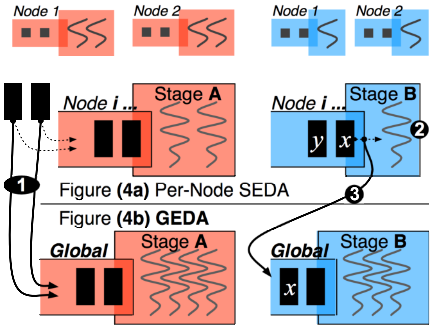


Figure 3.3: **Global Event Driven Arch.** *The figure format follows [173, Figure 6].*

3.2.2 White-box approaches

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 [71], 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 Architecture (GEDA)

To address the problem, we must redesign the target system, but with minimal changes. We leverage the staged event-driven architecture (SEDA) [173] (Figure 3.3(a)), 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 3.3(b). 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 3.3), we redirect them to GEDA-level event queues, to be later processed by GEDA worker threads. This only requires ∼10 LOC change per stage (as we use aspect-oriented programming [12]). 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 3.3 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 3.3), which costs ∼50 LOC change per stage.

Adding GEDA to Cassandra only costs us 581 LOC (table 3.1(d) 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 (section 3.5.4) but for some Cassandra tests, it only enables around 130-node colocation (table 3.1(d)), which we elaborate in the next section.

Processing Illusion (PIL)

Finally, the last challenge we address is: how to produce accurate results (*i.e.*, the same fault symptoms observed in real-scale deployment) when colocating hundreds of CPU-intensive nodes? We found that STEST is sufficient for accurately revealing fault 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 fault symptoms reported by STEST are not accurate. For example, for ca-6127 (figure 1.2), in 256-node real deployment, we observed around 2000 flappings (the fault 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()`. For example, for ca-6127, we can replace the expensive gossip/stage-changes processing (see figures 1.2 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" [171]. 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?** 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 fault symptoms. We name such functions as "PIL-safe functions." We identify two main characteristics of such functions: **(1) they have memoizable output**, this is, a PIL-safe function must have a memoizable (deterministic) output based on the input of the function, and **(2) they do not contain 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 ca-6127, there is a ring-table checkpoint (not shown) needed for fault tolerance but is irrelevant (never read) during bootstrapping.

We extend SFind to SFind$_{PIL}$, which includes a static analysis that finds code blocks in scale-dependent loops that can be safely PIL-ed. SFind$_{PIL}$ 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. 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. A complete description of the related algorithms, can be found at appendix B.

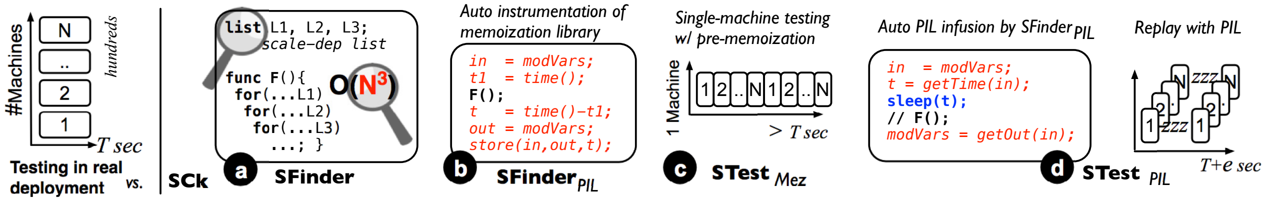
- **How we can produce the output and predict the timing "`t`" if the actual compute is skipped?** As PIL-safe functions no longer perform the actual computation, the next question to address is: how do we manufacture the output such that the global behavior is not altered (*e.g.*, rebalancing protocol 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 3.4(b)).

  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 [94] 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 3.1(e)).

3.3 SCALECHECK

Figure 3.4(a)-(d) summarizes the complete four stages of SCALECHECK: ⓐ SFIND searches for scale-dependent loops which helps developers create test workloads. ⓑ For test workloads that show CPU busyness in all nodes, SFIND$_{PIL}$ finds PIL-safe functions and inserts our pre-memoization library calls. Next, STEST now works in two parts. ⓒ STEST$_{mez}$ (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. ⓓ STEST$_{PIL}$ (with PIL) will then run by having SFIND$_{PIL}$


#Machines
N
..
2
1
hundreds
T sec
Testing in real deployment
vs.
SCk
a
SFinder
Auto instrumentation of memoization library
b
SFinder$_{PIL}$
Single-machine testing w/ pre-memoization
1 Machine
1 2 .. N 1 2 .. N
> T sec
c
STest $_{Mez}$
Auto PIL infusion by SFinder$_{PIL}$
d
STest $_{PIL}$
Replay with PIL
1 2 .. N zzz 1 2 .. N zzz
T+e sec

```
list L1, L2, L3;
  scale-dep list

func F(){
 for(...L1)
  for(...L2)
   for(...L3)
     ...; }
```

$O(N^3)$

```
in  = modVars;
t1  = time();
F();
t   = time()-t1;
out = modVars;
store(in,out,t);
```

```
in  = modVars;
t = getTime(in);
sleep(t);
// F();
modVars = getOut(in);
```

Figure 3.4: **SCALECHECK Testing Pipeline.** *"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 3.1 and section 3.2.* STEST *in stage (c) runs on one machine but will take some time (>T). PIL in stage (d) still runs on one machine but only consumes a similar time as in deployment testing (T +e) and can be replayed numerous times.*

remove the pre-memoization library calls, replace the expensive PIL-safe function with `sleep(t)`, and insert our code that constructs the memoized output data. SCALECHECK also records message ordering during STEST$_{mez}$ and replays the same order in STEST$_{PIL}$ (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 3.4(c)), up to 6x longer time than real-deployment testing (section 3.5.5). However, this is only a one-time overhead. Most importantly, developers can repeatedly re-run STEST$_{PIL}$ (figure 3.4(d)) as many times as needed (tens of iterations) until the fault behavior is completely understood. In STEST$_{PIL}$, 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 Φ algorithm (for ca-6127), caching slow methods (ca-3831), changing lock management (ca-5456), and enabling parallel processing (vd-1212). However, if the fixes involve a complete redesign (*e.g.*, optimized gossip processing in ca-3881, decentralized to centralized rebalancing in rk-3926), STEST$_{mez}$ must be repeated.

3.4 Application and implementation

Table 3.2 quantifies the application of SCALECHECK techniques to a variety of distributed systems, Cassandra [1], HDFS [4], Riak [94], and Voldemort [93]. The major system-specific change is

	Cass	**HDFS**	**Riak**	**Vold**
STest-able systems	918	179	217	800
SFind code	4026 (generic)			
STest library	6047 (generic)			

Table 3.2: **ScaleCheck integration effort (LOC).**

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* [130, 146]. 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 [35] to support Java programs. We left porting to Erlang's parser [36, 37] as future work.

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 (rebalance), and 1 Voldemort (rebalancing) protocols. A protocol can be built on top of other protocols (*e.g.*, bootstrap on gossip and failure detection protocols). Second, for exposing known faults, we applied ScaleCheck to a total of 10 earlier releases: 4 Cassandra (v0.8.9, v1.1.10, v1.2.0, v1.2.9), 4 HDFS (v0.12.3, v0.19.0, v0.23.6, v2.0.0), 1 Riak (v0.14.2), and 1 Voldemort old releases (v0.90.1). For finding unknown faults, we also ran ScaleCheck on recent releases of the four systems (Cassandra v2.2.5, HDFS v2.7.3, HDFS 2.9.0, Riak v.2.1.3, and Voldemort v1.10.21).

3.5 Evaluation

We now evaluate ScaleCheck: Is ScaleCheck effective in exposing scalability faults (sections 3.5.1-3.5.2), accurate (section 3.5.3), scalable and efficient (sections 3.5.4-3.5.5)? We com-

Fault#	**N**	**Protocol**	**Metric**	T_m	T_{pil}
ca-6127 [15]	≥256	Bootstrap	*#flaps*	2h	15m
ca-3831 [15]	≥256	Decomm.	*#flaps*	17m	9m
ca-3881 [15]	≥64	Add nodes	*#flaps*	7m	5m
ca-5456 [15]	≥256	Add nodes	*#flaps*	16m	4m
rk-3926 [95]	≥128	Rebalance	T_{Comp}	6h	2h
vd-1212 [105]	≥128	Rebalance	T_{Comp}	22h	–
hd-9198 [58]	≥256	Blk. report	Q_{Size}	8m	–
ha-4061 [57]	≥256	Decomm.	T_{Lock}	6h	–
ha-1073 [49]	≥512	Pick nodes	T_{Comp}	1m	–
hd-395 [56]	≥512	Blk. report	T_{Comp}	5m	–

Table 3.3: **Fault benchmark.** *The table lists the scalability faults we use for benchmarking* SCALECHECK*. "**ca**" stands for Cassandra, "**hd**" for HDFS, "**rk**" for Riak, and "**vd**" for Voldemort. The "**N**" column represents the #nodes for the fault symptoms to surface. The "**Metric**" column lists the quantifiable metrics of the fault 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 (*STEST$_{mez}$*) and PIL replay (*STEST$_{PIL}$*) stages when N≥256, as discussed in section 3.5.5. "–" implies PIL is unnecessary.*

pare 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.

3.5.1 Exposing scalability faults

Table 3.3 lists the 10 real-world faults we use for benchmarking SCALECHECK. We chose these (among the 55 faults we studied) because the reports contain detailed descriptions of the faults, which is important for us to create the "input" (*i.e.*, the test cases). Figure 3.5 shows the accuracy of SCALECHECK in exposing the 10 faults using the "bug-symptom" metrics in table 3.3 (the first fault, ca-6127, will be shown later in section 3.5.3 and the last fault, hd-395, is omitted in figure 3.5 for space).

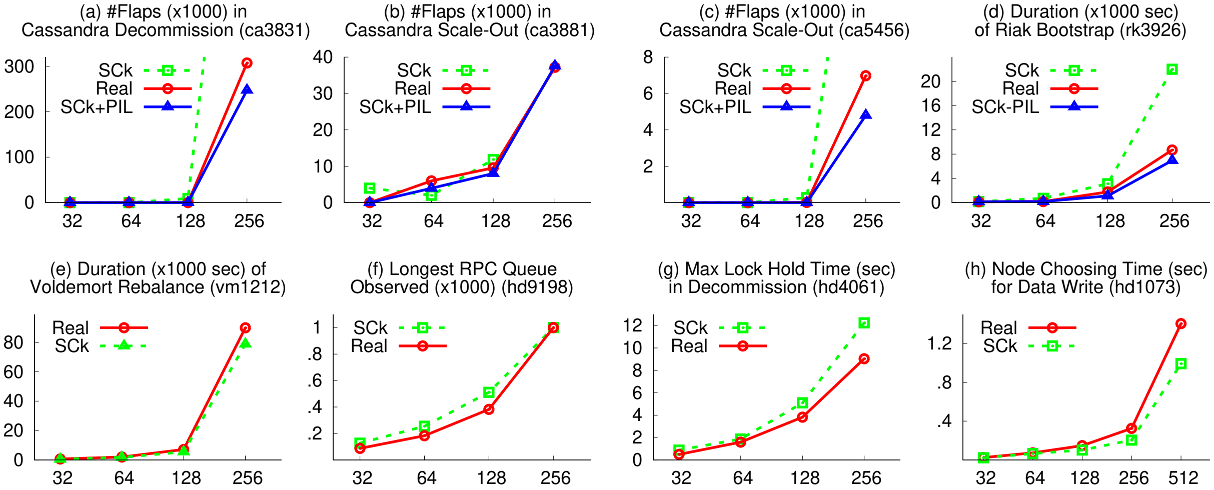


Figure 3.5: **SCALECHECK effectiveness in exposing scalability faults.** *"SCk" represents* SCALECHECK. *The faults are listed in table 3.3. The* ***x-axis*** *represents the number of nodes (N). The figure title describes the* ***y-axis****, i.e., the fault symptom metrics as recorded in "Real" deployment vs.* SCALECHECK. *For Cassandra and Riak faults (a-d), where all nodes are CPU-intensive, the fault symptoms are inaccurate without PIL ("*`SCk`*" lines). However, with PIL ("*`SCk+PIL`*" lines), the fault symptoms are relatively accurate as in the real deployment scenarios. For Voldemort and HDFS faults (e-h), where there is no concurrent CPU busyness, PIL is not needed.*

Results summary

First, SCALECHECK is effective and accurate in exposing scalability faults, some of which only surface in 256+ nodes. As shown, for Cassandra and Riak faults where all nodes are CPU intensive, PIL is needed for accuracy (`SCk+PIL` vs. `Real` lines in figures 3.5(a)-(d)), but for the rest, STEST suffices (`SCk` vs. `Real` in 3.5(e)-(f)).

Second, SCALECHECK can help developers prevent recurring faults; the series of Cassandra faults (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.

Fault descriptions

We now describe the faults and scale-dependent collections involved.

- In ca-6127 [15] protocol bootstrap (from scratch, without data) has complexity $\sim O((np)^3)$ (where n is the number of nodes and p is number of vnodes) because for each entry in the message, it needs to iterate on every vnode in the ring and clone the ring. 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 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 collection of live endpoints (each peer ip address, field of the class *Gossiper*) and token (peer data) metadata (fields of the class *TokenMetadata*).

- In ca-3831 [15] protocol commission/decommission has complexity $\sim O(n^4 * (log(n))^3)$ (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 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*).

- ca-3881 [15] is similar to ca-3831. After patching ca-3831 developers realized that even if the maximum practical size of a cluster was improved the solution was not robust yet.

- In ca-5456 [15] protocol commission/decommission acquires a lock for computation with complexity $\sim O((np)^2 * log(np))$ (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*).

- In rk-3926 [95] 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).

- In vd-1212 [105] 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.

- In hd-9198 [58], 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*).

- In ha-4061 [57], 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*).

- In ha-1073 [49], 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*).

- In hd-395 [56] is similar to hd-9198, but in this case the block report contained all blocks (and not only the ones with changes), thus wasting CPU (processing blocks that have not

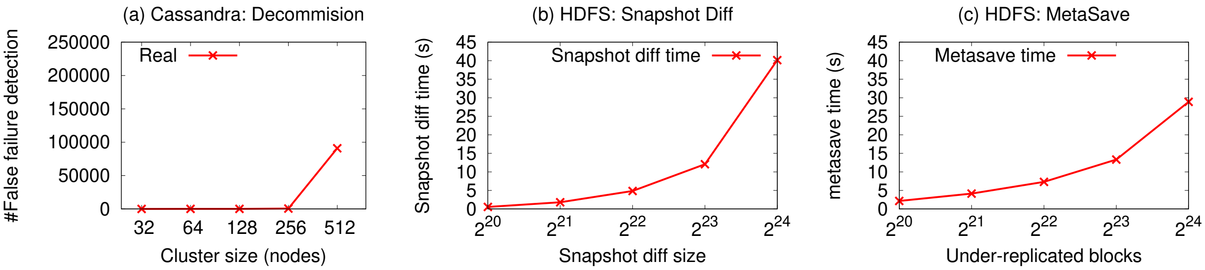


Figure 3.6: **Discovering unknown faults.** *The **x-axis** represents the number of nodes (a) and the number of blocks (b-c). The **y-axis** represents the observed fault symptom and the figure title shows the system and the workload under test. In (a) the symptom being observed is the number of flaps (number of times a node is declared "dead" incorrectly while the cluster membership is not settled). In (b-c) the symptom being observed in the amount of time (in seconds) the namenode global lock is being held (while the workload is being executed).*

changed) and degrading namenode performance.

3.5.2 Discovering unknown faults

We also integrated SCALECHECK to more recent stable versions of Cassandra, HDFS, Riak, and Voldemort, and found 1 unknown fault in Cassandra and 3 faults in HDFS. Below we describe the details of such faults.

- In the fist fault (Cassandra v2.2.5, figure 3.6(a)), SFIND pointed us to the method *Gossiper#applyStateLocally*, which is invoked for every gossip message processed. Given that the complexity of this method is $\sim O(n^3 p)$ (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 in the form of flappings (a node is declared "dead" incorrectly). Due to this and other issues related to this protocol, developers started a new initiative for designing "Gossip 2.0" to scale to 1000+ nodes [13], which until these days has unknown results.

- In the second fault (HDFS v2.7.3) SFIND pointed us to a code path involving the methods (1) *DatanodeManager#refreshNodes* (holding a global lock) and (2) *DatanodeMan-*

ager#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 fault (HDFS v2.7.3, figure 3.6(b)) our static analysis 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.

- Finally, in the fourth fault (HDFS v2.9.0, figure 3.6(c)) our static analysis 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 v.2.1.3 and Voldemort v1.10.21, we found that the bootstrap/rebalance protocols in our target versions do not exhibit any scalability faults, up to 512 nodes.

3.5.3 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 ca-6127 [15] (see section 1.1).

a) $\#flaps = f(\ \Phi > 8\ )$

b) $\Phi = f(\ T_{avgGossip}, T_{lastGossip}\ )$

$T_{avgGossip}$ = avg. of last 1000 $T_{lastGossip}$

c) $T_{lastGossip} = f(\ \#hops, T_{gossipExec}\ )$

$\#hops = log(N)$ on average

$T_{gossipExec} = T_{stateUpdate}$ (if new state changes)

d) $T_{stateUpdate} = f(\ Size_{ringTable}, Size_{newStates}\ )$

$Size_{ringTable} \leq N \times P$ and $Size_{newStates} \leq N$

Figure 3.7: **Cassandra internal metrics.** *Above are the metrics we measured within the Cassandra bootstrap protocol for measuring* SCALECHECK *accuracy (figure 3.8). "f" represents "a function of" (i.e., an arbitrary function).*

Figure 3.7(a)-(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 3.8(a)-(d) compare in detail the accuracy of STEST without PIL ("`SCk`") and STEST$_{PIL}$ with PIL ("`SCk+PIL`"), respective to the real-deployment testing ("`Real`").

Figure 3.8(a) shows the total number of flaps (alive-to-dead transitions) observed in the whole cluster during bootstrapping. STEST by itself will not be accurate if all nodes are CPU intensive (section 3.2.2). However, with PIL, SCALECHECK closely mimics real deployment scenarios. Next, Figure 3.7(a) defines that *#flaps* depends on Φ [129]. Every node A maintains a Φ for a peer B (a total of $N \times (N-1)$ variables to monitor).

Figure 3.8(b) shows the maximum Φ 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 Φ values observed for each of the 512 nodes. As shown, the larger the cluster, more Φ values exceeds the threshold value of 8, hence the flapping. Figure 3.7(b) points that Φ 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 $T_{avgGossip}$.

Figure 3.8(c) 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,

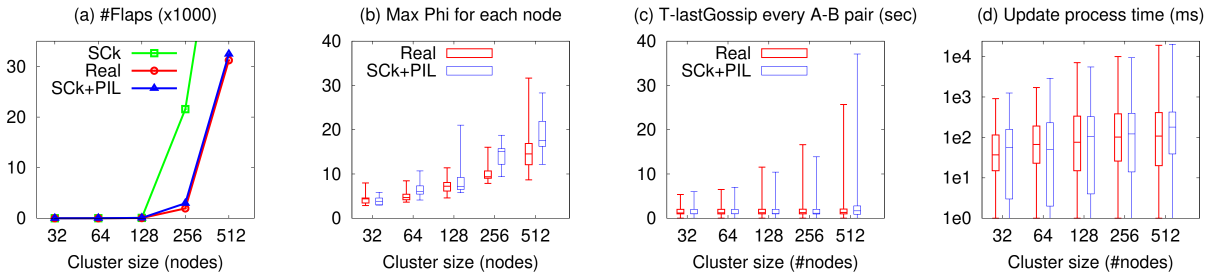


Figure 3.8: **Accuracy in exposing ca-6127.** *The figures represent the metrics presented in figure 3.7, measured in real deployment ("Real") and in* ScaleCheck *("SCk") with different cluster sizes (32, 64, 128, 256, and 512 in the* ***x-axis****). The* ***y-axes*** *(the metrics) are described in the figure titles.*

especially at high percentiles. Figure 3.7(c) 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.

Figure 3.8(d) (*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 3.7(d), $T_{stateUpdate}$ 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.

3.5.4 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.

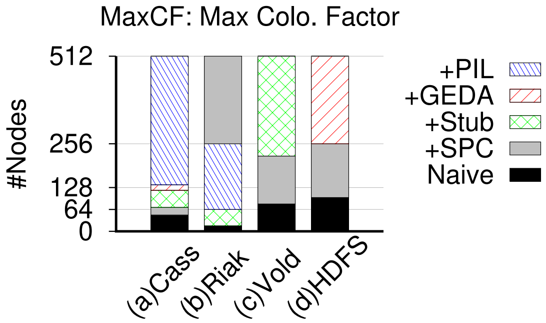


Figure 3.9: **Maximum colocation factor.** *The colocation factor reached as each technique is added.*

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 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 section 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 3.9 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 3.9(a), without using PIL in Cassandra, MaxCF only reaches 136. But with PIL, MaxCF significantly jumps to 512. 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 3.9(a)), our integration sequence is +SPC, +Stub, +GEDA, and +PIL (as we hit context switching overhead before CPU). For Riak (figure 3.9(b)), we began with PIL as we hit CPU limitation first before hitting Erlang VMM network overflow which requires SPC (section 3.2.1), and Riak does not require GEDA because Erlang, as an event-driven language, manages thread executions as events. For Voldemort (figure 3.9(c)), 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 3.9(d)), 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 3.9(a) 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.

3.5.5 Pre-memoization and replay time

The "T_m" and "T_{pil}" columns in table 3.3 quantifies the duration of the pre-memoization (STEST$_{mez}$) and PIL-based replay (STEST$_{PIL}$) stages when $N \geq 256$. For example, for CPU-intensive faults such as ca-6127, the pre-memoization time takes 2 hours while the PIL-based replay is only 15 minutes (similar to the real-deployment test); for rk-3926, 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.

3.5.6 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 section 3.1).

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

3.6 Limitations

At the moment, our work focuses on scale-dependent CPU/processing time (section 3.1), and the "scale" here implies the scale of *cluster size* ((section 2.1). The specific list of faults we studied are shown in table A.1.

However, there are other scaling problems that lead to IO and memory contentions [125, 157, 163], usually caused by the scale of *load* [112, 126] or *data size* [151]. For emulating data size, we are only aware of one work, Exalt [171], which is orthogonal to SCALECHECK. In our fault study, we learn that some load or data-size related fault can be addressed with accurate modeling [126] (*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 [163]).

3.7 Conclusions

Technical leaders of a large cloud provider emphasized that "the most critical problems today is how to improve testing coverage so that faults can be uncovered during testing and not in production" [122]. It is now evident that scalability faults are new-generation faults 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 4

SVIEW: IDENTIFYING AND ANALYZING POTENTIAL SCALABILITY FAULTS IN LARGE-SCALE DISTRIBUTED SYSTEMS

As discussed in section 1.3.3, this chapter presents SVIEW [166], a framework for identifying and analyzing potential scalability faults in large-scale distributed systems. The goal of this chapter is to address the following research questions:

- **RQ6: Which pieces of code are affected by dimensionality?** Mature distributed systems are usually comprised of hundreds of thousands of lines of code, but according to our observations, only some of those code fragments are **dimensional**, *i.e.*, as one or more system dimensions grow, the number of executions of these fragments also grows.

- **RQ7: How to detect dimensional code fragments?** *What are the necessary steps and tools we need to find such fragments?*. Moreover, *which techniques are useful in determining their correlation with certain dimensions and categorizing their growth trends?*.

- **RQ8: When do dimensional code fragments become problematic?** Dimensional code fragments are the building blocks of distributed systems, thus not all of them are inherently problematic. Then, *when do they become problematic?*.

The rest of this chapter is structured as follows: we first introduce the main motivation for this work, **dimensional code fragments (DCFs)** (section 4.1), and continue with a design overview of SVIEW (section 4.2). Then, in the following sections, we detail our contributions: study of DCFs (section 4.3), the design of LEAP (section 4.4) and the analysis modules built on top (section 4.5), and an in-depth evaluation of LEAP that covers 4 real distributed systems with a total of 15 versions (section 4.6). We then close with related work (section 4.7) and conclusions (section 4.8).

4.1 Dimensional Code Fragments

In the study presented at chapter 2, we found various dominant root causes such as unbounded resource usage, bloated data structure design, etc. While there are many root causes to dissect, this chapter specifically puts our attention to the largest dominant root cause that covers almost half of the reports, **dimensional code fragments (DCFs)**, pieces of code in which the number of executions (*e.g.*, loop iterations, method calls, etc) is positively correlated to the increase in the size of a dimension in the system, such as cluster size, amount of data processed, etc. Imagine this simple code segment below whose computation grows as the number of nodes and files in the cluster grows.

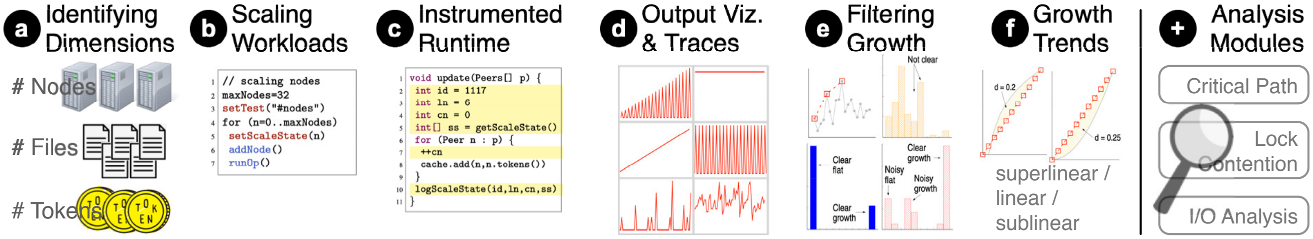
```
for (i=0...numNodes)
  for (j=0...numFiles)
    doSomething
```

If the cluster grows in number of nodes and number of files, one can imagine that this simple code segment can cause ripple effects to other parts of the system. Let's suppose that this segment is holding a lock used by a user-facing/foreground call, the long lock contention will cause performance issues that directly affect users. The `doSomething` segment could also involve I/O operations, hence generating excessive small I/Os, which is a known scalability anti-pattern. While we portray a simple example above, this paper shows various cases of dimensional code fragments including the challenges in finding which ones can potentially lead some harm to the system.

4.2 Overview

We present SView, a framework for identifying and analyzing potential scalability faults in large-scale distributed systems. SView outputs a list of dimensional code fragments and their relationships to the the system dimensions. For example, an output like

```
{AbstractReplicationStrategy.java, line:240, SuperLinear(tokens)}
```



```
// scaling nodes
maxNodes=32
setTest("#nodes")
for (n=0..maxNodes)
 setScaleState(n)
 addNode()
 runOp()
```

```
void update(Peers[] p) {
 int id = 1117
 int ln = 6
 int cn = 0
 int[] ss = getScaleState()
 for (Peer n : p) {
  ++cn
  cache.add(n,n.tokens())
 }
 logScaleState(id,ln,cn,ss)
}
```

Figure 4.1: **SVIEW design.** *The figure shows the 6 stages in* SVIEW *and the 3 analysis modules built on top.*

indicates that a code fragment located at that class file and starting at that line has a superlinear relationship with the number of tokens. If the number of tokens is large, this code can lead to a bottleneck at scale.

SVIEW comprises of 6 stages as illustrated in Figure 4.1. **(a)** In the first stage, SVIEW users (*e.g.*, developers) identify per-system dimensions, *i.e.*, the target system components that are scalable such as the number of nodes, partitions, files, tokens, etc., which typically can be found in design documents. **(b)** Next, the user writes **scaling workloads**, a set of tests that scale the identified dimensions. The workloads use the system APIs (*e.g.*, `addNodes()`) as well as SVIEW-specific APIs to flow important information to the target system's runtime to assist the subsequent stages in automatically identifying the DCFs and their relationship with the scaled dimensions. **(c)** Within the system runtime (*e.g.*, Java runtime), we instrument three types of loops as "potential DCFs": application, library, and "implicit" loops (*e.g.*, a namenode function called by *N* datanodes in the cluster). Then we monitor the complexity of potential DCFs automatically, that is how the number of iterations is increased as each of the dimensions is being scaled.

After the user runs the workloads on the instrumented system, **(d)** SVIEW outputs a trace file containing detailed information of potential DCF iterations and their relationships with the scale dimensions. SVIEW can visualize each of the potential DCFs as illustrated in Figure 4.1(d) where each small figure shows a growth or flat pattern with respect to the scaled dimension(s). **(e)** The previous stage generates potentially over 1000 graphs (one for each potential DCF), thus the next stage is to *identify true DCFs*, *i.e.*, determining those whose number of iterations truly increases with the scaled dimension being and discarding the flat ones. However, this phase is not as straightforward as expected due to "noises" stemming from real-world system behaviors such as complex protocol-specific `if-else` conditions and system optimizations such as batching and thread scheduling. To handle this, we introduce a 3-step empirical-driven filtering process that distinguishes four unique patterns (clear growth, clear flat, noisy growth, and noisy flat). **(f)** Finally, by filtering in the "true" DCFs, SVIEW categorizes the DCF *growth trend* with a similarity-measures technique to superlinear, linear, or sublinear trends. This last step can help developers prioritize their analysis.

SVIEW by itself is *not* a bug finding tool. Its goal primarily is to provide developers the "view" of their systems with respect to DCFs. In other words, not all DCFs are harmful, hence we try to find DCFs that could potentially bring harmful effects to the system. For this, developers can build *analysis modules* on top of SVIEW' results. In this work, we built three modules: *critical path analysis* to find DCFs within user-facing/foreground paths; *lock contention analysis* to find DCFs in background/operational protocols that contend with a foreground lock; and *I/O analysis* to find disk/network I/Os inside a DCF that could lead to the "chatty I/O" problem, a performance antipattern.

4.3 Characteristics of Scalability Faults and Dimensional Code Fragments

So far we have covered 66 bug reports, shown in table C.1. Below we characterize them, including the dimensions, the scenarios (when they were discovered), the related code paths, to which we refer as protocols, and the effort in terms of time, discussion length and patches, that was required to solve them.

High-level dimensions: Figure 4.2(a) categorizes the high-level dimensions over 3 axes of scale: *cluster size* (# nodes, # partitions, etc.), *data* (# files, # tables, # rows, etc.) and *load* (# clients, # requests, etc.). The majority is concentrated among the first two axes (around 42% for each) while a relatively small percentage (16%) is related to load. Later Section 4.4.1 will break

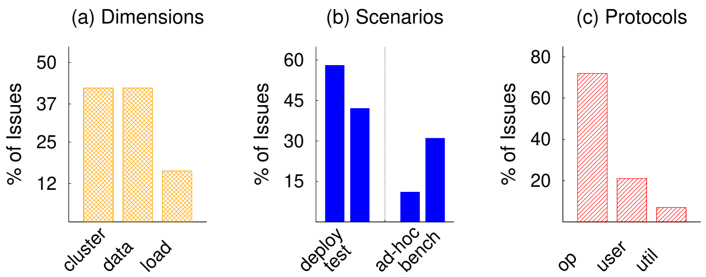


Figure 4.2: **Fault characterizations.** *The figures breaks down the characterizations of scalability issues in terms of the high-level dimensions, the scenarios in which they were found, and the protocols involved.*

down further the per-system dimensions that are necessary to create scalability test suites.

Scenarios: Figure 4.2(b) shows that 58% of the issues were found *in deployment* ("deploy" bar) while 42% *during testing* ("test" bar). Within the testing scenarios, we further subdivide them into ad-hoc testing ("ad-hoc" bar at 11% of total), where the developers create custom workloads to reproduce the issues, and benchmarking ("bench" at 31%), where the developers use popular benchmarking suites or stress-testing tools. This highlights that dimensional code fragments could be missed by existing tools (mainly because the root dimensions are not scale tested) and the developers have to resort to ad-hoc testing or waiting until the problems surface in deployment.

Protocol types: To understand why scalability issues are often hidden, we study the protocol types, which we simply break to *user-facing/foreground* paths (*e.g.*, read/write), *operational/background* protocols (*e.g.*, membership management, cleaning, compaction, backup, snapshot operations), and general *utilities* used by both. Figure 4.2(c) shows that 72% of scalability faults linger in operational protocols and only 21% and 7% are in user-facing protocols and general utilities, respectively. While user-facing protocols are technically implicitly tested "all the time" in deployment, operational protocols often do not receive the same treatment. Not to mention, some of the issues arise from the interaction of operational and user-facing protocols (*e.g.*, lock contention), which increases the difficulties in terms of the amount of effort required for testing.

Time to solve, discussions, and patches: The complexity of scalability issues can also be seen from the number of months needed to close the bug report and/or the number of comments made

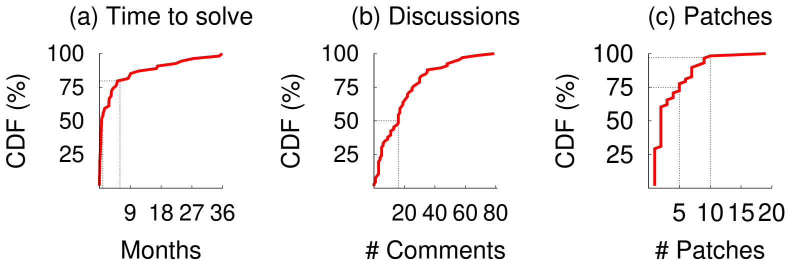


Figure 4.3: **Efforts to solve.** *The CDF figures summarize the developer's effort to address the problem in terms of time to mark the issue as solved, number of discussion comments, and number of patches submitted.*

by the developers in the bug report. Figure 4.3(a) shows the CDF of the solution time. 49% of them are closed after more than a month and in the long tail, 20% of them need more than 6 months to close. 40 of them are labeled `Major`, 4 `Urgent`, 10 `Critical` and 12 `Normal`. Figure 4.3(b) portrays the same complexity by showing the number of comments. Over 50% of them require over 16 back-and-forth comments. Finally, Figure 4.3(c) summarizes the number of patches submitted. 75% only require fewer than 5 patches, which might include patches for different versions of the system when necessary, and 3% include 10-20 patches, which arguably required a lot of time to spend.

Overall, our bug study shows that dimensional code fragments are not trivial to find and test. We found that the developer discussion was centered around the difficulties of correctly identifying the root cause of the problem and making sure the proposed solution was sound and the modified system passed all related test cases, new and existing ones. In some cases, this led to dozens of comments spawning a conversation over years. This study led us to the design of SVIEW.

4.4 SVIEW Design

This section details our design of the SVIEW pipeline: identifying per-system dimensions (section 4.4.1), writing scaling workloads (section 4.4.2), instrumenting system runtime (section 4.4.3), outputting the traces (section 4.4.4), identifying and filtering growth (section 4.4.5), and categoriz-

H.L.	Per-system							
Dims.	Dimensions	ca	ha	hb	hd	ig	kf	sp
Cluster	peers	✓				✓	✓	
	managers		✓	✓	✓		✓	✓
	workers		✓	✓	✓		✓	✓
	consumers						✓	
Load	clients	✓	✓	✓	✓	✓	✓	✓
	jobs/tasks		✓					✓
	requests	✓		✓	✓	✓	✓	
Data	rows	✓						✓
	columns	✓						✓
	tables/keyspaces	✓						✓
	logs	✓	✓	✓	✓	✓	✓	
	snapshots	✓			✓		✓	
	files/dirs	✓	✓	✓	✓	✓	✓	✓
	blocks				✓			
	partitions	✓	✓	✓		✓	✓	
	sstables	✓						

Table 4.1: **System Dimensions.** *This table lists the high-level and per-system dimensions in Cassandra (`ca`), Hadoop (`ha`), HBase (`hb`), HDFS (`hd`), Ignite (`ig`), Kafka (`kf`), and Spark (`sp`).*

ing growth trends (section 4.4.6).

4.4.1 Identifying Per-System Dimensions

The first stage of the pipeline is identifying the per-system dimensions, *i.e.*, the components of the system that can scale (increase in numbers). This process is a manual task that requires understanding the system design and in some cases their internal implementations. Per-system dimensions are not the high-level dimensions such as cluster size, data size, and load, but rather they are one level more detail. For example, cluster size can imply the number or nodes, logical partitions, tokens, and many others. Data size can be the number of files, blocks per file, directories, tables, segments, etc.Load size can represent the number of clients, connections, or more specifics such as consumers or producers.

Table 4.1 lists the per-system dimensions that we have identified for the systems that we studied. We collected the dimensions from the discussion comments of the fault reports as well as from their publicly available design documents. The names we use in Table 4.1 are abstract names for readability. In reality, every system has their own specific naming (*e.g.*, in Cassandra, a "table" is a `ColumnFamily` and a "partition" is a `Token`). A system can have more dimensions than others. The list in Table 4.1 is not necessarily complete, but we expect developers (can) have the full list.

It is important to identify each of the per-system dimensions because unlike prior works, our goal is to provide a clear "view" of which code fragments are dependent on which dimensions. We do not want to merely output that a code fragment can grow linearly or quadratically in compute cost but without the root cause. For example, a sample of output like

{`AbstractReplicationStrategy.java`, `line:240`, `SuperLinear(tokens)`},

identifies that a code fragment starting at this line has a super linear relationship where if the number of tokens is large this code can be a potential bottleneck. As the number of per-system dimensions is likely constant and small (*e.g.*, fewer than 20) and described in the design documents, we expect that developers can bear the manual process.

4.4.2 Writing Scaling Workloads

The next step is to write **scaling workloads**, a set of tests to exercise the growth of the identified dimensions and expose the behavior of the dimensional code segments in relation to the growth. These workloads can be programmed in common languages like Python or bash and are entirely external to the system under test.

Figure 4.4(a) shows a pseudo-code of a scaling workload that tests the cluster size dimension. In lines 2 and 4, the developer set the maximum number of nodes and the `for` loop that keeps adding one node at a time. Line 6, `addNode()` is the existing **system API** provided by the target system to scale this particular dimension. Line 7, `runOp()` is a sample function where the developer can choose to decide to run some additional foreground/background operations [17, 59, 65, 79].

The developer also must call the `LEAP APIs`, specifically `setTest()` on line 3 that declares the dimension name that is being scaled (*e.g.*, "`#nodes`") and `setScaleState` on line 5 that sets the *current* **scale state** of the workload, which is varied within the loop (*e.g.*, `n` = 10 nodes currently).

```
1  // scaling nodes
2  maxNodes=32
3  setTest("#nodes")
4  for (n=0..maxNodes)
5   setScaleState(n)
6   addNode()
7   runOp()
```

```
1  // scaling both
2  maxNodes=32
3  step=10000
4  curRow=step
5  setTest("#nodes", "#rows")
6  for (n=0..maxNodes)
7   setScaleState(n, curRow)
8   addNode()
9   addRows(step)
10  curRow += step
```

(a) One Dimension (b) Two Dimensions

Figure 4.4: **Sample scaling workloads.** *The figures show sample pseudo-code of one and two dimensional scaling workloads (nodes and nodes+table rows, respectively).*

These APIs form an important information flow to the target system's runtime that we automatically instrument in the next section. The issue we address via these APIs is that our instrumented runtime is calculating the complexity of the dimensional code fragments but it has no knowledge of the current scale status of the system. Providing such a simple information proves to be useful for the subsequent steps.

Figure 4.4(b) shows another simplified example where we scale two dimensions at the same time, number of nodes and rows of a key-value table. A couple of differences include: the use of system APIs that allow batch update such as adding 10,000 rows at a time (`addRows(step)` on line 9), declaration of the two dimensions (`setTest()` on line 5), and update of the scale status of the two dimensions (`setScaleState(n,curRow)` on line 7).

4.4.3 Instrumenting the Runtime

Next we monitor the complexity of dimensional code segments automatically, that is how the number of iterations is increased as each of the dimensions is being scaled. As our systems under test are all written in Java, we instrument both the target system code ("application" code) and Java libraries by combining `Javassist` [85], a bytecode manipulation framework, with instrumentation agents [84], which directs the instrumentation process and provides an entry point for parame-

```
 1 void update(Peers[] p) {
 2  int id = 1117
 3  int ln = 6
 4  int cn = 0
 5  int[] ss = getScaleState()
 6  for (Peer n : p) {
 7   ++cn
 8   cache.add(n,n.tokens())
 9  }
10  logScaleState(id,ln,cn,ss)
11 }
```

(a) Application Code

```
void add(K k, V[] v) {
 int id = callerId()
 int ln = callerLine()
 int cn = 0
 int[] ss = getScaleState()
 for (V val : v) {
  ++cn
  map.put(k,val)
 }
 logScaleState(id,ln,cn,ss)
}
```

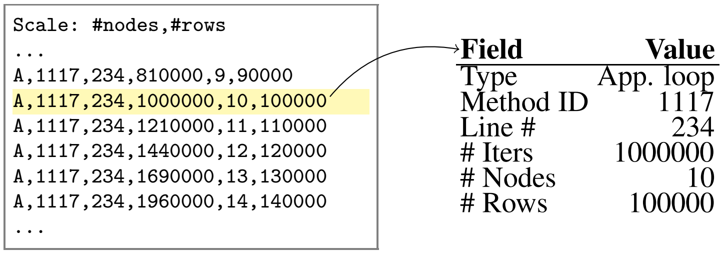
(b) Library Code

Figure 4.5: **Instrumentation example.** *The highlighted lines correspond to code added by our instrumentation around (a) application and (b) library loops.*

ter passing. The former provides the necessary APIs to detect specific types of code structures, such as loops in bytecode, while the latter allows us to pass the necessary filters to distinguish between application and library code (*e.g.*which package prefixes should be considered part of the application). We instrument three important types of loops: application, library and implicit loops.

Application loops: A simple example of an application loop instrumentation is shown 4.5(a), where the code automatically added by our instrumentation is highlighted. As our tool detects an application loop on line 6 (`for (Peer n:p)`) that belongs to the system under test, it adds a unique ID, line number, and a count variable (lines 2 to 4). More important, it also retrieves the scale state (`getScaleState()` on line 5), which provides the information about the dimension that is being grown. This way we can correlate the dimension being exercised and the increasing number of iterations via the counter added by our tool (`cn++` on line 7).

Library loops: Beyond application loops, we found that root causes of scalability bugs can remain unseen due to "hidden" dimensional loops inside library calls. The application loop can look $O(N)$ but if the body calls another $O(N)$ library then the complexity becomes quadratic. Here, libraries imply external code where the original source code might not be available. However, since we instrument at bytecode we can identify such loops and differentiate them from the application loops by using the Java agents, specifically by filtering external classes to the system (*e.g.*, not



```
Scale: #nodes,#rows
...
A,1117,234,810000,9,90000
A,1117,234,1000000,10,100000
A,1117,234,1210000,11,110000
A,1117,234,1440000,12,120000
A,1117,234,1690000,13,130000
A,1117,234,1960000,14,140000
...
```

Field	Value
Type	App. loop
Method ID	1117
Line #	234
# Iters	1000000
# Nodes	10
# Rows	100000

(a) Two-dimensional trace (b) Details

Figure 4.6: **Output trace.** *The figure shows a sample output trace from running a two-dimensional (#nodes and #rows) scaling workload.*

`org.apache.cassandra.*` classes). We then associate the library loops with the caller in the target system code. Figure 4.5(b) shows an instrumentation result where the `id` is not a unique ID but rather the caller ID (line 2). The line number (`ln`) is also not accessible inside a library bytecode, hence the line is the caller's line. The rest are the same as described for application loops.

Implicit loops: We also found scalability issues where the dimensionality exists but there is *no* explicit loop, but rather an "implicit loop" forms. Imagine a function `f()` in a single master node being called by every data node in the cluster (tens to potentially thousands) in a very frequent manner with heavy tasks. Here `f()` acts like a body of a loop that is being iterated many times. Thus, our instrumentation also handles this kind of scenario by recording the number of times each method is invoked.

We emphasize again the advantages of interposing all the loop types above in the runtime compared to doing loop analysis statically. With static approaches [117, 124, 152, 172, 174], we must first know which data structures are dimension dependent and then perform a dataflow analysis [165]. However, we find that there are many ways that the original dimension-dependent data structures (*e.g.*, `List nodes;`) can be copied to other "temporary" data structures that are copied as a body of a transient message (*e.g.*, `message.nodeList = copy(nodes)`). If static analysis fails to track them properly, then the loop analysis will also be impaired, *i.e.*, loops on temporary dimensional data structures will not be marked as growing. Furthermore, static analysis also cannot cover implicit loops unless the developer annotates which functions can be called by many components

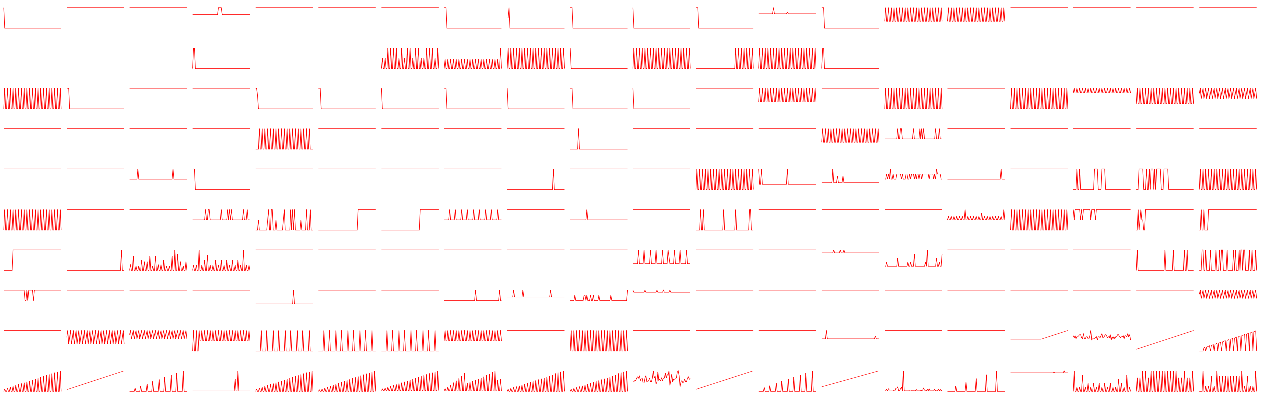


Figure 4.7: **Sample output visualization.** *Here* SVIEW *visualizes the growth of each loop in the output trace. Each small figure represents a single loop. The number of iterations (in the y-axis of each small figure) is projected over the scaled dimension (in the x-axis of each small figure). For simplicity, here we only visualize the output of a single-dimensional scaling workload.*

of the system in parallel.

4.4.4 Outputting Traces

With the instrumentation, every time a scaling workload is run, SVIEW will output a trace file that records the information of its loop iterations. Figure 4.6 shows an example of a segment of an output trace file for a two-dimensional scaling workload (*#nodes* and *#rows*). Every line is a tuple in the form of $\langle A/L/F, id, ln, cn, [ss] \rangle$, where $A/L/F$ represent an application, library, or function loop (i.e., implicit loop), *id* the unique method ID, *ln* the line number of the loop, *cn* the iteration count, and $[ss]$ the scale state in an array with one or two numbers for one- or two-dimensional tests, respectively. As a concrete example, $\langle A, 1117, 234, 1000000, 10, 100000 \rangle$ implies that an application loop in method ID 1117 at line 234 has iterated 1000000 times when the number of nodes and rows have reached 10 and 10000, respectively. LEAP' subsequent steps will then clean and analyze the traces.

After a scaling workload completes, the resulting trace will show all the loops that are exercised and their *growth* with respect to the dimension being scaled. Figure 4.7 shows a sample of trace output visualization. For simplicity, here we only show an output of a 1-dimensional test where we

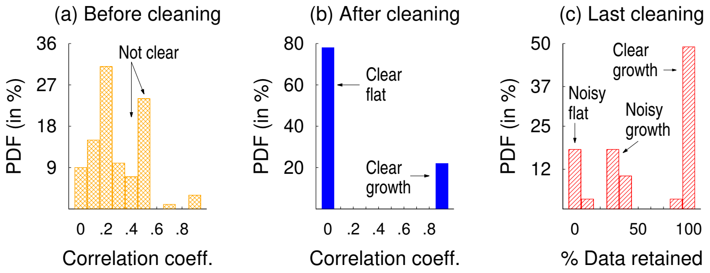


Figure 4.8: **Growth analysis .** *Figure (a) shows the probability distribution function (PDF) of Spearman correlation coefficient of the raw datasets before cleaning, bucketed in 0.1 step (in the x-axis); Figure (b) plots the filtered data after cleaning, separating loops with clear non-growth (flat) and growth; and Figure (c) shows the distribution of the percentage of retained datapoints where the result clearly distinguishes the three growth patterns.*

scale the number of keyspaces (tables) in Cassandra. The trace shows 880 application, 104 library, and 566 implicit loops being exercised, and only some of them are shown in the figure. Each small graph in Figure 4.7 represents a loop. The x- and the y-axis in each small figure are the dimension (in this case the number of keyspaces being added) and the number of iterations, respectively.

4.4.5 Identifying and Filtering Growth

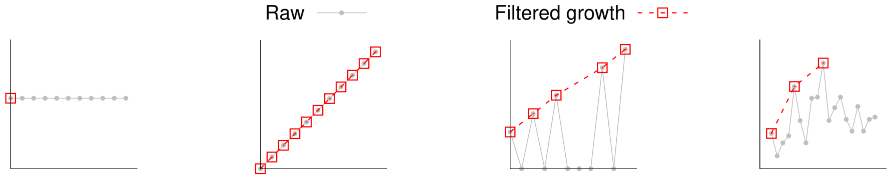
The next step is to *identify growth*, *i.e.*, determining the code fragments whose number of iterations increases with the dimension being scaled (*e.g.*, a loop at the bottom left corner graph in Figure **??**) and ignoring the flat ones (*e.g.*, a loop at the upper right corner of the same figure). In other words, flat loops are not harmful and should be discarded but growing loops *can* be harmful and should be filtered in. By manually "eyeballing" the graphs, we can easily identify the growing ones but this is not a viable approach given the thousands of graphs plotted by SVIEW.

One straightforward way of automating this filtering process is to plug in the numbers to statistical algorithms such as Pearson, Kendall and Spearman [113] correlation coefficients, and let them decide which datasets exhibit high correlations. However we found naively doing so generates a lot of noise. Using the same example above, we use Spearman on 1550 sets of data (representing all the loops) and receive 1550 correlation coefficients. In our case, a coefficient near 0 implies no correlation (flat) and a coefficient near 1 identifies growth. 4.8(a) shows the probabilistic distribution function (PDF) of the 1550 coefficients bucketed in every 0.1 coefficient range. We were expecting a bimodal distribution with many cases separated to the two opposite sides, 0 and 1 (in the x-axis), with the hope that we can easily filter in the highly correlated ones (near $x = 1$). However, we see around 31% (in the y-axis) of the datasets do not have a clear outcome ($0.4 \leq x \leq 0.6$).

Upon further analysis, we found two expected flat/growth patterns and two other interesting patterns that make the noise, as illustrated in Figures 4.9(a)-(d) (please focus on the gray dots and lines at this point and ignore the red dots and lines). The first one (4.9(a)) is a clear flat pattern where the loop iterations do not grow (in the y-axis) even when the dimension being exercised continues to grow (in the x-axis). The second one (4.9(b)) is a clear growth pattern, in this example a loop with a linear, $O(N)$ complexity. The third one (4.9(c)) is an interesting *noisy growth* pattern. Here we observe a trend of growth but often times there are data points in the output traces that show 0 iterations ($y = 0$ in the middle area of Figure 4.9(c)). We will explain the fourth figure (4.9(d)) later.

After further debugging, we found two main root causes for the noise. First, complex distributed systems have `if-else` conditions specific to the logic of their protocols. For example, a key-range rebalancing process might discard a message containing a list of nodes if that message is outdated (*e.g.*, has an older ballot), hence the associated `for` loop will not be executed. Second, system optimization techniques such as batching and thread scheduling also will make some actions being skipped, making the loops potentially do more work in future iterations. To sum up, the sources of noise are “real” stemming from real-world system behaviors, which led us to data cleaning.

To perform cleaning of the noise, we perform a 3-step empirical-driven filtering process. **(i)** We first only retain a data point (the number of iterations) that is larger than the maximum of all the previous data points. For some *illustrations*, in Figure 4.9(a), there is only one point (in red box) retained while the others are removed as they are smaller than the first point. In Figure 4.9(b),



(a) Clear flat (b) Clear growth (c) Noisy growth (d) Noisy flat

Figure 4.9: **Growth patterns.** *The figures summarize the four major patterns that we observed from all the output visualization akin to Figure 4.8.*

all data points are kept as they are monotonically increasing. In Figure 4.9(c), only 5 data points are considered, and this simple process converts the noisy growth into a cleaner growth pattern (the red dashed line). **(ii)** Next, for the retained points, we use Spearman to give us the correlation coefficient and the result is much clearer, as shown in Figure 4.8(b); it is either a flat (the correlation coefficient is $0 \leq x \leq 0.1$) or growth pattern ($0.9 \leq x \leq 1$). **(iii)** Finally, among the datasets with clear growth pattern, we need to handle one more case, the *noisy flat* pattern in Figure 4.9(d) where the overall pattern is flat but the small number of filtered datapoints shows growth. To exclude cases like this, we analyze the percentage of retained datapoints after the cleaning in first step. Figure 4.8(c) shows the distribution and three groups appear: the normal growth group (Figure 4.9(b)) where over 80% of the datapoints are retained ($x > 80\%$); the noisy growth group (Figure 4.9(c)) where about 30-60% of the datapoints are kept ($30\% < x < 60\%$); and finally the noisy flat group (Figure 4.9(d)) where only less than 10% of data is kept. Thus, in this last cleaning we remove the last group because they do not have enough datapoints to confirm the growth pattern.

4.4.6 Categorizing Growth Trends

The final step is to categorize the complexity of each dimensional code fragment that has been filtered from the previous step. Based on our fault study and experience running LEAP, they can be categorized into three trends: **superlinear** (47% of the faults in our study), **linear** (53%) or **sublinear** (0% but we found a few cases in LEAP' filtered output). This simple categorization can be useful for developers in order to estimate the urgency or priority of the issue. For example,

Theoretical model	# Dimensions	Type
$I = D_1^2$	1	superlinear
$I = D_1 * D_2$	2	superlinear
$I = D_1$	1	linear
$I = D_1 + D_2$	2	linear
$I = log(D_1)$	1	sublinear
$I = \sqrt[2]{D_1}$	1	sublinear

Table 4.2: **Theoretical models to growth trends.** *The table shows different types of theoretical models that represent the number of iterations ("I") as a function of the size of one or more dimensions ("D"), and how they can be mapped to the three growth trends (superlinear, linear and sublinear).*

among the fault reports involving superlinear fragments, 35% are marked as "critical/urgent" while it's only 8% for linear fragments.

To categorize the computational complexity to one of the three trends, we empirically selected a few computational theoretical models (Table 4.2) based on our study. Then we determine which model best fits with our data points by exploring *similarity measures*, more specifically *Fréchet Distance* ("*d*") [111]. Such measures are employed to determine how close two curves are within the same metric space. To apply this method, we first take the filtered datapoints from the previous phase (the red dots in Figure 4.9); normalize the y-values to [0, 1] range; generate datapoints for each of the theoretical models in the same range; compute the distance value d between our datapoints and each model; and finally pick the model (trend) that has the smallest distance value. Figure 4.10 illustrates an example of when we analyze a *linear* fragment. In the first and last graphs (4.10(a) and 4.10(c)) the linear filtered datapoints are far from the superlinear and sublinear models, respectively, generating a positive d value. In the middle one (4.10(b)), the datapoints match the linear model, hence this final phase labels the fragment with "linear" trend.

Other than Fréchet Distance, we also experimented with other algorithms such as r^2 [140] and *mean absolute error* [159], however we find them not reliable for our datasets. We observed an imprecision of up to 20%; "imprecise" means that for around 20% of 10k randomly generated datasets with linear, superlinear and sublinear filtered datapoints, those algorithms reported more than one category (*e.g.*the same d value was reported when comparing against linear and sublinear

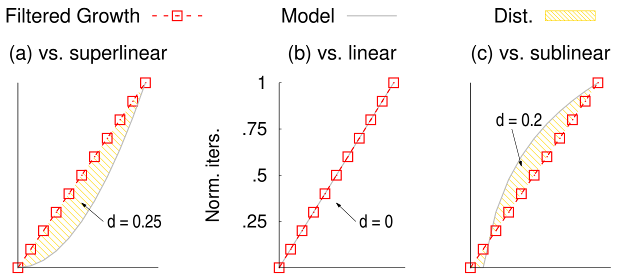


Figure 4.10: **Similarity measures.** *The three graphs illustrate how Fréchet Distance categorizes computational complexity to three growth patterns.*

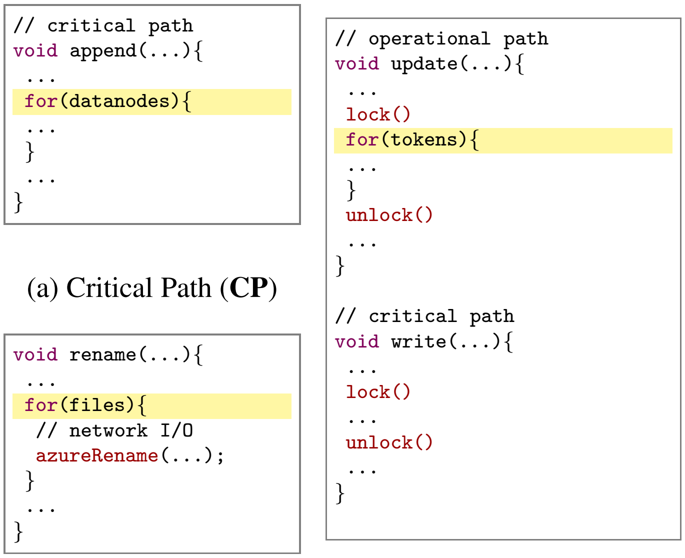
models) as correct. The reason is that they are not that robust in analyzing impartial datasets (*e.g.*, noisy growth patterns with gaps in between filtered datapoints). For Fréchet Distance, so far we observed a 100% precision.

4.5 Analysis Modules

In complex distributed systems, dimensional loops are one of the basic building blocks, hence not all of them are harmful, even the superlinear ones. Thus, the next question to address is: which dimensional loops could potentially bring harmful effects to the system? For this, we turned to our fault study again (§4.3) and found three simple but potentially harmful loop patterns that can pose a performance threat to the system, as summarized in Figure 4.11. By formalizing those cases we build three analysis modules on top of SVIEW. Using Spoon [158], we implemented the analysis using static call-graph analysis that analyzes the linear/superlinear loops that SVIEW has triaged, as described below. We also label them (CP, LC and IO) for evaluation purposes later.

4.5.1 Critical Path Analysis (CP)

We define *critical path* as a code path that is user facing, also often called a "foreground" operation such as file reading/writing, key-value querying, table creation/manipulation or job/task submission. User-facing/foreground operations are expected to be latency sensitive. Unfortunately, dimensional loops are sometimes present in critical paths (21% in our fault study), as illustrated in



```
// critical path
void append(...){
 ...
 for(datanodes){
 ...
 }
 ...
}
```

(a) Critical Path (**CP**)

```
// operational path
void update(...){
 ...
 lock()
 for(tokens){
 ...
 }
 unlock()
 ...
}

// critical path
void write(...){
 ...
 lock()
 ...
 unlock()
 ...
}
```

(b) Lock Contention (**LC**)

```
void rename(...){
 ...
 for(files){
  // network I/O
  azureRename(...);
 }
 ...
}
```

(c) I/O (**IO**)

Figure 4.11: **Potentially harmful loops.** *These code snapshots, based on hd-14366, ca-14660 and ha-13403 respectively, represent three harmful loops that (a) exist in a critical/foreground/user-facing path, (b) hold a lock contending with a foreground lock, and (c) contain I/O operations.*

Figure 4.11(a). Albeit, mechanisms like stress testing [23, 60, 80, 109] are often useful to detect such bottlenecks, the lack of dimensionality, *i.e.* not scaling the relevant dimensions in the test setups, reduces their effectiveness when looking for scalability issues.

For this, our critical-path (CP) analysis module takes an input of all foreground APIs as the starting points and then performs a call-graph analysis by traversing all functions and lines reachable from the APIs. If the analysis observes a linear/superlinear loop (that was already triaged by SVIEW), the CP module will throw a *warning*. It is important that we do not quickly declare them as "bugs" because the developers might be willing to take the performance hit in cases where there is no obvious alternative to fix the problem. Our CP module essentially performs a forward slicing because the number of user-facing/foreground APIs is bounded. A backward slicing method that starts the analysis from the dimensional loop locations is also possible.

4.5.2 Lock Contention Analysis (LC)

The second problematic pattern relates to lock contention at scale. As show in Figure 4.11(b), a thread executing a dimensional loop (`update`) is holding a lock (potentially for a long time) that is also used by another thread (`write`). The developer of the latter thread might assume a fast operation, but such an assumption breaks when a large dimensional loop holds the lock for a long time. This case covers 74% of our fault study and can happen in foreground/background interactions (*e.g.*, a message processing operation must wait for a long time until a background rebalancing operation completes).

Our lock contention (LC) analysis module basically performs a backward slicing from every dimensional loop triaged by SVIEW. If during the backward slice we find a lock acquire method, our analysis searches for other places that use the lock and throw a warning. We throw warnings not only in foreground/background lock contention but also in background/background interactions. Again, in complex systems different pieces of code might be written by different developers and some background operations also need some latency bound.

4.5.3 I/O Analysis (IO)

Finally, the last potentially problematic pattern relates to dimensional I/Os (*e.g.*, disk or network I/Os) as shown in Figure 4.11(c) (`azureRename`). Here, the body of loop contains either a disk read/write operation or a synchronous network message, thus the amount of I/O operations is tied to the size of one or more dimensions. This leads to the "chatty I/O" problem, a performance antipattern [88], and furthermore a large number of small I/Os is often known to cause a significant performance overhead.

Our I/O analysis module essentially analyzes the body of the dimensional loop. Obtaining the list of I/O APIs was done as a preprocessing step (*e.g.*, `File#write`, `Socket#read`). Such APIs can be native method calls or external library calls. Unit/stress testing or benchmarking often fails to surface the severity of dimensional I/O loops because they need to be tested at a sufficient scale.

Fault ID	Dim.	Trend [DCF]	CP	LC	IO
a) ca-15141 [40]	N,T	superlinear(N,P) [26]		✓	
b) ca-14660 [39]	N,T	superlinear(N,P) [25]	✓	✓	
c) ca-13923 [38]	N,T	superlinear(P) [27]	✓	✓	
d) ha-16850 [42]	R	linear(R) [29]	✓		
e) ha-13403 [41]	F	linear(F) [28]	✓		✓
f) hd-15415 [44]	B	linear(B) [30]		✓	
g) hd-14366 [43]	N	linear(N) [31]	✓		
h) ig-12087 [45]	R,E	superlinear(R,E) [32]	✓		
i) kf-9393 [47]	R,P	superlinear(P) [34]	✓	✓	✓
j) kf-8736 [46]	R,E	superlinear(R,E) [33]	✓		

Table 4.3: **Existing faults reproduced.** *The table lists the fault IDs, the corresponding dimensions ("Dim.") being scaled, the trend* Leap *identifies including links to the harmful dimensional code fragments, and the analysis modules (CP/LC/IO) that threw the warnings. For the dimensions, "N" stands for nodes, "T" for tokens, "R" for requests, "F" for files, "B" for blocks, "P" for partitions, and "E" for cache entries.*

However with SView, with just a small scale, the growth projection (*e.g.*, superlinear) can be identified, and combined with I/O analysis, the corresponding warning is also thrown.

4.6 Evaluation

We now evaluate SView and the three analysis modules, by specifically answer these questions: **i** Can they help find root causes of known scalability faults? **ii** Do they run well on most recent versions of our target systems and what are the output statistics? **iii** Can they identify potential scalability issues in most recent versions?

To answer these questions, we integrate SView with 4 popular scalable distributed systems (Cassandra, HDFS, Ignite and Kafka) and do so across a total of 11 old and 4 recent versions. All of the experiments are run using a single machine with 48 AMD EPYC cores and 256-GB DRAM on Chameleon [24]. Some of the experiments in reproducing old fault symptoms also have been packaged using Chameleon Trovi [83].

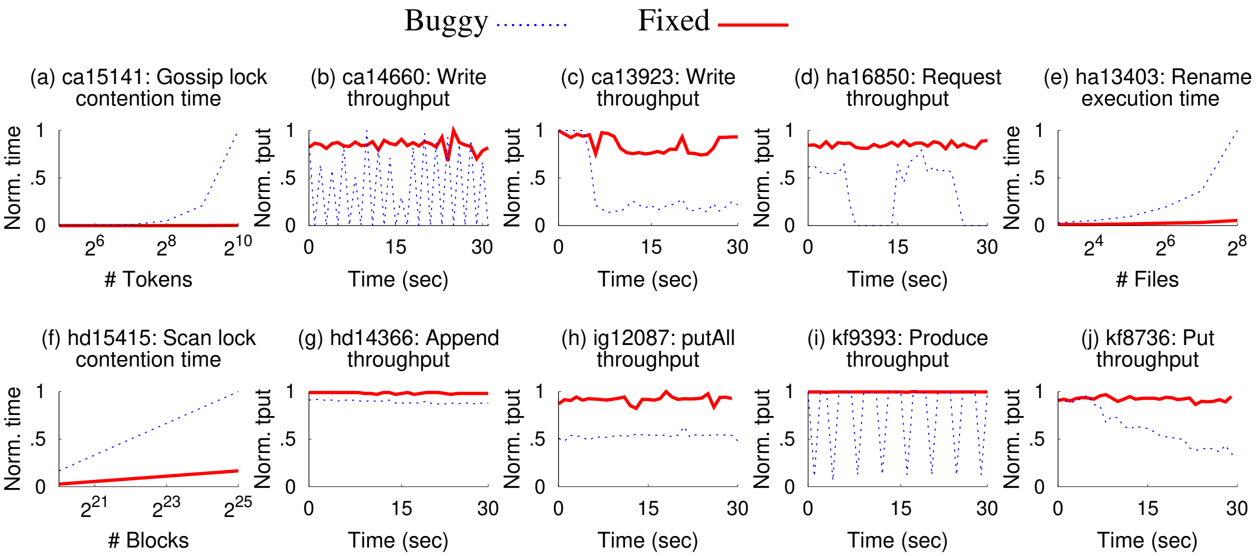


Figure 4.12: **Reproduced symptoms of existing faults.** *The figures show the performance implications (in the y-axis, normalized to* $[0-1]$ *range) of the scalability issues before the issue was fixed (blue dashed line) and after we apply the patch (red solid line).*

4.6.1 Reproducing Scalability Issues

To reproduce existing scalability issues from our fault study, we performed the following steps. **(i)** We picked 10 fault samples that have clear issue descriptions, as listed in Table **??**. **(ii)** The descriptions guide us in writing the scaling workloads that will likely cover the root cause. Note at this point, we don't have to "reproduce" the fault symptoms, because there are various performance metrics to analyze to measure the symptoms, as explained later. The second column (**"Dim."**) of the table lists the dimensions being exercised in our scaling workloads. **(iii)** We use SVIEW (section 4.4) to filter dimensional code fragments that have growing trends as the dimension(s) being scaled. The third column in the table (**"Trend [DCF]"**) cites the GitHub links to the harmful dimensional code fragments (DCFs) and shows the trends reported by SVIEW on these DCFs. **(iv)** We use the three analysis modules (section 4.5) to help us point the dimensional code fragments that are potential harmful. The last three columns in the table show the CP/LC/IO warnings that the three analysis modules throw on the fragments. **(v)** We analyze the fault reports again to understand specific performance metrics that we need to measure as we run the scaling workload.

These steps serve two purposes. First, it proves that we have successfully reproduced the faults. Second, when we re-use the same scaling workloads to find potential issues in newer versions (section 4.6.3), we can measure the same metrics again to know the ground truth whether there are still problems in the newer versions. Overall, Figures 4.12(a)-(j) show the performance implications before the issue was fixed (blue dotted line) and after we apply the patch (red bold). Note that the figure x-axis uses *scale* when the direct impact is on latency and *time* when the impact is on throughput.

(a) In ca-15141 (Figure 4.12(a)), when Cassandra observes nodes being added/removed, a single-thread Cassandra's gossip (operational) protocol with $O(tokens^2)$ complexity is triggered [26] while holding a write lock. The figure shows the lock contention time skyrockets as the number of tokens jumps, causing delays in other write operations. After the developers fixed the design, the contention went away.

(b) In ca-14660 (Figure 4.12(b)), the (critical) write path in Cassandra is contending on a `synchronized` block with an operational protocol with a $O(nodes \times tokens)$ logic [25] on a Guava [48] collection, external to Cassandra's application code. User write throughput experienced an unstable (up and down) behavior.

(c) In ca-13923 (Figure 4.12(c)), similarly the (critical) write path this time collides with another background $O(tokens^2)$ method that flushes in-memory data to the storage [27]. The critical write threads need to wait for memory availability for a long time, collapsing the write throughput.

(d) In ha-16850 (Figure 4.12(d)), HDFS's request path for JVM metrics iterates over every outstanding request thread [29] in an $O(outstandingRequests)$ loop that causes increased execution time of other types of requests due to internal JVM-level locking mechanisms, affecting throughput.

(e) In ha-13403 (Figure 4.12(e)), HDFS's user-facing `rename` operation performs $O(files)$ I/O calls [28], and when the number of files is large and they are stored in a third party cloud storage, the operation makes many long network I/O calls that increase the `rename` latency.

(f) In hd-15415 (Figure 4.12(f)), HDFS's `DirectoryScanner` feature performs several $O(blocks)$ operations [30] while holding the same lock used by other operations.

(g) In hd-14366 (Figure 4.12(g)), hd-'s `append` (critical) path contains an $O(dataNode)$ operation in order to identify the live replicas of the file that is being appended [31]. This is an example where the patch only made a small throughput improvement because the loop was necessary and there was no easy way to remove it entirely from the critical path.

(h) In ig-12087 (Figure 4.12(h)), Ignite cache's `putAll` critical path for data map performs a check on all possible grid candidates with an $O(cacheSize \times inputDataSize)$ complexity while holding a lock [32], causing a 50% throughput drop.

(i) In kf-9393 (Figure 4.12(i)), Kafka's `produce()` critical path collides in a `synchronized` block with an operational protocol in charge of cleaning the write logs with an $O(partitions)$ command that lists directory files [34], causing throughput instability.

(j) In kf-8736 (Figure 4.12(j)), Kafka cache's `put()` critical path performs evictions when the cache becomes full and to do that `put()` calls the `size()` method [33], which is typically $O(1)$ in most Java Collection implementations but can be $O(entries)$ if the configuration chooses a specific data structure such as `ConcurrentSkipListMap`. SVIEW caught the superlinear complexity under this configuration as it covers both the $O(request)$ implicit and $O(entries)$ library loops in this specific configuration.

4.6.2 SVIEW *Statistics*

SVIEW is able to identify the harmful DCFs of all the 10 faults we have reproduced. Of course, SVIEW also reports some other DCFs as potentially harmful as well. Although we are interested in investigating whether these reported DCFs are real problems, the developers are typically unwilling to comment on issues reported on old versions of their software. Therefore, we decide to repeat the same exercise on newer versions of these software.

In the previous section, in order to generate prior faults, we ran SVIEW on old versions of our

	Cassandra	**HDFS**	**Ignite**	**Kafka**
Workloads	14	3	1	7
Avg. LOC	71	91	61	67
#APIs/Ops exercised	5	3	1	1
Avg. time (hours)	1.1	0.3	0.2	0.2
#Fragments (unique)	1964	2632	762	1132
#Growth (unique)	167	123	32	148
#Growth (%)	9%	5%	4%	13%
Super Linear (%)	10%	9%	3%	6%
Linear (%)	76%	81%	75%	79%
Sub Linear (%)	14%	10%	22%	15%
CP	7	17	2	0
LC	17	11	1	12
IO	0	2	0	0

Table 4.4: **SVIEW statistics.** *The table shows the number of workloads and the statistic of the resulting outputs.*

target systems specifically 7 Cassandra, 3 HDFS, 2 Ignite, and 3 Kafka old versions. In addition to these, we also run SVIEW on one of their recent stable versions, Cassandra v4.0.0, Hadoop/HDFS v3.3.3, Ignite v2.14, and Kafka v3.2.

Table 4.4 shows the statistics of the SVIEW outputs when running on the more recent versions of our 4 target systems. For each system, we wrote from 1 to 14 scaling workloads (Cassandra has the highest number because that is our first target system); the average line of code is from 61 to 91 LOC (trivial effort); 3 to 5 dimensions being scaled; 1 to 5 system foreground and background operations being exercised; and overall took between 0.2 to 1.1 hours per workload on average. Table 4.5 further details the dimensions and scales we configure for our scaling workloads.

In the middle row section of Table 4.4, we can also see that across the 4 systems, SVIEW found between 762 to 1964 application/library/implicit loops (section 4.4.3), with only 32 to 167 of them that are being marked with real growth (section 4.4.5), and among these growing dimensional code fragments, 3-10% of them have a superlinear trend, 75-81% linear trend, and 10-22% sublinear trends.

Finally in the bottom section of Table 4.4, the three analysis modules (section 4.5) throw 26 warnings for the critical path analysis (CP), 41 for the lock contention analysis (LC) and 2 for I/O

H.L. Dims.	Per-system Dimensions	ca	hd	ig	kf
Cluster	peers	32	.	32	32
	datanodes	.	32	.	.
	consumers	.	.	.	32
Load	requests	32000	32000	32000	32000
	clients	.	32	.	32
Data	blocks	.	1024	.	.
	files		1024	.	.
	tables	64	.	.	.
	keyspaces	64	.	.	.
	rows	32000	.	.	.
	partitions	128	.	128	128
	topics	.	.	.	128

Table 4.5: **Scaling workload configurations.** *The table shows the maximum scale for every dimension that we exercised in our 5 target systems. The step value is 1000 for requests, 1000 for rows, 2 for keyspaces, and 1 for the rest.*

analysis (IO). We emphasize again that they are warnings, because although they might be harmful, there might be no easy way for the developers to change the design. However, we believe these warnings can provide developers with some awareness of a potential problem in the future.

4.6.3 Reporting Potential Issues to Developers

Among all the 69 warnings reported in the previous section, we prioritize analyzing the superlinear trends (2 warnings) and 1 linear trend, since the related dimension can easily reach the scale of millions. Our goal is not to swamp the developers with too many fault reports, but rather only report incrementally starting with the ones that likely could impact performance. We also would like to emphasize that we name them as "potential issues" because scalability faults are not merely implementation bugs but usually point to "design bugs." The former usually can be fixed in a few days if not weeks, while the latter can take weeks, if not months. Going back to our fault study in Section 4.3, 49%, 33%, 20%, and 13% of the scalability issues took more than 1, 3, 6, and 12 months to close. Below we describe 3 clear patterns of potential scalability issues reported by SView.

In Cassandra v4.0.0, the Gossip protocol processes changes in the cluster and gossip messages. When nodes are (de)commissioned, it will execute an $O(nodes \times tokens)$ operation while holding a lock. This means incoming gossip messages from other live nodes must wait. If the execution takes more than the timeout value, the live nodes will be marked unavailable (as their gossips “never arrived”). The developer confirmed that this is the expected behavior. The solution is not to fix the “bug” but rather this information can help operators of large Cassandra clusters to adjust the timeout value (*e.g.*, dynamically [147]) when (de)commissioning a large number of nodes.

In HDFS v3.3.3, there is an array data structure that will be resized (create-new-then-copy) in an $O(blocks)$ operation whenever it is full in order to to accommodate more blocks. This operation is found in at least 4 critical/foreground and 3 operational/background paths while holding a global lock, potentially making requests wait a long time if the number of blocks is in the order of millions. We found 5 older related reports (2 of them still open) where a performance issue is being reported and the stack traces reported include this resize operation. Our report is still marked Open and under discussion with the developers. Two potential solutions are to keep multiple small arrays that don’t require a create-new-then-copy procedure or perform the resize proactively when it is almost full and the load is low.

In Kafka v3.2, response handlers are responsible to handle incoming messages, but when one of the handlers needs to process a `partition-assignment` message, it serializes the operation in $O(consumers \times partitions)$ iterations while holding a global lock, making all the response handlers stall. We felt that this expensive process can be pushed to a background thread. The developers have not responded to our report (after two months).

4.7 Related Work

Approaches for guiding developers on analyzing potential scalability faults can be divided into three main categories: modeling, emulation and extrapolation.

First, efforts have been made to find scalability faults in parallel applications by accurately modeling execution time in terms of number of concurrent processes, processor counters and other low-level hardware resources [115, 120, 132]. As these approaches are designed for parallel applications, their effectiveness or applicability to distributed systems, the main target of SVIEW, remains unclear.

Second, emulation techniques [127, 148, 165, 171, 179] focus on simulating large-scale deployments in a small set of machines, in the hopes of exposing specific fault *symptoms* (*e.g.*, high cpu/memory/storage/network usage). Albeit useful, these approaches are inherently limited by the amount of instances that can be emulated, which varies depending on the type of resource that is being emulated. Moreover, they often require sizable source code modifications that hindrance their adoption. SVIEW is not subject to any of those constrains.

Third, extrapolation techniques [135, 162, 165, 181, 182] are used to identify potential scalability faults using small scale experiments to project systems behavior at large scale. Albeit SVIEW follows a similar approach, it neither focuses on symptoms [135, 162, 165, 182], targets specific types of architectures [162, 181], nor depends on specific types of data structures (*e.g.*, `Java Collections`) [165] for its analysis, as it targets code structures (*e.g.*, loops) that belong to the implementation language, not to specific libraries.

Other related works include instrumentation for performance bottleneck identification [118, 153, 168, 169], static/dynamic analysis tools to detect performance anti-patterns [121, 141, 164, 177], and purely static tools to detect potential performance issues [117, 124, 152, 154, 172, 174]. These approaches are orthogonal to SVIEW as the proposed tools could contribute on specific parts of the process (*e.g.*, loop complexity bounding). However, none of them are specifically focused on scalability faults nor inspect the relationship between a distributed system and its dimensionality.

4.8 Conclusion

According to our findings, issues related to dimensional code fragments (DCFs) often appear after the system has been deployed and only a small percentage of them is found using traditional

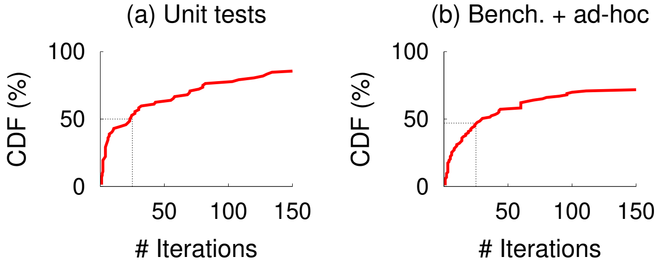


Figure 4.13: **DCF iterations in Cassandra testing tools.** *The figures show the maximum number of iterations performed by 93 DCFs (application loops) counted while using (a) Cassandra v4.0.0 unit tests and (b) benchmarks/ad-hoc testing tools (included in their main repository).*

testing tools (Figure 4.2(b)). Figure 4.13(a)-(b) shows the CDFs (%) of the maximum number of iterations performed in unit tests and benchmarks/ad-hoc tests by 93 (application loops only) of the 167 dimensional code fragments we found in a recent stable version of Cassandra (Table 4.4).

Roughly 50% of the tested DCFs performed less than 25 iterations in both types of test suites. With such a low iteration count, developers cannot identify the performance impact caused by DCFs. Moreover, as only 77% and 81% of the selected fragments are respectively covered by these types of tests, a myriad of scenarios are being ignored and left for post-mortem root cause analysis. The latter depicts how complex it is to detect potential scalability faults. As they are not caused solely by DCFs, linear/superlinear complexity nor specific patterns in source code, but by a combination of all those circumstances, traditional testing tools lack the necessary *guidance* and *depth*.

We believe that this situation gives rise to the need for frameworks like SVIEW. By identifying DCFs, we provide a guided approach that, complemented by analysis modules, delivers the necessary depth in detecting potential scalability faults. Nevertheless, more exciting challenges are in the horizon as more scalability fault root causes need to be addressed. We hope SVIEW motivates more advancements in this research area.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

CHAPTER 6

OTHER WORKS

In this section, I present the abstracts of the papers I have participated on during my Ph.D. program that are not directly related to my main research focus. In all of these papers, I had the pleasure of making intellectual and technical contributions along with my colleagues, to whom I sincerely extend my gratitude.

6.1 Layered And Uniform Contention Mitigation Capabilities For Cloud Storage

In this paper [170], we introduce an ecosystem of contention mitigation supports within the operating system, runtime and library layers. This ecosystem provides an end-to-end request abstraction that enables a uniform type of contention mitigation capabilities, namely request cancellation and delay prediction, that can be stackable together across multiple resource layers. Our evaluation shows that in our ecosystem, multi-resource storage applications are faster by 5-70% starting at 90P (the 90th percentile) compared to popular practices such as speculative execution and is only 3% slower on average compared to a best-case (no contention) scenario.

6.2 TRANSACTUATIONS: Where Transactions Meet The Physical World

A large class of IoT applications read sensors, execute application logic, and actuate actuators. However, the lack of high-level programming abstractions compromises correctness especially in presence of failures and unwanted interleaving between applications. A key problem arises when operations on IoT devices or the application itself fails, which leads to inconsistencies between the physical state and application state, breaking application semantics and causing undesired consequences. Transactions are a well-established abstraction for correctness, but assume properties that are absent in an IoT context. In our **award wining paper** [160, 161], we studied one such environment, smart home, and established inconsistencies manifesting out of failures. We proposed an abstraction called TRANSACTUATION that empowers developers to build reliable applications. Our runtime, RELACS, implemented the abstraction atop a real smart-home platform. We evaluated programmability, performance, and effectiveness of transactuations to demonstrate its potential as a powerful abstraction and execution model.

6.3 FLYMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems

In this paper [144], we present a fast and scalable testing approach for datacenter/cloud systems such as Cassandra, Hadoop, Spark, and ZooKeeper. The uniqueness of our approach is in its ability to overcome the path/state-space explosion problem in testing workloads with complex interleavings of messages and faults. We introduce three powerful algorithms: state symmetry, event independence, and parallel flips, which collectively makes our approach on average 16X (up to 78X) faster than other state-of-the-art solutions. We have integrated our techniques with 8 popular datacenter systems, successfully reproduced 12 old bugs, and found 10 new bugs — all were done without random walks or manual checkpoints.

6.4 MITTOS: Supporting Millisecond Tail Tolerance With Fast Rejecting SLO-Aware OS Interface

In this paper we presented MITTOS [128], a tail-latency support mechanism for distributed storage systems. MITTOS provides operating system support to cut millisecond-level tail latencies for data-parallel applications. In MITTOS, we advocate a new principle that operating system should quickly reject IOs that cannot be promptly served. To achieve this, MITTOS exposes a fast rejecting SLO-aware interface wherein applications can provide their SLOs (e.g., IO deadlines). If

MITTOS predicts that the IO SLOs cannot be met, MITTOS will promptly return *EBUSY* signal, allowing the application to failover (retry) to another less-busy node without waiting. We build MITTOS within the storage stack (disk, SSD, and OS cache managements), but the principle is extensible to CPU and runtime memory managements as well. MITTOS' no-wait approach helps reduce IO completion time up to 35% compared to wait-then-speculate approaches.

6.5 PBSE: A Robust Path-Based Speculative Execution For DegradedNetwork Tail Tolerance In Data-Parallel Frameworks

In this paper [167], we reveal loopholes of Speculative Execution (SE) implementations under a unique fault model: node-level network throughput degradation. This problem appears in many data-parallel frameworks such as Hadoop MapReduce and Spark. To address this, we present PBSE, a robust, path-based speculative execution that employs three key ingredients: path progress, path diversity, and path-straggler detection and speculation. We show how PBSE is superior to other approaches such as cloning and aggressive speculation under the aforementioned fault model. PBSE is a general solution, applicable to many data-parallel frameworks such as Hadoop/HDFS+QFS, Spark and Flume.

REFERENCES

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

[2] Apache Hadoop. https://hadoop.apache.org/.

[3] Apache HBase. https://hbase.apache.org/.

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

[5] Apache Ignite. http://ignite.apache.org.

[6] Apache JIRA. https://issues.apache.org/jira/secure/Dashboard.jspa.

[7] Apache Kafka. https://kafka.apache.org/.

[8] Apache Software Foundation. https://www.apache.org/.

[9] Apache Spark. https://spark.apache.org/.

[10] Apache Storm. http://storm.apache.org/.

[11] Apache Yarn. http://hadoop.apache.org/.

[12] AspectJ. www.eclipse.org/aspectj.

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

[14] CASSANDRA-14821: Make it possible to run multi-node coordinator/replica tests in a single JVM. https://issues.apache.org/jira/browse/CASSANDRA-14821.

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

[16] CASSANDRA-9100: Gossip is inadequately tested. https://issues.apache.org/jira/browse/CASSANDRA-9100.

[17] CASSANDRA admin commands. https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/tools/toolsNodetool.html.

[18] Cassandra Compaction. https://cassandra.apache.org/doc/latest/operating/compaction/index.html.

[19] Cassandra Distributed Tests (DTests). https://github.com/apache/cassandra-dtest.

[20] Cassandra Internal Benchmarks. https://github.com/apache/cassandra/tree/trunk/test/microbench/org/apache/cassandra/test/microbench.

[21] Cassandra NodeTool. https://cassandra.apache.org/doc/latest/tools/nodetool/nodetool.html.

[22] Cassandra Scrub Command. https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/tools/toolsScrub.html.

[23] Cassandra Stress. https://cassandra.apache.org/doc/latest/tools/cassandra_stress.html.

[24] Chameleon. https://www.chameleoncloud.org.

[25] (DCF) CASSANDRA 3.0.17: create. https://github.com/apache/cassandra/blob/d52c7b8c595cc0d06fc3607bf16e3f595f016bb6/src/java/org/apache/cassandra/utils/SortedBiMultiValMap.java#L59.

[26] (DCF) CASSANDRA 3.11.0: getAddressReplicas. https://github.com/Instagram/cassandra/blob/15141-trunk/src/java/org/apache/cassandra/locator/AbstractReplicationStrategy.java#L240.

[27] (DCF) CASSANDRA 3.11.0: getRangeAddresses. https://github.com/apache/cassandra/blob/88dee7e9d515ad94ecf8f2309f1e6138ec79e1a2/src/java/org/apache/cassandra/locator/AbstractReplicationStrategy.java#L193.

[28] (DCF) HADOOP 2.7.2: execute. https://github.com/apache/hadoop/blob/branch-2.7.2/hadoop-tools/hadoop-azure/src/main/java/org/apache/hadoop/fs/azure/NativeAzureFileSystem.java#L399.

[29] (DCF) HADOOP 2.8.2: getThreadUsage. https://github.com/apache/hadoop/blob/branch-2.8.2/hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/metrics2/source/JvmMetrics.java#L174.

[30] (DCF) HDFS 3.2.0: scan. https://github.com/apache/hadoop/blob/f6c4e006cd1190e27fadbe0a38ce09782f45ca04/hadoop-hdfs-project/hadoop-hdfs/src/main/java/org/apache/hadoop/hdfs/server/datanode/DirectoryScanner.java#L398.

[31] (DCF) HDFS 3.3.0: getNumLiveDatanodes. https://github.com/apache/hadoop/blob/rel/release-3.3.0/hadoop-hdfs-project/hadoop-hdfs/src/main/java/org/apache/hadoop/hdfs/server/blockmanagement/DatanodeManager.java#L1259.

[32] (DCF) Ignite 2.7.6: checkThreadChain. https://github.com/apache/ignite/blob/626c4cda245940ad87958b3698ce2a46ec72ea66/modules/core/src/main/java/org/apache/ignite/internal/processors/cache/distributed/GridDistributedCacheEntry.java#L748.

[33] (DCF) KAFKA 2.3.0: maybeEvict. https://github.com/apache/kafka/blob/2.3.0/streams/src/main/java/org/apache/kafka/streams/state/internals/ThreadCache.java#L238.

[34] (DCF) KAFKA 2.4.0: listSnapshotFiles. https://github.com/apache/kafka/blob/2.4/core/src/main/scala/kafka/log/ProducerStateManager.scala#L458.

[35] Eclipse Java development tools. http://www.eclipse.org/jdt/.

[36] Elvis: Erlang Style Reviewer. https://github.com/inaka/elvis.

[37] Erlang man page: Dialyzer. http://erlang.org/doc/man/dialyzer.html.

[38] (Fault) CASSANDRA-13923: Flushers blocked due to many SSTables. https://issues.apache.org/jira/browse/CASSANDRA-13923.

[39] (Fault) CASSANDRA-14660: Improve TokenMetaData cache populating performance for large cluster. https://issues.apache.org/jira/browse/CASSANDRA-14660.

[40] (Fault) CASSANDRA-15141: Faster token ownership calculation for NetworkTopologyStrategy. https://issues.apache.org/jira/browse/CASSANDRA-15141.

[41] (Fault) HADOOP-13403: AzureNativeFileSystem rename/delete performance improvements. https://issues.apache.org/jira/browse/HADOOP-13403.

[42] (Fault) HADOOP-16850: Support getting thread info from thread group for JvmMetrics to improve the performance. https://issues.apache.org/jira/browse/HADOOP-16850.

[43] (Fault) HDFS-14366: Improve HDFS append performance. https://issues.apache.org/jira/browse/HDFS-14366.

[44] (Fault) HDFS-15415: Reduce locking in Datanode DirectoryScanner. https://issues.apache.org/jira/browse/HDFS-15415.

[45] (Fault) IGNITE-12087: Transactional putAll - significant performance drop on big batches of entries. https://issues.apache.org/jira/browse/IGNITE-12087.

[46] (Fault) KAFKA-8736: Performance: ThreadCache uses size() for empty cache check. https://issues.apache.org/jira/browse/KAFKA-8736.

[47] (Fault) KAFKA-9393: DeleteRecords may cause extreme lock contention for large partition directories. https://issues.apache.org/jira/browse/KAFKA-9393.

[48] Google Core Libraries for Java (Guava). https://github.com/google/guava.

[49] HADOOP-1073: DFS Scalability: high CPU usage in choosing replication targets and file open. https://issues.apache.org/jira/browse/HADOOP-1073.

[50] Hadoop MapReduce. http://hadoop.apache.org/.

[51] Hadoop MiniCluster. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/CLIMiniCluster.html.

[52] Hadoop NN Benchmarks. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/Benchmarking.html.

[53] Hadoop Stress. https://community.cloudera.com/t5/Community-Articles/Benchmarking-Hadoop-with-TeraGen-TeraSort-and-TeraValidate/ta-p/248381.

[54] HBase Stress. https://blog.cloudera.com/hbase-performance-testing-using-ycsb/.

[55] Hbase Test Utils. https://github.com/apache/hbase/tree/master/hbase-testing-util.

[56] HDFS-395: DFS Scalability: Incremental block reports. https://issues.apache.org/jira/browse/HDFS-395.

[57] HDFS-4061: Large number of decommission freezes the Namenode. https://issues.apache.org/jira/browse/HDFS-4061.

[58] HDFS-9198: Coalesce IBR processing in the NN. https://issues.apache.org/jira/browse/HDFS-9198.

[59] HDFS admin commands. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSCommands.html.

[60] HDFS Benchmarks. https://github.com/erikmuttersbach/hdfs-benchmark.

[61] HDFS Data Blocks. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Data+Blocks.

[62] How Microsoft drives exabyte analytics on the world's largest YARN cluster. https://bit.ly/3p8DEN4.

[63] How Netflix manages petabyte scale Apache Cassandra in the cloud. https://www.apachecon.com/acna19/s/#/scheduledEvent/1010.

[64] How to Write a Dtest. https://www.datastax.com/blog/how-write-dtest.

[65] IGNITE admin commands. https://ignite.apache.org/docs/latest/tools/control-script.

[66] Ignite Internal Benchmarks. https://github.com/apache/ignite/tree/master/modules/benchmarks/src/main/java/org/apache/ignite/internal/benchmarks.

[67] Ignite Yardstick Stress Test. https://ignite.apache.org/docs/latest/perf-and-troubleshooting/yardstick-benchmarking.

[68] Java Collections. https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html.

[69] Java Executors. https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html.

[70] Java Garbage Collection. https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html.

[71] Java NIO Selector. http://tutorials.jenkov.com/java-nio/selectors.html.

[72] Java Reflection API. https://docs.oracle.com/javase/tutorial/reflect/index.html.

[73] Java Thread Basics. https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/thread_basics.html.

[74] JBoss AS 7 classloading. http://www.mastertheboss.com/jboss-server/jboss-as-7/jboss-as-7-classloading.

[75] JConsole. https://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html.

[76] JMH. https://openjdk.java.net/projects/code-tools/jmh/.

[77] JMX. https://docs.oracle.com/javase/7/docs/technotes/guides/jmx/.

[78] JXray. https://jxray.com/.

[79] KAFKA admin commands. https://docs.cloudera.com/documentation/enterprise/6/6.3/topics/kafka_admin_cli.html.

[80] Kafka Stress. https://docs.cloudera.com/runtime/7.2.2/kafka-managing/topics/kafka-manage-cli-perf-test.html.

[81] Kafka Stress. https://github.com/apache/kafka/tree/trunk/jmh-benchmarks.

[82] Kafka Test API. https://github.com/apache/kafka/tree/trunk/tests/kafkatest/tests.

[83] (Misc) Chameleon Trovi. https://chameleoncloud.readthedocs.io/en/latest/technical/sharing.html.

[84] (Misc) Java Instrumentation Agents. https://docs.oracle.com/javase/8/docs/technotes/guides/instrumentation/index.html.

[85] (Misc) Javassist Github Repository. https://github.com/jboss-javassist/javassist.

[86] Most of Zoom runs on AWS, not Oracle - says AWS. https://www.datacenterdynamics.com/en/news/most-zoom-runs-aws-not-oracle-says-aws/.

[87] MRUnit. https://mrunit.apache.org/.

[88] Performance Antipattern: Chatty I/O. https://docs.microsoft.com/en-us/azure/architecture/antipatterns/chatty-io/.

[89] Performance Antipattern: Extraneous Fetching antipattern. https://docs.microsoft.com/en-us/azure/architecture/antipatterns/extraneous-fetching/.

[90] Performance Flame Graphs. http://www.brendangregg.com/flamegraphs.html.

[91] 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.

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

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

[94] Riak. http://basho.com/products/riak-kv.

[95] RIAK: Large ring_creation_size. https://docs.riak.com/riak/kv/latest/configuring/basic/index.html#ring-size.

[96] ScaleFault DB: A collection of scalability faults, root causes and solution patterns. Available after acceptance due to anonimization requirements.

[97] Singletons are pathological liars. https://testing.googleblog.com/2008/08/by-miko-hevery-so-you-join-new-project.html.

[98] Spark Stress. https://codait.github.io/spark-bench/.

[99] Spark Test API. https://zedar.gitbooks.io/spark-hadoop-notes/content/spark_unit_testing_with_hdfs.html.

[100] Storm Stress Test. https://github.com/yahoo/storm-perf-test.

[101] TestDFSIO Benchmark. https://github.com/tthx/testdfsio.

[102] The Infrastructure Behind Twitter: Scale. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html.

[103] The YARN Timeline Service v.2. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/TimelineServiceV2.html.

[104] Understanding Memory Management. https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html#:~:text=Java%20objects%20reside%20in%20an,making%20space%20for%20new%20objects.

[105] VOLDEMORT: Number of partitions. https://groups.google.com/forum/#!msg/project-voldemort/3vrZfZgQp2Y/Uqt8NgJHg4AJ.

[106] Werner Vogels: A Word on Scalability. https://www.allthingsdistributed.com/2006/03/a_word_on_scalability.html.

[107] Yarn Stress Test. https://hadoop.apache.org/docs/current/hadoop-sls/SchedulerLoadSimulator.html.

[108] Yarn Test API. https://github.com/naver/hadoop/tree/master/hadoop-yarn-project/hadoop-yarn/hadoop-yarn-server/hadoop-yarn-server-tests/src/test/java/org/apache/hadoop/yarn/server.

[109] YCSB. https://github.com/brianfrankcooper/YCSB.

[110] Zoom: A Message To Our Users. https://blog.zoom.us/a-message-to-our-users/.

[111] Boris Aronov, Sariel Har-Peled, Christian Knauer, Yusu Wang, and Carola Wenk. Fréchet Distance for Curves, Revisited. In *Proceedings of the 14th Annual European Symposium (AES)*, 2006.

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

[113] Sarah Boslaugh and Paul Andrew Watters. *Statistics in a nutshell - a desktop quick reference*. O'Reilly, 2008.

[114] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

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

[116] Natalia Chechina, Huiqing Li, Amir Ghaffari, Simon Thompson, and Phil Trindera. Improving the network scalability of Erlang. *Journal of Parallel and Distributed Computing*, 90-91:22–34, April 2016.

[117] Bihuan Chen, Yang Liu, and Wei Le. Generating Performance Distributions via Probabilistic Symbolic Execution. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016.

[118] Jinfu Chen, Weiyi Shang, and Emad Shihab. PerfJIT: Test-Level Just-in-Time Prediction for Performance Regression Introducing Commits. *IEEE Transactions on Software Engineering (TSE)*, 48(5):1529–1544, 2022.

[119] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Detecting Performance Anti-patterns for Applications Developed using Object-Relational Mapping. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014.

[120] Sai P. Chenna, Greg Stitt, and Herman Lam. Multi-Parameter Performance Modeling using Symbolic Regression. In *The 2019 International Conference on High Performance Computing and Simulation (HPCS)*, 2019.

[121] Ting Dai, Daniel Dean, Peipei Wang, Xiaohui Gu, and Shan Lu. Hytrace: A Hybrid Approach to Performance Bug Diagnosis in Production Cloud Infrastructures. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC)*, 2017.

[122] Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. Uncovering Bugs in Distributed Storage Systems during Testing (Not in Production!). In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[123] Anh-Tu Do-Mai, Thanh-Dang Diep, and Nam Thoai. Race Condition and Deadlock Detection for Large-Scale Applications. In *Proceedings of the 15th International Symposium on Parallel and Distributed Computing (ISPDC)*, 2016.

[124] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2009.

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

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

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

[128] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

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

[130] Val Henson, Zach Brown, Theodore Ts'o, and Arjan van de Ven. Reducing fsck time for ext2 file systems. In *Ottawa Linux Symposium (OLS)*, 2006.

[131] Michihiro Horie, Kazunori Ogata, Kiyokuni Kawachiya, and Tamiya Onodera. String Deduplication for Java-based Middleware in Virtualized Environments. In *The 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2014.

[132] Yuyang Jin, Haojie Wang, Teng Yu, Xiongchao Tang, Torsten Hoefler, Xu Liu, and Jidong Zhai. SCALANA: Automating Scaling Loss Detection with Graph Analysis. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[133] Anshul Jindal, Paul Staab, Jorge Cardoso, Michael Gerndt, and Vladimir Podolskiy. Online Memory Leak Detection in the Cloud-based Infrastructures. In *Proceedings of the International Workshop on Artificial Intelligence for IT Operations (AIOPS)*, 2020.

[134] Kamil Jezek and Richard Lipka. Antipatterns causing memory bloat: A case study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017.

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

[136] Avinash Lakshman and Prashant Malik. Cassandra - A Decentralized Structured Storage System. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2009.

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

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

[139] Tanakorn Leesatapornwongsa, Cesar A. Stuardo, Riza O. Suminto, Huan Ke, Jeffrey F. Lukman, and Haryadi S. Gunawi. Scalability Bugs: When 100-Node Testing is Not Enough. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS XVI)*, 2017.

[140] Colin Lewis-Beck and Michael Lewis-Beck. *Applied regression: An introduction*. Sage publications, 2015.

[141] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S. Gunawi, Xiaohui Gu, Dongsheng Li, and Xicheng Lu. PCatch: Automatically Detecting Performance Cascading Bugs in Cloud Systems. In *Proceedings of the 2018 EuroSys Conference (EuroSys)*, 2018.

[142] Thomas A. Limoncelli and Doug Hughe. LISA '11 Theme – DevOps: New Challenges, Proven Values. *USENIX ;login: Magazine*, 36(4), 2011.

[143] John D. C. Little. A Proof for the Queuing Formula. *Operations Research*, 9(3):383–387, June 1961.

[144] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the 2019 EuroSys Conference (EuroSys)*, 2019.

[145] Yingjun Lyu, Ali Alotaibi, and William G. J. Halfond. Quantifying the Performance Impact of SQL Antipatterns on Mobile Applications. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019.

[146] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ffsck: The Fast File System Checker. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST)*, 2013.

[147] Sixiang Ma and Yang Wang. Accurate Timeout Detection despite Arbitrary Processing Delays. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.

[148] Nuno Machado, Francisco Maia, Francisco Neves, Fábio Coelho, and José Pereira. Minha: Large-scale distributed systems testing made practical. In *Proceedings of the 23rd International Conference on Principles of Distributed Systems (OPODIS)*, 2019.

[149] Bo Mao, Hong Jiang, Suzhen Wu, and Lei Tian. Leveraging Data Deduplication to Improve the Performance of Primary Storage Systems in the Cloud. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SOCC)*, 2013.

[150] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. Four Trends Leading to Java Runtime Bloat. *Communications of the ACM (CACM)*, 27, 2010.

[151] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A High-Performance Big-Data-Friendly Garbage Collector. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[152] Adrian Nistor, Po-Chun Chang, Cosmin Rădoi, and Shan Lu. CARAMEL: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

[153] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting Performance Problems via Similar Memory-Access Patterns. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, 2013.

[154] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[155] Oracle. JVMTM Tool Interface version 1.2. https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html.

[156] John Ousterhout. Is Scale Your Enemy, Or Is Scale Your Friend?: Technical Perspective. *Communications of the ACM (CACM)*, 54(7), 2011.

[157] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[158] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015.

[159] Claude Sammut and Geoffrey I. Webb. *Encyclopedia of Machine Learning*. Springer Publishing Company, Incorporated, 2011.

[160] Aritra Sengupta, Tanakorn Leesatapornwongsa, Masoud Saeida Ardekani, Gustavo Petri, and Cesar A. Stuardo. Transactuations: Where Transactions Meet The Physical World. *ACM Transactions on Computer Systems (TOCS)*, 36:1–31, November 2018.

[161] Aritra Sengupta, Tanakorn Leesatapornwongsa, Masoud Saeida Ardekani, and Cesar A. Stuardo. Transactuations: Where Transactions Meet The Physical World. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[162] Rong Shi, Yifan Gan, and Yang Wang. Evaluating Scalability Bottlenecks by Workload Extrapolation. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2018.

[163] Konstantin V. Shvachko. HDFS Scalability: The Limits to Growth. *USENIX ;login:*, 35(2):6–16, 2010.

[164] Linhai Song and Shan Lu. Performance Diagnosis for Inefficient Loops. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, 2017.

[165] Cesar A. Stuardo, Tanakorn Leesatapornwongsa, Riza O. Suminto, Huan Ke, Jeffrey F. Lukman, Daniar H. Kurniawan, Wei-Chiu Chuang, Shan Lu, and Haryadi S. Gunawi. ScaleCheck: A Single-Machine Approach for Discovering Scalability Bugs in Large Distributed Systems. In *Proceedings of the 17th USENIX Symposium on File and Storage Technologies (FAST)*, 2019.

[166] Cesar A. Stuardo, Hao-Nan Zhu, Miao Yu, Patrick J. Chapman, Cindy Rubio-Gonzáles, Yang Wang, and Haryadi S. Gunawi. SView: Identifying and Analyzing Potential Scalability Faults in Large-Scale Distributed Systems. In *In submission.*, 2023.

[167] Riza O. Suminto, Cesar A. Stuardo, Alexandra Clark, Huan Ke, Tanakorn Leesatapornwongsa, Bo Fu, Daniar H. Kurniawan, Vincentius Martin, Uma Maheswara Rao G., and Haryadi S. Gunawi. PBSE: A Robust Path-Based Speculative Execution for Degraded-Network Tail Tolerance in Data-Parallel Frameworks. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC)*, 2017.

[168] Mert Toslali, Emre Ates, Alex Ellis, Zhaoqi Zhang, Darby Huye, Lan Liu, Samantha Puterman, Ayse K. Coskun, and Raja R. Sambasivan. Automating Instrumentation Choices for Performance Problems in Distributed Applications with VAIF. In *Proceedings of the 12th ACM Symposium on Cloud Computing (SoCC)*, 2021.

[169] Mohammad Mejbah ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs. In *Proceedings of the 2017 EuroSys Conference (EuroSys)*, 2017.

[170] Meng Wang, Cesar A. Stuardo, Daniar Heri Kurniawan, Ray A. O. Sinurat, and Haryadi S. Gunawi. Layered Contention Mitigation for Cloud Storage. In *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*, 2022.

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

[172] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. Singularity: Pattern Fuzzing for Worst Case Complexity. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2018.

[173] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[174] Xiaofei Xie, Bihuan Chen, Liang Zou, Yang Liu, Wei Le, and Xiaohong Li. Automatic Loop Summarization via Path Dependency Analysis. *IEEE Transactions on Software Engineering (TSE)*, 45(6):537–557, 2019.

[175] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. How not to structure your database-backed web applications: a study of performance bugs in the wild. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, 2018.

[176] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. How not to structure your database-backed web applications: a study of performance bugs in the wild. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, 2018.

[177] Junwen Yang, Cong Yan, Chengcheng WanS, Shan Lu, and Alvin Cheung. View-Centric Performance Optimization for Database-Backed Web Applications. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, 2019.

[178] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2011.

[179] Yukun Zeng, Mengyuan Chao, and Radu Stoleru. EmuEdge: A Hybrid Emulator for Reproducible and Realistic Edge Computing Experiments. In *Proceedings of the 2019 IEEE International Conference on Fog Computing (ICFC)*, 2019.

[180] Yutong Zhao, Lu Xiao, Xiao Wang, Lei Sun, Bihuan Chen, Yang Liu, and Andre B. Bondi. How Are Performance Issues Caused and Resolved? - An Empirical Study from a Design Perspective. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2020.

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

[182] Wenju Zhou, Jiepeng Zhang, Jingwei Sun, and Guangzhong Sun. Using Small-Scale History Data to Predict Large-Scale Performance of HPC Application. In *Proceedings of the 34th IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020.

APPENDIX A

SCALECHECK SELECTED SCALABILITY FAULTS

Table A.1 shows the complete list of faults that were considered in chapter 3.

#	System	Issue #	Title
1	CASSANDRA	2058	Load spikes due to MessagingService-generated garbage collection
2	CASSANDRA	3831	Scaling to large clusters in GossipStage impossible due to calculatePendingRanges
3	CASSANDRA	3881	Reduce computational complexity of processing topology changes
4	CASSANDRA	4288	Prevent thrift server from starting before gossip has settled
5	CASSANDRA	5220	Repair improvements when using vnodes
6	CASSANDRA	5456	Large number of bootstrapping nodes cause gossip to stop working
7	CASSANDRA	6127	vnodes don't scale to hundreds of nodes
8	CASSANDRA	6268	Poor performance of HADOOP if any DC is using VNodes
9	CASSANDRA	6345	Endpoint cache invalidation causes CPU spike (on vnode rings?)
10	CASSANDRA	6409	Gossip performance improvement at node startup
11	CASSANDRA	6485	NPE in calculateNaturalEndpoints
12	CASSANDRA	6862	Poor performance of HADOOP if any DC is using VNodes
13	CASSANDRA	13968	Cannot replace a live node in large clusters
14	COUCHBASE	1040	Improve bootstrapping speed by creating/initializing all nodes in parallel
15	COUCHBASE	8640	Rightscale template :: 15 of 120 node sized array stranded in booting: exited with 2, expected 0.
16	COUCHBASE	13102	Empty and idle node runs Flusher frequently, problem for scaling past 10 buckets
17	COUCHBASE	15757	Graceful failover either fails or takes very long time, delta rebalance fails - with latest build 3470

#	System	Issue #	Title
18	COUCHBASE	16807	New UI slightly slow at least on 130 node cluster
19	HADOOP	3656	Sort job on 350 scale is consistently failing with latest MRV2 code
20	HADOOP	3711	AppMaster recovery for Medium to large jobs take long time
21	HADOOP	4478	TaskTracker's heartbeat is out of control
22	HADOOP	4946	Type conversion of map completion events leads to performance problems with large jobs
23	HADOOP	5124	AM lacks flow control for task events
24	HADOOP	5508	JobTracker memory leak caused by unreleased FileSystem objects in JobInProgress#cleanupJob
25	HBASE	3620	Make HBCK Faster
26	HBASE	4742	Split dead servers log in parallel
27	HBASE	5422	StartupBulkAssigner would cause a lot of timeout on RIT when assigning large numbers of regions (timeout = 3 mins)
28	HBASE	6728	Prevent OOM possibility due to per connection responseQueue being unbounded
29	HBASE	7060	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
30	HBASE	7190	Add an option to hbck to check only meta and assignment
31	HBASE	8778	Region assigments scan table directory making them slow for huge tables
32	HBASE	9208	ReplicationLogCleaner slow at large scale
33	HBASE	9377	Backport HBASE- 9208 ”ReplicationLogCleaner slow at large scale”
34	HBASE	9775	Client write path perf issues
35	HBASE	10209	Speed region assign in failover
36	HBASE	11290	Unlock RegionStates

#	System	Issue #	Title
37	HBASE	12139	StochasticLoadBalancer doesn't work on large lightly loaded clusters
38	HDFS	354	Data node process consumes 180% cpu
39	HDFS	395	DFS Scalability: Incremental block reports
40	HDFS	611	Heartbeats times from Datanodes increase when there are plenty of blocks to delete
41	HDFS	1073	DFS Scalability: high CPU usage in choosing replication targets and file open
42	HDFS	1851	HDFS-15 scalability improvements
43	HDFS	2495	Increase granularity of write operations in ReplicationMonitor thus reducing contention for write lock
44	HDFS	2938	Recursive delete of a large directory makes namenode unresponsive
45	HDFS	3990	NN's health report has severe performance problems
46	HDFS	4061	Large number of decommission freezes the Namenode
47	HDFS	4075	Reduce recommissioning overhead
48	HDFS	4360	Multiple BlockFixer should be supported in order to improve scalability and reduce too much work on single BlockFixer
49	HDFS	4479	logSync() with the FSNamesystem lock held in commitBlockSynchronization
50	HDFS	4937	ReplicationMonitor can infinite-loop in BlockPlacementPolicyDefault#chooseRandom()
51	HDFS	9198	Coalesce IBR processing in the NN
52	HDFS	9287	Block placement completely fails if too many nodes are decommissioning
53	HDFS	10609	Uncaught InvalidEncryptionKeyException during pipeline recovery may abort downstream applications

#	System	Issue #	Title
54	RIAK	3926	Large ring_creation_size
55	VOLDEMORT	1212	Number of Partition

Table A.1: **SCALECHECK selected scalability faults.** *This table shows the* 55 *faults chosen when evaluating* SCALECHECK, *where the second column is a hyperlink to the corresponding JIRA report. All of these faults are dependent on the "cluster size" dimension, described in chapter* 2.

APPENDIX B

SFIND$_{PIL}$ ALGORITHMS

Algorithms 1 to 2 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 isPILCandidate
Input  loop : Loop
       loopContext : Method
Output pilType : int
       category : string
begin
 // 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 stamenent : loop.getStatements())
  if(!statement.isIOOPeration()) hasOnlyNonPertinentOperations = false
 // can be PIL-ed without memoization
 if(hasOnlyNonPertinentOperations) return 1
 // now second case
 for(Statement stamenent : 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)) {
   return 2
  }
 }
 // cannot be PIL-ed
 return 0
end
```

Algorithm 1: **isPILCandidate.**

Algorithm 1 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.

Algorithm 2 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 figures B.1 and B.2. We use the same algorithm to illustrate the two cases described above since the difference between them are minimal from an implementation perspective.

```
Algorithm insertPreMemoizationCode
Input  targetBlock : CodeBlock
       targetMethod : Method
       replayBlock  : CodeBlock
       memoizeBlock : CodeBlock
       pilType      : int
Output None
begin
 // we have identified the target block we want to modify, so we just insert code
 targetMethod.insertBefore(targetBlock, replayBlock);
 if (pilType == 2) targetMethod.insertBefore(targetBlock, memoizeBlock);
end
```

Algorithm 2: **insertPreMemoizationCode.**

```
Time time = now();
// execution of target block happens here
Time elapsed = now() - time;
if (isMemoizeEnabled()) {
 State state = StateManager.recordClustesState();
 StateSerializer.recordClusterState(state, elapsed);
}
// method continues
```

Figure B.1: **SFind$_{PIL}$ sample memoization block.**

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

```
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
}
```

Figure B.2: **SFIND$_{PIL}$ sample replay block.**

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.

APPENDIX C

SVIEW SELECTED SCALABILITY FAULTS

Table C.1 shows the complete list of faults that were considered in chapter 4.

#	System	Issue #	Title
1	CASSANDRA	15141	Faster token ownership calculation for NetworkTopologyStrategy
2	CASSANDRA	14660	Improve TokenMetaData cache populating performance for large cluster
3	CASSANDRA	13923	Flushers blocked due to many SSTables
4	CASSANDRA	13065	Skip building views during base table streams on range movements
5	CASSANDRA	12281	Gossip blocks on startup when there are pending range movements
6	CASSANDRA	12245	initial view build can be parallel
7	CASSANDRA	10654	Make MV streaming rebuild parallel
8	CASSANDRA	9258	Range movement causes CPU and performance impact
9	CASSANDRA	7758	Some gossip messages are very slow to process on vnode clusters
10	CASSANDRA	6488	Batchlog writes consume unnecessarily large amounts of CPU on vnodes clusters
11	CASSANDRA	6345	Endpoint cache invalidation causes CPU spike (on vnode rings?)
12	CASSANDRA	6297	Gossiper blocks when updating tokens and turns node down
13	CASSANDRA	5456	Large number of bootstrapping nodes cause gossip to stop working
14	CASSANDRA	3881	reduce computational complexity of processing topology changes
15	CASSANDRA	3831	scaling to large clusters in GossipStage impossible due to calculatePendingRanges
16	HADOOP	16850	Support getting thread info from thread group for JvmMetrics to improve the performance
17	HADOOP	14600	LocatedFileStatus constructor forces RawLocalFS to exec a process to get the permissions

#	System	Issue #	Title
18	HADOOP	14369	NetworkTopology calls expensive toString() when logging
19	HADOOP	13403	AzureNativeFileSystem rename/delete performance improvements
20	HADOOP	4061	Large number of decommission freezes the Namenode
21	HADOOP	1073	DFS Scalability: high CPU usage in choosing replication targets and file open
22	HBASE	11368	Multi-column family BulkLoad fails if compactions go on too long
23	HBASE	10209	Speed region assign in failover
24	HBASE	9377	Backport HBASE- 9208 ”ReplicationLogCleaner slow at large scale”
25	HBASE	9208	ReplicationLogCleaner slow at large scale
26	HBASE	8778	Region assigments scan table directory making them slow for huge tables
27	HDFS	15415	Reduce locking in Datanode DirectoryScanner
28	HDFS	15150	Introduce read write lock to Datanode
29	HDFS	14997	BPServiceActor processes commands from NameNode asynchronously
30	HDFS	14859	Prevent unnecessary evaluation of costly operation getNumLiveDataNodes when dfs.namenode.safemode.min.datanodes is not zero
31	HDFS	14854	Create improved decommission monitor implementation
32	HDFS	14657	Refine NameSystem lock usage during processing FBR
33	HDFS	14613	BlockManagerSafeMode should avoid to check datanode thresholds with default zero value.
34	HDFS	14497	Write lock held by metasave impact following RPC processing
35	HDFS	14476	lock too long when fix inconsistent blocks between disk and in-memory
36	HDFS	14366	Improve HDFS append performance

#	System	Issue #	Title
37	HDFS	14171	Performance improvement in Tailing EditLog
38	HDFS	13821	RBF: Add dfs.federation.router.mount-table.cache.enable so that users can disable cache
39	HDFS	13702	Remove HTrace hooks from DFSClient to reduce CPU usage
40	HDFS	13136	Avoid taking FSN lock while doing group member lookup for FSD permission check
41	HDFS	12998	SnapshotDiff - Provide an iterator-based listing API for calculating snapshotDiff
42	HDFS	12866	Recursive delete of a large directory or snapshot makes namenode unresponsive
43	HDFS	12749	DN may not send block report to NN after NN restart
44	HDFS	11225	NameNode crashed because deleteSnapshot held FSNamesystem lock too long
45	HDFS	10477	Stop decommission a rack of DataNodes caused NameNode fail over to standby
46	HDFS	7213	processIncrementalBlockReport performance degradation
47	HDFS	5790	LeaseManager.findPath is very slow when many leases need recovery
48	HDFS	5757	refreshNodes with many nodes at the same time could slow down NN
49	HDFS	5341	Reduce fsdataset lock duration during directory scanning.
50	HDFS	5153	Datanode should send block reports for each storage in a separate message
51	HDFS	4075	Reduce recommissioning overhead
52	IGNITE	12087	Transactional putAll - significant performance drop on big batches of entries.
53	IGNITE	8681	Using ExpiryPolicy with persistence causes significant slowdown.

#	System	Issue #	Title
54	IGNITE	5578	Discovery events coalescing
55	IGNITE	5521	Large near caches lead to cluster instability with metrics enabled
56	IGNITE	1837	Rebalancing on a big cluster (30 nodes and more)
57	KAFKA	9393	DeleteRecords may cause extreme lock contention for large partition directories
58	KAFKA	8736	Performance: ThreadCache uses size() for empty cache check
59	KAFKA	7142	Rebalancing large consumer group can block the coordinator broker for several seconds
60	KAFKA	5642	Use async ZookeeperClient in Controller
61	KAFKA	4851	SessionStore.fetch(key) is a performance bottleneck
62	KAFKA	4469	Consumer throughput regression caused by inefficient list removal and copy
63	KAFKA	4415	Reduce time to create and send MetadataUpdateRequest
64	SPARK	29351	Avoid full synchronization in ShuffleMapStage
65	SPARK	29048	Query optimizer slow when using Column.isInCollection() with a large size collection
66	SPARK	27801	InMemoryFileIndex.listLeafFiles should use listLocatedStatus for DistributedFileSystem

Table C.1: **SVIEW selected scalability faults.** *This table shows the* 66 *faults chosen when evaluation* SVIEW, *where the second column is a hyperlink to the corresponding JIRA report.*