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

Cesar A. Stuardo, *University of Chicago;* Tanakorn Leesatapornwongsa, *Samsung Research America;* Riza O. Suminto, Huan Ke, and Jeffrey F. Lukman, *University of Chicago;* Wei-Chiu Chuang, *Cloudera;* Shan Lu and Haryadi S. Gunawi, *University of Chicago*

https://www.usenix.org/conference/fast19/presentation/stuardo

This paper is included in the Proceedings of the
17th USENIX Conference on File and Storage Technologies (FAST '19).

February 25–28, 2019 • Boston, MA, USA

978-1-939133-09-0

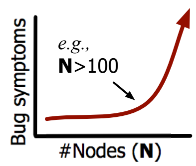
# ScaleCheck: A Single-Machine Approach for Discovering Scalability Bugs in Large Distributed Systems

Cesar A. Stuardo, Tanakorn Leesatapornwongsa*, Riza O. Suminto, Huan Ke, Jeffrey F. Lukman, Wei-Chiu Chuang†, Shan Lu, and Haryadi S. Gunawi

University of Chicago    *Samsung Research America    †Cloudera

## Abstract

*We present* SCALECHECK*, an approach for discovering scalability bugs (a new class of bug in large storage systems) and for democratizing large-scale testing.* SCALECHECK *employs a program analysis technique, for finding potential causes of scalability bugs, and a series of colocation techniques, for testing implementation code at real scales but doing so on just a commodity PC.* SCALECHECK *has been integrated to several large-scale storage systems, Cassandra, HDFS, Riak, and Voldemort, and successfully exposed known and unknown scalability bugs, up to 512-node scale on a 16-core PC.*




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

**Examples:**
"obvious symptom in 1000 nodes" [Cassandra bug #6127],
"with >500 nodes, ... trouble" [# 6409];
"16800 maps [recovery] was slow" [Hadoop #3711],
"1900 nodes, [namenode's] queue overflowed" [#4061];
"with >200 nodes, it doesn't work" [HBase #12139].

Figure 1: **Scalability bugs.** *Definition and quotes from scalability bug reports. Detailed examples are in §2a and §5.1.*

## 1 Introduction

Being a critical backend of many today's applications and services, storage systems must be highly reliable. Decades of research address a variety of storage dependability issues, including availability [44, 55], consistency [41, 77], durability [51, 72], integrity [36, 56], security [53, 71], and reliability [73, 74].

The dependability challenge grows as storage systems continue to *scale* in large distributed manners, especially in the last couple of years where the field witnesses a phenomenal deployment scale; Netflix runs tens of 500-node Cassandra clusters [34], Apple deploys a total of 100,000 Cassandra nodes [2], Yahoo! revealed the largest Hadoop/HDFS cluster with 4500 nodes [35], and Cloudera's customers deploy Spark on 1000 nodes [24, 27].

Is scale a friend or a foe [68]? On the positive side, scale surpasses the limit of a single machine in meeting increasing demands of compute and storage. On the negative side, this new era of "cloud-scale" storage systems has given birth to a new class of bug, *scalability bugs*, as defined in Figure 1.

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

Second, the common practice of debugging scalability bugs 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. As it stands today, many developers are heavily reliant on test clusters operated by large companies to do scale testing and only accessible to expert developers [26].

These realities raise the following question: how to discover latent scalability bugs and democratize large-scale testing? To this end, we introduce SCALECHECK, a concept that emphasizes the need to scale-check distributed system implementations at real scales, but do so cheaply on just one machine, hence empowering more developers to perform large-scale testing and debugging.

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

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

To show the generality and effectiveness of SCALECHECK, we have integrated SCALECHECK to a variety of large-scale storage systems, Cassandra [58], HDFS [18], Riak [30], and Voldemort [29], across a total of 15 earlier and newer releases. We scale-checked a total of 18 protocols (bootstrap, rebalance, add/decommission nodes, etc.), reproduced 10 known bugs and discovered 4 unknown critical scalability bugs (in Cassandra and HDFS). By only modifying the target systems in 179 to 918 LOC (and with a generic STEST library), we can colocate up to 512 nodes on a 16-core 32-GB commodity PC with high result accuracy (*i.e.*, observe a similar behavior as in the real-scale deployment).

SCALECHECK is unique compared to related work. For example, scalability simulation [39, 57] only checks models, but SCALECHECK checks implementation code. Extrapolation from "mini clusters" [57, 75, 80] does not work if the bug symptoms do not surface in small deployments, but SCALECHECK checks at real scales. Finally, emulation "tricks" run implementation code at real scale but in a smaller emulated environment [10, 48, 78] (the same category SCALECHECK can be put in), however existing techniques have limitations such as not addressing CPU contention and not finding potential causes automatically (more in §7). We also acknowledge many other works in improving storage scalability [42, 70], while our work emphasizes on scalability faults.

In summary, scalability bugs are new-generation bugs to combat in modern cloud-scale storage. Finding them without dependence of large clusters is a new research area to explore. In fact, this problem was discussed in a recent large meeting of Hadoop committee [26]. Currently, many new features in the alpha releases of Hadoop/HDFS still "sit on the shelf," *i.e.*, it is hard to test alpha (or even beta) releases at real scales as large production systems are not always accessible for testing. Some new features are still pushed and deployed but without much confidence. With this unideal reality, the committee agrees on the need for this new research, that it will increase their confidence on new releases [26]. Some companies began to invest in building scale-testing frameworks. For example, LinkedIn just released their scale-testing framework this year [9, 10] but it only emulates storage space specifically for HDFS.

For interested readers, we provide a supplemental file [1]. In the following sections, we present an extended motivation (§2), SCALECHECK design, application and implementation, and evaluation (§3-5) discussion, related work, and conclusion (§6-8).

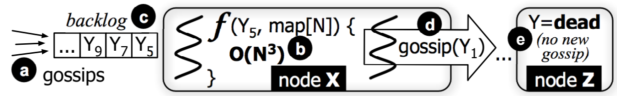



Figure 2: **An example bug (Section 2a).** *(a) Every second every node gossips to its peers its ring view and version number (e.g., $Y$ gossiped up to version $Y_9$), (b) the receiving node (e.g., $X$) executes "$f()$" to synchronize the view, (c) when $N$ is large, this $O(N^3)$ scale-dependent process creates a backlog of new gossips, (d) thus $X$ keeps gossiping only the latest (old) versions (e.g., $Y_1$), (e) as $Y$'s recent gossips are not propagated on time, other nodes (e.g., $Z$) mark $Y$ as dead.*

## 2 Scalability Bugs

Scalability bugs are not a well-understood problem. To the best of our knowledge, we provide the first in-depth look at scalability bugs in scale-out systems.

**(a) What is an example of scalability bugs?** In Cassandra issue #c6127 in Figure 2 [7], the bug surfaced when bootstrapping a large cluster. Here, every node receives gossips from peer nodes (with their ring views), then find any difference to synchronize their views of the ring. The *root cause* is that during bootstrapping with many view changes, the gossip processing is *scale-dependent*, $O(N^3)$, as it iterates through the node's and peer's ring data structures and uses a list-copy mechanism. When $N$ is large, this CPU-intensive process creates a backlog of new gossips, hence many nodes are inadvertently declared dead (and then alive after the gossips arrive). This repeating process leads to a *cluster instability* with thousands of "flappings" as $N$ grows; a "*flap*" is when a node marks a peer as down and alive again. More detailed examples are presented in §5.1.

**(b) Do they exist in many scalable systems?** We have collected a total of 55 bugs in many modern distributed systems (13 in Cassandra, 5 in Couchbase, 6 in Hadoop, 13 in HBase, 16 in HDFS, 1 in Riak, and 1 in Voldemort). This is an arduous process due to the lack of searchable keywords for "scalability bugs"; we might have missed some other bugs. We post the full list in Section 2 of [1]. All the bugs were reported from large deployments (100-1900 nodes). We emphasize again that all these bugs can *only* be reproduced at scale.

**(c) What are the root causes?** We study the buggy code, patches, and developer discussions and find that the majority (52) of the bugs are caused by *scale-dependent loops*, which iterate scale-dependent data structures (*e.g.*, list of nodes); the rest is about logic bugs that can be caught with single-function testing. We break them down to three categories: (1) CPU-intensive loops (15 bugs); Figure 2 shows an example. (2) Disk IO loops (26 bugs); the pattern is similar to Figure 2 but the nested-loops contain disk IOs. (3) Locking-related loops (11 bugs); they can be in the form of locks inside the loops or vice versa. These patterns suggest that this problem lends itself to program analysis (§3.1).

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

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

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

**(g) Why were the bugs not found before?** First, the workloads and the necessary scales to cover the buggy protocols are not captured in the unittests as creating a scalable test platform is not straightforward [26]. Second, protocols might be scalable in design, but not in practice. Related to c6127 (Figure 2), the failure detector/gossiper [50] was adopted for its "scalable" design [58]. However, the design does not account for the gossip processing time during bootstrap/cluster-changes, which can be long, and the subsequent backlogs. To debug, the developers tried to "do the [simple] math" but failed [7]. Specific implementation choices such as overloading gossips with many other purposes (*e.g.*, announcing boot/rebalance changes) deviate from the original design sketch, hence the need for scale-testing the implementation code at real scales.

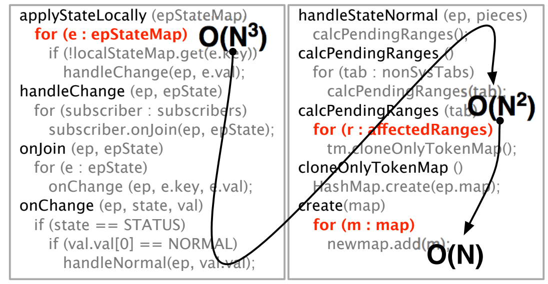



Figure 3: $O(N^3)$ **scale-depended loops (§3.1).** *The partial code segment above depicts the* $O(N^3)$ *loops in Figure 2.* SFIND *automatically tags* `epStateMap`*,* `affected- Ranges`*, and* `map` *as scale-dependent collections.*

**(h) Are scalability bugs easy to debug and fix?** The bugs took 1 month to fix on average with tens of back-and-forth discussions. One big factor of delayed fixes is the lack of budget for large test clusters as such luxury tends to only be accessible in large companies, but not to open-source developers [26]. Another factor is that debugging and fixing are not a single-iteration task; developers must repeatedly instrument the system and re-run at scale to pinpoint the root cause and test the patch.

# 3 SCALECHECK

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

## 3.1 SFIND

The first challenge to address is: how to find scale-dependent loops? Unfortunately, it is not trivial as such loops can span multiple functions and iterate many scale-dependent collections (iterable data-structure instances such as `list`). In Figure 3, the $O(N^3)$ loops span 1000+ LOC, 3 classes, and 10 functions and iterate 3 scale-dependent collections. This difficulty motivates SFIND, a generic program analysis that helps developers pinpoint scale-dependent loops. Below are the three main steps of SFIND. For space, the pseudo-code can be found in our supplement, Section 3.1 of [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. After each step, we record the size of each instantiated collection. When all the steps are done, we check each collection's growth tendency and mark as scale dependent those whose size increases as the cluster/data size grows.

This, however, is insufficient due to two reasons. First, there are collections that only grow when background/operational tasks are triggered (§2d); thus, we must also run all non-foreground tasks. Second, there are "ephemeral" collections (*e.g.*, messages) whose content are scale-dependent but might have been garbage collected by the runtime. Given that the measurements are taken in steps, garbage collection can happen in between them so these collections will not be detected consistently, thus this phase must be iterated multiple times to remove such noise.

For Java systems, we track heap objects and map them to their instance names by writing around 1042 LOC of analysis on top of Java language supports such as JVMTI [67] and Reflection [22]. This phase also performs a dataflow analysis to taint all other variables derived from scale-dependent collections. In our experience, by scaling out to just 30 nodes (30 steps), which can be done easily on one machine, scale-dependent collections can be clearly observed (though not the symptoms). This phase found 32 scale-dependent collections in Cassandra (three in Figure 3) and 12 in HDFS.

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

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

**(3) Reporting and triaging:** SFIND finds 131 scale-dependent loops in Cassandra and 92 in HDFS, hence the need for triaging. For example, if a function $g$ has lower complexity than $f$, and $g$ is within the call path of $f$, then testing $f$ can be prioritized. For every nested loop to test, SFIND reports the relevant control- and data-flows from the outer-most to inner-most loop, along with the entry points (either client/admin RPCs or background daemon threads). The entry points are finally ranked by counting the number of spanned scale-dependent lines of code, the theoretical complexity (in terms of scale-dependent data structures), the number of IO operations (including reads/writes) and the number of blocking operations (including locking and operations that block waiting for a future result) in that path. The theoretical complexity is not by itself a complete indicator of potential bottlenecks. For example, an entry point reported with high complexity ,*e.g.* $O(N^3)$, but with no IO/Blocking operations on its code path might not be as bottleneck prone as one reported with less complexity, *e.g.* $O(N)$, but many IO/Blocking operations on its code path. This ranking helps developers prioritize and create the necessary test workloads. For example, in Figure 3, 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 [38], 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 [62] 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 1 using Cassandra as an example. Abbreviations of our methods (*e.g.*, NP, SPC, GEDA) are added for ease of reference in the evaluation.

### 3.2.1 Black-Box Approaches

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

*(a) Memory bottlenecks:* Many distributed systems today are implemented in managed languages (*e.g.*, Java, Erlang) whose runtimes consume non-negligible memory overhead. Java and Erlang VMs, for example, use around 70 and 64

| | #Nodes per PC | LOC added | Colocation bottlenecks |
|---|---|---|---|
| *Black/gray-box approaches (§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 (§3.2.2)* | | | |
| (d) GEDA | 130 | +581 | CPU |
| (e) GEDA+PIL | 512 | +246 | CPU |

Table 1: **Colocation strategies and bottlenecks (§3.2).**

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

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

*(c) Managed-language VM limitations:* We also found that managed-language VMs are backed by advanced services. For example, Erlang VMM contains a DNS service that sends heartbeat messages among connected VMs. When hundreds of Erlang VMs (one for each Riak node) run on one Erlang VMM, the heartbeat messages cause a "network" overflow that undesirably disconnects Erlang VMs (also reported in [40]). Naive packing is infeasible.

• **Single-Process Cluster (SPC) + Network Stub:** To address the bottlenecks above, we deployed all nodes as threads in a single process. Surprisingly, our target systems are not easy to run in this "single-process cluster." For example, Cassandra developers bemoan the fact that their gossip/failure-detector protocols are not adequately scale-tested [15, 28] because Cassandra (and many other systems) uses "singleton" design pattern for simplicity (but bad for modularity) [32]. 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 [15, 28]. We found another way: leveraging class loader isolation support from the language runtime [23], which is rarely used but fits SPC purpose. In

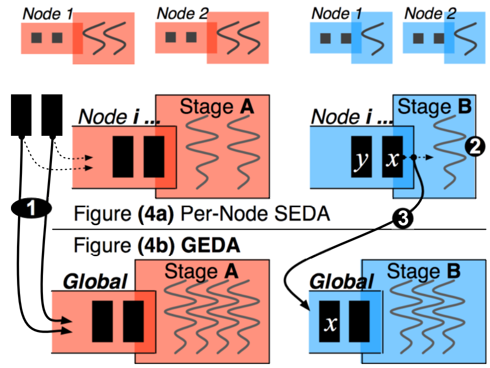



Figure 4: **Global Event Driven Arch. (Section 3.2.2).** *The figure format follows [79, Figure 6].*

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. Very recently, we found that Cassandra developers also begin to develop a similar method to address this problem [8].

By SPC-ing Cassandra, we now hit a colocation limit of 70 nodes (Table 1b), but still have not reached the memory or CPU bottlenecks. We suspected thread and/or user-kernel context switching as a root cause. We removed the latter by creating a generic network stub that (de)marshalls inter-node messages and skips the OS. This stub is also helpful in reducing network memory footprints under higher colocation. For example, in Voldemort, the nodes communicate via Java NIO [25] which is fast but contains buffers and connection metadata that take up memory space and prevent >200-node colocation (more in §5.4). For Cassandra, the network stub allows up to 120-node colocation (Table 1c).

#### 3.2.2 A White-Box Approach

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 [21], which would reduce the problem to $N$ threads. However, we still observed around $N\times 26$ active threads – each node still runs multiple service stages (gossiper, failure detector, etc.), each can be multi-threaded. A high colocation factor will spawn thousands of threads.

• **Global Event Driven Arch. (GEDA):** To address the problem, we must redesign the target system, but with minimal changes. We leverage the staged event-driven architecture (SEDA) [79] (Figure 4a), 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

4b). 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 4), 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 [3]). 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 4 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 4), which costs ~50 LOC change per stage (details in §3.2 of [1]). Thus, by default we interpose at enqueue (small changes) and at dequeue for single-threaded stage (more changes).

Adding GEDA to Cassandra only costs us 581 LOC (Table 1d) and is simple; the same 10-50 LOC method above is simply repeated across all the stages. Overall, GEDA does not change the logic of the target systems, but successfully removes some delays that should have never existed in the first place, as if the nodes run exclusively on independent machines. For HDFS tests, GEDA enables 512-node colocation (§5.4) but for some Cassandra tests, it only enables around 130-node colocation (Table 1d), which we elaborate in the next section.

## 3.3 Processing Illusion (PIL)

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

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

To address this, we need to emulate CPU-intensive processing by supplementing STEST with *processing illusion* (PIL), an approach that replaces an actual processing with `sleep()`. For example, for c6127, we can replace the expensive gossip/stage-changes processing (see Figures 2 and 3), 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" [78]. Similarly, PIL provides an illusion of compute processing; our insight is that *"the key to computation is not the intermediate results, but rather the execution time and eventual output."* In other words, with PIL, we will still observe the overall timing behaviors and the corresponding impacts accurately.

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

### 3.3.1 PIL-Safe Functions

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

We extend SFIND to 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 [1, §3.3]. We consider all variables involved in the execution of a target protocol as relevant states. While our static analysis tool eases the identification of these variables, programmer intervention can help for additional verification. In (2), the loop performs non-pertinent operations only (such as IO). In this case, we can automatically replace the loop with a *sleep* call without affecting the behavior of the protocol.

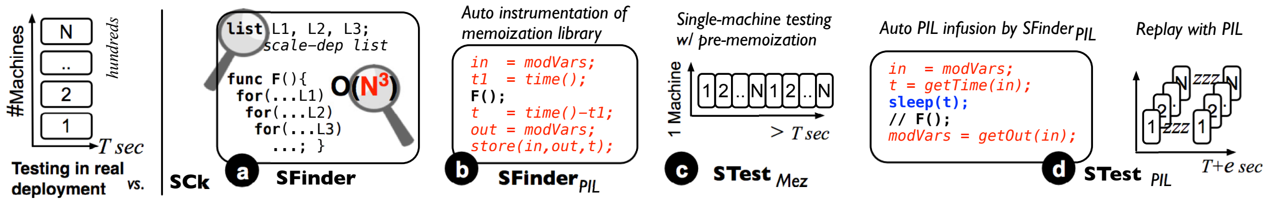



Figure 5: **SCALECHECK complete automated flow (Section 3.4).** *"SCk" represents* SCALECHECK. *The left-most figure illustrates testing in real deployments, where testing time is fast ($T$) but requires $N$ machines. Stages (a) to (d) reflect the automated* SCALECHECK *process as described in Section 3.4.* STEST$_{mez}$ *in stage (c) runs on one machine but will take some time ($>T$).* STEST$_{PIL}$ *in stage (d) still runs on one machine but only consumes a similar time as in deployment testing ($T+e$) and can be replayed numerous times.*

#### 3.3.2 Pre-Memoization (with Determinism)

As PIL-safe functions no longer perform the actual computation, the next question to address is: how do we manufacture the output such that the global behavior is not altered (*e.g.*, rebalancing protocol should terminate successfully)?. For functions with no pertinent outputs, we just need to do time profiling but not output recording. For functions with pertinent outputs, our solution is *pre-memoization*, which records input-output pairs and the processing time, specifically a tuple of three items (`ByteString in`, `out`, `long nanoSec`) indexed by `hash(in)`), which represent the to-be-modified variables before and after the function is executed and the processing time, respectively (Figure 5b).

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

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

Other than this, similar to the theme in the previous section that existing systems are not amenable to single-machine testing, we found similar issues such as the use of wall-clock time which essentially incapacitates memoization and replay. Here, we convert wall-clock time to "cluster start time + elapse time" in 296 LOC (Table 1e).

### 3.4 Putting It All Together

Figure 5a-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$_{PIL}$ 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 5c), up to 6x longer time than real-deployment testing (§5.5). However, this is only a one-time overhead. Most importantly, developers can repeatedly re-run STEST$_{PIL}$ (Figure 5d) as many times as needed (tens of iterations) until the bug 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 $\Phi$ algorithm (for c6127), caching slow methods (c3831), changing lock management (c5456), and enabling parallel processing (v1212). However, if the fixes involve a complete redesign (*e.g.*, optimized gossip processing in c3881, decentralized to centralized rebalancing in r3926), STEST$_{mez}$ must be repeated.

| | Cass | HDFS | Riak | Vold |
|---|---|---|---|---|
| STEST-able systems | 918 | 179 | 217 | 800 |
| SFIND code | 4026 (generic) | | | |
| STEST library | 6047 (generic) | | | |

Table 2: **Integrations LOC (Section 4).** *More explanations are in Section 4 of [1]. We will release our code publicly.*

# 4 Application and Implementation

Table 2 quantifies the application of SCALECHECK techniques to a variety of distributed systems, Cassandra [58], HDFS [18], Riak [30], and Voldemort [29]. The major system-specific change is achieving "STEST-able systems" (*i.e.*, supporting SPC and GEDA), which range between 179 to 918 LOC (less than 1 % of the target code size). This is analogous to how file systems code are modified to make them "friendlier" to `fsck` [52, 63]. The rest is the generic SFIND and STEST library code (pre-memoization, auto PIL insertion, message order determinism support, AspectJ utilities). SFIND was built with Eclipse AST Parser [11] to support Java programs. We leave porting to Erlang's parser [12, 13] as future work.

**Generality:** We show the generality of SCALECHECK with two major efforts. First, we scale-checked a total of 18 protocols: 8 Cassandra (*e.g.*, bootstrap, scale-out, decommission), 8 HDFS (*e.g.*, decommission, block reports, snapshot), 1 Riak (rebalance), and 1 Voldemort (rebalancing) protocols (full list in §4 of [1]). A protocol can be built on top of other protocols (*e.g.*, bootstrap on gossip and failure detection protocols). Second, for exposing known bugs, we applied SCALECHECK to a total of 10 earlier releases: 4 Cassandra, 4 HDFS, 1 Riak, and 1 Voldemort old releases. For finding unknown bugs, we also ran SCALECHECK on recent releases of the four systems.

# 5 Evaluation

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

## 5.1 Exposing Scalability Bugs

Table 3 lists the 10 real-world bugs we use for benchmarking SCALECHECK. We chose these 10 bugs (among the 55 bugs we studied) because the reports contain detailed descriptions of the bugs, which is important for us to create the "input" (*i.e.*, the test cases). Figure 6 shows the accuracy

| Bug# | N | Protocol | Metric | $T_m$ | $T_{pil}$ |
|---|---|---|---|---|---|
| c6127 [7] | ≥256 | Bootstrap | *#flaps* | 2h | 15m |
| c3831 [6] | ≥256 | Decomm. | *#flaps* | 17m | 9m |
| c3881 [5] | ≥64 | Add nodes | *#flaps* | 7m | 5m |
| c5456 [4] | ≥256 | Add nodes | *#flaps* | 16m | 4m |
| r3926 [31] | ≥128 | Rebalance | $T_{Comp}$ | 6h | 2h |
| v1212 [33] | ≥128 | Rebalance | $T_{Comp}$ | 22h | – |
| h9198 [19] | ≥256 | Blk. report | $Q_{Size}$ | 8m | – |
| h4061 [17] | ≥256 | Decomm. | $T_{Lock}$ | 6h | – |
| h1073 [16] | ≥512 | Pick nodes | $T_{Comp}$ | 1m | – |
| h395 [20] | ≥512 | Blk. report | $T_{Comp}$ | 5m | – |

Table 3: **Bug benchmark (§5.1).** *The table lists the scalability bugs we use for benchmarking SCALECHECK. "**c**" stands for Cassandra, "**h**" for HDFS, "**r**" for Riak, and "**v**" for Voldemort. The "**N**" column represents the #nodes for the bug symptoms to surface. The "**Metric**" column lists the quantifiable metrics of the bug symptoms; $T_{Comp}$, $T_{Lock}$, and $Q_{Size}$ denote computation time, lock time, and queue size, respectively. The "$T_m$" and "$T_{pil}$" columns quantify the duration of the pre-memoization (STEST$_{mez}$) and PIL replay (STEST$_{PIL}$) stages when $N{\geq}256$, as discussed in §5.5. "–" implies PIL is unnecessary.*

of SCALECHECK in exposing the 10 bugs using the "bug-symptom" metrics in Table 3 (the first bug c6127 will be shown later in Section 5.3 and the last bug h395 is omitted in Figure 6 for space).

**Results summary:** First, SCALECHECK is effective and accurate in exposing scalability bugs, some of which only surface in 256+ nodes. As shown, for Cassandra and Riak bugs where all nodes are CPU intensive, PIL is needed for accuracy (`SCk+PIL` vs. `Real` lines in Figures 6a-d), but for the rest, STEST suffices (`SCk` vs. `Real` in 6e-f).

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

Third, different systems of the same type (*e.g.*, key-value stores, master-worker file systems) implement similar protocols. The effectiveness of SCALECHECK methods in scale-checking the different protocols above can be useful to many other distributed systems.

**Bug descriptions:** We now briefly describe the bugs. Longer descriptions can be found in Section 5.1 of [1].

**(a)** Figure 6a: In Cassandra c3831 [6] when a node X is removed, all other nodes must own X's key-partitions. This scale-dependent, CPU-intensive "pending keyrange calculation" cause cluster-wide flapping (the y-axis), observable in 256+ nodes. The fix caches the outputs of slow methods.

**(b)** Figure 6b: c3881 [5] is similar to the previous bug (c3831), but the fix was obsolete as the concept of multi-

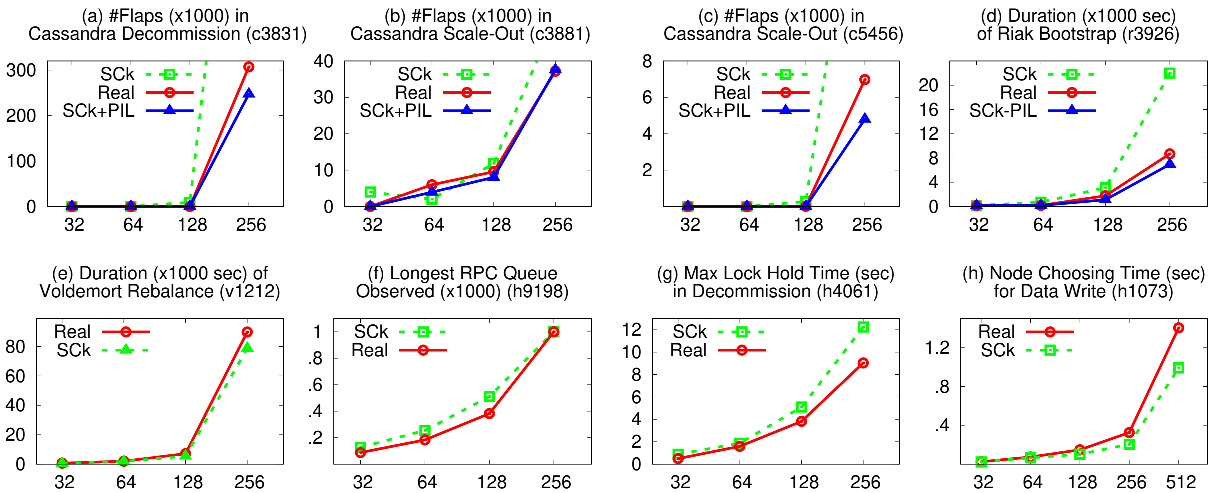



Figure 6: **SCALECHECK effectiveness in exposing scalability bugs (Section 5.1).** *"SCk" represents* SCALECHECK. *The bugs are listed in Table 3. The **x-axis** represents the number of nodes ($N$). The figure title describes the **y-axis**, i.e., the bug symptom metrics as recorded in "Real" deployment vs.* SCALECHECK. *For Cassandra and Riak bugs (a-d), where all nodes are CPU-intensive, the bug symptoms are inaccurate without PIL ("*`SCk`*" lines). However, with PIL ("*`SCk+PIL`*" lines), the bug symptoms are relatively accurate as in the real deployment scenarios. For Voldemort and HDFS bugs (e-h), where there is no concurrent CPU busyness, PIL is not needed.*

ple key-partitions per node was added. The calculation is now scale-dependent on $N \times P$. This causes CPU spikes and massive flapping during scaling out; the bug surfaced in 64+ nodes (when 32+ new nodes are added to existing 32+ nodes). The bug was fixed with a complete redesign of the pending keyrange calculation.

**(c)** Figure 6c: Interestingly, c5456 [4] is a bug in the *same* protocol as above. The previous fix was obsolete again as pending range calculation is now multi-threaded; range calculations can happen concurrently. However, this new design introduces a new coarse-grained lock that can block gossip processing for a long time, thus introduces flapping (in 256+ nodes). The fix changed the lock management.

**(d)** Figure 6d: In r3926 [31], 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).

**(e)** Figure 6e: In v1212 [33], Voldemort's rebalancing was not optimized for large clusters; it led to more stealer-donor partition transitions as the cluster size grows (128+ nodes). The fix changed the stealer-donor transition algorithm.

**(f)** Figure 6f: In h9198 [19], incremental block reports (IBRs) from HDFS datanodes to the namenode acquire the global master lock (*i.e.*, a special worker-to-master "loop" as explained in §3.1). 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 ($\times$1000). When this happens, user requests are undesirably dropped by the namenode. The fix batches the IBR request processing. In HDFS, to emulate large blocks, we reuse the "`TinyDataNode`" class (1KB blocks) that the developers already use in the unit tests.

**(g)** Figure 6g: In h4061 [17], when $D$ datanodes are decommissioned, the blocks must be replicated to the other $N - D$ nodes. Every 5 minutes, the `DecommissionMonitor` thread in the namenode iterates all the block descriptors to check if the $D$ nodes can be safely decommissioned (when all data replications complete). This thread, unfortunately, must hold the global file system lock. When $N$ is 256+, this process can hold the lock (*i.e.*, stall user requests) for more than 10 seconds ($y$>10). The fix used a dedicated thread to manage decommissioning and refined the algorithm.

**(h)** Figure 6h: In h1073 [16], for a new file creation, the namenode calls a `chooseTarget` function to sort a list of target datanodes from their distances from the writer and choose the best nodes. When $N$ and the replication factor are large, it can take more than one second to choose. The fix modified the sorting algorithm.

**(i)** Finally, in h395 [20] (figure not shown for space), datanodes send block reports too frequently and when $N$>512 nodes, the namenode spends more time in this background process as opposed to serving users.

## 5.2 Discovering Unknown Bugs

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

For Cassandra, SFIND pointed us to another nested scale-dependent loop. We created the corresponding test case and SCALECHECK showed that cluster-wide flapping resurfaces again but only in 512-node deployment. As an example, decommissioning just only one node already caused almost 100,000 flaps. The developers confirmed that the bug is related to a design problem. To prevent flappings, the devel-

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 7: **Cassandra internal metrics (§5.3).** *Above are the metrics we measured within the Cassandra bootstrap protocol for measuring* SCALECHECK *accuracy (Figure 8). "f" represents "a function of" (i.e., an arbitrary function).*

opers suggested us to add/remove node one at a time with 2-minute separation, which means scaling-out/down 100 nodes will take over 3 hours (*i.e.*, this bug impedes instant elasticity). The developers recently started a new initiative for designing "Gossip 2.0" to scale to 1000+ nodes [14].

For Riak and Voldemort, we found that their latest-stable bootstrap/rebalance protocols do not exhibit any scalability bug, up to 512 nodes.

For HDFS, we found 3 instances of scale-dependent loops that hold the entire namenode read/write lock (also confirmed by the developers). Specifically, SFIND reports the following number of lines executed:

```
FSNamesystem.getSnapshotDiff        N*(85*B+17)
DatanodeManager.refreshDatanodes    N*(136*B+137)
FSNamesystem.metaSave               N*(50*B+21)
```

Here, "B" represents the number of blocks per datanode (*e.g.*, 10,000). The first function, `getSnapshotDiff`, contains a bug that the HDFS developers were hunting for 4 weeks, as the unresponsive-namenode impact recently affected a customer. In this path, there is a recursive function iterating on a list of files and blocks and a conditional path that makes ACL lookups which causes the namenode to be unresponsive for more than 40 seconds in at least a 512-node deployment. Similar symptoms were also reproduced for the second and third bugs (`refreshDatanodes` and `metaSave`). The developers say these bugs are dangerous because if the namenode is paused for 45 seconds, it will cause a heavy failover. They also say these bugs are hard to find in a million-plus lines of code. More details/graphs are in §5.2 of [1].

## 5.3 Accuracy

The goal of our next evaluation is to show that PIL-infused SCALECHECK mimics similar behaviors as in real-deployment testing and is accurate not only in the final bug-symptom metric but also in the detailed internal metrics. For this, we collected roughly 18 million values. For space, we only focus on c6127 [7] (see §2a).

Figure 7a-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 8a-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").

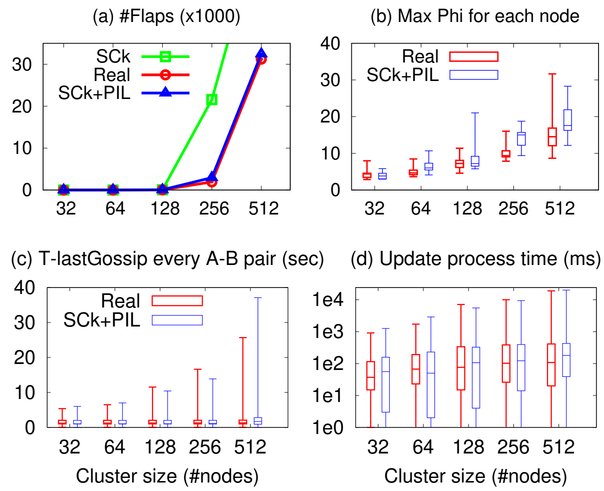



Figure 8: **Accuracy in exposing c6127 (§5.3).** *The figures represent the metrics presented in Figure 7, measured in real deployment ("Real") and in* SCALECHECK *("SCk") with different cluster sizes (32, 64, 128, 256, and 512 in the* ***x-axis****). The* ***y-axes*** *(the metrics) are described in the figure titles.*

**(a)** Figure 8a 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 (§3.3). However, with PIL, SCALECHECK closely mimics real deployment scenarios. Next, Figure 7a defines that *#flaps* depends on Φ [50]. Every node A maintains a Φ for a peer B (a total of $N \times (N-1)$ variables to monitor).

**(b)** Figure 8b 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 7b 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 $T_{avgGossip}$.

**(c)** Figure 8c shows the whisker plots of gossip inter-arrival times ($T_{lastGossip}$) that we collected for every A-B pair (millions of gossips as a gossip message contains $N$ gossips of the peer nodes). The figure shows that in larger clusters, new gossips do not arrive as fast as in smaller clusters, especially at high percentiles. Figure 7c shows that $T_{lastGossip}$ depends on how far B's new gossips propagate through other nodes to A (*#hops*) and the gossip processing time in each hop ($T_{gossipExec}$). The latter ($T_{gossipExec}$) is essentially the state-update processing time ($T_{stateUpdate}$), triggered whenever there are state changes.

**(d)** Figure 8d (*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 7d, $T_{stateUpdate}$ complicatedly depends on a scale-dependent 2-dimensional input ($Size_{ringTable}$ and $Size_{newStates}$). A node's $Size_{ringTable}$ depends on how many nodes it knows, including the partition arrangement ($\leq N \times P$) and $Size_{newStates}$ ($\leq N$), which increases as cluster size grows.

## 5.4 Colocation Factor

This section shows the maximum colocation factor ScaleCheck can achieve as each technique is added one at a time on top of the other. To recap, the techniques are: single-process cluster (SPC), network stub (Stub), global event driven architecture (GEDA), and processing illusion (PIL). The results are based on a 16-core machine.[1]

**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 §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 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 9a, 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 9a), our integration sequence is +SPC, +Stub, +GEDA, and +PIL (as we hit context switching overhead before CPU). For Riak (Figure 9b), we began with PIL as we hit CPU limitation first before hitting Erlang VMM network overflow which requires SPC (§3.2.1), and Riak does not require GEDA because Erlang, as an event-driven language, manages thread executions as events (more in Section 5.4 of [1]). For Voldemort (Figure 9c), 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 9d), 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 9a 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.

[1] So far, we consistently use the same testbed, but a higher-end machine can be used in the future.

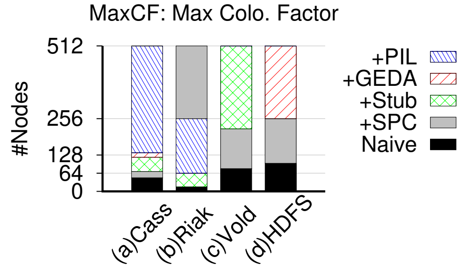



Figure 9: **Maximum colocation factor (Section 5.4).** *The colocation factor reached as each technique is added.*

## 5.5 Pre-Memoization and Replay Time

The "$T_m$" and "$T_{pil}$" columns in Table 3 on page 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 bugs such as c6127, the pre-memoization time takes 2 hours while the PIL-based replay is only 15 minutes (similar to the real-deployment test); for r3926, it is 6 vs. 2 hours. Pre-memoization does not necessarily take $N\times$ longer time because one node only consumes 2 cores (while the machine has 16 cores) and also not every node is busy all the time.

## 5.6 Test Coverage

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

We emphasize that SFIND is *not* a bug-finding tool, hence the reason why we do not report false positives. A more complete picture of SFIND's output can be found in Section 5.6 of our supplemental document [1].

# 6 Discussion

At the moment, our work focuses on scale-dependent CPU/processing time (§2c), and the "scale" here implies the scale of *cluster size*. However, there are other scaling problems that lead to IO and memory contentions [46, 69, 76], usually caused by the scale of *load* [37, 47] or *data size* [64]. For emulating data size, we are only aware of one work, Exalt [78], which is orthogonal to SCALECHECK (more in §7). In our bug study, we learn that some load or data-size related bugs can be addressed with accurate modeling [47] (*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 [76]). Nevertheless, we will continue our study of these other scaling dimensions, especially as scaling bugs in datacenter distributed systems is not a well-understood problem.

So far, SCALECHECK is limited as a single-machine framework, which integrates well to the de-facto unit-test style. To increase colocation factor, a higher-end machine can be used. Another approach is to extend SCALECHECK to run on multiple machines. However, this means that we need to enable back the networking library, which originally already caused a colocation bottleneck. We also acknowledge as a limitation that adding new code will also add new maintenance costs. In future work, we intend to approach zero-effort emulation.

Finally, SFIND by itself is not sufficient to reveal scalability bugs. Building a program analysis that covers all paths and understands the cascading impacts is challenging. Not all scale-dependent loops imply buggy code.

# 7 Related Work

In Section 1, we briefly discussed related work in four categories: real-scale testing/benchmarking (direct, but not economical) [26, 59], large-scale simulation (easy to run, but rarely used for server infrastructure code) [39, 54, 57], extrapolation (easy to run, but missing bugs in small training scale) [57, 61, 75, 80], and emulation. SCALECHECK falls in this category and below discuss three closely related works [10, 48, 78].

Exalt [78] targets IO-intensive (Big Data) scalability problems where storage capacity is the colocation bottleneck. Exalt's library (Tardis) compresses users' data to zero bytes on disk. With this, Exalt can co-locate 100 space-emulated HDFS datanodes per machine. As the authors stated, their approach "may not discover scalability problems that arise at the nodes that are being emulated" [78]. Thus, it cannot cover P2P systems where the scale-dependent code is in all the nodes. However, as Exalt targets storage space emulation and SCALECHECK addresses processing time emulation, we believe they complement each other. LinkedIn's Dynamometer is similar to Exalt [10].

DieCast [48], invented for network emulation, can colocate processes/VMs on a single machine as if they run individually, by "dilating" time. The trick is adding a "time dilation factor" (TDF) support [49] into the VMM. For example, TDF=5 implies that for every second of wall-clock time, each emulated VM believes that time has advanced by only 200 ms (1/TDS second). DieCast was only evaluated with a colocation factor (TDF) of 10 as the testing time significantly increases proportionally to the TDF; colocating 500 nodes will increase testing time by 500 times. DieCast was introduced for answering "what if the network is much faster?", but not specifically for single-machine scale-testing. Another significant difference is that both Exalt and DieCast papers do not present an in-depth bug study.

In terms of related work in the static/program analysis space, Clarity [66] and Speed [45] use static analysis to look for potential performance bottlenecks by focusing on redundant traversals and precise complexity bounding. Both approaches are evaluated in libraries. However, for distributed systems, real-scale testing can help reveal unintended complex component interactions, and not all scale-dependent loops cause problems.

Finally, a recent work also highlights the urgency of combating scalability bugs [60]. The work, however, does not employ methodical and incremental changes, only suggests a manual approach, and reproduces only 4 bugs in 1 system.

# 8 Conclusion

Technical leaders of a large cloud provider emphasized that "the most critical problems today is how to improve testing coverage so that bugs can be uncovered during testing and not in production" [43]. It is now evident that scalability bugs are new-generation bugs to combat, that existing large-scale testing is arduous, expensive, and slow, and that today's distributed systems are not single-machine scale-testable. Our work addresses these contemporary issues and will hopefully spur more solutions in this new area.

# 9 Acknowledgments

We thank Cheng Huang, our shepherd, and the anonymous reviewers for their tremendous feedback and comments. This material was supported by funding from NSF (grant Nos. CNS-1350499, CNS-1526304, CNS-1405959, and CNS-1563956) as well as generous donations from Dell EMC,Google, Huawei, and NetApp, and CERES center.

## References


[1] Anonymized document for ScaleCheck supplementary materials (also submitted to HotCRP), for interested reviewers. https://tinyurl.com/sck-supp-mat.

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

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

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

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

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

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

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

[9] Dynamometer Github Repository. https://github.com/linkedin/dynamometer.

[10] Dynamometer: Scale Testing HDFS on Minimal Hardware with Maximum Fidelity. https://engineering.linkedin.com/blog/2018/02/dynamometer--scale-testing-hdfs-on-minimal-hardware-with-maximum.

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

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

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

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

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

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

[17] Hadoop bug: Large number of decommission freezes the Namenode. https://issues.apache.org/jira/browse/HADOOP-4061.

[18] HDFS. https://hortonworks.com/apache/hdfs/.

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

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

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

[22] Java Reflection API. https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/index.html.

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

[24] Meet Cloudera's Apache Spark Committers. http://blog.cloudera.com/blog/2015/09/meet-clouderas-apache-spark-committers/.

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

[26] 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.

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

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

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

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

[31] Riak bug: Large ring_creation_size. http://lists.basho.com/pipermail/riak-users_lists.basho.com/2011-April/003895.html.

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

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

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

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

[36] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST)*, 2008.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[42] James Cipar, Gregory R. Ganger, Kimberly Keeton, Charles B. Morrey III, Craig A. N. Soules, and Alistair C. Veitch. LazyBase: trading freshness for performance in a scalable database. In *Proceedings of the 2012 EuroSys Conference (EuroSys)*, 2012.

[43] 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.

[44] Daniel Ford, Franis Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlna. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

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

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

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

[52] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *IEEE 2nd Workshop on Hot Topics in System Dependability (HotDep)*, 2006.

[53] Bernard Dickens III, Haryadi S. Gunawi, Ariel J. Feldman, and Henry Hoffmann. StrongBox: Confidentiality, Integrity, and Performance using Stream Ciphers for Full Drive Encryption. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[54] Håvard D. Johansen, Robbert Van Renesse, Ymir Vigfusson, and Dag Johansen. Fireflies: A secure and scalable membership and gossip service. *ACM Transactions on Computer Systems*, 33:5:1–5:32, June 2015.

[55] Kimberly Keeton, Cipriano A. Santos, Dirk Beyer, Jeffrey S. Chase, and John Wilkes. Designing for disasters. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST)*, 2004.

[56] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

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

[58] 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.

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

[60] 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 *The 16th Workshop on Hot Topics in Operating Systems (HotOS XVII)*, 2017.

[61] 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.

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

[63] 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.

[64] 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.

[65] Kazunori Ogata and Tamiya Onodera. Increasing the transparent page sharing in java. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[66] 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.

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

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

[69] 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.

[70] Swapnil Patil and Garth Gibson. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *Proceedings of the 9th USENIX Symposium on File and Storage Technologies (FAST)*, 2011.

[71] Jason K. Resch and James S. Plank. AONT-RS: Blending Security and Performance in Dispersed Storage Systems. In *Proceedings of the 9th USENIX Symposium on File and Storage Technologies (FAST)*, 2011.

[72] Mohit Saxena, Michael M. Swift, and Yiying Zhang. FlashTier: a Lightweight, Consistent and Durable Storage Cache. In *Proceedings of the 2012 EuroSys Conference (EuroSys)*, 2012.

[73] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding Latent Sector Errors and How to Protect Against Them. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST)*, 2010.

[74] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[75] 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.

[76] Konstantin V. Shvachko. HDFS Scalability: The Limits to Growth. *USENIX ;login:*, 35(2), April 2010.

[77] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-Based Service Level Agreements for Cloud Storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[78] 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.

[79] 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.

[80] 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.