PreFail: Programmable and Efficient Failure Testing Framework

Pallavi Joshi
Haryadi S. Gunawi
Koushik Sen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-3
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-3.html

January 18, 2011

Copyright © 2011, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

This material is based upon work supported by Computing Innovation Fellowship and the National Science Foundation under grant
Nos. CCF-1018729 and CCF-0747390. We also thank Eli Collins and Todd
Lipcon from Cloudera Inc. for helping us confirm the HDFS bugs that we found. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

PREFAIL: Programmable and Efficient Failure Testing Framework

Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen
University of California, Berkeley

Abstract

As hardware failures are no longer rare in the era of cloud computing, reliability has become a first-class design goal of today's cloud software systems. To ensure that software's fault-tolerance "prevails" against hardware failures, cloud systems have to be tested against multiple, diverse failures that are likely to occur in the real-world. Such failure testing poses several challenges including the need to explore a large number of combinations of failures, and also by implication, to debug a large number of test runs that fail during testing. In this paper, we present PREFAIL, *a programmable and efficient failure testing framework. With* PREFAIL, *a tester can express a variety of failure exploration policies, skip redundant fault-injection tests, run failure testing in parallel, and reduce the time to debug failed test runs.*

1 Introduction

With the arrival of the cloud computing era, large-scale distributed systems are increasingly in use. These systems are built out of tens of thousands of commodity machines that are not fully reliable and can fail from time to time. In fact, it has been observed in real deployments that one can expect to lose ten machines a day out of ten thousand machines [8]; not to mention that catastrophic failures could make many more machines unavailable at the same time. Thus, the software that runs on large-scale distributed systems has a great responsibility to correctly recover from frequent, diverse failures such as machine crashes, disk errors, and network failures.

Even if distributed systems are built with reliability and fault tolerance as primary goals [7, 8, 9], their recovery protocols are often buggy. For example, the developers of Hadoop File System [21] have dealt with 91 recovery issues over its four years of development [12]. There are many reasons for this. Sometimes developers fail to anticipate the kind of failures that a system can face in a real setting. They might have built the system to tolerate only fail-stop failures like crashes, but the system might face non fail-stop failures like data corruption. Even if the developers could anticipate all kinds of failures that the system could face, they still might be incorrect in the way in which they program the system to recover from those failures. There have been many serious consequences (*e.g.*, data loss, unavailability) of the presence of recovery bugs in real deployed systems [3, 4, 5, 12]. Therefore, thorough failure testing becomes essential to improve the reliability of distributed systems.

The state-of-the-art of failure testing of distributed systems, unfortunately, only injects sequences of failures in a random fashion [4, 13, 14, 22, 23]. Randomness is simple but can easily miss specific failure scenarios that can lead to corner-case bugs. Another approach to failure testing is to exhaustively explore all possible failure scenarios. However, the number of such scenarios is too many to explore with reasonable computing resources within a reasonable time [12, 16]. Thus, there is a need for more efficient failure testing techniques that can smartly explore the space of failure scenarios, and find bugs efficiently.

There has been some work that proposes novel techniques for smart exploration of failures [16, 18]. They primarily address single failures during program execution. However, large-scale distributed systems face frequent, multiple, and diverse failures. And thus, there is a need to advance the state-of-the-art of failure testing for large-scale distributed systems.

Due to the reasons mentioned above, in previous work [12], we began addressing the challenges of failure testing, specifically by building a failure testing framework that can explore failures systematically, including multiple failures. Due to its contributions, several companies are exploring the use of our technology [11], which we believe shows the importance of such a testing framework. Although it has some initial success, our previous work does not address many other challenges of failure testing; the framework has some fundamental design limitations such as rigid exploration techniques, lack of programmability, ad-hoc fault injections in some cases, non-parallelizable design, and the absence of triaging techniques that make it impractical for use by real testers. These limitations inform us about more "complete" features that a failure testing framework should be equipped with.

In this work, we introduce PREFAIL, a programmable and efficient failure testing framework. PREFAIL comes with the following features:

1. PREFAIL provides a way for testers to write policies in which they can express the kind of failure sequences (single failures or combinations of multiple failures) that they want to prioritize during testing. By writing appropriate policies, a tester can achieve different high-level objectives (*e.g.*, faster code coverage, faster recovery behavior coverage, etc.).

2. PREFAIL is equipped with well-defined fault-injection optimizations that completely remove redundant fault-injection tests.

3. PREFAIL is designed such that its test workflow is parallelizable. Thus, it can explore multiple failure sequences simultaneously, and hence achieve a considerable speed-up in its performance.

4. PREFAIL is equipped with configurable triaging heuristics that cluster failed experiments into different classes based on the bugs that caused them. Therefore, the tester can significantly reduce unnecessary debugging effort of every failed experiment, which can take hours or even days.

With these new contributions, PREFAIL is a novel, more complete, and practical failure testing framework that can help today's large-scale distributed systems "prevail" against failures. Currently, PREFAIL has been adopted by Cloudera Inc. to test their Hadoop software releases [6].

In the rest of the paper, we discuss related work (§2), briefly describe our approach to failure testing (§3), outline contributions of PREFAIL in more detail (§4), describe each new aspect of PREFAIL (§5 to §8), present the evaluation (§9), and finally conclude (§10).

2 Related Work

In the last couple of years, many large-scale failure statistics have been made publicly available [1, 2, 8, 13, 19, 20], and they all reach to the same conclusion that large-scale systems see frequent hardware failures. Thus, there is a shift in reliability paradigm: today's software should assume that hardware does not have perfect reliability. As an implication, failure testing has become a mainstream technique to test software reliability [4, 13, 14, 22].

One major challenge of failure testing is that the failure space to explore is potentially large [12, 16]. One direct way to explore the space is via randomness. For example, random injection of failures is employed by the developers at Google [4], Yahoo! [22], Microsoft [23], Amazon [13], and others [14]. Random fault-injection is relatively simple to implement, but the downside is that it could easily miss corner-case failure scenarios.

Another approach is to exhaustively explore all possible failure scenarios by injecting sequences of failures in all possible ways during execution. However, we found that within an execution of a protocol (*e.g.*, distributed write protocol, log recovery), there are potentially thousands of possible combinations of failures that can be exercised [12], which could take tens of hours of testing time. Thus, exhaustive testing is plausible if the tester has enough time budget and computing resources.

Other than random and exhaustive approaches, we found little work that addresses smart failure exploration. Two novel pieces of work that are closest to ours are LFI [18] and AFEX [16]. LFI is a framework that allows a tester to write a fault injection trigger to specify the failure scenarios that she is interested in exploring [18]. The framework also has support that aids in the generation of triggers by automatically analyzing the system to find code that is potentially buggy in its handling of failures. AFEX [16] is a system that automatically figures out the set of failure scenarios that when explored can meet a certain given coverage criterion like a given level of code coverage. It uses a variation of stochastic beam search to find the failure scenarios that would have the maximal effect on the coverage criterion. To our best understanding, the authors do not address multiple failures in distributed systems.

EXPLODE [24] and FiSC [25] are system model-checkers that explore thousands of program states and inject crashes at every unique program state. They only perform at most two crashes per run (one within the target system and one within the fsck) and do not support flexible policies to explore failures.

In summary, there is only a small amount of work that addresses smart failure exploration. Thus, it is not surprising that practitioners still consider the current state of recovery testing to be behind the times [4]. Compared to other work, our framework targets distributed systems and addresses multiple failures in detail. To the best of our knowledge, we are the first to address this issue of exponential increase of multiple-failure combinations in the context of distributed systems. Thus, we also hope that this challenge attracts system researchers to present other alternatives.

3 Failure Testing

In this section, we first describe some basic information regarding our approach to failure testing, and in the following section we describe our new contributions in more detail. We begin with an example of failure testing (§3.1), then present our abstraction for a failure (§3.2) and the overall architecture (§3.3), and briefly describe the limitations of our previous work (§3.4).

3.1 Example

To illustrate the need for failure testing, consider the following code that runs on a distributed system with two nodes, A and B:

```
       Node A                   Node B
L1. write(B, msg);       L1. write(A, msg);
L2. read(B, header);     L2. read(A, header);
L3. read(B, body);       L3. read(A, body);
L4. write(B, msg);       L4. write(A, msg);
L5. write(Disk, buf);    L5. write(Disk, buf);
```

This code executes reads and writes (to the network and the disk) in each node. Given this code, hardware failures such as crashes, data corruptions, and network failures, can happen around the I/O operations. Let us consider crash-only failures for now. A crash can happen before or after the execution of a `read` or a `write` call. Since there are 10 I/O calls that are executed here, there are 20 possible ways in which a failure testing framework can inject a crash (before and after every I/O). Thus, this program could be executed in 20 experiments, where each experiment would inject a crash at a point that has not been explored before.

Beyond injecting a single failure per experiment, a tester might wish to inject a sequence of multiple failures (*e.g.*, a crash after the write at L4 in node A and then a crash before the write at L5 in node B). Thus, a failure testing framework should facilitate some form of abstraction for exploring different failure scenarios.

In addition to abstraction, a failure testing framework should support smart exploration policies. This is due to the exponential explosion of multiple failures [12]. Let's consider the code above. Since we could inject 10 possible crashes at every node, there are $10^2 * N(N-1)$ possible ways to inject two crashes, where N is the number of participating nodes. Considering many other factors such as different failure types and more failures during recovery, the number of all possible sequences of failures can be too many to explore with reasonable computing resources and time. On the other hand, a tester wants to find bugs quickly.

3.2 Failure IDs

To systematically inject different types of failures at all possible points during program execution, we need to keep track of what failures have already been injected so that the same failures will not be explored at the same execution points again. For this, we introduce a failure abstraction called the failure ID (FID). We use failure IDs to record in history the sequences of failures that have been explored, and identify the sequences that have not been explored yet.

I/O ID Fields		Values
Static	func	write()
	src loc	Write.java (line L1)
Dynamic	node	A
	target	B
	stack	(the stack trace)
Domain Specific	network msg.	"Heartbeat Msg"
Failure ID = hash (I/O ID + Crash) = 2849067135		

Table 1: **A Failure ID.** *A failure ID comprises an I/O ID plus the injected failure (e.g., crash). Hash is used to record a failure ID. For space, some fields are not shown.*

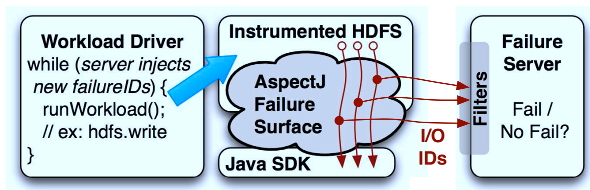


Figure 1: **Failure Testing Architecture.**

Table 1 shows an example of a failure ID. It is composed of an I/O ID (abstract information of an I/O call) and the injected failure. There are three different kinds of information that are included in the abstraction of an I/O call: static (*e.g.*, source location), dynamic (*e.g.*, stack trace, node ID), and domain-specific information (*e.g.*, the semantic meaning of a network message).

Given an I/O ID, we generate a list of possible failures that could happen before and after (*e.g.*, a bad-disk exception before a disk write, a crash after a node receives a message). Currently, we support failures such as crash, disk failure, disk corruption, node-level and rack-level network partitioning, and transient failure (one-time exception during an I/O call). A more complete description can be found here [12].

3.3 Architecture

Figure 1 shows the overall architecture of our failure testing framework. We first instrument the target system (*e.g.*, HDFS [21]) by inserting a "failure surface". The failure surface builds I/O ID at each I/O point (execution of an I/O call) and checks if a persistent failure injected in the past affects this I/O point (*e.g.*, permanent disk failures). If so, the surface naturally returns an error to emulate the failure. Otherwise, it sends the I/O ID to the server and receives a failure decision. The workload driver is where the developer attaches the workload to be tested (*e.g.*, HDFS write) and specifies the maximum number of failures to be injected per run. As the workload runs, the failure server receives I/O IDs from the failure surface, combines the I/O IDs with possible fail-

ures into failure IDs, and makes failure decisions based on the failure history. The workload driver terminates when the server has no new failure scenario to explore.

3.4 Limitations of Previous Work

As mentioned above, there is a need for smart failure exploration policies. In our previous work [12], we introduced two rigid prioritization techniques that exercise a subset of all possible sequences. Unfortunately, they are hard-coded in the framework, and cannot be extended or changed by a tester; a tester might want to express some policies to achieve certain testing objectives. Beyond lack of programmability, our previous work also has some fundamental design limitations that make it impractical to be used by real testers, limitations such as ad-hoc fault injections in some cases, non-parallelizable design, and the absence of triaging techniques.

4 PREFAIL: Programmable and Efficient Failure Testing

In this work, we introduce PREFAIL, a programmable and efficient failure testing framework. PREFAIL is a much improved failure testing framework that addresses many challenges of failure testing. More specifically, PREFAIL comes with the following core features:

1. Increased programmability to support flexible policies: Our goal is to allow testers to express failure exploration policies of different complexities so that they can use the right ones at the right times. For example, they might want to use some coarse policies that result in few experiments in the development mode, finer policies during nightly builds, and more elaborate policies that test a greater number of failures before a big release.

One challenge to support programmable failure testing is to decide the level of abstraction at which testers will write the policies. This includes deciding the language in which testers will write their policies, the abstraction of a failure that should be exposed to the testers, and the libraries that should be available to the testers so that they can easily access and use the failure abstraction.

In PREFAIL, we expose a simple abstraction of a failure with supporting library functions that testers can use to write policies in Python. The abstraction is similar to the failure abstraction described in Section 3.2, but testers do not need to have any knowledge of PREFAIL to access and use the fields of the abstraction. We also provide useful information from previous experiments that testers can use to decide which failures to prioritize next. With all the information that PREFAIL provides, a tester can write policies that can achieve important coverage criteria like high coverage of function calls where failures can happen, or high coverage of recovery behaviors.

2. Well-defined failure optimizations: There are optimizations that we can do for some types of failures that can greatly reduce the number of experiments for those types. For example, while exploring crashes, we only need to inject crashes before write I/Os; a naive failure testing framework would try to inject crashes before and after all read and write I/Os. In PREFAIL, we have clearly identified the set of optimizations that are applicable to all distributed systems, and have enabled them by default (*i.e.*, testers need not worry about these low-level optimizations). Our previous work does not have such optimizations built into it.

3. Triaging support for efficient debugging: In automated failure testing, a number of experiments can fail because of the same bug. Thus, when tens or hundreds of experiments fail, a tester could get easily overwhelmed from debugging each of the failed experiments (as is the case in our experience). To reduce the debugging effort, PREFAIL automatically triages failed experiments by clustering failed experiments according to the bugs that caused them to fail. The triaging support can also sort failed experiments according to the importance of bugs that caused them.

4. Parallelizable testing workflow: Testers might want to write failure exploration policies that exercise many failures to gain a high confidence regarding the reliability of their systems. Exploring many failures might not be feasible with reasonable computing resources within reasonable time. Thus, we threw away many of our old internal designs, and designed PREFAIL's test workflow to be parallelizable so that it could explore different failures simultaneously. Not to parallelize our failure testing framework would be a bit embarrassing because we are targeting developers of large-scale systems who tend to have many machines that they can commit for testing purposes.

As we had mentioned earlier, several companies are considering adopting our technology. However, one serious concern that they have is its long testing (and debugging) time; their unit tests already take hours to run, and integrating automated failure testing into their unit tests can increase the total testing time by several hours. Thus, it is crucial to develop techniques that can bring down the testing and debugging time. All the aforementioned enhancements in PREFAIL are necessary for making the failure testing tool useful and amenable to testers. In the following sections, we describe each of the four enhancements in detail.

5 Programmable Failure Testing

This section presents the PreFail programmable framework. We describe in sequence: the overview of the testing workflow (§5.1), the HDFS write protocol which we use as an example (§5.2), recovery profiling and clustering which form the core mechanisms of PreFail (§5.3), the abstractions and libraries that PreFail provides to testers (§5.4), and finally some example policies that testers can specify (§5.5).

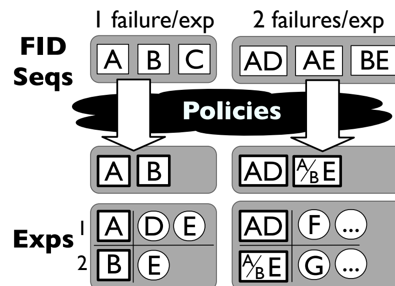


Figure 3: **PreFail Testing Workflow.**

	Initial	Recovery	Difference
Setup stage	**1** 4 5	2 4 5	2 - -
	1 **4** 5	1 5 6	- 5 6
	1 4 **5**	1 4 6	- - 6
Data transfer stage	**1** 4 5	- 4 5	- 4 5
	1 **4** 5	1 - 5	1 - 5
	1 4 **5**	1 4 -	1 4 -

Table 2: **HDFS Write Recovery.** *The table illustrates at a high-level the different recovery behaviors of the two stages of write. The first column shows the initial pipeline; the crashed node is marked in bold. The second column shows the final pipeline that contains the replicas after write recovery. The third column summarizes the I/O differences that PreFail observes between the I/Os happening in the initial pipeline and in recovery. Setup-stage recovery is basically a retry, therefore the I/O calls in recovery are the same as in the initial case, except that we see new I/Os in new nodes (e.g., 2 and 6). However, the last node in the pipeline executes different code segment. That is why Node 5 appears in the last column because it has become the middle node after recovery. Data-transfer recovery on the other hand executes new I/O calls in the surviving nodes (e.g., to synchronize block offsets). That is why the surviving nodes appear in the difference. The "diff" algorithm is explained more in §5.3.*

5.1 Test Workflow

Figure 3 presents the high-level overview of how failure testing policies programmed by testers are used to prune down the space of all possible failures and explore only a subset of the space. During testing, a tester can specify the maximum number of failures to inject in any execution of the system under test. Let us say that she sets it to 2. Then, PreFail first runs the system with zero failure during execution (i.e. without injecting any failure during execution). During this execution, it obtains the set of all failure IDs (*e.g.*, `A`, `B`, and `C`) where failures can be injected. Using the tester-specified policies, PreFail prunes down the set (*e.g.*, to `A` and `B`), and then injects failure at each failure ID in the pruned down set.

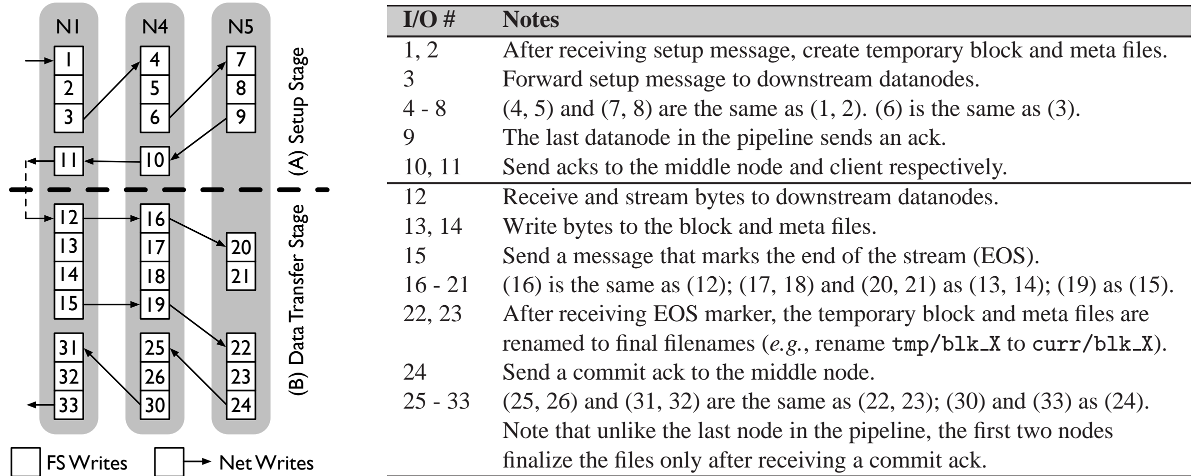
While exercising a failure ID in an experiment, PreFail traces all I/O calls that occurred in the experiment. This I/O tracing technique is one of the core mechanisms of our framework; for distributed systems, I/Os are fundamental operations that represent system behaviors. For simplicity of abstraction, we trace I/Os by recording the failure IDs. This is acceptable because when PreFail sees an I/O, it builds all possible failure IDs that can be exercised at that I/O. Thus, a failure ID implicitly represents an I/O. This way, we don't need to keep track of different types of IDs (*e.g.*, I/O IDs and failure IDs).

With I/O tracing, PreFail can capture all failures IDs that it sees in every fault-injection experiment. For example, after injecting `A` in the first experiment, PreFail observes the failure IDs `D` and `E`. From this information, PreFail creates the set of all combinations of two failure IDs (*e.g.*, `AD`, `AE`, and `BE`) that can be exercised while injecting two failures per experiment. As mentioned before, the number of all combinations of failure IDs that can be exercised tends to be large. Thus, PreFail again uses the tester-specified policies to reduce this number. For example, a tester might want to run just one experiment that exercises `E` as the second failure. Thus, PreFail would automatically exercise just one of `AE` and `BE` to satisfy this policy instead of exercising both of them.

5.2 HDFS Write Protocol

Now, we describe the HDFS [21] write protocol as an example workload. Figure 2 shows the write I/Os (both file system and network writes) occurring within the HDFS write protocol. The protocol by default stores three replicas in three nodes, which form a pipeline initiated by the client (left-hand side of the first node, not shown). If multiple racks are available, it makes sure that the nodes come from a minimum of two racks such that a single rack failure does not make the data block unavailable.

The HDFS write protocol is divided mainly into two stages: the setup stage and the data transfer stage. The recovery for each stage is different. The first and second columns of Table 2 illustrate at a high level the recovery for each stage (the third column is explained in



I/O #	Notes
1, 2	After receiving setup message, create temporary block and meta files.
3	Forward setup message to downstream datanodes.
4 - 8	(4, 5) and (7, 8) are the same as (1, 2). (6) is the same as (3).
9	The last datanode in the pipeline sends an ack.
10, 11	Send acks to the middle node and client respectively.
12	Receive and stream bytes to downstream datanodes.
13, 14	Write bytes to the block and meta files.
15	Send a message that marks the end of the stream (EOS).
16 - 21	(16) is the same as (12); (17, 18) and (20, 21) as (13, 14); (19) as (15).
22, 23	After receiving EOS marker, the temporary block and meta files are renamed to final filenames (*e.g.*, rename `tmp/blk_X` to `curr/blk_X`).
24	Send a commit ack to the middle node.
25 - 33	(25, 26) and (31, 32) are the same as (22, 23); (30) and (33) as (24). Note that unlike the last node in the pipeline, the first two nodes finalize the files only after receiving a commit ack.

Figure 2: **HDFS Write Protocol.** *The figure shows the file system and network write I/O calls within the HDFS protocol. In this configuration, nodes N1-N3 and nodes N4-N6 are separated in two racks. When the client asks the master for three nodes to put the replicas, the master node gives to client N1, N4, and N5 (communication between client and master is not shown).*

§5.3). In the setup stage, if a node crashes (*e.g.*, `N1`), the client will ask for a new node from the master node, and begin the whole write process again with the new pipeline (*e.g.*, `N2-N4-N5`). However, if a node crashes in the second stage, it will continue on the surviving datanodes [12, 21].

Although at a high level it seems that there are only two main recovery stages, we will see in the following sections how failures at different I/O points result in uniquely different recovery behaviors. This motivates the need for recovery clustering; this will help testers identify different recovery behaviors and to test them.

5.3 Recovery Clustering

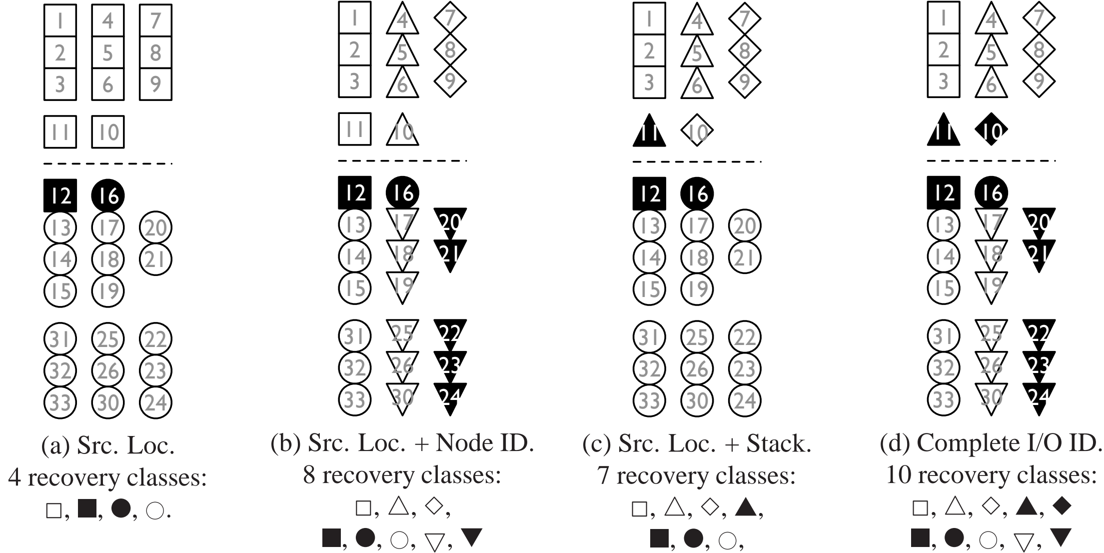
With all the I/O information that PREFAIL collects, testers can write policies that use the information to prune the space of all failures (more in section 5.5). However, since in failure testing, we are concerned about testing the correctness of *recovery* behaviors of a system, it is intuitive to prune the failure space via some form of *recovery clustering*. That is, if we can find out which failures result in a similar recovery, then we can cluster those failures together into a single recovery class. We can then simply exercise one failure from each recovery class. This way we can efficiently test different recovery behaviors of the system.

To perform recovery clustering, we first capture recovery behavior that results because of a given failure. This is performed by taking the "difference" of the execution that is seen when the failure is injected and the execution that is seen when no failure is injected. This difference can be thought of the "extra" execution or the recovery behavior that is observed when the failure is injected.

To characterize an execution, we use the list of all I/Os that we obtain during I/O profiling for that execution. PREFAIL gives the tester the power and flexibility to decide how to use the list of all I/Os to characterize an execution, and also how to define the difference of the two lists to characterize a recovery behavior. For example, a tester might want to consider two executions to be the same if they execute the code at the same source locations. Thus, she might characterize an execution by the set of all source locations of the I/Os observed during the execution. She can then define the difference of two executions to be the difference of the sets of source locations of the I/Os observed during the two executions.

After we profile the recovery behavior for every failure, developers can decide how to cluster them together. Unlike previous work, PREFAIL allows *flexible* recovery clustering. For example, the developer can decide to consider two recovery behaviors to be the same if both of them pass through the same locations in the source code, or same locations executed by the same set of nodes, and so on.

Let's consider Figure 3 as an example where `D` and `E` are two I/Os that execute at the same source location (*e.g.*, `X.java`, line 5) but in different nodes (*e.g.*, `N1` and `N4`). If a developer decides to use only source locations to characterize recovery behavior, then failure IDs `A` and `B` will fall into the same recovery class as their corresponding executions have the same set of source locations of I/Os in executed recovery code. But, if the developer decides to use both the source locations and node IDs of I/Os to characterize recovery behavior, then `A` and `B` will fall into different recovery classes.



(a) Src. Loc. 4 recovery classes: □, ■, ●, ○.

(b) Src. Loc. + Node ID. 8 recovery classes: □, △, ◇, ■, ●, ○, ▽, ▼

(c) Src. Loc. + Stack. 7 recovery classes: □, △, ◇, ▲, ■, ●, ○,

(d) Complete I/O ID. 10 recovery classes: □, △, ◇, ▲, ◆ ■, ●, ○, ▽, ▼

Figure 4: **Recovery Clusters of HDFS Write Protocol.** *The I/O numbers 1-33 represent crashes before the I/Os described in Figure 2. The three nodes in every figure represent the initial write pipeline (i.e.,* `N1-N4-N5`*). A shape (e.g., □) surrounding I/O #X represents the recovery from a crash that happens at that I/O. Different shapes represent different recovery behaviors. The four figures show the different results if recovery is characterized in different ways (e.g., (a) using source location only, (b) using source location and Node ID, and so on).*

5.3.1 Flexible Clustering of HDFS Write Recovery

To illustrate the power of flexible recovery clustering, Figure 4 shows how crashes at different I/Os shown in Figure 2 result in different recovery classes (*e.g.*, □ vs. ○). The figure also shows the result of characterizing recovery by using different elements (source location, Node ID, stack trace, target I/O, etc.) of the I/Os in the recovery path. For example, figure 4a shows four recovery classes that result from the use of only source location to distinguish different recovery behaviors. Simply by using source location, PREFAIL automatically profiles the two main recovery classes in the protocol (□ and ○) (§5.2). Furthermore, PREFAIL also finds two unique cases of failures that result in two more recovery classes (■ and ●). In the first one (■), a crash at `N1` which happens before I/O #12 leaves the surviving nodes (`N4` and `N5`) with zero-length blocks, and thus the recovery protocol executes different source locations for this particular scenario. In the second one (●), a crash happens before I/O #16 at `N4` that leaves the surviving nodes (`N1` and `N5`) with different block sizes (the first node has received the bytes, but not the last node), and thus the recovery behavior includes execution of `syncOffset` operation that truncates the surviving blocks to the lowest offset before streaming the rest of the bytes.

Figure 4b shows the 8 recovery classes that we get if the developer uses the node where the code executes in addition to source location for failure ID. The first stage recovery is distinguished into three recovery classes (□, △, and ◇). The reason behind this classification is shown in the last column of Table 2. Crashes that happen in different nodes in the first stage result in recovery code that is executed in different sets of nodes. Similarly, the second stage recovery is broken down into three classes (○, ▽, and ▼). The two unique cases stay the same (■ and ●).

Figure 4c and Figure 4d show other different ways to characterize recovery behaviors (not explained due to space constraint). In general, the more elements of I/Os considered, the more unique recovery behaviors uncovered. Fewer elements lead to fewer experiments, but might miss some corner-case bugs. For example, we have seen a bug that could only be produced if a transfer-stage crash happens at the second or last node in the pipeline. Clustering in Figure 4a might not uncover this bug if PREFAIL chooses a crash at the first node to represent the transfer-stage recovery class (○).

In summary, flexible recovery clustering provides a systematic way of reducing fault-injection experiments. Among many other things, this feature is a core mechanism of PREFAIL. Its major advantage is to reduce the tester's burden of choosing which failures to inject (*e.g.*, which I/Os at which nodes); the tester can simply specify how she wants to characterize distinct recovery behaviors (*e.g.*, depending on time and resource constraints), and PREFAIL will automatically choose only one representative failure for each of the clustered recovery behaviors.

5.4 Abstractions and Libraries

Previous sections have presented all the information that PreFail gathers and profiles. This information is accessible to testers using the following abstractions:

• **Failure Sequence (`fidSeq`):** `fidSeq` is a data structure used to describe a sequence of failure IDs that has already been exercised, or that could have been exercised but was pruned down, or that can be exercised in a subsequent experiment. For example, Figure 3 shows six failure sequences (`A`, `B`, `C`, `AD`, `AE`, and `BE`).

• **History of each experiment (`allMap` and `subqMap`):** `allMap` carries information about all failure IDs observed in every experiment; each map entry is essentially the I/O trace of an experiment. `subqMap` carries information about the failure IDs that are observed in an experiment *after* a failure sequence is injected. This information can be used to write recovery clustering policies.

PreFail also provides a library of functions for easy information extraction:

• **Failure Sequence (`reducedFid()` and `explored()`):** `reducedFid()` takes a failure ID and a list of field names, and returns a "reduced" ID that is built only from the given fields. Reduced IDs can be used during recovery clustering to characterize recovery paths by only using the given fields (*e.g.*, source location and node ID). The function `explored()` returns true if its argument field values (particular source location, node ID, etc.) have already been explored.

• **History of each experiment (`allFids()` and `postInjectionFids()`):** `allFids()` takes a failure sequence as argument, and returns the list of all failure IDs that are observed in the experiment in which the sequence is injected. Similarly, `postInjectionFids()` returns the list of all failure IDs observed after the sequence is injected.

5.5 Policies

PreFail provides support for writing two different types of policies using which testers can express how to prune the failure space: filter and cluster policies. A filter policy allows a tester to express the failure sequences that she wants to exercise. A cluster policy lets a tester express which failure sequences she considers equivalent, and thus only one failure sequence from an equivalence class will be exercised. A tester can also compose these two types of policies to achieve high-level testing objectives (*e.g.*, code coverage, recovery coverage).

Policies are in general simple to express, and are written as simple python functions that take a single failure sequence (filter policy) or two failure sequences (cluster policy) as arguments, and implement a condition involving the fields of the failure IDs in the sequence(s). Below

```
1 def filter (fidSeq):
2  for fid in fidSeq:
3   if not contains (fid.stack,
                     'setupStage'):
4    return False
5  return True
```

Figure 5: **Setup-stage filter.** *Return true if all FIDs in `fidSeq` are executed under the `setupStage` function.*

```
1 def filter (fidSeq):
2  last = fidSeq [ len(fidSeq) - 1 ]
3  return not explored (last.loc)
```

Figure 6: **New source location filter.** *Return true if the source location of the last FID has not been explored.*

we present some example policies. As a note, it is often the case that when considering whether to exercise a failure sequence (*e.g.*, `AD`), a policy analyzes the prefix (*e.g.*, `A`) and the last failure ID (*e.g.*, `B`) in different ways.

5.5.1 Filter Policies

The main use of a filter policy is to let a tester focus on a subset of recovery behaviors at a time. For example, the policy in Figure 5 only tests failures occurring in the setup stage. In our experience, filter policies are useful for testing a small subset of recovery behaviors repeatedly (*e.g.*, for testing our fixes for some of the recovery bugs that we found).

Figure 6 shows a filter policy that explores failures at previously unexplored source locations. The policy filters out a failure sequence if its latest failure ID (`last`) has an unexplored source location. This filter can be used to rapidly achieve a high coverage of failures at distinct source locations (more in §5.5.3).

5.5.2 Cluster Policies

A cluster policy lets a tester answer the following question: If a sequence of failures (*e.g.*, `AE`) has been exercised, then should another sequence (*e.g.*, `BE`) be also exercised? To help testers answer this question, a cluster policy takes two failure sequences as arguments, and returns true if the tester considers them to be the same, and false otherwise. For every cluster, PreFail automatically exercises only one representative failure sequence.

Figure 7 shows a policy that collapses all possible failures within the HDFS write setup stage into one cluster. As a result, the 11 possible single crashes (□) shown in Figure 4a are clustered into one group. PreFail will only run a single experiment to cover the cluster.

Figure 8 shows a policy that clusters failure sequences

```
1 def cluster(fidSeq1, fidSeq2):
2  isSetup1 = True
3  isSetup2 = True
4  for fid in fidSeq1:
5   if not contains (fid.stack,
     'setupStage'):
6       isSetup1 = False
7  for fid in fidSeq12:
8   if not contains (fid.stack,
     'setupStage'):
9       isSetup2 = False
10 return (isSetup1 and isSetup2)
```

Figure 7: **Setup-stage cluster.** *Cluster two failure sequences if both of them are within the setup stage context.*

```
1 def cluster(fidSeq1, fidSeq2):
2  last1 = fidSeq1 [ len(fidSeq1) - 1]
3  last2 = fidSeq2 [ len(fidSeq2) - 1]
4  return (last1 ==  last2)
```

Figure 8: **Last FIDs cluster.** *Return true if the last FIDs of two failure sequences are the same.*

that end with the same failure ID. For example, as shown in Figure 3, PreFail can exercise AE and BE, but a tester might wish to just run one of them. This kind of clustering policy is useful if the tester wants to rapidly explore new recovery paths; it does not matter how those paths are reached. The policy can also be relaxed easily. For example, a tester might wish to cluster failure sequences whose last failure IDs have the same source location even if the failure IDs are different (Figure 9); the only difference from the previous policy is in the last line.

5.5.3 Expressing Testing Objectives

In the previous examples, we have shown the benefits of filter and cluster policies. In reality, developers might want to achieve some high-level testing objectives. One common objective in the world of testing is to have some notion of "high coverage". In the case of failure testing, we can express policies that achieve different types of coverage. For example, a developer might want to achieve a high coverage of source locations of I/O calls where failures can happen. Another useful coverage criterion is high coverage of distinct recovery behaviors (as illustrated in Figure 4). Below we show how filter and cluster policies can be composed together to achieve these objectives.

Code coverage: To achieve high code-coverage with as few experiments as possible, the tester can simply compose the filter policy shown in Figure 6 and the cluster policy shown in Figure 9. As explained before, the

```
1 def cluster(fidSeq1, fidSeq2):
2  last1 = fidSeq1 [ len(fidSeq1) - 1]
3  last2 = fidSeq2 [ len(fidSeq2) - 1]
4  return (last1.loc == last2.loc)
```

Figure 9: **Source location cluster.** *Return true if the last FIDs have the same source location.*

```
1  def eqv (seq1, seq2):
2   rPath1 = getRecoveryPath (seq1)
3   rPath2 = getRecoveryPath (seq2)
4   return rPath1 == rPath2

5  def getRecoveryPath (fidSeq):
6   a = allFids(fidSeq)
7   r = reducedFids(a, ['loc'])
8   a0 = allFids([])
9   r0 = reducedFids(a0, ['loc'])
10  rPath = r - r0
11  return rPath
```

Figure 10: **Tester-defined recovery characterization.** *Line 6 uses the library function* `allFids()` *(§5.4) to get the set of all I/Os,* `a`*, seen during the execution in which* `fidSeq` *is injected. Line 7 obtains the reduced failure IDs from this set that include only the source locations of the I/Os. Line 10 filters out the reduced failure IDs that correspond to normal program execution where no failure has been injected (lines 8 and 9). Line 11 returns the set of remaining reduced failure IDs.*

filter policy explores only those failure sequences that have unexplored source locations, and the cluster policy clsuters into one group the failure sequences that have the same unexplored source location.

Recovery path coverage: Another possible testing goal is to rapidly explore multiple failures that lead to different recovery paths. To do this, the tester needs to specify how she wants to characterize a recovery behavior. As mentioned in Section 5.3 (and illustrated in Figure 4), there are many ways to characterize a recovery path. The function `getRecoveryPath(fidSeq)` shown in Figure 10 shows an example that uses only source locations to characterize a recovery path (lines 7 and 9). The function also performs one possible "diff" algorithm as explained in earlier section (§5.3). In essence, this function returns the source locations of recovery I/Os of a given failure sequence. The `eqv` function simply returns true if two recovery paths are equivalent (line 4).

After specifying an equivalent-class algorithm, the tester can simply write a policy such as the one in Figure 11. Given this function, if there are two failure sequences, `(prefix1, last)` and `(prefix2, last)`, where `prefix1` and `prefix2` result in the same recovery behavior, then PreFail will explore only one of the

```
1 def cluster(fsq1, fsq2):
2   last1 = fsq1 [ len(fsq1) - 1 ]
3   last2 = fsq2 [ len(fsq2) - 1 ]
4   prefix1 = fsq1 [ 0 : len(fsq1) - 1 ]
5   prefix2 = fsq2 [ 0 : len(fsq2) - 1 ]
6   isEqv = eqv (prefix1, prefix2)
7   return isEqv and (last1 == last2)
```

Figure 11: **Equivalent-recovery clustering.** *This policy clusters two failure sequences if their prefixes have the same recovery path and their last failure IDs are the same. Line 6 uses the equivalent-class characterization function in Figure 10.*

two sequences.

To illustrate the result of this policy, let's say there is a failure ID L reachable from all crashes at I/Os #1-13 in Figure 4a (*e.g.*, failure IDs P_1 to P_{13}). Without the specified equivalent-recovery clustering, PREFAIL will run 13 experiments (P_1L .. $P_{13}L$). But with this policy, PREFAIL will only run one experiment ($P_1/P_2/../P_{13} + L$) as all the prefixes have the same recovery class (□). If the developer changes the clustering function such that it uses source location and node ID to characterize different recovery behaviors (Figure 4b), then PREFAIL will run three experiments as the prefixes now fall into three different recovery classes (□, △, and ◇).

In summary, programmable failure testing framework allows testers to write various failure exploration policies in order to achieve different testing objectives. In addition, as different systems and workloads employ different recovery strategies, we believe this programmability is valuable in terms of systematically exploring failures that are appropriate for each strategy.

6 Well-Defined Optimizations

Generally, failures can be injected before and/or after every read and write I/Os. This section presents well-defined reasonings that throw away unnecessary fault-injections. Note that unlike policies, these optimizations are enabled by default (*i.e.*, testers only need to worry about reducing the failure space via high-level policies, but they do not have to worry about these low-level optimizations). These optimizations also help us to emulate the semantics of the effects of the injected failures more accurately. We explain optimizations for failures that PREFAIL supports.

6.1 Crashes

In a distributed system, read I/Os performed by a node affect only the local state of the node, while write I/Os potentially affect the states and execution of other nodes. Therefore, in PREFAIL, we do not explore crashing of nodes around read I/Os. We just explore crashing of nodes before write I/Os. The second optimization that we do for crashes is that we do not crash a node before the node performs a network write I/O that sends message to an already crashed node. This is because crashing a node before a network write I/O can only affect the node to which the message is being sent, but the receiver node is itself dead in this case. This optimization is particularly useful for pruning multiple-failure experiments.

6.2 Disk Failures

For disk failures (permanent and transient), we insert failure hooks before every write I/O call, but *not* before *every* read I/O call. Consider two adjacent Java read I/Os from the same input file (*e.g.*, `f.readInt()` and `f.readLong()`). It is unlikely that the second call throws an I/O exception, but not the first one. This is because the file is typically already buffered by the OS. Thus, if there is a disk failure, it is more likely the case that an exception is already thrown by the first call. Thus, due to OS buffering mechanism, and the fact that our framework interposes at the application level (Java SDK), we need to do extra work to emulate realistic failure implications. More specifically, we only insert read disk failures on the first read of every file (*i.e.*, we assume that files are always buffered after the first read). The subsequent reads to the file will naturally fail. We note that this is a simple but not a perfect emulation.

6.3 Network Failures

Optimizing network failures brings a similar challenge as in disk failures. The problem is still due to OS buffering, but we cannot use the same optimization since there is no notion of file in network I/Os. Thus, what we do is we keep information about the latest network read that a thread of a node performs. If a particular thread performs a read call that has the same sender as the previous call, then we assume that it is a subsequent read on the same network message from the same sender to this thread (potentially buffered by the OS), and thus we do not need to explicitly inject a network failure on this subsequent read. In addition, we must clear the read history if the node performs a network write, so that we can insert network failures when the node performs future reads on different network messages. Again, we note that this is not a perfect emulation, however it is effective enough for the systems that we analyze. In addition, we do not inject a network failure if one of the nodes participating in the message is already dead.

6.4 Disk Corruption

In the case of disk corruption, after data gets corrupted, all reads of the data give unexpected values for the data. It is possibly but very unlikely that the first read of the data gives a non-corrupt value and the second read in the near future gives a corrupt one. Thus, we perform an optimization similar to the disk-failure case.

7 Triaging Failure Testing Results

Multiple experiments can fail during testing because of a single bug. Since debugging each failed experiment can take a significant amount of time (many hours or even days), we designed a triaging framework in PREFAIL which allows testers to write heuristics to cluster failed experiments; an ideal heuristic should be able to put failed experiments due to the same recovery bug in one cluster. To write heuristics, testers can use all the information that we gather on every experiment (*e.g.*, I/O traces, the injected failure IDs, error messages that the target system logs). Beyond clustering failed experiments, testers could also write heuristics to sort the failed experiments based on the bug types (*e.g.*, data loss bug vs. availability bugs). Our approach is similar to other triaging techniques for other domains [10], but ours are specific to failure testing.

We explain some heuristics that we find useful while clustering failed experiments. These heuristics might not be applicable to all systems, but testers can decide on whether to use these heuristics or to design their own heuristics based on their knowledge about the system under test and the intuition that they obtain after debugging a few failed experiments.

Similar prefixes of failure sequence: Two failed experiments with failure sequences (P_1, L_1) and (P_2, L_2) are typically caused by the same bug if P_1 and P_2 are the same; `P` and `L` stands for prefix and last failure IDs (§5.5). This is because the recovery for failure P_1 or P_2 might already be incorrect, but the observable violation is only visible after more failures happen (*e.g.*, L_1 or L_2).

One might wonder why the experiments that exercised P_1 or P_2 did not fail. In our experience, we have seen that often without detailed specifications of system correctness, we cannot mark an experiment as failed (even though it already deviates from the correct behavior). Thus, only after more failures are injected does the incorrect behavior become visible (*e.g.*, by high-level system specifications). We believe that this experience actually motivates the need for injecting multiple failures.

Similar last failures: Two failed experiments with failure sequences (P_1, L_1) and (P_2, L_2) could be caused by the same bug if L_1 and L_2 are the same. This is usually the case when the system recovers correctly when P_1 or P_2 is injected, but not when the last failure is injected. That is, the bug is related to the last failure, but not to the earlier ones.

Same program execution context: Two failure sequences with failure IDs that have the same program execution context (source location and the stack trace) could be caused by the same bug. In other words, since the same code runs on different nodes in a distributed system, the same bug in the code might manifest multiple times in different nodes.

8 Parallelizing PREFAIL

Even after pruning the number of experiments, there might still be too many failure sequences to be explored within a reasonable time. Our previous framework [12], was hard to parallelize, and thus we designed PREFAIL's testing workflow in a way that it can be parallelized easily and we can get a good speed-up on its performance.

As shown in figure 3, when PREFAIL explores failure sequences of a particular length i (*e.g.*, `A` and `B` are sequences of length 1), it finds out the list of all possible sequences of length $(i + 1)$ that can be explored next (*e.g.*, `AD`, `AE`, and `BE`). Thus, after it finishes all experiments of length i, it has a list of all experiments of length $(i + 1)$ that it can explore, and it prunes the list with tester-written policies. Since the experiments that exercise the sequences in the pruned down list are independent of each other, we can easily divide the list into m parts, and send each part to a different machine to explore if we have m machines available. Since the search in PREFAIL is depth-first (not described due to space constraint), when we divide the list into m sub-lists, we keep the sequences that would be closer in a depth-first search together. Thus, to explore all sequences in its sub-list, a system would have to do lesser work while searching the failure space in a depth-first manner.

As of now, synchronization barriers exist when PREFAIL goes from step i to step $i + 1$. That is, tester cannot begin step $i + 1$ before all experiments in step i have finished. Between two consecutive steps, PREFAIL combines the per-experiment information from the worker machines (machines that run the experiments) to update the global history such as `allMap` and `subqMap` (§5.4). To handle a dead machine during testing, simple heartbeat and checkpoint-restart mechanisms can be easily implemented.

9 Evaluation

In this section, we evaluate the different aspects of PREFAIL. We first list our target systems and workloads, along with the bugs that we found (§9.1 and §9.2). Then, we quantify the effectiveness of many core features of PREFAIL (§9.3). Finally, we show the implementation complexity of PREFAIL (§9.4).

9.1 Target Systems, Workloads, and Bugs

We have integrated PREFAIL on different releases of 3 popular "cloud" systems: HDFS [21] v0.20.0, v0.20.2+320, and v0.20.2+737 (the last one is a release used by Cloudera customers [6]), ZooKeeper [15] v3.2.2 and v3.3.1, and Cassandra [17] v0.6.1 and v0.6.5. These many integrations show how easy it is to port our framework to many systems and releases. We evaluate PREFAIL on four HDFS workloads (log recovery, read, write, and append), 2 Cassandra workload (key-value insert and log recovery), and 1 ZooKeeper workload (leader election). In this submission, we only present extensive evaluation numbers for Cloudera's HDFS, which we have prioritized in the last couple of months. For other releases we only present partial results.

9.2 Bugs Found

In our previous work, we have shown the power of failure testing: we found 16 new bugs in HDFS v0.20.0 [12]. We were told that many internal designs of HDFS have changed since that version. After we integrated PREFAIL to a much newer HDFS (v0.20.2+737), we found 6 new bugs (three have been confirmed, and three are still under consideration). Since explaining the bugs requires space for detailed description, we do not describe the bugs here. Importantly, the developers believe that the bugs are crucial ones and are hard to find without a failure testing framework (their quote: "great finds!" [6]). These bugs are basically availability bugs (*e.g.*, the HDFS master node is unable to reboot permanently) and reliability bugs (*e.g.*, user data is permanently lost).

9.3 Effectiveness of PREFAIL Features

We now show the effectiveness of the core features of PREFAIL: failure optimizations, failure exploration policies, triaging heuristics, and parallelizable workflow.

9.3.1 Optimization Benefits

Table 3 shows the effectiveness of the optimizations of four different failure types that we described in Section 6. Overall, depending on the workload, the optimizations

Workload	Crash	Disk Failure	Net Failure	Data Corruption
H. Read	2/42	1/4	4/17	1/4
H. Write	57/454	27/27*	45/200	N.A.
H. Append	111/880	43/60	117/380	1/18
H. LogRecovery	36/128	39/64	N.A.	3/28
C. Insert	33/102	25/25*	12/26	N.A.
C. LogRecovery	84/196	89/98	N.A.	5/14
Z. Leader	39/132	21/21*	31/45	N.A.

Table 3: **Optimization Benefits.** *The table shows the benefits of the optimizations of four different failure types (§6) on four HDFS workloads (H), two Cassandra workloads (C), and one ZooKeeper workload (Z). Each cell shows two numbers X/Y where Y and X are the numbers of fault-injection experiments before and after the optimization respectively. N.A. represents a not applicable case; the failure type never occurs for the workload. For write workloads, the replication factor is 3 (i.e., 3 nodes participating). (*) These write workloads do not perform any disk read, and thus the optimization does not work here.*

bring 21 to 1 times (5 on average) of reduction in the number of fault-injection experiments. In the following sections, we no longer use the numbers from the unoptimized cases; since the optimizations are well-defined, they are enabled by default.

9.3.2 Policy Benefits

We now show the benefits of using different failure exploration policies to prune down the fault-injection space in different ways. Table 4 shows different number of experiments that PREFAIL runs for different policies. Here, we inject crash-only failures so that the numbers are easy to compare. The table only shows numbers for multiple-failure experiments because injecting multiple failures is where the major bottleneck is.

Compared to our previous work [12] which has one unprogrammable policy, this work enables testers to choose between different levels of pruning. In the previous work, the rigid policy gives around 10 times reduction compared to a brute-force mode (this brute-force mode exercises all possible sequences of failure IDs [12]). Even with 10 times reduction, there are still hundreds of experiments to run which could take a couple of hours to finish; each experiment usually takes between 5 to 9 seconds to run (due to pre-setups and post-checks).

With PREFAIL, a tester can choose different policies, and hence different speed-ups, depending on her time and resource constraints. For example, the code-coverage policy (CC) described in Section 5.5 gives the least number of experiments because it simply explores

Workload	#F	CC	R_L	R_{LN}	R_{All}	BF
Write	2	9	189	262	328	2229
	3	0	563	1071	2154	>10K
Append	2	3	79	126	233	3396
	3	0	42	144	334	>10K
LogRecovery	2	1	48	48	318	1365
	3	0	48	48	782	>10K
Write (v0.20.0)	2	10	197	205	367	1431
	3	0	94	206	621	>10K
Append (v0.20.0)	2	6	207	230	268	3398
	3	0	121	207	(*)	>10K

Table 4: **#Experiments run with different policies.** *Each cell shows the number of fault-injection (just crash-only failure) experiments for a given policy and a workload. #F represents the number of crashes injected per experiment. CC and BF represent the code-coverage policy and brute-force exploration, respectively. R_L, R_{LN}, and R_{All} represent recovery-coverage policies that use three different ways to characterize recovery (§5.5.3): using source location only (L), source location and node ID (SN), and all information in failure ID (All). We also include numbers from the old HDFS release v0.20.0, which has a different design (and hence different results) compared to the Cloudera's version. (*) At the time of submission, we do not have this number.*

possible crashes at source locations that it has not exercised before (*e.g.*, after exploring two crashes, there is no new source location to cover in 3-crash cases). Recovery clustering policies (R_L, R_{LN}, etc.) on the other hand run more experiments. The more relaxed the recovery characterization, the lesser the number of experiments (*e.g.*, R_L vs. R_{All}).

Pruning is not a benefit if it is not effective in finding bugs. In our experience, the recovery clustering policies are effective enough in rapidly finding important bugs in the system. To capture recovery bugs in the system, we wrote simple recovery specifications for every target workload. For example, for HDFS write, we can write a specification that says "if a crash happens during the data transfer stage, there should be two surviving replicas at the end" (more discussion on recovery specification can be found here [12]). If a specification is not met, the corresponding experiment is marked as failed.

Table 5 shows the number of bugs that we found even with the use of the most relaxed recovery clustering policy (R_L, which only uses source location to characterize recovery). But again, a more exhaustive policy could find bugs that were not caught by a more relaxed one. For example, we know an old bug that might not surface with R_L policy, but does surface with R_{LN} policy which uses source location and node ID to characterize recovery.

Workload	#F	#Failed Exps	#Bugs	#Triaged Clusters
Write	2	0	0	0
	3	46	1	10
Append	2	14	2	5
	3	31	2	9
LogRecovery	2	6	3	3
	3	3	3	3

Table 5: **#Bugs and #triaged clusters.** *The table shows the number of failed experiments for a given workload and the number of crashes per run (#F), along with the actual number of bugs that trigger the failed experiments (#Bugs). The last column shows how many clusters resulted with our triaging heuristics (#Cluster). For this table, we use the simplest recovery clustering policy (R_L column in Table 4).*

9.3.3 Triaging Benefits

The last column of Table 5 shows the benefits of triaging. The numbers show that it is unnecessary to debug all the failed experiments (*e.g.*, instead of debugging 46 experiments, the tester only debugs 10 representative clusters). An ideal triaging should give one cluster for every bug found.

We also note that the numbers of failed experiments here are relatively small because these numbers come from policy R_L which gives a smaller number of experiments compared to other policies. If we use more exhaustive policies, we get more failed experiments, and our triaging framework becomes more useful.

9.3.4 Parallelization Benefits

In failure testing, since we are basically running a particular workload repeatedly but with different failures in every run, all runs roughly have the same elapsed time. Thus, we do not have any issue of straggler; using N number of machines roughly gives N times speed-up.

9.4 Complexity

PREFAIL is composed of the main framework (Figure 1) and the workflow generator (Figure 3). The main framework (implemented in around 6000 lines of Java code) interposes the target system, runs workloads, injects failures, and records all information gathered in every run into local files. These files are then processed by the workflow generator (written in 1266 lines of Python code). The workflow generator collects files from worker machines (if PREFAIL runs in parallel mode), merges and processes the files into abstractions accessible to testers, processes the tester-defined policies, generates new jobs, and sends them to worker machines. Tester-defined policies are also written in Python. In our expe-

rience, on an average, each policy can be written in 17 lines of code, and policies are composable.

10 Conclusion

We have presented PREFAIL, a multi-thronged approach to solving the problem of long testing and debugging time of state-of-the-art failure testing. With PREFAIL, we try to help today's large-scale distributed systems "prevail" against failures. Although so far we have used PREFAIL to find only reliability bugs, we envision PREFAIL will empower many more program analyses "under failures". That is, we note that many program analyses (related to data races, deadlocks, security, etc.) are often done when the target system faces no failure. However, we did find data races and deadlock cases under some failure scenarios. Therefore, for today's pervasive cloud systems, we believe that existing analysis tools should also run when the target system undergoes failures. The challenge is that some program analyses might already be time-consuming. Running them with failures will prolong the testing time. Thus, the efficiency of PREFAIL will be a great contribution.

11 Acknowledgments

This material is based upon work supported by Computing Innovation Fellowship and the National Science Foundation under grant Nos. CCF-1018729 and CCF-0747390. We also thank Eli Collins and Todd Lipcon from Cloudera Inc. for helping us confirm the HDFS bugs that we found. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *SIGMETRICS '07*.

[2] 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 *FAST '08*.

[3] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI '06*.

[4] Tushar Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live - An Engineering Perspective. In *PODC '07*.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI '06*.

[6] Eli Collins and Todd Lipcon. Contact Person at Cloudera.

[7] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC '10*.

[8] Jeffrey Dean. Underneath the covers at google: Current systems and future directions. In *Google I/O*.

[9] Garth Gibson. Reliability/Resilience Panel. In *HEC FSIO '10*.

[10] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, , and Galen Hunt. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *SOSP '09*.

[11] Haryadi S. Gunawi. Results of visits to Cloudera, NetApp, and Twitter, November 2010.

[12] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Koushik Sen. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *NSDI '11 (to Appear; currently available as a technical report: UCB/EECS-2010-127*.

[13] Alyssa Henry. Cloud Storage FUD: Failure and Uncertainty and Durability. In *FAST '09*.

[14] Todd Hoff. Netflix: Continually Test by Failing Servers with Chaos Monkey. http://highscalability.com, December 2010.

[15] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Usenix ATC '10*.

[16] Lorenzo Keller, Paul Marinescu, and George Candea. AFEX: An Automated Fault Explorer for Faster System Testing. 2008.

[17] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. In *LADIS '09*.

[18] Paul D. Marinescu, Radu Banabic, and George Candea. An Extensible Technique for High-Precision Testing of Recovery Code. In *Usenix ATC '10*.

[19] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andre Barroso. Failure Trends in a Large Disk Drive Population. In *FAST '07*.

[20] Bianca Schroeder and Garth Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *FAST '07*.

[21] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST '10*.

[22] Hadoop Team. Fault Injection framework: How to use it, test using artificial faults, and develop new faults. http://issues.apache.org.

[23] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI '09*.

[24] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *OSDI '06*.

[25] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI '04*.