PreFail: A Programmable Failure-Injection Framework

Pallavi Joshi
Haryadi S. Gunawi
Koushik Sen

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: A Programmable Failure-Injection Framework

Pallavi Joshi
University of California, Berkeley
pallavi@cs.berkeley.edu

Haryadi S. Gunawi
University of California, Berkeley
haryadi@cs.berkeley.edu

Koushik Sen
University of California, Berkeley
ksen@cs.berkeley.edu

Abstract

As hardware failures are no longer rare in the era of cloud computing, cloud software systems must "prevail" against multiple, diverse failures that are likely to occur. Testing software against multiple failures poses the problem of combinatorial explosion of multiple failures. To address this, a tester can write diverse policies that prune down the space of multiple failures while meeting her testing objective. In this paper, we present PreFail*, a programmable failure-injection framework that enables testers to write a wide range of pruning policies. Using the principle of separation of mechanism and policy, we decouple a failure-injection framework into two components: failure-injection engine and driver. The policies written in the driver decide which failures should be injected by the engine. We define clear abstractions on which the two components interact. We integrate* PreFail *to three cloud software systems, show a wide variety of pruning policies that we can write for them and the speed-ups that we obtain with those policies.*

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 the last couple of years, many large-scale failure statistics have been made publicly available [6, 7, 18, 27, 42, 47], and they all reach 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. Thus, the software that runs on large-scale distributed systems has a great responsibility to correctly recover from diverse hardware failures such as machine crashes, disk errors, and network failures.

Even if distributed systems are built with reliability and fault tolerance as primary goals [15, 18, 24], their recovery protocols are often buggy. For example, the developers of Hadoop File System [48] have dealt with 91 recovery issues over its four years of development [25]. 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 [10, 12, 13, 25]. As an implication, failure testing has become a mainstream technique to test software failure recovery in large distributed systems [12, 27, 29, 51].

There has been some work that has proposed novel failure-injection frameworks which address single failures during program execution. However, large-scale distributed systems face frequent, multiple, and diverse failures. In this regard, there is a need to advance the state-of-the-art of failure testing for large-scale distributed systems. Exercising multiple failures is unfortunately not straight-forward. The challenge to deal with is *the combinatorial explosion of multiple failures* that can be exercised.

To address this challenge, from our personal experience and our conversation with some developers of cloud software systems, a tester can employ many different pruning strategies to reduce the large combinations of multiple failures. For example, a tester might only want to fail a representative subset of the components of a system, or inject only a subset of all possible failure types, or reduce the number of places to inject the failures with some optimizations, or explore failure points that satisfy some code coverage objectives, or fail probabilistically. Furthermore, the tester might want to compose multiple strategies together.

The goal of this paper is to build a failure-injection framework that is capable of supporting many different pruning strategies. Thus, we design, implement, and evaluate PreFail, a programmable failure-injection framework. At the heart of PreFail is the the classic principle of separation of mechanism and policy [3, 5, 20, 38, 49, 50]. That is, we decouple a failure-injection framework into two pieces: the *failure-injection engine* which is capable of interposing different execution points of the system under test and is responsible for performing failure injection at those points, and the *failure-injection driver* where testers can write pruning policies that "drive" the engine (*i.e.*, make decisions about which failures to inject).

One challenge in decoupling a failure-injection framework is to define the level of abstraction at which testers write the policies. We provide suitable abstractions of failures and the executions in which failures are injected and a library of functions for using the abstractions. With the library and abstractions that PREFAIL provides, we show how we can easily write a wide range of pruning policies that reduce the number of multiple-failure combinations.

We have integrated PREFAIL to three popular cloud systems: Hadoop File System (HDFS) [48], ZooKeeper [31], and Cassandra [37]. We have written a variety of pruning policies for these systems, and have evaluated the speed-ups that we achieve using the policies. We have also found a number of bugs in these systems using PREFAIL. Real-world adoption of PREFAIL is in progress.

In the rest of the paper, we present an extended motivation (§2) for having a programmable failure-injection framework, the design and implementation of such a framework PREFAIL (§3), examples of a wide range of policies that we can write in PREFAIL to prune down large failure spaces (§4), evaluation of PREFAIL (§5), related work (§6), and finally conclusion (§7).

2. Extended Motivation

As we enter the cloud era, it is becoming common to see a single software system running on thousands of commodity machines. For example, Hadoop MapReduce [2] and Hadoop File System (HDFS) [48] (open-source versions of Google MapReduce [19] and Google File System [23] respectively) are run on 4000 nodes at Yahoo! [4]. At this scale, hardware failures are no longer a rarity, and thus the chances of cloud software systems experiencing *multiple* failures are increasing. Thus, it is not surprising when large-scale developers ask to advance the state-of-the-art of testing, specifically by providing ways to exercise complex combinations of multiple, diverse failures [12].

2.1 The Combinatorial Explosion of Multiple Failures

Testing systems against multiple failures is unfortunately not straight-forward – the challenge to deal with is the *combinatorial explosion of multiple failures*. This explosion is attributed to the complex characteristics of failures that can arise: different types of failures (*e.g.*, crashes, disk failures, rack failures, network partitioning), different parts of the hardware (*e.g.*, two among four nodes fail), and different timings (*e.g.*, failures happen at different stages of the protocol). Exhaustively exploring all possible failure sequences can take a lot of computing resources and time.

Let's consider the code segment in Figure 1 that runs on a distributed system with two nodes, A and B. This program executes reads and writes (to the network and the disk) in each node. Given this program, hardware failures such as crashes, data corruptions, and network failures, can happen around the I/O operations. Let us consider crash-only failure

```
         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);
```

Figure 1. Example code.

that can be injected before a `read` or a `write` call. Let's suppose that the tester would like to inject two crashes. One possible combination is to crash before the write at L4 in node A and then to crash before the write at L5 in node B. Overall, since there are 5 possible points to inject a crash on every node, there are $5^2 * N(N-1)$ possible ways to inject two crashes, where N is the number of participating nodes ($N = 2$ in the above example). Again, considering many other factors such as different failure types and more failures that can be injected during recovery, the number of all possible failure sequences can be too many to explore with reasonable computing resources and time.

2.2 The Need For Programmable Failure-Injection

To address the aforementioned challenge, we believe that there are many different ways in which a tester could reduce the number of failures to inject. Below, we present some examples based on our personal experience and our conversation with some developers of cloud software systems. Our goal is to allow testers to express failure space pruning strategies of different complexities so that they can use the right ones at the right times.

Failing a component subset: Let's suppose a tester wants to test a distributed write protocol that writes four replicas to four machines (or nodes), and let's suppose that the tester wants to inject two crashes in all possible ways in this execution to show that the protocol could survive and continue writing to the two surviving machines. A brute-force technique will inject failures on all possible combinations of two nodes (*i.e.*, $\binom{4}{2}$). However, to do this quickly, the tester might wish to specify a policy that just injects failures in *any* two nodes.

This type of policy can also be applied to other target systems (*e.g.*, RAID systems [41]). For example, let's imagine a tester of a N-disk RAID system who wants to test that the RAID system can survive two disk failures (*e.g.*, as in RAID-6 or RAID-DP [16]). Again, a brute-force technique will exercise $\binom{N}{2}$ combinations. However, with the policy above, the tester can reduce the number of combinations. One take-away point here is that a pruning policy could be re-usable for different target systems (*e.g.*, distributed systems and RAID systems).

Failing a subset of failure types: Another way to prune down a large failure space is to focus on a subset of the possible failure types. For example, let's imagine a testing process for a storage system, where at every disk I/O, the testing process can inject a machine crash or a disk I/O failure. Furthemore, let's say the tester knows that the system is designed as a crash-only software [11], that is, all I/O failures are supposed to translate to system crash (followed by a reboot) in order to simplify the recovery mechanism. A concrete example of this type of system is the HDFS master node; a single I/O failure observed by the master will cause shutdown and reboot. In this environment, the tester might want to just inject I/O failures but not crashes because it is useless to inject additional crashes as I/O failures will lead to crashes anyway.

Exploring a subset of possible failure types is also useful when the tester wants to test specific protocols against specific failure types. One good example is the rack-aware data placement protocol common in many cloud systems to ensure high availability [21, 48]. The protocol should ensure that file replicas should be placed on multiple racks such that if one rack goes down, the file can be accessed from other racks. In this scenario, if the tester wants to test the rack-awareness property of the protocol, only rack failures need to be injected (*e.g.*, vs. individual node or disk failures).

Domain-specific optimization: In some cases, system-specific knowledge can be used to reduce the number of failures. For example, consider ten consecutive Java read I/Os that read from the same input file (*e.g.*, `f.readInt()`, `f.readLong()`, ...). In this scenario, disk failure can start to happen at any of these ten calls. In a brute-force manner, a tester would run ten experiments where disk failure begins at 10 different calls. However, with some operating system knowledge, the tester might inject disk failure only on the first read. The reasoning behind this is that a file is typically already buffered by the operating system after the first call. Thus, it is unlikely (although possible) to have earlier reads be successful and the subsequent reads fail. In our experience, by reducing these individual failures, we greatly reduce the combinations of multiple failures.

Coverage-based policies: A tester might want to speed up the testing process with some coverage-based policies. For example, let's imagine two different I/Os (A and B) that if failed could initiate the same recovery path that performs another two I/Os (M and N). To ensure correct recovery, a tester should inject more failures in the recovery path. A brute-force method will exercise 4 experiments by injecting two failures at AM, AN, BM, and BN (M and N cannot be exercised by themselves unless A or B have been failed). But a tester might wish to finish the testing process when she has satisfied some code coverage policy, for example, by stopping after all I/O failures in the recovery path (at M and N) have been exercised. With this policy, she only needs to run 2 experiments with failures at AM and AN.

Failing probabilistically: Multiple failures can also be reduced by only injecting them if the likelihood of their occurrence is greater than a predefined threshold [45, 51]. This technique is useful especially if the tester is interested in correlated failures. For example, two machines put within the same rack are likely to fail together compared to those put across different racks [21]. In the case of sector-level disk failures, if a sector fails, its neighboring sectors are also likely to fail [46]. A tester can use real-world statistical data to implement some failure probability distributions and write failure-injection policies based on the distributions.

In summary, there are many different ways in which a tester can reduce the number of injected failures. Thus, we believe that there is a need for a programmable failure-injection framework that enables testers to express different pruning policies. In the following section, we describe our approach in detail.

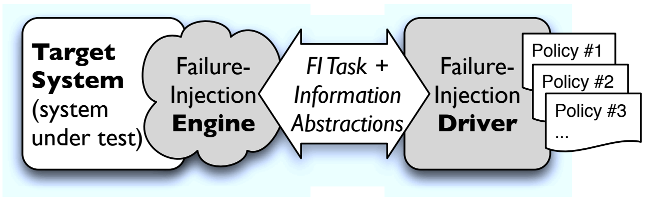


Figure 2. PREFAIL Architecture. *The figure shows the separation of failure-injection engine (mechanism) and driver (policy). The pruning policies written in the driver make failure decisions that drive the engine.*

3. Programmable Failure Injection

In this section we present PREFAIL, a programmable failure-injection framework that allows testers to express different ways to prune down a large failure space. At the heart of PREFAIL is the classic principle of separation of mechanism and policy [3, 5, 20, 38, 49, 50]. As an informal definition, one can view the policy as the scheme for deciding what should be done and the mechanism as the tool for implementing a set of policies.

With this principle, we decouple a failure-injection framework into two pieces: the FI engine and the FI driver as depicted in Figure 2 (FI stands for failure-injection). The FI engine is the component that injects failures in the system under test, and the FI driver is the component that takes tester-specified policies to decide where to inject failures. One challenge in decoupling a failure-injection framework is to define the level of abstraction at which testers will write the policies. This includes deciding clear roles of the FI engine and the FI driver, the failure abstraction that should be exposed to the FI driver, the language in which to write the policies, and the libraries that should be made available to the testers so that they can easily access the failure abstraction and re-use existing pruning strategies. In following sections, we explain the FI engine, the abstraction interface, the FI driver, and finally the overall testing workflow.

3.1 FI Engine

The FI engine is the component that interposes the different execution points in the system under test and injects failures at those points. The target failure points and the range of failures that can be injected all depend on the objective of the tester. For example, interposition can be done at Java/C library calls [25, 39], TCP-level I/Os [17], disk-level I/Os [44], POSIX system calls [36], OS-driver interfaces [32], and at many other points. Depending on the target failure points, the range of failures that can be injected varies.

To prove the concept of programmable failure-injection framework, we take a failure-injection framework that we built in prior work [25] as an example of a FI engine. This particular FI engine interposes all I/O related to calls to Java libraries and emulates hardware failures by supporting diverse failure types such as crashes, disk failures, and network partitioning at node and rack levels.

The FI driver tells the FI engine to run a set of experiments that satisfy the written policies. An *experiment* is an execution of the system under test with a particular failure scenario (could be one or multiple failures). For example, using the example in Figure 1, the FI driver could tell the FI engine to run one experiment with one specific failure (*e.g.*, a crash before the write at L4 in node A) or two concurrent failures (*e.g.*, the same crash plus a crash before the write at L5 in node B).

As mentioned above, one major challenge in supporting programmable failure injection is to decide the information abstraction that a FI engine should expose to the FI driver. Below, we present how we address this.

3.2 Abstractions and Libraries

We believe that, no matter what the supported failure types are and where the failures are injected, every failure-injection mechanism can describe an injected failure as a task. Thus, we first define the abstraction *failure-injection task* (or `fit` in short). Testers then can write pruning policies on top of this `fit` abstraction. Only exposing `fit` however is not enough – to enable testers to write powerful policies (*i.e.*, a variety of pruning policies), the FI engine must expose to the FI driver more information regarding the execution of the system under test. We record this information in *per-experiment profile*.

• **Failure-Injection Task (`fit`)**. A failure-injection task (`fit`) is a list of key-value pairs, with a key being a string representing a part of the context associated with the failure represented in the task.

The goal of using key-value pairs to represent a `fit` is two-fold. First, we want to provide the tester more flexibility with adding as much contextual information as she desires. Usually, the more the information provided to the FI driver, the more powerful the policies that one can write. Second, we envision that some pruning policies (*e.g.*, failing a component subset of a system in Section 2) can be used not only for one specific system, but also for many other systems (*e.g.*, distributed systems and RAID systems). Thus, the structure of the pruning algorithm stays the same, and all the tester needs to change is the key that represents the system component. We show an example of this scenario in Section 4.2.

Key	Value
failure	crash
func	write()
loc	Write.java (line L1)
node	A
target	B
stack	⟨stack trace⟩
...	...

Table 1. A Failure-Injection Task (`fit`). *A failure-injection task comprises a set of key-value pairs that represent the type of injected failure and the contextual information of the failure. In this example, the task is to inject a crash (defined by the `failure` key). This `fit` also contains more contextual information about when and where the failure would be exercised: The key `func` is the function call where the failure is injected, `loc` is the location of the function call in the source code, `node` is the node where the failure occurs, `target` is the target of the I/O (e.g., the node to which a network message is sent by the function call), and `stack` is the stack in node A when the failure is injected.*

Failure-injection tasks can be built automatically within the FI engine. To do that, the tester first needs to decide what failures and what contextual information about the failures that she wants to include in a `fit`. We modified our old failure-injection framework [25] to create a `fit` for every I/O call and a possible failure that it can inject at that I/O call. Let's use as an example the code segment in Figure 1. In this example, Table 1 shows the `fit` that our FI engine builds to represent a crash before the `write` call at line L1 in node A. The FI engine passes this `fit` to the FI driver. Using the detailed contextual information in the `fit`, a tester can decide whether to ask the FI engine to exercise the failure at the `fit` or not. In general, the contextual information present in `fit`s allows testers to decide how to prune down the set of all possible failure sequences that can be exercised. The FI engine injects failures only at the failure sequences in the pruned-down set returned by the FI driver.

• **Failure Sequence (`fitSeq`):** Since one of our primary goals is to exercise multiple failures, to easily express a combination of multiple failure-injection tasks, we define a failure sequence (`fitSeq`). A failure sequence is a data structure used to describe a sequence of `fit`s.

• **Per-experiment profile**. As mentioned above, to enable a variety of pruning policies, the FI driver needs more information. One type of information that we find really useful is the execution profile of every experiment. In every experiment, the FI engine not only injects the failure(s), but also records all failure-injection tasks corresponding to the failures that were exercised or that could have been exercised in the experiment. The intuition is that failure-injection tasks are built out of failure-injection points (*e.g.*, I/O calls, library calls, or system calls), and the set of these failure-injection points can be used to represent the execution of the target system. Thus, the set of all failure-injection tasks observed during an execution can be used to represent that execution.

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

• **Reduced fit:** Sometimes testers might want to use a 'reduced' `fit` that retains a subset of the key-value pairs in the original `fit` (for example, when they are interested in only some specific context of failures). Thus, PREFAIL provides a function `reducedFit` that takes a `fit`, a list of keys, and a boolean as arguments. If the boolean is true, then `reducedFit` returns a 'reduced' `fit` that includes only those key-value pairs in `fit` that involve the keys in the argument list. If the boolean is false, then it returns a 'reduced' `fit` that excludes the key-value pairs from `fit` that involve the keys in the argument list, and retains the rest of the pairs. There are variations of the `reducedFit` function that take a set of `fits` or a sequence of `fits` as arguments instead of a single `fit`. The function `reducedFits` takes a set of `fits`, a list of keys, and a boolean as arguments, applies the `reducedFit` function on each `fit`, and returns a set of 'reduced' `fits`. Similarly, the function `reducedFitSeq` deals with a sequence of `fits`.

• **Failure-Injection Tasks:** PREFAIL provides a function `allFitSeqs` that returns the list of all `fit` sequences that can be exercised by the FI engine. The FI driver uses policies to determine which of these sequences should the FI engine actually exercise. PREFAIL also provides a function `explored` that takes a list of key-value pairs and returns true if a failure with those key-value pairs has already been explored in the past by the FI engine, else false.

• **Profiling information:** To let testers easily access the profiling information and use it in their policies, there are two functions that are provided by the PREFAIL: `allFits` and `postInjectionFits`. The function `allFits` takes a failure sequence as argument and returns the set of all `fits` observed in the experiment in which its argument failure sequence is injected. The function `postInjectionFits` takes a failure sequence as argument and returns the set of all `fits` observed after its argument failure sequence is injected.

3.3 FI Driver

The FI driver provides support for writing policies using which testers can express how to prune the failure space. A policy is a function that takes a set of failure sequences and returns a subset of the sequences to explore. The FI engine

```
def filter (fitSeq):
 for fit in fitSeq:
  if not contains (fit['stack'],
                    'setupStage'):
   return False
 return True
```

Figure 3. Setup-stage filter. *Return true if all `fits` in `fitSeq` are executed under the `setupStage` function.*

```
def filter (fitSeq):
 last = fitSeq [ len(fitSeq) - 1 ]
 return not explored [('loc',
                       last['loc'])]
```

Figure 4. New source location filter. *Return true if the source location of the last `fit` has not been explored.*

exercises only the sequences present in the output. Testers can use the failure and profiling information abstractions provided by the FI engine in their policies. There are two different kinds of policies that can be written: filter and cluster policies.

3.3.1 Filter Policy

A filter policy uses a `filter` function that takes a failure sequence as an argument and implements a condition that decides whether to exercise the sequence or not. Thus, the policy takes a set of failure sequences, applies the `filter` function on each sequence, and retains the sequence in its output set if the `filter` function returns true for it. Figure 3 shows a `filter` function written in Python that selects only those failure sequences that involve failures that occur when the target system is in the setup stage (*e.g.*, the stack trace contains the function `setupStage`). The main use of a filter policy is to let a tester focus on a subset of all failure-injection tasks at a time.

Figure 4 shows another example of a `filter` function. The policy that uses this function explores failures at previously unexplored source locations by filtering out a failure sequence if its latest `fit` (`last`) has an unexplored source location. This policy can be used to rapidly achieve a high coverage of failures at distinct source locations (more in §4.3).

3.3.2 Cluster Policy

A cluster policy lets a tester answer the following question: If a failure sequence (*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 uses a `cluster` function that takes two failure sequences as arguments, and returns true if the tester considers them to be the same, and false otherwise. The `cluster` function implements an equivalence relation between failure sequences that

```
def cluster(fitSeq1, fitSeq2):
 isSetup1 = True
 isSetup2 = True
 for fit in fitSeq1:
  if not contains (fit['stack'],
   'setupStage'):
      isSetup1 = False
 for fit in fitSeq2:
  if not contains (fit['stack'],
   'setupStage'):
      isSetup2 = False
 return (isSetup1 and isSetup2)
```

Figure 5. Setup-stage cluster. *Cluster two failure sequences if both of them are within the setup stage context.*

```
def cluster(fitSeq1, fitSeq2):
 last1 = fitSeq1 [ len(fitSeq1) - 1]
 last2 = fitSeq2 [ len(fitSeq2) - 1]
 return (last1 ==  last2)
```

Figure 6. Last `fits` cluster. *Return true if the last `fits` of two failure sequences are the same.*

is used by the cluster policy to partition its argument set of failure sequences into disjoint subsets. The policy then randomly selects one failure sequence from each equivalence class. Thus, the tester implements her notion of equivalence of failure sequences, and the policy uses the equivalence relation to select failure sequences such that all equivalence classes in its argument set of failure sequences are covered.

Figure 5 shows a `cluster` function that collapses all possible failures within the setup stage into one cluster. Thus, even if there are N ($N >> 1$) possible single failures that can be injected during the setup stage in a system, all of the failures will be clustered into one group, and only one of the failures will be exercised by the FI engine to cover the cluster.

Figure 6 shows another example of `cluster` function. The `cluster` function clusters failure sequences that end with the same `fit`. For example, there might be two failure sequences `AE` and `BE` that can be exercised in two experiments, but a tester might wish to run just one experiment that has `E` as the second failure. This kind of cluster 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 `fits` have the same source location even if the `fits` are different (Figure 10); the only difference from the previous `cluster` function is in the last line.

3.4 Test Workflow

Figure 7 shows an example scenario of the testing process in PREFAIL. The tester specifies the maximum number of

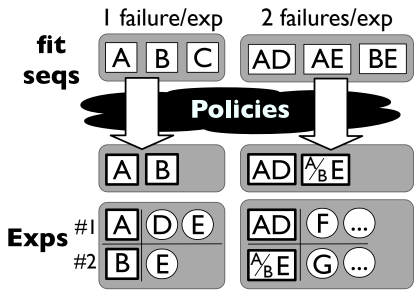


Figure 7. PREFAIL Test Workflow.

failures to inject in an execution of the system under test as 2. The FI engine 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 `fits`: `A`, `B`, and `C` where failures can be injected. Using the tester-specified policies, PREFAIL prunes down the set to `A` and `B`, and then injects failure at each `fit` in the pruned down set.

After injecting the failure in an experiment, the FI engine records all `fits` seen to build sequences of two `fits` that can be exercised while injecting two failures per experiment. For example, after injecting `A` in the first experiment, the FI engine observes the `fits` `D` and `E`. From this information, the FI engine creates the set of sequences of two `fits` `AD` and `AE`. Similarly, it creates `BE` after observing the `fit` `E` in the experiment that exercises `B`. As mentioned before, the number of all sequences of `fits` 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.

Algorithm 1 outlines the overall testing process with PREFAIL. PREFAIL takes a system S to test, a list of tester-written policies P, and the maximum number of failures N to inject in an execution of the system. The testing process runs in $N + 1$ steps. At step i ($0 \le i \le N$), the FI engine of PREFAIL executes the system S once for each failure sequence of length i that it wants to test, and injects the failure sequence during the execution of the system. FS_c is the set of all failure sequences that should be tested in the current step, and FS_n is the set of failure sequences that should be tested in the next step. Initially FS_c is set to a singleton set with the empty failure sequence as the only element. Thus, at step 0, the FI engine executes S and injects an empty sequence of failures, that is, it does not inject any failure. The FI engine observes the `fits` that are seen during execution, and adds singleton failure sequences with these `fits` to FS_n. Thus, FS_n has failure sequences that the FI engine can exercise in the next step, that is, in the $i = 1$ step. Before PREFAIL proceeds to the next step, it prunes down

Algorithm 1 PREFAILTest Workflow

```
1: INPUT: System under test ($S$), List of policies ($P$), Maximum number of failures per execution ($N$)
2: $FS_c$ = {()}
3: $FS_n$ = {}
4: for $0 \leq i \leq N$ do
5:   for each failure sequence $fs$ in $FS_c$ do
6:     Execute $S$ and inject $fs$ during execution
7:     Profile execution using fits observed during execution
8:     for each fit $f$ observed after injecting $fs$ do
9:       Append $(fs, f)$ to $FS_n$
10:    end for
11:  end for
12:  $FS_n$ = Prune($P$, $FS_n$)
13:  $FS_c$ = $FS_n$
14:  $FS_n$ = {}
15: end for
```

the set FS_n using the policies written by testers using the policy framework. The failure and profiling information abstractions provided by the FI engine can be used by testers to write rich and useful policies. In step $i = 1$, FS_c is set to the pruned down FS_n from the previous step. For each failure sequence fs in FS_c, the failure-injection framework executes S and injects fs during execution. For each `fit` f that it observes after fs has been injected, it adds the failure sequence (fs, f) to FS_n. After S has been executed once for each failure sequence in FS_c, PREFAIL prunes down the set FS_n with policies and moves to the next step. This process is repeated till the last step.

4. Crafting Pruning Policies

As discussed earlier in Section 2, when dealing with multiple failures, one needs to deal with the combinatorial explosion of failures. We believe that there is no single way to prune down the failure space – it depends on the objectives of the tester. In Section 2, we have listed some pruning strategies that we and real developers find valuable. For example, the tester can fail only a subset of all the components of the target system, inject a subset of possible failure types, reduce the number of failure-injection points with some domain-specific optimization, inject failures that satisfy some coverage-based policies, or fail probabilistically.

Overall, we make three major points in this section. First, by clearly separating the failure-injection mechanism and policies and by providing useful abstractions, we can write many different pruning strategies clearly and concisely. Second, we show that policies can be easily composed together to achieve different testing objectives. Finally, we show that some policies can be reused for different target systems. We believe these advantages show the power of PREFAIL.

	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 -

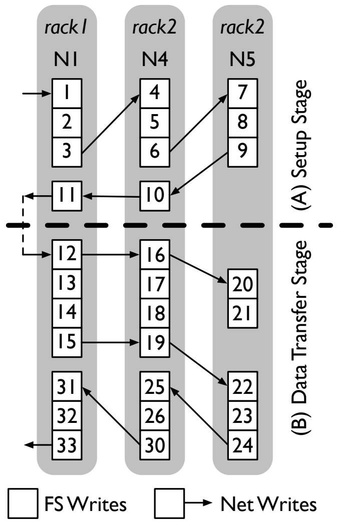
Table 2. HDFS Write Recovery. *The table illustrates at a high-level the different recovery behaviors of the two stages of write. The first column shows the initial pipeline; the crashed node is marked in bold. The second column shows the final pipeline that contains the replicas after write recovery. 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.*

To show all the policies that we have written and the advantages, we integrate PREFAIL to Hadoop File System (HDFS) [48], an underlying storage system for Hadoop MapReduce [2]. HDFS has been widely deployed in hundreds of organizations including Amazon, Yahoo!, Twitter, and Facebook [1]. We begin with an introduction to HDFS and then present the policies that we've written to prune failures in HDFS.

4.1 HDFS Primer

HDFS is a distributed file system. Here we describe the HDFS write workload in detail. Figure 8 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.

Our FI engine is able to emulate hardware failures on every I/O (every box in Figure 8). As depicted, there are at least 33 failure points that the FI engine interposes in this write protocol. At every I/O, the FI engine can inject a crash, a disk failure (if it's a disk I/O), a node-level network partitioning (if it's a network I/O), and a rack-level network partitioing (if it's a network I/O across two racks). The figure also depicts many possible ways in which multiple failures can occur. For example, two crashes can happen simultaneously at I/O #13 and #17, or a crash at I/O #21 and rack partitioning at I/O #15, and many more.



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 8. 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).*

The figure only shows the I/Os that occur in the normal execution path, it does not show the I/Os in the recovery path. Often, developers are interested in multiple failures some of which occur in the normal path and the rest in the recovery path. Later, we will show policies that quickly exercise different recovery paths. Before that, we explain the HDFS write recovery protocol here.

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. In the setup stage, if a node crashes (*e.g.*, `N1`), the client asks for a new node from the master node, and begins the whole write process again with the new pipeline (*e.g.*, `N2-N4-N5`). However, if a node crashes in the second stage, the write continues on the surviving datanodes. The third column specifies the nodes where new I/Os are seen during recovery. Although at a high level it seems that there are only two main recovery paths, we will see in a subsequent section how failures at different I/Os result in uniquely different recovery behaviors. We can use recovery clustering policies of different granularities to cluster these recovery behaviors.

4.2 Pruning by Failing a Component Subset

In distributed systems like HDFS, it is common to have multiple nodes participating in a distributed protocol. As mentioned earlier, let's say we have N participating nodes, and the developer wants to inject failures on two nodes, then there are $\binom{N}{2}$ combinations of failures that one could inject. Worse, on every node (as depicted in Figure 8), there could be many possible points to exercise the failure.

```
def cluster(fitSeq1, fitSeq2):
 rs1 = reducedFitSeq(fitSeq1, ['node'],
                          False)
 rs2 = reducedFitSeq(fitSeq2, ['node'],
                          False)
 return (rs1 == rs2)
```

Figure 9. Ignore nodes cluster. *Return true if two failure sequences have the same failures with the same contexts not considering the nodes in which they occur.*

To reduce this, a developer might just wish to inject failures at all possible failure points in *any* two nodes. She can write a cluster policy that uses the function in Figure 9 to cluster failure sequences that have the same sequence of failures occurring in the same contexts. We do not consider the nodes in which the failures occur as part of contexts of the failures. Thus, a failure sequence with a crash at node 1 and a subsequent disk failure at node 2 would be considered the same as a sequence with a crash at node 3 and a disk failure at node 4 if the rest of the contexts of the crashes and the disk failures are the same. The developer can then use this cluster policy to direct the FI engine to exercise failure sequences with two failures such that if the FI engine has already explored failures on a pair of nodes then it should not explore the same failures on a different pair of nodes.

Earlier, we mentioned that this type of pruning strategy might work for other systems such as RAID systems. Let us assume that there are N disks in a RAID system and the tester wants to inject failures at any two of these N disks. To do this, we definitely need a FI engine that works for RAID systems, but we can re-use much of the policy that we wrote for distributed systems for RAID systems. The

```
def cluster(fitSeq1, fitSeq2):
 last1 = fitSeq1 [ len(fitSeq1) - 1]
 last2 = fitSeq2 [ len(fitSeq2) - 1]
 return (last1[‘loc’] == last2[‘loc’])
```

Figure 10. Source location cluster. *Return true if the last* `fits` *have the same source location.*

only difference would be in the keys in the `fits` (*i.e.*, for distributed systems we used the ‘node’ key in Figure 9, for RAID systems we use the ‘disk’ key).

4.3 Pruning via Code-Coverage Objectives

In the previous sections, 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.

To achieve high code-coverage with as few experiments as possible, the tester can simply compose the policies that use the `filter` function shown in Figure 4 and the `cluster` function shown in Figure 10. As explained before, the filter policy explores only those failure sequences that have unexplored source locations, and the cluster policy clusters into one group the failure sequences that have the same unexplored source location.

4.4 Pruning via Recovery-Coverage Objectives

Since in failure testing, we are concerned with testing the correctness of *recovery* behaviors of a system, another useful testing goal is to rapidly explore failures that lead to different recovery paths. To do this, a tester can write a cluster policy that clusters failure sequences that lead to the same recovery behavior into a single class. PREFAIL can then use this policy to exercise a failure sequence from each cluster, and thus exercise a different recovery behavior with each failure sequence. We explain this whole process in two steps: characterizing recovery behavior, and clustering failure sequences based on the recovery characterization.

4.4.1 Characterizing Recovery Behavior

To write a recovery clustering policy, a tester has to specify how she wants to characterize a recovery behavior. One possible characterization would be to consider the “difference” of the execution that is seen when the failure sequence is injected and the execution that is seen when no failure is injected. The difference can be thought of as the “extra” execution or the recovery behavior that is observed when the failure sequence is injected.

PREFAIL gives the tester the power and flexibility to decide how to use the profiling information provided by the FI engine to characterize an execution, and also how to define

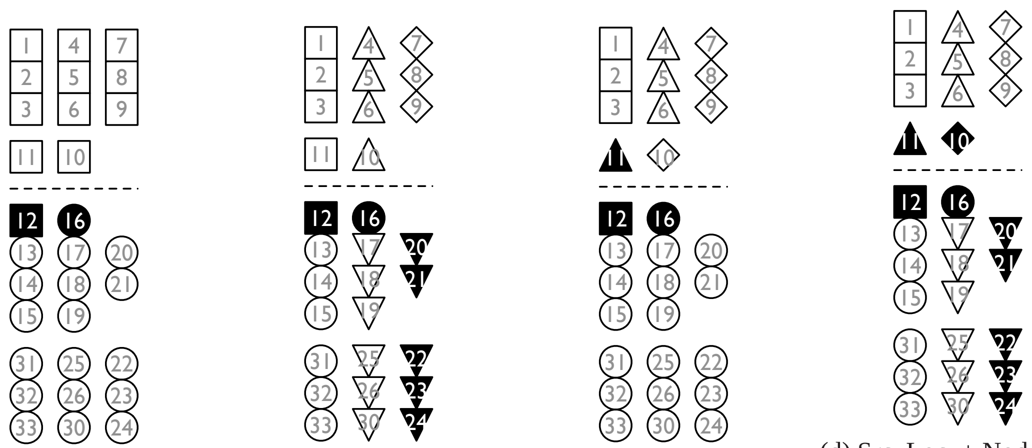
```
1   def getRecoveryPath (fitSeq):
2    a = allFits(fitSeq)
3    r = reducedFits(a, [‘loc’], True)
4    a0 = allFits([])
5    r0 = reducedFits(a0, [‘loc’], True)
6    rPath = r - r0
7    return rPath
```

Figure 11. Characterizing recovery behavior. *Line 2 uses the library function* `allFits` *(§3.2) to get the set of all* `fits`*,* `a`*, seen during the execution in which* `fitSeq` *is injected. Line 3 obtains the reduced* `fits` *from this set that include only the source locations of the* `fits`*. Line 6 filters out the reduced* `fits` *that correspond to normal program execution where no failure has been injected (lines 4 and 5). Line 7 returns the set of remaining reduced* `fits`*.*

the difference of two executions 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 `fits` 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 `fits` observed during the two executions. Figure 11 shows a function that uses only the source locations of observed `fits` to get the recovery behavior when a particular failure sequence is injected.

Let’s consider Figure 7 as an example where `D` and `E` are two `fits` at 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 `fits` `A` and `B` will fall into the same recovery class as their corresponding executions have the same set of source locations of `fits` in executed recovery code. But, if the developer decides to use both the source locations and node IDs of `fits` to characterize recovery behavior, then `A` and `B` will fall into different recovery classes.

We show how we obtain different recovery classes using different recovery characterizations in the HDFS write protocol. Figure 12 shows how crashes at different I/Os shown in Figure 8 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 12a 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 ○) (§4.1). 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


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

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

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

(d) Src. Loc. + Node ID + Stack
10 recovery classes:
□, △, ◇, ▲, ◆
■, ●, ○, ▽, ▼

Figure 12. Recovery Clusters of HDFS Write Protocol. *The I/O numbers 1-33 represent crashes before the I/Os described in Figure 8. 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 when recovery is characterized in different ways (e.g., (a) using source location only, (b) using source location and node ID, and so on).*

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 12b shows the 8 recovery classes that we get if the developer uses the node where the I/O executes in addition to the source location of the I/O. 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 12c and Figure 12d show other different ways to characterize recovery behaviors. In general, the more elements of I/Os considered, the more unique recovery behaviors uncovered. Fewer elements lead to fewer recovery classes and thus fewer failure-injection 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 12a might not uncover this bug if PreFail chooses a crash at the first node to represent the transfer-stage recovery class (○).

```
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)

8  def eqv (seq1, seq2):
9    rPath1 = getRecoveryPath (seq1)
10   rPath2 = getRecoveryPath (seq2)
11   return rPath1 == rPath2
```

Figure 13. Equivalent-recovery clustering. *Cluster two failure sequences if their prefixes (that exclude the last* `fit`*s) result in the same recovery path and their last* `fit`*s are the same. Lines 9 and 10 use the recovery path characterization in Figure 11.*

4.4.2 Clustering Failure-Injection Sequences

After specifying a characterization for a recovery path, the tester can simply write a cluster policy that uses a `cluster` function such as the one in Figure 13. 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 two sequences.

To illustrate the result of this policy, let's say there is a `fit` L reachable from all crashes at I/Os #1-11 in Figure 12a (*e.g.*, `fits` P_1 to P_{11}). Without the specified equivalent-

```
def filter (fitSeq):
 for f in fitSeq:
   isCrash = (f['failure'] == 'crash')
   isWrite = (f['ioType'] == 'write')
   isBefore = (f['place'] == 'before')
   if isCrash and
          (not (isWrite and isBefore)):
      return False
 return True
```

Figure 14. Generic crash optimization. *Return true if all crashes occur before write I/Os.*

recovery clustering, PREFAIL will run 11 experiments (P_1L .. $P_{11}L$). But with this policy, PREFAIL will run only one experiment ($P_1/P_2/../P_{11}$ + 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 12b), then PREFAIL will run three experiments as the prefixes now fall into three different recovery classes (□, △, and ◇).

4.5 Pruning via Optimizations

Generally, failures can be injected before and/or after every read and write I/O, system call or library call. For some types of failures like crashes or network failures at I/O calls in distributed systems, there are optimizations that can be performed to eliminate unnecessary failure-injection experiments. These optimizations can also be implemented as policies by testers. In the following sections, we give examples of optimizations implemented as policies for crashes, disk failures, network failures, and disk corruption in distributed systems.

4.5.1 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, we do not need to explore crashing of nodes around read I/Os. We can just explore crashing of nodes before write I/Os. Figure 14 shows a `filter` function that can be used by a filter policy to implement this optimization. The function accepts a failure sequence if all crash failures in the sequence are injected before write I/Os. If a failure sequence has a crash that is not injected before a write I/O, then that sequence is rejected, and thus not exercised by the failure-injection engine.

The second optimization that we can do for crashes is that we do not crash a node before the node performs a network write I/O that sends a 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. The `filter` function in Figure 15 implements this optimization. It accepts a failure sequence if for each crash

```
def filter (fitSeq):
 for i in range(len(fitSeq)):
   f = fitSeq[i]
   isNet = (f['ioTarget'] == 'net')
   isWrite = (f['ioType'] == 'write')
   isCrash = (f['failure'] == 'crash')
   rNode = f['receiver']
   pfx = fitSeq[0:i]
   if isNet and isWrite and isCrash and
      nodeAlreadyCrashed(pfx, rNode):
      return False
return True
```

Figure 15. Crash optimization for network writes. *Return true if there is no crash before a network write IO that sends message to an already crashed node.*

at a network write to a receiver node `rNode` in the sequence, there is no preceding crash in the sequence that occurs in the node `rNode`. The function `nodeAlreadyCrashed` (also implemented by the tester but not shown) takes a failure sequence and a node as arguments, and returns true if there is a crash failure in the sequence that occurs in the given node.

4.5.2 Disk Failures

For disk failures (permanent and transient), we inject failures 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, we can optimize and only inject 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. The policy for this optimization is similar to the one for network failure optimization (Figure 16) as explained in the next section.

4.5.3 Network Failures

For network failures, we can perform an optimization similar to disk failures. Since there is no notion of file in network I/Os, 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 explicitly inject a network failure on this subsequent read. In addition, we clear the read history if the node performs a network write, so that we can inject network failures when the node performs future reads on different network messages. In addition, we do not inject a network failure if one of the nodes participating in the message is already dead.

```
def filter (fitSeq):
 for i in range(len(fitSeq)):
   f = fitSeq[i]
   isNetFail = (fitSeq['failure'] ==
                          'netfail')
   isRead = (f['ioType'] == 'read')
   sender = f['sender']
   node = f['node']
   thread = f['thread']
   time = f['time']
   pfx = fitSeq[0:i]
   fitSeqs = allFitSeqs()
   if isNetFail and isRead and
      (not first(pfx, node, thread,
            time, sender, fitSeqs)):
      return False
 return True
```

Figure 16. Network failure optimization. *Return true if all network failures at read I/Os are at reads that are the first reads for their respective senders.*

The `filter` function in Figure 16 can be used by a filter policy to implement the optimization for network failures. The function checks for each network failure at a read I/O in a failure sequence to see if it is the first read of data in its thread that is sent by its sender to its node. The function `first` (also implemented by the tester, but not shown) determines this condition for each network failure in the failure sequence. The key `time` in a `fit` records the time when the `fit` was observed during execution in the FI engine. This key helps in determining the temporal position of a read in the list of all failure sequences `fitSeqs` passed on by the FI engine.

4.5.4 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 possible 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 can perform an optimization similar to the disk-failure case.

4.6 Failing Probabilistically

Finally, a tester can inject multiple failures if they satisfy some probabilistic criteria. We have not explored this strategy in great extent because we need some real-world failure statistic to perform real evaluation. However, we believe specifying this type of policy in PREFAIL will be straightforward. For example, the tester can write a policy as simple as: return true if `prob(fitSeq)` $>$ 0.1. That is, inject a sequence of failures `fitSeq` only if the probability of the failures happening together is larger than 0.1. The tester needs to implement the `prob` function that ideally uses some real-world failure statistic (*e.g.*, a statistic that shows the probability distribution of two machine crashes happening at the same time).

In summary, the programmable policy framework allows testers to write various failure exploration policies in order to achieve different testing and optimization 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.

5. 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 (§5.1 and §5.2). Then, we quantify the effectiveness of pruning policies that we have written (§5.3). Finally, we show the implementation complexity of PREFAIL (§5.4).

5.1 Target Systems, Workloads, and Bugs

We have integrated PREFAIL on different releases of 3 popular "cloud" systems: HDFS [48] v0.20.0, v0.20.2+320, and v0.20.2+737 (the last one is a release used by Cloudera customers [14]), ZooKeeper [31] v3.2.2 and v3.3.1, and Cassandra [37] 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 workloads (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.

5.2 Bugs Found

With PREFAIL, we were able to find the 16 new bugs in HDFS v0.20.0 that we had reported in previous work [25]. We were told that many internal designs of HDFS have changed since that version. After we integrated PREFAIL to a much newer HDFS version (v0.20.2+737), we found 6 *newer* bugs (three have been confirmed, and three are still under consideration). Importantly, the developers believe that the bugs are crucial ones and are hard to find without a multiple-failure testing framework. 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). For brevity of space, we explain below only one of the new recovery bugs. This bug is present in the HDFS append protocol, and it happens because of multiple failures.

The task of the append protocol is to atomically append new bytes to three replicas of a file that are stored in three nodes. With two node failures and three replicas, append should be successful as there is still one working replica. However, we found a recovery bug when two failures were injected; the append protocol returns error to the caller and the surviving replica (that has the old bytes) is inaccessible. Here are the events that lead to the bug: The first node-crash causes the append protocol to initiate a quite complex distributed recovery protocol. Somewhere in the middle of this recovery, a second node-crash happens, which leaves the system in an unclean state. The protocol then initiates another recovery again. However, since the previous recovery did not finish and the system state was not properly cleaned, this last initiation of recovery (which should be successful) cannot proceed. Thus, an error is returned to the append caller, and worse since the surviving replica is in an unclean state, the file cannot be accessed.

5.3 Effectiveness of Policies

We now show the effectiveness of some of the pruning policies that we have written. We first present the code-coverage (Section 4.3) and recovery-coverage (Section 4.4) based policies, and then the optimization-based policies (Section 4.5).

5.3.1 Coverage-Based Policies

We show the benefits of using different coverage-based failure exploration policies to prune down the failure space in different ways. Figure 17 shows the different number of experiments that PREFAIL runs for different policies. An experiment takes between 5 to 9 seconds to run. Here, we inject crash-only failures so that the numbers are easy to compare. The figure only shows numbers for multiple-failure experiments because injecting multiple failures is where the major bottleneck is.

With PREFAIL, a tester can choose different policies, and hence different numbers of experiments and speed-ups, depending on her time and resource constraints. For example, the code-coverage policy (CC) gives two orders of magnitude improvement over the brute-force approach because it simply explores 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, but still give an order of magnitude improvement over the brute-force approach. 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". If a specification is not met, the corresponding experiment is marked as failed.

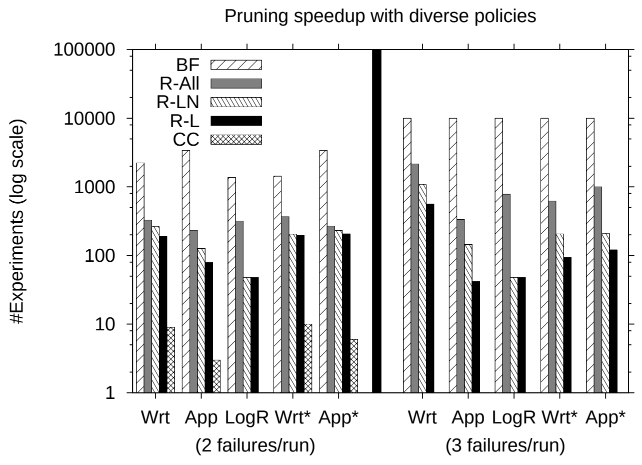


Figure 17. #Experiments run with different coverage-based policies. *The y-axis shows the number of failure-injection (just crash-only failure) experiments for a given policy and a workload. The x-axis shows the workloads: the write (Wrt), append (App), and log recovery (LogR) protocols from Cloudera's version of HDFS. We also run workloads from the old HDFS release v0.20.0 (marked with *), which has a different design (and hence different results). Two and three crashes were injected per experiment for the bars on the left- and right-hand sides respectively. 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 (§4.3): using source location only (L), source location and node (LN), and all information in `fit` (All). We stopped our experiments when they reached 10,000 (Hence, the maximum number of experiments is 10,000).*

Table 3 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.

5.3.2 Optimization-Based Policies

Table 4 shows the effectiveness of the optimizations of four different failure types that we described in Section 4.5. Each cell presents two numbers X/Y where Y and X are the numbers of failure-injection experiments for single failures without using and with using the optimization respectively. Overall, depending on the workload, the optimizations bring 21 to 1 times (5 on average) of reduction in the number of failure-injection experiments.

Workload	#F	#Failed Exps	#Bugs
Write	2	0	0
	3	46	1
Append	2	14	2
	3	31	(*) 2
LogRecovery	2	6	3
	3	3	(*) 3

Table 3. #Bugs found. *The table shows the number of failed experiments (#Failed Exps) 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). For this table, we use the simplest recovery clustering policy (R-L in Figure 17). (*) implies that these are the same bugs (i.e., bugs in 2-failure cases often appear again in 3-failure cases).*

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. LogR	36/128	39/64	N.A.	3/28
C. Insert	33/102	25/25*	12/26	N.A.
C. LogR	84/196	89/98	N.A.	5/14
Z. Leader	39/132	21/21*	31/45	N.A.

Table 4. Benefits of Optimization-based Policies. *The table shows the benefits of the optimization-based policies 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 failure-injection experiments for single failures without using and with using 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.*

5.4 Complexity

The FI engine is based on our previous work [25], which is written in 6000 lines of Java code. We added around 160 lines of code in this old framework so that it passes on appropriate `fits` and execution profiles to the FI driver. The FI driver is implemented in 1266 lines of Python code. It implements a library of functions that testers can use to access `fits` and execution profiles passed on by the FI engine. It also uses the policies written by testers to prune down the set of failure sequences that can be exercised by FI engine. We have written a number of different pruning policies in Python using the library provided by the FI driver. On an average, we wrote a policy in 17 lines of code.

6. Related Work

In this section, we compare our work with other work that relates to failure-injection. More specifically, we discuss other related work that decouple failure-injection mechanism and policy, provide some language support for specifying failure-injection tasks, and present techniques to prune down large failure spaces.

Several previous works have also suggested similar ideas to separate the component that injects the failures (*e.g.*, the "fault-injector") and the component that controls the failure-injection tasks (*e.g.*, the "controller") [8, 30, 34, 43]. In some cases, the controller can be seen as an interface for the testers to specify the failures to be injected. However, they do not present any appropriate abstractions of information that should flow between the two components. Thus, it is unclear how developers can write policies (*e.g.*, pruning policies) on top of the controller.

There has been some work in designing a clear language support for expressing which failures to inject. FAIL (Fault Injection Language) is a domain-specific language that describes failure scenarios for Grid middleware [28]. FIG also uses a domain-specific language to inject failures at library level [9]. Orchestra uses TCL scripts to inject failures at TCP level [17]. Genesis2 uses a scripting language to specify service-level failures [33]. LFI uses an XML-based language to trigger failures at library level [39]. These works however do not describe how a wide range of policies can be written in their languages. Furthemore, the tester might need to write *from scratch* the failure-injection tasks in these languages. In contrast, in our work, we abstract out a failure-injection task, and let testers easily use the information in the abstraction to write policies.

Our work is motivated by the need to exercise multiple failures especially to test cloud software systems. As mentioned before, one major challenge is the large number of combinations of failures to explore. One direct way to explore the space is via randomness. For example, random injection of failures is employed by the developers at Google [12], Yahoo! [51], Microsoft [52], Amazon [27], and other places [29]. Random failure-injection is relatively simple to implement, but the downside is that it can easily miss corner-case bugs that manifest only when specific failure sequences are injected.

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 the execution of a protocol (*e.g.*, distributed write protocol, log recovery), there are potentially thousands of possible combinations of failures that can be exercised, which can take hundreds of hours of testing time [26]. Thus, exhaustive testing is plausible only if the tester has enough time budget and computing resources.

Other than random and exhaustive approaches, there has been some work in devising smart techniques that systematically prune down large failure spaces. Extensible LFI [40] for example automatically analyzes the system to find code that is potentially buggy in its handling of failures (*e.g.*, system calls that do not check some error-codes that could be returned). AFEX [35] 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. Fu *et al.* [22] use compile-time analysis to find which failure-injection points would lead to the execution of which error recovery code. They use this information to guide failure injection to obtain a high coverage of recovery code. To the best of our knowledge, the authors of these works do not address pruning of combinations of multiple failures in distributed systems.

In our previous work [25], we begun the quest of finding techniques to prune down multiple-failure sequences. In this prior work, we only presented two rigid pruning techniques which are hard-coded in the failure-injection engine that we built. Based on more experience and conversation with some developers of cloud software systems, we found that there were many more pruning policies that a tester would like to use. This led us to re-think and re-structure our failure-injection framework so that it can let testers easily and rapidly write various kinds of policies.

The multiple-failure combinatorial explosion problem is similar to the state explosion problem in model checking. Existing system model-checkers [52, 53] use domain-specific optimization techniques to address the state explosion problem. However, when it comes to multiple failures, we did not find any system model-checker that is able to effectively prune down combinations of multiple failures. We believe that some of the pruning strategies that we have introduced in our work can be integrated within a system model checker.

In summary, there is only a small amount of work that addresses smart failure exploration. Thus, it is not surprising that practitioners of cloud systems still consider the current state of recovery testing to be behind the times [12]. Compared to other work, our framework targets distributed systems and addresses multiple failures in detail. We hope that our work attracts other researchers to present other alternatives to prune down multiple failure combinations.

7. Conclusion

We have presented PREFAIL, a programmable failure-injection framework. With PREFAIL, we have made three main contributions: (1) We show PREFAIL as a strong case of how the principle of seperation of policy and mechanism can be applied to failure-injection frameworks. (2) We design, implement, and evaluate PREFAIL. In particular, we present clear roles of the FI engine and driver, along with the clear and rich information abstractions that flow between the two components. (3) We present many policies to prune down the large number of combinations of multiple failures. Real-world adoption of PREFAIL is in progress.

Currently, we are also adding two other important features to PREFAIL: support for triaging of failed experiments, and parallelizing the whole architecture of PREFAIL. Since debugging each failed experiment can take a significant amount of time (many hours or even days), being able to automatically triage failed experiments according to the bugs that caused them can be very useful. Policies in PREFAIL already prune down a failure space and result in a speed-up of the entire failure testing process, but parallelizing PREFAIL would lead to an even greater speed-up. The test workflow of PREFAIL can in fact be very easily parallelized.

Overall, our goal in building PREFAIL is to help today's large-scale distributed systems "prevail" against possible hardware failures that can arise. Although so far we use PREFAIL primarily to find 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 deadlocks under some failure scenarios. Therefore, for today's pervasive cloud systems, we believe that existing analysis tools should also run when the target system faces failures. The challenge is that some program analyses might already be time-consuming. Running them with failures will prolong the testing time. We believe the pruning policies that PREFAIL supports will be valuable in reducing the testing time for these analyses. And again, we hope that our work attracts other researchers to present other pruning alternatives.

8. 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] Applications and organizations using Hadoop/HDFS. http://wiki.apache.org/hadoop/PoweredBy.

[2] Hadoop MapReduce. http://hadoop.apache.org/mapreduce.

[3] Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP '04)*, Oslo, Norway, June 2004.

[4] Ajay Anand. Scaling Hadoop to 4000 nodes at Yahoo! http://developer.yahoo.com/blogs/hadoop/

posts/2008/09/scaling_hadoop_to_4000_nodes_a.

[5] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James Nugent, and Florentina I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[6] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, San Diego, California, June 2007.

[7] 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 '08)*, San Jose, California, February 2008.

[8] J.H. Barton, E.W. Czeck, Z.Z. Segall, and D.P. Siewiorek. Fault Injection Experiments Using FIAT. *IEEE Transactions on Computers*, 39(4):1105–1118, April 1990.

[9] Pete Broadwell, Naveen Sastry, and Jonathan Traupman. FIG: A Prototype Tool for Online Verification of Recovery Mechanisms. In *Workshop on Self-Healing, Adaptive and Self-Managed Systems*.

[10] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[11] George Candea and Armando Fox. Crash-Only Software. In *The Ninth Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, Hawaii, May 2003.

[12] Tushar Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live - An Engineering Perspective. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC '07)*, Portland, Oregon, August 2007.

[13] 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 *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[14] Eli Collins and Todd Lipcon. *Contact Persons at Cloudera Inc.*, 2011.

[15] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 2010 ACM Symposium on Cloud Computing (SoCC '10)*, Indianapolis, Indiana, June 2010.

[16] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, California, April 2004.

[17] Scott Dawson, Farnam Jahanian, and Todd Mitton. Experiments on Six Commercial TCP Implementations Using a Software Fault Injection Tool. *Software—Practice and Experience*, 27:1385–1410, 1997.

[18] Jeffrey Dean. Underneath the covers at google: Current systems and future directions. In *Google I/O*, 2008.

[19] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

[20] Joan Feigenbaum, Rahul Sami, and Scott Shenker. Mechanism Design for Policy Routing. *Distributed Computing*, 18(4):293–305, 2006.

[21] 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 '10)*, Vancouver, Canada, October 2010.

[22] Chen Fu, Barbara G. Ryder, Ana Milanova, and David Wonnacott. Testing of Java Web Services for Robustness. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '04)*, Boston, Massachusetts, July 2004.

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[24] Garth Gibson. Reliability/Resilience Panel. In *High-End Computing File Systems and I/O Workshop (HEC FSIO '10)*, Arlington, VA, August 2010.

[25] 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 *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI '11)*, Boston, Massachusetts, March 2011.

[26] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Koushik Sen. Towards Automatically Checking Thousands of Failures with Micro-specifications. In *The 6th Workshop on Hot Topics in System Dependability (HotDep '10)*, Vancouver, Canada, October 2010.

[27] Alyssa Henry. Cloud Storage FUD: Failure and Uncertainty and Durability. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST '09)*, San Francisco, California, February 2009.

[28] William Hoarau, Sebastien Tixeuil, and Fabien Vauchelles. FAIL-FCI: Versatile fault injection. Journal of Future Generation Computer Systems archive, Volume 23 Issue 7, August, 2007.

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

[30] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault Injection Techniques and Tools. *IEEE Computer*, April 1997.

[31] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC '10)*, Boston, Massachusetts, June 2010.

[32] Andreas Johansson and Neeraj Suri. Error Propagation Profiling of Operating Systems . In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '05)*, Yokohama, Japan, June 2005.

[33] Lukasz Juszczyk and Schahram Dustdar. Programmable Fault Injection Testbeds for Complex SOA. In *Proceedings of the 8th International Conference on Service Oriented Computing (ICSOC '10)*, San Francisco, California, December 2010.

[34] Wei-lun Kao, Ravishankar K. Iyer, and Dong Tang. FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior Under Faults. In *IEEE Transactions on Software Engineering*, pages 1105–1118, 1993.

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

[36] Philip Koopman and John DeVale. Comparing the Robustness of POSIX Operating Systems. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing (FTCS-29)*, Madison, Wisconsin, June 1999.

[37] 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 '09)*, Florianopolis, Brazil, October 2009.

[38] R. Levin, E. Cohen, W. Corwin, F. J. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP '75)*, Austin, TX, November 1975.

[39] Paul Marinescu and George Candea. LFI: A Practical and General Library-Level Fault Injector. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '09)*, Lisbon, Portugal, June 2009.

[40] Paul D. Marinescu, Radu Banabic, and George Candea. An Extensible Technique for High-Precision Testing of Recovery Code. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC '10)*, Boston, Massachusetts, June 2010.

[41] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, Chicago, Illinois, June 1988.

[42] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andre Barroso. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

[43] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Model-Based Failure Analysis of Journaling File Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '05)*, Yokohama, Japan, June 2005.

[44] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, United Kingdom, October 2005.

[45] C. J. Price and N. S. Taylor. Automated multiple failure FMEA. *Reliability Engineering and System Safety*, 76(1):1–10, April 2002.

[46] 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 '10)*, San Jose, California, February 2010.

[47] Bianca Schroeder and Garth Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

[48] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Massive Storage Systems and Technologies (MSST '10)*, Incline Village, Nevada, May 2010.

[49] Alex C. Snoeren and Barath Raghavan. Decoupling Policy from Mechanism in Internet Routing. *ACM SIGCOMM Computer Communication Review*, 34(1), January 2004.

[50] Evan Speight, Hazim Shafi, Lixin Zhang, and Ramakrishnan Rajamony. Adaptive mechanisms and policies for managing cache hierarchies in chip multiprocessors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*, Madison, Wisconsin, June 2005.

[51] Hadoop Team. Hadoop Fault Injection Framework and Development Guide. `http://hadoop.apache.org/hdfs/docs/r0.21.0/faultinject_framework.html`.

[52] 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 *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI '09)*, Boston, Massachusetts, April 2009.

[53] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.