PREFAIL: A Programmable Tool for Multiple-Failure Injection

Pallavi Joshi
EECS, UC Berkeley, USA
pallavi@cs.berkeley.edu

Haryadi S. Gunawi
EECS, UC Berkeley, USA
haryadi@cs.berkeley.edu

Koushik Sen
EECS, UC Berkeley, USA
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 problem, we present PreFail, a programmable failure-injection tool that enables testers to write a wide range of policies to prune down the large space of multiple failures. We integrate PreFail to three cloud software systems (HDFS, Cassandra, and ZooKeeper), show a wide variety of useful pruning policies that we can write for them, and evaluate the speed-ups in testing time that we obtain by using the policies. In our experiments, our testing approach with appropriate policies found all the bugs that one can find using exhaustive testing while spending 10X–200X less time than exhaustive testing.

General Terms Reliability, Verification

Keywords fault injection, distributed systems, testing

Categories and Subject Descriptors D.4.5 [*Operating Systems*]: Reliability; D.2.5 [*Software Engineering*]: Testing and Debugging

1. Introduction

With the arrival of the cloud computing era, large-scale distributed systems are increasingly in use. These systems are built out of hundreds or thousands of commodity machines that are not fully reliable and can exhibit frequent failures [17, 24, 38, 41, 45]. Due to this reason, today's "cloud software" (*i.e.*, software that runs on large-scale deployments) does not assume perfect hardware reliability. Cloud software has a great responsibility to correctly recover from diverse hardware failures such as machine crashes, disk errors, and network failures.

Even if existing cloud software systems are built with reliability and failure tolerance as primary goals [14, 17, 20], their recovery protocols are often buggy. For example, the developers of Hadoop File System [42] have dealt with 91 recovery issues over its four years of development [22]. There are two main reasons for this. Sometimes developers fail to anticipate the kind of failures that a system can face in a real setting (*e.g.*, only anticipate fail-stop failures like crashes, but forget to deal with data corruption), or they incorrectly design/implement the failure recovery code. There have been many serious consequences (*e.g.*, data loss, unavailability) of the presence of recovery bugs in real cloud systems [9, 11, 12, 22].

To test recovery, there has been some work that has proposed novel failure-injection tools and frameworks, but they primarily address single failures during program execution. However, cloud software systems face *frequent*, *multiple*, and *diverse* failures. In this regard, there is a need to advance the state-of-the-art of failure testing – multiple failures need to be systematically explored in program execution. Unfortunately, exercising multiple failures is not straight-forward. The challenge to deal with is the *combinatorial explosion of multiple failures* that can be exercised.

From our personal experience and our conversation with some developers of cloud software systems, we found that a tester can employ many different heuristics to prune 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 failure-injection points with some optimizations, or explore failure-injection points that satisfy some code-coverage objectives, or fail probabilistically. Furthermore, the tester might want to use multiple heuristics together.

To enable testers to express many different pruning heuristics or policies, we design, implement, and evaluate PREFAIL, a programmable failure-injection tool for multiple-failure injection. More specifically, we make the following contributions in this paper.

1. We build a programmable failure-injection tool that allows testers to write policies to express the set of multiple-failure combinations (or sequences) that they want to explore. This way, we alleviate the need to ex-

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

plore all possible multiple-failure combinations, which can be too huge to test with reasonable resources and time. To enable programmability, we decouple PREFAIL 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). PREFAIL provides suitable abstractions of failures and the execution points where the failures can be injected, and also profiles of executions where failures are injected. These abstractions can be used by testers to easily write a wide variety of pruning policies.

2. We present a number of pruning policies that we have written for distributed systems. If a tester has a good knowledge about the system being tested, then she can easily write appropriate policies to explore the failures that meet her testing objectives. But, even if the tester does not know much about the system, we show that she can still use generic coverage based policies (*e.g.*, code-coverage and recovery-coverage based policies) to systematically test the system. In our experiments, we found all bugs in a system with appropriate policies in time that is much lesser than the time to exhaustively test all possible failure sequences (*e.g.*, 20 hours vs. 1/2 hour).
3. We have integrated PREFAIL to three popular cloud systems: Hadoop File System (HDFS) [42], ZooKeeper [27], and Cassandra [33]. We provide a thorough evaluation of the speed-ups in the testing process that we obtain by using the pruning policies that we wrote. In terms of bug finding, so far we have focused more on HDFS. We found all of the 16 new bugs in HDFS that we had found in previous work [22], and also found 6 newer bugs.

We have made PREFAIL publicly available for download from http://sourceforge.net/projects/prefail/. We have, in fact, already worked with engineers from Cloudera Inc. to test their version of Hadoop software. More real-world adoption of PREFAIL is in progress.

In our previous work [22], we had begun the quest of finding techniques to prune down multiple-failure sequences. In this prior work, we only presented two rigid pruning policies which are hard-coded in the failure-injection tool 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 tool so that it can let testers easily and rapidly write various kinds of policies.

In the rest of the paper, we present an extended motivation for having a programmable failure-injection tool (§2), the design and implementation of PREFAIL (§3), examples of a wide range of pruning heuristics that we can write in PREFAIL (§4), evaluation of PREFAIL (§5), limitations (§6), related work (§7), and finally conclusion (§8) .

```
          Node A                     Node B
A1. write(B, msg);       B1. write(A, msg);
A2. read(B, header);     B2. read(A, header);
A3. read(B, body);       B3. read(A, body);
A4. write(B, msg);       B4. write(A, msg);
A5. write(Disk, buf);    B5. read(Disk, buf);
```

Figure 1. Example code.

2. Extended Motivation

In this section, we present an extended motivation for having a programmable tool for multiple-failure injection.

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 transient I/O failures, crashes, data corruptions, and network failures, can happen around the I/O operations. Thus, we would like to test the tolerance of this code against different kinds of failures by injecting those failures during its execution. For example, we can inject a transient I/O failure at the write call on line A1 by executing code that throws an IOException instead of executing the write call.

Let us suppose that a tester wants to test against crashes before `read` and `write` calls, and that she wants to inject two crashes in an execution. One possible combination is to crash before the write at A4 and then to crash before the write at B5. 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 policies of different complexities so that they can choose a suitable policy based on testing budget and requirement.

Failing a component subset: Let's suppose a tester wants to test a distributed write protocol that writes four replicas to four machines, 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.

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 that, at every disk I/O, 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 [10], that is, all I/O failures are supposed to translate to system crash (followed by a reboot) in order to simplify the recovery mechanism. 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. Another good example is the rack-aware data placement protocol common in many cloud systems to ensure high availability [18, 42]. 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).

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 perform 4 experiments by injecting two failures at AM, AN, BM, and BN (M and N cannot be exercised by themselves unless A or B has 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.

Domain-specific optimization: In some cases, system-specific knowledge can be used to reduce the number of failures. For example, consider 10 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 10 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 succeed and the subsequent reads fail. In our experience, by reducing these individual failures, we greatly reduce the combinations of multiple failures.

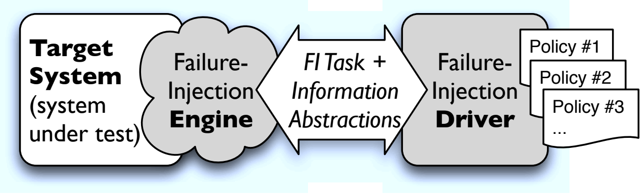


Figure 2. PreFail Architecture. *The figure shows the separation of failure-injection engine (mechanism) and driver (policy). The pruning policies written in the driver make failure decisions that drive the engine.*

Failing probabilistically: Multiple failures can also be reduced by only injecting them if the likelihood of their occurrence is greater than a predefined threshold [40, 44]. This technique is useful especially if the tester is interested in correlated failures. For example, two machines put within the same rack are more likely to fail together compared to those put across in different racks [18]. A tester can use real-world statistical data to implement policies that employ some failure probability distributions.

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

3. Programmable Failure Injection

In this section we present the design of PreFail. To enable programmability, we borrow the classic principle of separation of mechanism and policy [6, 34, 43]. With this principle, we decouple our 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. The FI engine exposes failure related abstractions to the FI driver that can be used by the testers in their policies. In the following sections, we first illustrate the test workflow in PreFail (§3.1), and then we explain the FI engine (§3.2), the abstraction interface (§3.3), the FI driver and policies (§3.4), and finally the detailed al-

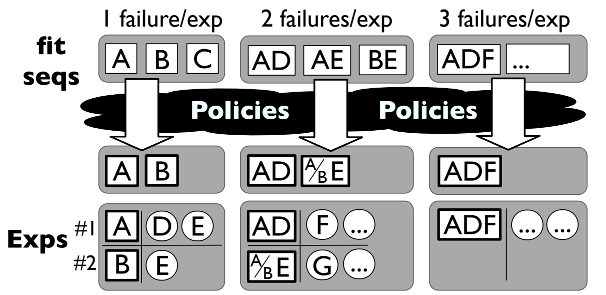


Figure 3. PreFail Test Workflow. *An alphabetical symbol represents a failure-injection point or task. A box represents a failure sequence (a sequence of failure-injection tasks) to be exercised. A letter in a circle represents a failure-injection point observed in an experiment. For ease of reading, a letter in a box represents a crash failure-injection task (e.g.,* `A` *in a box should be read as* $\mathtt{A}_c$*).*

fip B5

Key	Value
func	read()
loc	Read.java (line L5)
node	B
target	file *f*
stack	⟨stack trace⟩
...	...

Possible Failures at fip B5	fit
Crash	(crash, B5) / $\mathtt{B5}_c$
Corruption	(corruption, B5) / $\mathtt{B5}_{cr}$
Disk failure	(disk failure, B5) / $\mathtt{B5}_d$

Table 1. Failure-Injection Point (fip) and Failure-Injection Task (fit). *The left table illustrates a* fip *with label* B5*. More context can be added by adding more key-value mappings. The right-hand side table shows different* fit*s that can be formed for the* fip.

gorithm of the test workflow (§3.5). We give some examples of policies in Section 3.4.3, but for more examples and details, readers can refer to Section 4.

3.1 Test Workflow

Figure 3 shows an example scenario of the testing process in PreFail. The tester specifies three failures as the maximum number of failures to inject in an execution of the system under test. 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 execution points where failures can be injected (*i.e.*, *failure-injection points* as described in Section 3.3): A, B, and C. Let us assume that we are interested only in crashes, and let $\mathtt{A}_c$, $\mathtt{B}_c$, and $\mathtt{C}_c$ denote the injection of crashes (*i.e.*, *failure-injection tasks* as described in Section 3.3) at the failure-injection points A, B, and C, respectively (for ease of reading, a failure-injection task $\mathtt{X}_c$ is represented as X in a box in Figure 3). Using the tester-specified policies, suppose PreFail prunes down the set of failure-injection tasks to $\mathtt{A}_c$ and $\mathtt{B}_c$, and then exercises each failure-injection task in the pruned down set.

After exercising a failure-injection task, the FI engine records all failure-injection points seen where further crashes can be injected. For example, after exercising $\mathtt{A}_c$ (that is, injecting a crash at A), the FI engine observes the failure-injection points D and E. From this information, the FI engine creates the set of sequences of two failure-injection tasks $\mathtt{A}_c\mathtt{D}_c$ and $\mathtt{A}_c\mathtt{E}_c$ that can be exercised while injecting two crashes in an execution. Similarly, it creates $\mathtt{B}_c\mathtt{E}_c$ after observing the failure-injection point E in the execution that exercises $\mathtt{B}_c$.

As mentioned before, the number of all sequences of failure-injection tasks 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 test just one sequence of two crashes that exercises $\mathtt{E}_c$ as the second crash. Thus, PreFail would automatically exercise just one of $\mathtt{A}_c\mathtt{E}_c$ and $\mathtt{B}_c\mathtt{E}_c$ to satisfy this policy instead of exercising both of them. The step from injecting two failures to three failures per execution is similar.

3.2 FI Engine

The failure-injection tasks described above are created by the FI engine. The FI engine interposes different execution points in the system under test and injects failures at those points. The target failure-injection 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 [22, 35], TCP-level I/Os [16], disk-level I/Os [39], POSIX system calls [32], OS-driver interfaces [28], and at many other points. Depending on the target failure-injection points, the range of failures that can be injected varies.

In our work, we use a failure-injection tool that we had built in prior work [22] as the 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 A4) or two concurrent failures (*e.g.*, the same crash plus a crash before the write at B4).

3.3 Abstractions

In this section, we provide the abstractions that bridge the FI engine and the FI driver. The FI engine provides the following abstractions of failures and execution points where failures can be injected, and of failure-injection experiments. These abstractions can be used by testers in their pruning policies.

1. **Failure-Injection Point** (`fip`). A failure-injection point (`fip`) is a map from a set of keys $\mathcal{K}$ to a set of values $\mathcal{V}$. It is a static abstraction of an execution point where a failure can be injected. A key k in $\mathcal{K}$ represents a part of the static or dynamic context associated with the execution point. For example, k could be 'func' to represent the function call being executed. It would be mapped to the name of the function in the `fip`. Other examples for k are: 'loc' for the location of the function call in the source code, 'node' for the node ID on which the execution occurs, 'target' for the target of the I/O executed by the function call (*e.g.*, the name of the file being written to in case of a disk write I/O), and 'stack' for the stack trace. Table 1 shows the `fip` corresponding to the execution point at the `read` call at line L5 in node B in Figure 1. We denote the set of all failure-injection points by $\mathcal{P}$.

2. **Failure-Injection Task** (`fit`). A failure-injection task (`fit`) is a pair of a failure type (*e.g.*, crash, disk failure) and a failure-injection point. Thus, a `fit` $f \in \mathcal{F} \times \mathcal{P}$, where $\mathcal{F}$ denotes the set of all failure types. Given a failure-injection point, there are different types of failures that can be injected at that point. For example, Table 1 shows different `fit`s that can be formed for the `fip` illustrated in the same table for three different types of failures (crash, data corruption, and disk failure). Exercising a `fit` $f = (ft, fp)$ means injecting the failure type *ft* at the `fip` *fp*.

   Since we are interested in injecting multiple failures during execution in addition to single failures, we also consider sequences of failure-injection tasks. We denote the set of all sequences of failure-injection tasks by $\mathcal{Q}$. We call a sequence of failure-injection tasks as a *failure sequence* in short.

3. **Per-experiment profile**. To allow powerful policies (a variety of policies) to be written, the FI driver profiles the execution of the system in every experiment, and makes the profiling information available to testers. Testers can use the profiles of already executed experiments to decide in their policies which failure sequences to exercise in future experiments. Our strategy in profiling an execution is by recording the set of failure-injection points observed during the execution. The reasoning behind this is that failure-injection points are typically built out of I/O calls, library calls, or system calls, and these calls can be used to approximately represent an execution of the system under test. Thus, an execution profile $exp \in 2^{\mathcal{P}}$.

   Let `allFips`: $\mathcal{Q} \rightarrow 2^{\mathcal{P}}$ and `postInjectionFips`: $\mathcal{Q} \rightarrow 2^{\mathcal{P}}$ be the functions that return execution profiles of failure-injection experiments. Given a failure sequence *fs*, `allFips`(*fs*) returns the execution profile consisting of all `fips` observed during the experiment in which *fs* is injected, and `postInjectionFips`(*fs*) returns the set of all `fips` observed *after fs* has been injected. For the empty sequence (), `allFips` and `postInjectionFips` both return the set of all `fips` seen in the execution in which no failure is injected.

Algorithm 1 `fpGen`

```
Inputs: A filter predicate flt and a set of failure sequences $FS$
Output: A set of failure sequences $FS_P$
$FS_P = \{\}$
for $fs$ in $FS$ do
    if flt($fs$) then
        $FS_P = FS_P \cup \{fs\}$
    end if
end for
return $FS_P$
```

Algorithm 2 `cpGen`

```
Inputs: A cluster predicate cls and a set of failure sequences $FS$
Output: A set of failure sequences $FS_P$
$FS_P = \{\}$
$E = FS/R_{\texttt{cls}}$
for $e$ in $E$ do
    $fs$ = select an element from $e$ randomly
    $FS_P = FS_P \cup \{fs\}$
end for
return $FS_P$
```

3.4 FI Driver

Based on the abstractions above, the FI driver provides support for writing predicates that it uses to generate policies that express how to prune the failure space. For convenience and brevity, whenever we say that a tester writes a policy, we mean that the tester writes the predicate that is later used by the FI driver to generate the policy. A policy is a function $p : 2^{\mathcal{Q}} \rightarrow 2^{\mathcal{Q}}$. It takes a set of failure sequences, and returns a subset of the sequences to be explored by the FI engine. Testers can use the failure and execution point abstractions, and execution profiles provided by the FI engine in their predicates. There are two different kinds of predicates that can be written to generate two different kinds of policies: *filter* and *cluster* policies. PreFail can also compose the policies generated from different predicates to obtain more complex policies.

3.4.1 Filter Policy

A filter policy uses a tester-written predicate `flt`: $\mathcal{Q} \rightarrow$ `Boolean`. The predicate takes a failure sequence *fs* as an argument and implements a condition that decides whether to exercise *fs* or not. Algorithm 1 explains how a filter policy works. Given a predicate `flt`, the function `fpGen` : $(\mathcal{Q} \rightarrow$ `Boolean`$) \rightarrow (2^{\mathcal{Q}} \rightarrow 2^{\mathcal{Q}})$ (implemented in PreFail) generates a filter policy out of it. The policy takes a set of failure sequences *FS*, applies the `flt` predicate on each sequence

```
$\lambda fs$. (
    let (($ft_1$, $fp_1$), ..., ($ft_n$, $fp_n$)) = $fs$ in
    let isCrash($ft$) = ($ft$ == crash) in
    let inSetup($fp$) = $fp$['stack'] has 'setup' in
        $\bigwedge_{i \in \{1,\ldots,n\}}$ isCrash($ft_i$) $\wedge$ inSetup($fp_i$)
)
```

Figure 4. Setup-stage filter. *Return true if all* `fit`*s (ft_1, fp_1), ..., (ft_n, fp_n) in fs correspond to a crash within the* `setup` *function.*

fs, and retains fs in its result set FS_P if the predicate holds for it.

3.4.2 Cluster Policy

A cluster policy uses a tester-implemented predicate `cls`: $\mathcal{Q} \times \mathcal{Q} \rightarrow$ `Boolean`. The predicate takes two failure sequences as arguments, and returns true if the tester considers them to be similar (*e.g.*, exercising either of them would result in the same test coverage), and false otherwise. The predicate implicitly implements an equivalence relation $\mathrm{R}_{\mathtt{cls}}$ = $\{(fs_1, fs_2) \mid \mathtt{cls}(fs_1, fs_2)\}$. Algorithm 2 shows how a cluster policy works. Given a `cls` predicate, the function `cpGen` : ($\mathcal{Q} \times \mathcal{Q} \rightarrow$ `Boolean`) $\rightarrow$ ($2^{\mathcal{Q}} \rightarrow 2^{\mathcal{Q}}$) (implemented in PreFail) generates a cluster policy out of it. The policy uses the predicate to partition its argument set of failure sequences FS into disjoint subsets $FS/\mathrm{R}_{\mathtt{cls}}$. It then randomly selects one failure sequence fs 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.

3.4.3 Example Policies

We give brief examples of how one can use the filter and cluster policies. Suppose that a tester is interested in testing the tolerance of the setup stage of a distributed systems protocol against crashes. The tester can write the `flt` predicate in Figure 4. The filter policy `fpGen(flt)` would retain a failure sequence fs only if every `fit` in fs corresponds to a crash in the setup stage (execution of the `setup` function).

In failure testing, since we are concerned with testing the correctness of *recovery* paths of a system, one way to reduce the number of failure sequences to test would be to cluster them according to the recovery paths that they would lead to. Out of all failure sequences that would lead to a particular recovery path, we can just choose and test one. To achieve this, we can write the cluster predicate in Figure 5. If two failure sequences fs_1 and fs_2 have the same last `fit`, and their prefixes that leave the last `fit` out (fs_{1P} and fs_{2P} respectively) result in the same recovery path, then we can consider fs_1 and fs_2 to be equivalent in terms of the recovery paths that they would lead to since they involve injecting the same failure at the same execution point in the same recovery path. PreFail's test workflow is such that when deciding whether to test a failure sequence (*e.g.*, fs_1), all of its prefixal sequences (*e.g.*, fs_{1P}) would have already been tested, and thus we would have already seen the recovery paths that they lead to. Figure 5 uses the function `rec` to characterize a recovery path. It uses the set of all `fip`s seen in the recovery path to characterize it. From all `fip`s observed during an execution in which a failure sequence is injected, we subtract out the `fip`s that are observed during normal program execution (that is, when no failure is injected) to obtain the `fip`s seen in the recovery path. More details about recovery path clustering can be found in Section 4.4.

```
$\lambda fs_1, fs_2$. (
    let rec($fs$) = allFips($fs$) $\setminus$ allFips(()) in
    let eq($fs_1$, $fs_2$) = (rec($fs_1$) == rec($fs_2$)) in
    let ($f_{11}$, ..., $f_{1m}$) = $fs_1$ in
    let $fs_{1P}$ = ($f_{11}$, ..., $f_{1(m-1)}$) in
    let ($f_{21}$, ..., $f_{2n}$) = $fs_2$ in
    let $fs_{2P}$ = ($f_{21}$, ..., $f_{2(n-1)}$) in
        (eq($fs_{1P}$, $fs_{2P}$) $\wedge$ ($f_{1m} == f_{2n}$) $\wedge$ ($m \geq 2$)
            $\wedge$ ($n \geq 2$))
)
```

Figure 5. Recovery path cluster. *Cluster two failure sequences if their last* `fit`*s are the same and their prefixes (that exclude the last* `fit`*s) result in the same recovery path.*

PreFail also enables composition of policies. For example, the policies that use the predicates in Figures 4 and 5 (`fpGen(flt)` and `cpGen(cls)`) can be composed to first filter out only those failure sequences that have crashes in the setup stage, and then to cluster the filtered sequences according to the recovery paths that they would lead to. Section 4 shows how to write the policies in Python in PreFail, and also gives many other examples of policies.

3.5 Test Workflow Algorithm

Having outlined the major components of PreFail, this section presents the detailed algorithm of PreFail's test workflow (Algorithm 3). PreFail takes a system `Sys` to test, a list of tester-written predicates `Preds`, 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 ($0 \leq i \leq N$), the FI engine of PreFail executes the system `Sys` 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. Therefore, in step 0 the FI engine executes `Sys` and injects an empty sequence of failures, i.e. it does not inject any failure. The FI engine observes the `fip`s that are seen during execution, computes `fit`s from them, and adds singleton failure sequences with these `fit`s

Algorithm 3 PreFailTest Workflow

```
INPUT: System under test (Sys), List of flt and cls predicates (Preds), Maximum number of failures per execution ($N$)
$FS_c$ = {()}
$FS_n$ = {}
for $0 \leq i \leq N$ do
    for each failure sequence $fs$ in $FS_c$ do
        Execute Sys and inject $fs$ during execution
        Profile execution using fips observed during execution
        for each fit $f$ computed from a fip in postInjectionFips($fs$) do
            $fs'$ = Append $f$ to $fs$
            $FS_n$ = $FS_n \cup fs'$
        end for
    end for
    $FS_n$ = Prune(Preds, $FS_n$)
    $FS_c$ = $FS_n$
    $FS_n$ = {}
end for
```

Algorithm 4 Prune(Preds, FS)

```
$FS_P$ = $FS$
for predicate pr in Preds do
    if pr is a filter predicate then
        p = fpGen(pr)
    end if
    if pr is a cluster predicate then
        p = cpGen(pr)
    end if
    $FS_P$ = p($FS_P$)
end for
return $FS_P$
```

to FS_n. Therefore, FS_n has failure sequences that the FI engine can exercise in the next step, i.e. in the $i = 1$ step. Before PreFail proceeds to the next step, it prunes down the set FS_n using the predicates written by testers. The predicates in Preds are used to generate policies that are then applied to FS_n (Algorithm 4). The policy generated from the first predicate is applied first to FS_n, the second policy is then applied to the result of the first policy and so on. Note that the order of predicates is important since the policies generated from them may not commute. In step $i = 1$, FS_c is set to the pruned down FS_n from the previous step, and FS_n is reset to the empty set. For each failure sequence fs in FS_c, the failure-injection tool executes Sys and injects fs during execution. For each fit f that it computes from a fip observed after fs has been injected (that is, a fip in postInjectionFips(fs)), it generates a new failure sequence fs' by appending f to fs, and adds fs' to FS_n. After Sys has been executed once for each failure sequence in FS_c, PreFail prunes down the set FS_n with predicates and moves to the next step. This process is repeated till the last step.

4. Crafting Pruning Policies

In this section, we present the pruning policies that we have written and their advantages. More specifically, we present our integration of PreFail to Hadoop File System (HDFS) [42], an underlying storage system for Hadoop MapReduce [1], and show the policies that we wrote for it. We begin with an introduction to HDFS and then present the policies.

Overall, we make three major points in this section. First, by clearly separating the failure-injection mechanism and policy and by providing useful abstractions, we can write many different pruning policies 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. We chose Python as the language in which testers can write policies in PreFail, though any other language could have also been chosen.

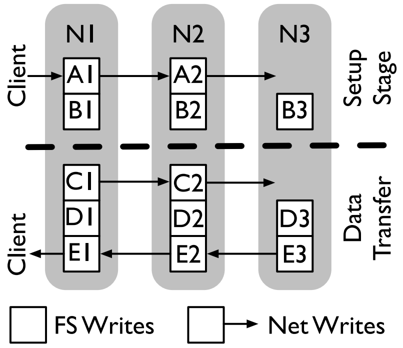
4.1 HDFS Primer

HDFS is a distributed file system that can scale to thousands of nodes. Here we describe the HDFS write protocol in detail. Figure 6 shows a simplified illustration of the write I/Os (both file system and network writes) occurring within the protocol. The protocol by default stores three replicas in three nodes, and is divided mainly into two stages: the setup stage and the data transfer stage; later, we will see how the recovery for each stage is different.

Our FI engine is able to emulate hardware failures on every I/O (every box in Figure 6). As illustrated, there are 13 failure points that the FI engine interposes in this write protocol. (Note that, in reality, the write protocol performs more than 40 I/Os). At every I/O, the FI engine can inject a crash, a disk failure (if it's a disk I/O), or a network failure (if it's a network I/O). The figure also depicts many possible ways in which multiple failures can occur. For example, two crashes can happen simultaneously at failure-injection points B1 and B2, or a disk failure at D1 and a network failure at E3, and many more. Interested readers can learn more about HDFS from here [42, 46] and our extended technical report (which depicts the write protocol in more detail) [29].

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 two failures on two nodes. Then there are $\binom{N}{2}$ failure sequences that one could inject. Worse, on every node (as depicted in Figure 6), there could be many possible points to exercise the failure on that node.



I/O #	Notes
A	Forward setup message to downstream nodes.
	(The last node does not need to forward the setup message)
B	After receiving setup message, create temporary block and meta files.
C	Stream data bytes to downstream nodes.
	(The last node does not need to stream the data bytes)
D	Write bytes to the block and meta files.
E	Send commit acks to upstream nodes.

Figure 6. HDFS Write Protocol. *The figure presents a simplified illustration of the HDFS write protocol. Each box represents an I/O (file system or network), and thus a failure-injection point. For ease of reading, we label each failure-injection point with an alphabetical symbol plus the node ID. The protocol begins with the client forming a pipeline (Client-N1-N2-N3) to the three nodes where replicas of a file will be stored. The client obtains these target nodes from the master (communication between client and master is not shown). For simplicity, we don't show many other I/Os such as other acknolwedgment and disk I/Os. We also do not show the rack-aware placement of replicas.*

```
def cls (fs1, fs2):
  rs1 = abstractOut(fs1, ‘node’)
  rs2 = abstractOut(fs2, ‘node’)
  return (rs1 == rs2)
```

Figure 7. 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. The function `abstractOut` removes the mappings for nodes from the `fip`s in its argument failure sequence.*

To reduce the number of failure sequences to test, a developer might just wish to inject failures at all possible failure-injection points in *any* two nodes. She can write a cluster policy that uses the function in Figure 7 to cluster failure sequences that have the same context when the node is not considered as part of the context. With this policy, the developer can 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. Using Figure 6 as an example, a failure sequence with simultaneous crashes at D1 and D2 is equivalent to another with crashes at D2 and D3.

We also want to emphasize that this type of pruning policy could be used for other systems. Consider a RAID system [37] with N disks that a tester wishes to test by injecting failures at any two of its 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 only difference would be in the keys in the `fips` whose mappings we want to remove (*i.e.*, for distributed systems we removed the mappings for the ‘node’ key in Figure 7, for RAID systems we remove the mappings for ‘disk’ key).

```
def flt (fs):
  last = FIP (fs [ len(fs) - 1 ])
  return not explored (last, ‘loc’)
```

Figure 8. New source location filter. *Return true if the source location of the last `fip` has not been explored. The function FIP returns the `fip` in the argument `fit`.*

```
def cls (fs1, fs2):
  last1 = FIP (fs1[len(fs1) - 1])
  last2 = FIP (fs2[len(fs2) - 1])
  return (last1[‘loc’] == last2[‘loc’])
```

Figure 9. Source location cluster. *Return true if the `fip`s in the last `fit`s have the same source location.*

4.3 Pruning via Code-Coverage Objectives

Developers can achieve high-level testing objectives using policies. One common objective in the world of testing is to have some notion of "high coverage". In the case of failure testing, we can write 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 `flt` function shown in Figure 8 and the `cls` function shown in Figure 9. The filter policy explores failures at previously unexplored source locations by filtering out a failure sequence if the `fip` in its latest `fit` (`last`) has an unexplored source location. The function `FIP` in Figure 8 returns the `fip` in the argument `fit`. The function `explored` returns true if a failure has already been injected at the source location in the last `fip` in a previous failure-injection experiment. For brevity, we do not show the source code of

fit	Recovery Path (Fig. 10)	SL	SL+N
A1$_c$	{ABCDE}×{234}	□	□
B2$_c$	{ABCDE}×{134}	□	△
C1$_c$	{FGI}×{23}, {CDE}×{23}	■	■
C2$_c$	{FGJ}×{13}, {CDE}×{13}	●	●
D1$_c$	{FG}×{23}, {CDE}×{23}	○	○
E2$_c$	{FG}×{13}, {CDE}×{13}	○	▽

Table 2. HDFS Write Recovery. *The table shows the detailed recovery I/Os of some* `fit`*s within the HDFS write protocol. The first column shows the* `fit`*s. A1$_c$ is the* `fit` *for crash at the I/O A1. (For simplicity, we do not distinguish here between an I/O and the failure-injection point that corresponds to the execution of the I/O). The second column shows the recovery paths returned by the* `getRecoveryPath` *function (Figure 10) for every* `fit` *shown in the first column*[1]*. To save space, we use ×; {AB}×{12} represents the I/Os A1, A2, B1, and B2. The third and fourth columns represent two ways of characterizing the recovery path; the same shape represents the same class of recovery path. For example, the third column represents the characterization shown in Figure 11 which uses source location (SL) to characterize recovery. The fourth column uses source location and node ID (SL+N) to characterize recovery.*

these functions. The `cls` function in Figure 9 clusters failure sequences that have the same source location in their last `fit`s. Thus, after the filter policy has filtered out failure sequences that have unexplored source locations, the cluster policy would cluster the failure sequences with the same unexplored source location into one group. With these policies, PREFAIL would exercise a failure sequence for each unexplored source location.

4.4 Pruning via Recovery-Coverage Objectives

In failure testing, since we are concerned with testing the correctness of *recovery* paths 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 leading to the same recovery path into a single class. PREFAIL can then use this policy to exercise a failure sequence from each cluster, and thus exercise a different recovery path with each failure sequence. Below, we first describe the HDFS write recovery protocol, and then explain the whole process of recovery-coverage based pruning in two steps: characterizing recovery path, and clustering failure sequences based on the recovery characterization.

```
1  def getRecoveryPath (fs):
2    a = allFips(fs)
3    a0 = allFips([])
4    rPath = a - a0
5    return rPath
```

Figure 10. Obtaining Recovery Path FIPs. *Line 2 uses the function* `allFips` *(§3.3) to get the set of all* `fip`*s,* `a`*, observed during the execution in which* `fs` *is injected. Line 3 obtains the set of* `fip`*s observed when no failure is injected (represented by "[]"). Line 4 performs the "diff" of the two sets to obtain the* `fip`*s in the recovery path taken when* `fs` *is injected.*

4.4.1 HDFS Write Recovery

As mentioned before, 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. Table 2 shows in detail the recovery I/Os, that is, the I/Os that occur during execution while recovering from an injected failure (or failure sequence). We will gradually discuss the contents of the table in the following sections. In the setup stage, if a node crashes, the recovery protocol will repeat the whole write process again with a new pipeline. For example, in the first row of Table 2, after `N1` crashes at I/O A1 (A1$_c$), the protocol executes the entire set of I/Os again (ABCDE) in the new pipeline (`N2-N3-N4`). However, if a node crashes in the second stage, the recovery protocol will only repeat the second stage with some extra recovery I/Os on the surviving datanodes. For example, in the fifth row of Table 2, after `N1` crashes at D1 (D1$_c$), the protocol first performs some synchronization I/Os (FG), and then repeats the second stage I/Os (CDE) on the surviving nodes (`N2` and `N3`).

4.4.2 Characterizing Recovery Path

To write a recovery clustering policy, a tester has to first decide how to characterize the recovery path taken by a system. One way to characterize would be to use the set of `fips` observed in the recovery path. Figure 10 returns the "difference" of the `fips` observed in the execution in which a failure sequence is injected and the `fips` in the execution in which no failure is injected. The difference can be thought of as the `fips` that are observed in the "extra" execution that results or the recovery path that is taken when the failure sequence is injected.

A tester can use the set of `fips` observed in the recovery path to characterize the recovery path. Thus, two failure sequences that result in the same set of `fips` in the recovery path are considered to be equivalent. Instead of using all of

[1] The reader might wonder why the I/Os A, B, C, D, and E appear again in the recovery paths even though the `getRecoveryPath` function returns the "diff" between the I/Os in the execution with failures and in the normal execution path, and thus should exclude those I/Os. The answer is that these I/Os are executed in the recovery path too, but with different contexts (e.g. different message content, different generation number) that we incorporate in the `fip`. For simplicity, we do not discuss these detailed contexts here.

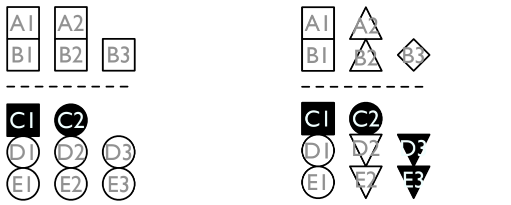
```
def eqvBySrcLoc (fs1, fs2):
  r1 = getRecoveryPath (fs1)
  r2 = getRecoveryPath (fs2)
  c1 = abstractIn(r1, ‘loc’)
  c2 = abstractIn(r2, ‘loc’)
  return c1 == c2
```

Figure 11. Equivalence of recovery paths. *Return true if two failure sequences result in the system executing I/Os at the same set of source locations during recovery. The function* `abstractIn` *retains only the mappings for the source locations (‘loc’) in the* `fips` *in its argument set.*

the context in the `fips`, the tester might abstract out the `fips` and use only part of the context in them to characterize a recovery path. For example, the tester might want to use only the source locations of `fips`. Thus, she might consider two recovery paths to be the same if the I/Os in them occur at the same set of source locations. The function in Figure 11 considers this relaxed characterization of recovery paths. Thus, in PreFail, a tester has the power and flexibility to decide how to characterize and cluster recovery paths.

If we use the equivalence function in Figure 11 to cluster failure sequences that result in the same recovery path into the same class, then we would obtain four different equivalence classes for the HDFS write protocol. The third column in Figure 2 shows the four classes: □, ■, ●, and ○ which represent the recovery paths {ABCDE}, {CDEF}, {CDEG}, and {CDE} respectively. Note that the recovery paths of $A1_c$ and $B2_c$ are considered to be equivalent (□) as they have I/Os at the same set of source locations {ABCDE} even if the I/Os are executed in different nodes. However, if the tester decides to characterize recovery paths using both source location and node ID, then the recovery paths of $A1_c$ and $B2_c$ would be considered to be different (□ and △), as shown in the last column in Table 2.

Figure 12 provides more details of how different I/Os shown in Figure 6 are grouped into different recovery classes. The left figure shows 4 recovery classes that result from the use of only source location to distinguish between different recovery paths. Even by just using source location, PreFail is able to distinguish between the two main recovery classes in the protocol (□ and ○). Furthermore, PreFail also finds two unique cases of failures that result in two more recovery classes (■ and ●). In the first one (■), a crash at C1 leaves the surviving nodes (`N2` and `N3`) with zero-length blocks, and thus the recovery protocol executes I/Os at a different source location (labeled with I in Table 2). In the second one (●), a crash at C2 leaves the surviving nodes (`N1` and `N3`) with different block sizes (the first node has received the bytes, but not the last node), and thus I/Os at yet another different source location (labeled as J) are executed.



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

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

Figure 12. Recovery Classes of HDFS Write Protocol. *The symbols (e.g., A1, A2) represent I/Os described in Figure 6. A shape (e.g., □) surrounding an I/O #X represents the equivalence class of the I/O with regard to the recovery path that is taken by HDFS when a crash occurs at that I/O. Different shapes represent different equivalence classes. The two figures show how the I/Os are grouped differently into equivalence classes when recovery paths are characterized in different ways (e.g., (a) using source location only and (b) using source location and node ID).*

```
1 def cls (fs1, fs2):
2   last1 = fs1 [ len(fs1) - 1 ]
3   last2 = fs2 [ len(fs2) - 1 ]
4   prefix1 = fs1 [ 0 : len(fs1) - 1 ]
5   prefix2 = fs2 [ 0 : len(fs2) - 1 ]
6   isEqv = eqvBySrcLoc (prefix1, prefix2)
7   return isEqv and (last1 == last2)
```

Figure 13. Equivalent-recovery clustering. *Cluster two failure sequences if their prefixes (that exclude the last* `fits`*) result in the same recovery path and their last* `fits` *are the same. Line 6 uses the* `eqvBySrcLoc` *function in Figure 11 to compute the equivalence of the recovery paths of the prefixes.*

Figure 12b shows the 8 recovery classes that result when node ID is used in addition to the source location to characterize recovery paths. If the tester uses all of the context present in a `fip`, the I/Os in the write protocol will be grouped into 10 recovery classes. Interested readers can find the explanations behind the different numbers of recovery classes in [29]. In general, the more context information in `fips` considered, the more we can distinguish between different recovery paths, and hence the more the number of recovery classes of I/Os. Lesser context leads to fewer recovery classes and thus fewer failure-injection experiments, but might miss some corner-case bugs.

4.4.3 Clustering Failure Sequences

After specifying the characterization of a recovery path, the tester can simply write a cluster policy that uses the `cls` function in Figure 13. Given this policy, if there are two failure sequences, `(prefix1, last)` and `(prefix2, last)`, such that `prefix1` and `prefix2` result in the same recovery path, then PreFail will exercise only one of the two sequences.

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

Figure 14. Generic crash 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 FI engine.*

To illustrate the result of this policy, let's consider the example in Table 2. The fit F_c (crash at I/O F) can be exercised after any of the crashes at $\{DE\} \times \{123\}$ (*i.e.*, 6 fits). Without the specified equivalent-recovery clustering, PREFAIL will run 6 experiments ($D1_cF_c$.. $E3_cF_c$). But with this policy, PREFAIL will group all of the 6 failure sequences into a single class ($D1_c$/../$E1_c$ + F_c) as all the prefixes have the same recovery class (○, as shown in Figure 12), and thus will run only 1 experiment to exercise any of the 6 failure sequences. If the tester changes the clustering function such that it uses both source location and node ID to characterize a recovery path (Figure 12b), then PREFAIL will run three experiments as the prefixes now fall into three different recovery classes (○, ▽, and ▼).

4.5 Pruning via Optimizations

In general, 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 disk failures, there are optimizations that can be performed to eliminate unnecessary failure-injection experiments. In the following sections, we present policies that implement optimizations for crashes and disk failures in distributed systems. Appendix A describes the optimizations for network failures and disk corruption. By reducing the number of individual failure-injection tasks, these optimizations also help in reducing the number of multiple-failure sequences.

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 `flt` function that can be used to implement this optimization.

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 `flt` function that implements this optimization is shown in Figure 16 in Appendix A.

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 that is explained in Appendix A (Figure 17).

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 that specifying this type of policy in PREFAIL will be straightforward. For example, the tester can write a policy as simple as: return true if $\texttt{prob}(fs) > 0.1$. That is, inject a failure sequence fs 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 three popular "cloud" systems: HDFS [42] v0.20.0, v0.20.2+320, and v0.20.2+737 (the last one is a release used by Cloudera customers [13]), ZooKeeper [27] v3.2.2 and v3.3.1, and Cassandra [33] v0.6.1 and v0.6.5. These integrations show that it is easy to port our tool to real-world systems and releases. We evaluate PreFail on four HDFS workloads (log recovery, read, write, and append), two Cassandra workloads (key-value insert and log recovery), and one ZooKeeper workload (leader election). In this work, we focused more on Cloudera's HDFS, and thus present extensive evaluation numbers for it. We present partial results for other releases.

5.2 Bugs Found

With PreFail, we were able to find all of the 16 bugs in HDFS v0.20.0 that we had reported in our previous work [22]. 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 more previously unknown 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 tool. These bugs are basically availability (*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 report 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).

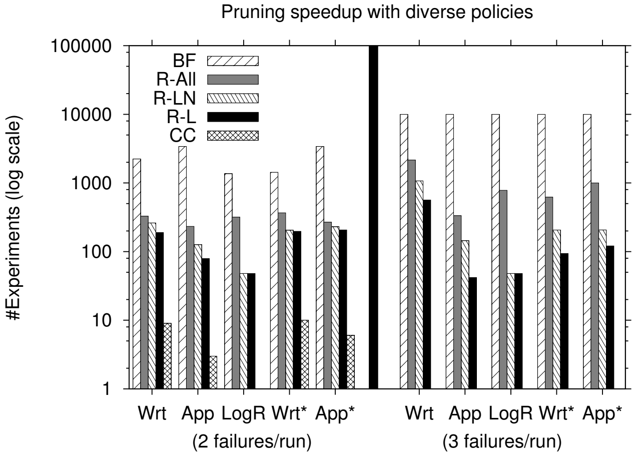


Figure 15. #Experiments run with different coverage-based policies. *The y-axis shows the number of failure-injection (just crash-only failure) experiments for a given policy and a workload. The x-axis shows the workloads: the write (Wrt), append (App), and log recovery (LogR) protocols from Cloudera's 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 `fip` (All). We stopped our experiments when they reached 10,000 (Hence, the maximum number of experiments is 10,000).*

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 15 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

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

Table 3. #Bugs found. *The table shows the number of failed experiments (#Failed Exps) for a given workload using the simplest recovery clustering policy (R-L in Figure 15) and the number of crashes per run (#F), along with the actual number of bugs that trigger the failed experiments (#Bugs). The last column (#Bugs$_R$) is the number of bugs that can be found using randomized failure injection. (*) implies that these are the same bugs (i.e., bugs in 2-failure cases often appear again in 3-failure cases).*

relaxed the recovery characterization, the lesser the number of experiments (*e.g.*, R-L vs. R-All).

Pruning is not beneficial 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.

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. The last column in the table shows the number of bugs that we can find by using randomized failure injection, that is, by randomly choosing the execution points at which to inject crashes. For each workload, we execute the system as many times as we do for the recovery clustering policy, and randomly inject crashes in each execution. Randomized failure injection can find the bugs for the write and append workloads, but not for the log recovery workload. This is because the bugs for the log recovery workload are corner-case bugs; the proportion of failure sequences that lead to a log recovery bug is much smaller than that for a write or an append bug. This shows that randomized failure injection, though simple to implement, is not effective in finding corner-case bugs that manifest only in specific failure scenarios.

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.*

5.3.2 Optimization-Based Policies

Table 4 shows the effectiveness of the optimizations of different failure types that we described in Section 4.5. The optimizations for network failures and data corruption are shown in Appendix A. 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.

5.4 Complexity

The FI engine is based on our previous work [22], which is written in 6000 lines of Java code. We added around 160 lines of code to this old tool so that it passes on appropriate failure and execution abstractions 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`, `fips` 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 Python code.

6. Limitations

PreFail does not control all kinds of non-determinism present in a system execution (*e.g.*, network message ordering). Therefore, two executions of the same system against the same workload might be different, and PreFail might not be able to inject a failure sequence that seemed possible to inject from a previous system execution. In future work, we plan to control the non-determinism that arises out of network message ordering and also expose it to testers and provide support for writing policies that can express the message orderings to test for.

7. 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 provide some language support for specifying failure-injection tasks, and present techniques to prune down large failure spaces.

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 [25]. FIG also uses a domain-specific language to inject failures at library level [8]. Orchestra uses TCL scripts to inject failures at TCP level [16]. Genesis2 uses a scripting language to specify service-level failures [30]. LFI uses an XML-based language to trigger failures at library level [35]. 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 code *from scratch* to build 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 [11], Yahoo! [44], Microsoft [47], Amazon [24], and other places [26]. 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 [23]. 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 [36] 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 [31] 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.* [19] 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.

The multiple-failure combinatorial explosion problem is similar to the state explosion problem in model checking. Existing system model-checkers [47, 48] 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.

There has been some work in program testing [7, 15, 21] that uses tester-written specifications or input generators to produce all non-isomorphic test inputs bounded by a given size. The specifications or generators can be thought of as being analogous to the tester-written pruning policies in PREFAIL, and the process of generating inputs from them by pruning down the input space can be thought of as being analogous to the process of pruning down the failure space using policies. The specifications are used only for the purpose of generating test inputs, and there is no support to address failures in the specifications.

8. Conclusion

We have presented PREFAIL, a programmable failure-injection tool that provides appropriate failure abstractions and execution profiles to let testers write a wide variety of policies to prune down large spaces of multiple-failure combinations. Currently, we are 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.

9. Acknowledgments

This material is based upon work supported by the NSF/CRA Computing Innovation Fellowship and the National Science Foundation under grant numbers CCF-1016924, CNS-0720906, CCF-0747390, CCF-1018729, and CCF-0747390. The third author is supported in part by an Alfred P. Sloan Foundation Fellowship. We also thank Eli Collins and Todd Lipcon from Cloudera Inc. for helping us confirm the HDFS bugs that we found. Any opinions, findings, 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] Hadoop MapReduce. http://hadoop.apache.org/mapreduce.

[2] Seattle, Washington, November 2006.

[3] San Jose, California, February 2007.

[4] Indianapolis, Indiana, June 2010.

[5] Boston, Massachusetts, June 2010.

[6] 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.

[7] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated Testing Based on Java Predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '02)*, pages 123–133, Rome, Italy, July 2002.

[8] 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*.

[9] 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)* [2].

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

[11] 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.

[12] 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)* [2].

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

[14] 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)* [4].

[15] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated Testing of Refactoring Engines. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '07)*, Dubrovnik, Croatia, September 2007.

[16] 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.

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

[18] 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.

[19] 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.

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

[21] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*, pages 225–234, Cape Town, South Africa, May 2010.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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)* [5].

[28] 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.

[29] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. PREFAIL: A Programmable Failure-Injection Framework. UC Berkeley Technical Report UCB/EECS-2011-30, April 2011.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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)* [5].

[37] 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.

[38] 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)* [3].

[39] 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.

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

[41] 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)* [3].

[42] 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.

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

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

[45] Kashi Vishwanath and Nachi Nagappan. Characterizing Cloud Computing Hardware Reliability. In *Proceedings of the 2010 ACM Symposium on Cloud Computing (SoCC '10)* [4].

[46] Tom White. *Hadoop The Definitive Guide*. O'Reilly, 2009.

[47] 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.

[48] 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.

A. Appendix

We explain the optimizations that we perform to eliminate redundant failure-injection experiments for network failures and disk corruption.

A.1 Network Failures

For network failures, we can perform an optimization similar to disk failures (Section 4.5.2). 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

```
def flt (fs):
 for i in range(len(fs)):
   fp = FIP(fs[i])
   isNet = (fp[‘ioTarget’] == ‘net’)
   isWrite = (fp[‘ioType’] == ‘write’)
   isCrash = (fp[‘failure’] == ‘crash’)
   rNode = fp[‘receiver’]
   pfx = fs[0:i]
   if isNet and isWrite and isCrash and
      nodeAlreadyCrashed(pfx, rNode):
      return False
 return True
```

Figure 16. Crash optimization for network writes. *The function accepts a failure sequence if for each crash 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.*

```
def flt (fs):
 for i in range(len(fs)):
   fp = FIP(fs[i])
   isNetFail = (fp[‘failure’] ==
                        ‘netfail’)
   isRead = (fp[‘ioType’] == ‘read’)
   sender = fp[‘sender’]
   node = fp[‘node’]
   thread = fp[‘thread’]
   time = fp[‘time’]
   pfx = fs[0:i]
   allFS = allFitSeqs()
   if isNetFail and isRead and
      (not first(pfx, node, thread,
            time, sender, allFS)):
      return False
 return True
```

Figure 17. Network failure optimization. *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* `fip` *records the time when the* `fip` *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* `allFS` *passed on by the FI engine.*

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. Also, we do not inject a network failure if one of the nodes participating in the message is already dead. Figure 17 shows the `flt` function that can be used to implement the optimization for network failures.

A.2 Data 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 (Section 4.5.2).