Failure Scenario as a Service (FSaaS) for Hadoop Clusters

Faraz Faghri
University of Illinois at Urbana-Champaign, USA
faghri2@illinois.edu

Sobir Bazarbayev
University of Illinois at Urbana-Champaign, USA
sbazarb2@illinois.edu

Mark Overholt
University of Illinois at Urbana-Champaign, USA
overhol2@illinois.edu

Reza Farivar
University of Illinois at Urbana-Champaign, USA
farivar2@illinois.edu

Roy H. Campbell
University of Illinois at Urbana-Champaign, USA
rhc@illinois.edu

William H. Sanders
University of Illinois at Urbana-Champaign, USA
whs@illinois.edu

ABSTRACT

As the use of cloud computing resources grows in academic research and industry, so does the likelihood of failures that catastrophically affect the applications being run on the cloud. For that reason, cloud service providers as well as cloud applications need to expect failures and shield their services accordingly. We propose a new model called *Failure Scenario as a Service (FSaaS)*. FSaaS will be utilized across the cloud for testing the resilience of cloud applications. In an effort to provide both Hadoop service and application vendors with the means to test their applications against the risk of massive failure, we focus our efforts on the Hadoop platform. We have generated a series of failure scenarios for certain types of jobs. Customers will be able to choose specific scenarios based on their jobs to evaluate their systems.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*Fault tolerance*; D.4.3 [**File Systems Management**]: Distributed File Systems

General Terms

Design, Performance, Reliability

Keywords

Cloud computing, MapReduce, Hadoop, failure injection

1. INTRODUCTION

Cloud computing has recently become a mainstream commodity in the IT industry [21]. Consequently, the impact on overall system dependability of equipment failure or software bugs in the cloud infrastructure has increased. Applications not only need to prepare for occasional failures of their cloud infrastructure, they need to expect different types of failures (ranging from hard drive errors to crash of whole racks) and combinations of them as part of an application's normal operating procedure [7]. Combinations of failures may have a major impact on performance of an application, sometimes even leading to applications being temporarily out of service [13]. Unfortunately, the use of small sandbox-testing models is not sufficient to predict the effects of failures on realistic applications running on large cloud-based infrastructures.

A few teams have proposed failure injection as a possible way to address the reliability challenges of cloud infrastructures. The first publicized system developed specifically for failure injection in cloud computing infrastructure is the "Chaos Monkey" (ChM) from Netflix [5], which was recently open-sourced (July 2012). In ChM, a simple failure model (namely a whole node failure) is simulated across a virtual cluster running in the Amazon Elastic Compute Cloud (Amazon EC2). The model injects a continuous sequence of failures into the Amazon auto-scaling group service, causing frequent failures of different nodes within the infrastructure in order to identify weaknesses in the applications that run on it.

Later, Gunawi et al. introduced "Failure as a Service" (FaaS) as an approach to the challenge, and proposed a basic design for such a service [15]. Their motivation stemmed from the ChM system, and they intended to build upon that idea.

Both of the previous efforts simulated certain failure modes, for example, crashing of a node. However, real-world faults and errors result in many diverse failure scenarios, potentially including combinations of different failure modes [10]. In addition, many failures in a cloud environment tend to cascade from being small problems to being large problems very quickly [13]. Therefore, we believe that a system designed to assess the real-world dependability of a cloud-based system is needed, and that a failure system should be able to handle complex fault and error injection scenarios and also simulate combinations of different failure modes.

In this paper, we introduce a *Failure Scenario as a Service (FSaaS)* model, designed and implemented for Hadoop clusters. Our FSaaS model can currently be used by cloud service providers and clients who rely on Hadoop MapReduce clusters. By targeting Hadoop, we aim to provide FSaaS services to a wide spectrum of users. Consider that an average of one thousand MapReduce jobs were executed on Google's clusters every day in 2008, and that more than ten thousand

SDMCMM'12, December 3-4, 2012, Montreal, Quebec, Canada.

distinct MapReduce programs were implemented internally at Google in a four-year period [12]. There exist many types of Hadoop workloads [17], and this paper shows that for the workloads we study, they behave very differently under various failure scenarios. Because it is difficult to design general failure scenarios that fit all types of Hadoop workloads, we profile the behavior of several different Hadoop workload types against failures and generate a series of template failure scenarios that have high impact on these particular job types. As an example, in Section 4 we compare data-intensive application versus CPU-intensive application workload behavior under different failures and find that the behaviors are different in important ways. Using those scenarios, we are creating an FSaaS model that would allow users of Hadoop clusters to pick the proper template failure scenario to run against their applications. Failures in the model are randomized and are synthetically introduced into components, allowing a Hadoop application to see a series of failures that simulate real-world failures. Our tools can also inject real faults into running Hadoop applications, as a mechanism to best mimic real-world dependability challenges.

The rest of this paper is arranged as follows. In Section 2, we will talk about related work in this area and compare it to our own work. In Section 3, we will discuss our methodology and the design of the experiment we conducted. Section 4 describes experimental results. We conclude in Section 5.

2. RELATED WORK

When Netflix moved from their own data centers to Amazon Web Services, they developed Chaos Monkey [5] in order to evaluate how potential failures in AWS would affect their ability to provide continuous services. Chaos Monkey would kill EC2 instances to test how they affected overall services to clients. Netflix performed the tests in real scale, enabling them to find the bottlenecks in their system and areas where improvements were necessary. Because failures in data centers are common [7], this type of testing will be important for most organizations running their services on the cloud.

Dinu et al. performed an evaluation of Hadoop's performance under compute node and process failures [13]. They observed that even single failures had detrimental effects on running times of jobs. It was observed that several design decisions in Hadoop, such as delayed speculative execution (SE), the lack of sharing of failure information, and overloading of connection failure semantics, make Hadoop's performance sluggish in the presence of failures. Jin et al. derived a stochastic model to predict the performance of MapReduce applications under failures [18]. They generated synthetic data to run their MapReduce simulator to confirm the accuracy of their model.

[19, 16] present tools for efficient injection of failures into cloud software systems, like HDFS, and evaluation of cloud recovery. Cloud software systems like Hadoop include fault tolerance and failure recovery, but some failures may have unpredictable effects on performance of Hadoop [13]. Also, some failures may not be accounted for when failure recovery is implemented in cloud software systems; or failure recovery may even be buggy [13]. Hence, [19] is based on the need for state-of-the-art failure-testing techniques. The authors address the challenges of dealing with *combinatorial explosion of multiple failures* through their work in PreFail [19]. PreFail is a programmable failure injection tool that provides failure abstractions to let testers write policies to prune down large spaces of multiple-failure combinations. The main goal of PreFail is to find reliability bugs in large-scale distributed systems. [16] presents a similar tool called Fate, a framework for cloud recovery testing. Fate is designed to *systematically* push cloud systems into many possible failure scenarios. Similar to [19], Fate aims to solve the challenge of massive combinatorial explosion of failures by implementing smart and efficient exploration strategies of multiple-failure scenarios. Fate achieves fast exploration of failure scenarios by using strategies that prioritize failure scenarios that result in distinct recovery actions. In summary, PreFail [19] and Fate [16] are tools that allow users to programmatically inject failures into cloud systems to analyze their failure recovery. They both aim to solve the challenge of combinatorial explosion of multiple failures, PreFail through pruning policies and Fate through prioritization strategies. However Fate is deployed in only three cloud systems (HDFS, ZooKeeper, and Cassandra) and their systematic behavior during failure scenarios. Instead, we focus on failure impact on jobs running on Hadoop as a more high-level framework. Our approach is to systematically analyze the MapReduce jobs rather than the frameworks.

In [15], the idea of Failure as a Service (FaaS) was introduced. The authors used Netflix's ChaosMonkey [5] and DevOps GameDay [6] as motivations for FaaS. Netflix's Chaos Monkey and DevOps have proven to be very effective in finding bottlenecks and evaluating system recovery from failures, but there are only a few large organizations that do such failure testing. Hence, the primary goal of our work is to provide FSaaS, a cloud service for performing large-scale online failure drills.

3. FSaaS DESIGN

Our goal is to implement FSaaS for Hadoop clusters by designing a set of failure scenarios that contain collections of effective failures for specific types of applications. FSaaS and the scenarios can be used by organizations as a Quality Control tool to improve their applications. Also, it can be used as a method of determining upper bounds on required resources for applications with possibilities of failure during operation, to satisfy service-level agreements (SLA).

We first identified a set of various common failures that may affect Hadoop jobs, described in Section 3.1. Then we evaluated effects of individual failures on different types of job, and evaluated the performance of the jobs against the failures and a combination of them. As a result, we have developed a set of sample scenarios (described in Section 3.2) for a few different types of workloads (Section 3.3).

To make efficient use of the FSaaS service, we allow users to find a failure template to evaluate their job by selecting a job type (I/O intensive, CPU intensive, or Network intensive) and appropriate matching failure scenario for their particular Hadoop job. The template injects the set of failures into a Hadoop cluster, and helps the users to identify weaknesses, bottlenecks, and hot spots in their application in the presence of failures.

Next we describe the failure injection framework, the different MapReduce applications representing MapReduce job types in our case study, the utilized evaluation metrics, and our experimental testbed.

3.1 Failure Injection Framework

We are using AnarchyApe [3] as our failure injection base code. AnarchyApe is an open-source project, created by Yahoo!, developed to inject failures in Hadoop clusters. About ten common failures have been implemented in AnarchyApe, and more can be added. Each failure in AnarchyApe is implemented as a failure command. In our templates, we execute these commands to inject the failures into nodes in the Hadoop cluster (see Section 3.2).

Here are some common failures in Hadoop environments, proposed in [3]:

- Data node is killed
- Application Master (AM) is killed
- Application Master is suspended
- Node Manager (NM) is killed
- Node Manager is suspended
- Data node is suspended
- Tasktracker is suspended
- Node panics and restarts
- Node hangs and does not restart
- Random thread within data node is killed
- Random thread within data node is suspended
- Random thread within tasktracker is killed
- Random thread within tasktracker is suspended
- Network becomes slow
- Network is dropping significant numbers of packets
- Network disconnect (simulate cable pull)
- One disk gets VERY slow
- CPU hog consumes x% of CPU cycles
- Mem hog consumes x% of memory
- Corrupt ext3 data block on disk
- Corrupt ext3 metadata block on disk

In our case studies, we have used a handful of these possible failures (Section 4).

3.2 Failure Scenarios

Currently, to create a scenario, the user constructs a shell script specifying the types of errors to be injected or failures to be simulated, one after another. A sample line in a scenario file could be as follows:

```
java -jar ape.jar -remote cluster-ip-list.xml
        -fb lambda -k lambda
```

where the `-fb` is a "Fork Bomb" injection, the `-k` is a "Kill One Node" command, and the `lambda` specifies the failure rates.

Users can define `lambda` parameters by computing Mean Time Between Failures (MTBF) of a system. MTBF is defined to be the average (or expected) lifetime of a system and is one of the key decision-making criteria for data center infrastructure systems [20]. Equipment in data centers is going to fail, and MTBF helps with predicting which systems are the likeliest to fail at any given moment. Based on previous failure statistics, users can develop an estimate of MTBF for various equipment failures; however, determining MTBFs for many software failures is challenging.

To evaluate our case studies in this paper, we have developed a set of ready-to-use failure scenarios. However, the easy programmability of FSaaS ensures its use in other use cases. Users can create user-defined scenarios; as an example, a user can categorize data center failures into two groups: equipmental failures and HDFS failures, and use their known failure rates or some publicly available reports in order to set `lambda` parameters. Some publicly available data for data center failures in Google can be find at [11] and for HDFS failures from Hortonworks at [8].

3.3 MapReduce Applications

At Google alone [12], more than ten thousand distinct MapReduce programs have been implemented; hence, for the FSaaS to be applicable to all types of programs, good program classifications are needed. We categorize MapReduce programs as network-(data transfer), CPU-(computation), or I/O-(local data access) intensive.

We have used the following representative applications for the above resource-intensive categories:

- **Sort** sorts a set of records that have been distributed into S partitions and is inherently network-intensive [17].
- **WordCount** computes the frequency of occurrence of each word in a large corpus and reports the ten most common words. WordCount performs extensive data reduction on its inputs, so it is CPU-intensive and transfers very little data over the network [17, 14].
- **RandomWriter** randomly chooses words from a small vocabulary (100 words) and forms them into lines in Map task. The map outputs are directly committed to the distributed file system, so there is no Reduce task in RandomWriter. RandomWriter is I/O-intensive [14].

3.4 Profiling

Since our testbed is set up on Amazon AWS [2], we have used the Amazon CloudWatch [1] service to monitor AWS cloud resources. Amazon CloudWatch monitors AWS resources such as Amazon EC2, and with it, one can gain system-wide visibility into resource utilization, application performance, and operational health [1].

3.5 Evaluation Metrics

We can use various metrics to measure the impacts of failures on different application types. As different applications have different properties, our FSaaS service can use more than one metric. Some commonly used metrics in large-scale networks are as follows:

- **Makespan:** total time taken by an experiment until the last job completes.
- **System Normalized Performance (SNP):** the geometric mean of all the ANP values for the jobs in an experiment, where ANP stands for the "Application Normalized Performance" of a job [22].
- **Slowdown norm:** some scaled l_p norms of the slowdown factors of the jobs across each experiment [17].
- **Data Transfer (DT):** the total amount of data transferred by all tasks during an experiment. DT is split into 3 components: data read from local disk, data transfer across a rack switch, and data transfer across the central switch.

In this paper we have used Makespan for evaluations. We intend to use additional metrics in future work.

Table 1: Amazon EC2 Instance Specs

Memory	1.7 GB
EC2 Compute Unit	1
Storage	160 GB
Platform	32-bit or 64-bit
I/O Performance	Moderate
API name	m1.small

3.6 Testbed Description

For our testbed, we used Amazon Web Services (AWS) [2] EC2 machines and ran MapReduce jobs through the AWS Elastic MapReduce service. AWS is a widely used Infrastructure as a Service (IaaS) for running cloud services and it simplifies the setting up of Hadoop clusters to run MapReduce jobs through the Elastic MapReduce service. AWS provides a highly scalable infrastructure, which is essential for our experiments.

We set up our Hadoop cluster with 10 AWS EC2 machines. Each machine was a standard small instance type. Table 1 gives details on that instance type.

4. EVALUATION

In this section, we present the results gathered from our experiments and discuss the data that led to the creation of the failure scenarios. As stated in Section 3, we gathered a variety of different failures that could happen inside a Hadoop cluster and injected them into different Hadoop jobs. The Hadoop jobs we selected represented three different types of workloads, namely I/O-intensive, CPU-intensive, and network-intensive. Those workloads put heavy pressure on different resources within a Hadoop cluster; hence, we can use them to identify better set of failures that have larger impacts in these different failure scenarios. Our results show that the behavior of each of the workloads under our failure scenarios varied significantly according to the type of workload. In our experiments, we chose the following three Hadoop jobs: Random Writer, Hadoop Sort, and Word Count. They correspond to I/O-intensive, network-intensive, and CPU-intensive workloads, respectively.

Initially, we observed the impacts of different failures on different Hadoop jobs. First, we implemented the following failures in AnarchyApe [3]: kill nodes, disconnect network for duration of period, slow down network for duration of period, fork bomb at a particular node, and drop network packets for duration of period at a certain rate. Second, we ran our three different types of Hadoop jobs many times, each time with one of the above failures injected. Our goal in the experiments was to determine the impact levels of the above failures on the different job types.

In Figure 1, bars "None" through "Packet Drop" show job completion times for Word Count, Random Writer, and Sort Hadoop jobs with injection of different failures. We can see that different failures had different levels of impact on the completion times of the Hadoop jobs. The "None" bars show the Hadoop job completion time with no failures. When no failures were injected, the Word Count job took, on average, about 40 minutes. For a "Kill One Node" injection, one of the Hadoop nodes was terminated during the execution. Killing one node during the execution did not affect the running time at all. This shows that for CPU-intensive jobs, Hadoop is well-built to recover from individual node

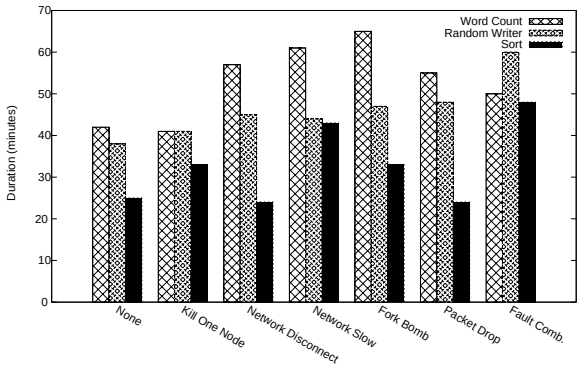

Figure 1: Job durations for Word Count, Random Writer, and Sort under different failure scenarios.

failures without loss of computation time. In a "Network Disconnect" failure, we brought down the network interface for a few minutes in several nodes. In the "Network Slow" failure, we slowed down network packets for a few milliseconds at a few nodes. In a "Fork Bomb" failure, we started a fork bomb at a few Hadoop nodes. Finally, in the "Packet Drop" failure, we dropped a particular percentage of packets at several Hadoop nodes. These additional failures had a significant impact on completion time, Fork Bomb being the worst culprit. For a network-intensive workload (Sort), Network Disconnect and Packet Drop had surprisingly little impact on job completion times. After further investigation, we discovered that the Sort job has many bursty network communications, so Packet Drop and Network Disconnect failures may not have a big impact unless they occur during times of bursty network communication. For the I/O-intensive workload, Random Writer, there were no failures that affected the job completion time significantly more than others. The last column shows the combination of the top three failures for each job.

Figure 2 shows the remaining map tasks measured against time for the Word Count, Sort, and Random Writer Hadoop jobs. The slope of the lines gives us some insight into how Hadoop handles failures as they are being injected. For Word Count, the lines are fairly level for the Network Slow and Network Disconnect failure injections, meaning that Hadoop did not do anything to help when those failures occurred, and that the failures caused a steady increase in completion time for each map task. The Fork Bomb data show exactly what we suspected. The more horizontal part of the graph shows that for a period of 7 minutes, no map tasks were completed, severely degrading the performance of the jobs. For Random Writer jobs, the remaining map tasks were affected differently by the different failure injections, but they all ended up leveling out to similar results towards the end.

After identifying failures with the most impact on each of the different jobs, we proceeded to inject combinations of failures into each of the jobs. For each job (Word Count, Random Writer, and Sort), we selected three of the failures with the most impact on that job and injected combinations of these three failures to observe the resulting behavior. For Word Count jobs, we injected a combination of Network Disconnect, Network Slow, and Fork Bomb failures. For Random Writer, we injected Network Disconnect, Fork Bomb, and Packet Drop. Finally, for Sort jobs, we injected Kill

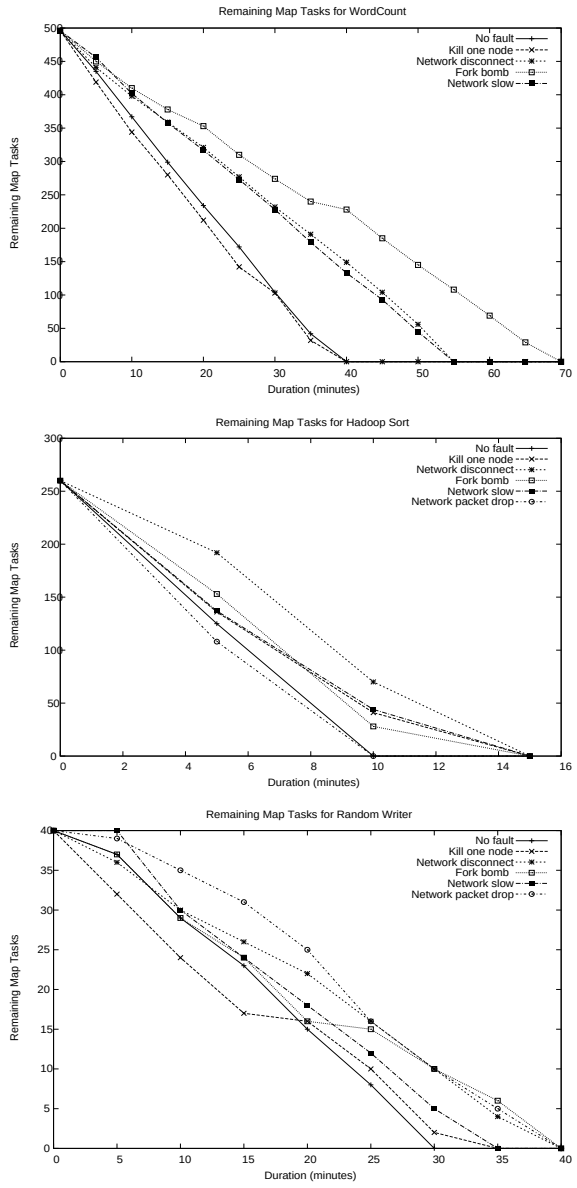

Figure 2: Remaining Map tasks for Word Count, Sort, and Random Writer.

Node, Network Slow, and Fork Bomb failures. Interestingly, the Word Count job completed in less time compared to completion times under the three failures injected individually and separately. The most reasonable explanation is that failures have different impacts depending on the times they are injected in relation to the status of the running jobs, which relates to the Hadoop job phases at the moment of injection. On the other hand, and as expected, the Random Writer and Sort jobs took longer to complete under the combination of failures than under any individual failures.

In review, even though we did not experiment with a huge number of applications, our study does show that application behavior varies considerably, depending on the type of job and the phase of the job that is executing when a failure occurs.

5. CONCLUSION AND FUTURE WORK

Failure in the cloud is a normal occurrence now, and ensuring that an application can withstand large and widespread failure is essential for the longevity of the service. Hadoop has seen many different types of applications across industry and academia, and, like other cloud services, it needs to be tested against different types of failures. Running an FSaaS against Hadoop applications can help to identify failure vulnerabilities in those applications, allowing developers to fix them and provide better service.

Services running in the cloud are difficult to test using traditional methods. Having a background service continually causing different parts of the infrastructure to fail would go a long way towards identifying faults in a running application on a production environment. One of the reasons it is difficult to test at such a large scale is that the number of failures that can occur at any one time is very large. We have presented a system for profiling Hadoop applications in an attempt to narrow down the testing strategy and allow for a Hadoop application to be failure-tested as efficiently as possible. Such testing can serve two purposes. One is to identify weak spots in an application and attempt to fix them. The other is to identify the quantity of running cloud resources you need to stave off complete failure in the event of isolated system crashes.

Much future work is needed in this space. A long-term study of a real running Hadoop application would be necessary to completely performance-test our application and workload profiles. Following such s study, a real live system could be deployed and given to all users of public Hadoop clouds, such as Amazon's Elastic Map Reduce. Our eventual goal is to develop our system into a general cloud-based failure scenario execution engine, suitable for many different cloud usage scenarios. (Our immediate future plans include extension of support to cloud-based databases like HIVE [4], and streaming systems like STORM [9]).

6. ACKNOWLEDGEMENTS

The authors would like to thank Nathan Roberts, Robert Evans, and Mark Holderbaugh of Yahoo! for their guidance and advice in the early stages of the project. They provided guidelines, support, and invaluable assistance in getting the project started.

7. REFERENCES

[1] Amazon CloudWatch. `http://aws.amazon.com/cloudwatch/`.
[2] Amazon Web Services. `http://aws.amazon.com/`.
[3] AnarchyApe. `https://github.com/yahoo/anarchyape`.
[4] Apache Hive. `http://hive.apache.org/`.
[5] ChaosMonkey. `http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html`.
[6] DevOps GameDay. `https://github.com/cloudworkshop/devopsgameday/wiki`.
[7] Failure rates in Google data centers. `http://www.datacenterknowledge.com/archives/2008/05/30/failure-rates-in-google-data-centers`.
[8] Hortonworks: Data integrity and availability in Apache Hadoop HDFS.

http://hortonworks.com/blog/data-integrity-and-availability-in-apache-hadoop-hdfs/.

[9] Storm. https://github.com/nathanmarz/storm.

[10] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, Jan. 2004.

[11] J. Dean. Software engineering advice from building large-scale distributed systems. http://research.google.com/people/jeff/stanford-295-talk.pdf.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[13] F. Dinu and T. E. Ng. Understanding the effects and implications of compute node related failures in Hadoop. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 187–198. ACM, 2012.

[14] Y. Geng, S. Chen, Y. Wu, R. Wu, G. Yang, and W. Zheng. Location-aware mapreduce in virtual cloud. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 275–284, Washington, DC, USA, 2011. IEEE Computer Society.

[15] H. S. Gunawi, T. Do, J. M. Hellerstein, I. Stoica, D. Borthakur, and J. Robbins. Failure as a Service (FaaS): A cloud service for large-scale, online failure drills. Technical Report UCB/EECS-2011-87, EECS Department, University of California, Berkeley, Jul 2011.

[16] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur. FATE and DESTINI: a framework for cloud recovery testing. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 18–18, Berkeley, CA, USA, 2011. USENIX Association.

[17] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[18] H. Jin, K. Qiao, X.-H. Sun, and Y. Li. Performance under failures of MapReduce applications. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '11, pages 608–609, Washington, DC, USA, 2011. IEEE Computer Society.

[19] P. Joshi, H. S. Gunawi, and K. Sen. PREFAIL: a programmable tool for multiple-failure injection. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 171–188, New York, NY, USA, 2011. ACM.

[20] W. Torell and V. Avelar. Performing effective MTBF comparisons for data center infrastructure. http://www.apcmedia.com/salestools/ASTE-5ZYQF2_R1_EN.pdf.

[21] J. Weinman. Cloudonomics: A rigorous approach to cloud benefit quantification. *The Journal of Software Technology*, 14:10–18, October 2011.

[22] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen. Processor hardware counter statistics as a first-class system resource. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, HOTOS'07, pages 14:1–14:6, Berkeley, CA, USA, 2007. USENIX Association.