Silent Data Corruption Simulation in Apache Spark and Hadoop Tested with PageRank

SAM WILLIAMS, Virginia Tech, Virginia

Apache Spark[8] running on top of a Hadoop[7] distributed file system provides a distributed and fault-tolerant mechanism for running distributed applications like PageRank. But silent data corruption (SDC), though rare, can still silently affect the correctness of programs running on Spark and Hadoop. For this project, I developed a basic framework for simulating silent data corruption in Spark applications in Python.

I also examined the effects of silent data corruption on an example Spark application, PageRank. PageRank is an interesting application to test correctness for because of its property of converging at around 50 iterations [6]. Results from an experiment simulating SDC with variable probabilities and iteration counts implicate that the correctness of PageRank is not significantly affected by the number of PageRank iterations, given a probability of SDC occurring. However, there is slightly higher relative error and average deviation experienced by PageRank running 25 iterations for smaller probabilities of SDC occurring tested, over the 10 and 50 iteration tests, which could be explored in future work.

CCS Concepts: • **Computer systems organization** → **Data corruption**; *Redundancy*; • **Networks** → Network reliability.

Additional Key Words and Phrases: silent data corruption, distributed systems, simulations, fault injections, PageRank

ACM Reference Format:
Sam Williams. 2022. Silent Data Corruption Simulation in Apache Spark and Hadoop Tested with PageRank. 1, 1 (November 2022), 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Application correctness an important metric for almost all programs and applications. Many applications anticipate errors and handle them appropriately. However, not all errors are reported to applications, specifically forms of silent data corruption (SDC). When data is corrupted silently, via radiation, faulty hardware, or otherwise, the application may never know it is using corrupted data unless it constantly checks for it. The corrupted data may perpetuate and cause a chain-reaction of incorrect data and execution branches. Clearly, this is undesirable and may have devastating consequences for some applications.

Methods of identifying, solving and probing silent data corruption have been explored [1–3, 5]. Dixit et. al [2] explored debugging methods for SDC in a Facebook datacenter, providing libraries and practices for identifying and debugging errors caused by SDC. Bacon [1] explores SDC in a planet-scale distributed database owned by Google that is used by more than one billion people, and discusses software and operations-based approaches to detect and correct SDC.

Different types of DRAM have different forms (or none) of silent error detection and correction, like cyclic redundancy checks (CRCs) and single device data correction (SDDC). Kim et. al [5] propose an innovative alternative approach to other error correction coding schemes (ECCs), but introduces trade offs, i.e. CRC requirements, for the proposed SDDC algorithm to work.

Author's address: Sam Williams, Virginia Tech, Blacksburg, Virginia, shwilliams@vt.edu.

XXXX-XXXX/2022/11-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

Fiala et. al [3] explore silent data corruption in OpenMPI, an open source Message Passing Interface implementation, and introduce a fault injector to simulate SDC on OpenMPI as well as an MPI library called RedMPI that can detect and correct SDC in MPI applications without requiring changes to application code with double or triple redundancy.

As explored above, silent data corruption is a serious modern issue. The unique challenges posed by SDC are motivating researchers to find solutions to these unpredictable and invisible kinds of errors. Additionally, these errors are fundamentally still present in fault-tolerant applications like Apache Spark and Hadoop [2]. So, it makes sense to explore how SDC affects applications running on Spark/Hadoop. I could not find any previous work on Spark-specific fault injectors, though Dixit et al. discuss practices for debugging SDC in Spark. So, for this project, an Apache Spark Python-based fault injector is designed and run on a test Spark application, PageRank, where SDC is simulated. The results are aggregated and then analyzed to see how SDC affects the correctness.

I found specific interest in the PageRank application due to its property of convergence (meaning that every PageRank value stops changing significantly after around 50 iterations) [6]. Specifically, I wished to see if it would still converge after variable SDC and to what extent.

2 BACKGROUND

2.1 Silent Data Corruption

Silent Data Corruption (SDC) is a form of data corruption where incorrect data is generated or calculated by hardware and passed to software. More generally, SDC occurs when some data's bits are corrupted in storage or during calculations but no errors are raised. This can lead to application level errors because of incorrect data begin read or used in subsequent calculations.

Oftentimes, SDC is caused by radiation (e.g. single bit flips) in memory, in disk, RAM and even CPUs, though it can also be caused by faulty hardware [3]. Single bit flips can be detected with error detecting codes like cyclic redundancy checks (CRCs) and fixed with error correction codes (ECCs). Although some forms of these measures have been adopted in some DRAM hardware [5], these are not always implemented in caches, registers and ALUs. However, double flips can go undetectable by these systems [3]. Redundancy detection may help, but requires at least dual redundancy for valid detected (i.e. two different copies of the same data) but correction requires at least triple redundancy (i.e. one is incorrect, if two are the same then the incorrect one is obvious. If they are all different, that's another issue, though unlikely.) Redundancy can be expensive both computationally and monetarily. Thus, SDC is one of the trickiest and most invisible forms of data corruption in computing, and must often be checked for manually.

2.2 Apache Spark

Apache Spark is an engine for unified data processing, providing a system of performing parallel data processing based on applications in languages like Java, Python and Scala [8]. Additionally, Apache Spark is frequently used with/on top of Apache Hadoop, a distributed file system [7]. Together, the Spark/Hadoop stack provides a Map-Reduce-like API for programming distributed data-processing applications.

Spark's main programming abstraction is a resilient distributed dataset (RDD) [8]. RDDs are fault-tolerant collections of a generic object that can have particular *transformations* applied to them. These transformations include operations such as *mapping*, *filtering* and *grouping* that are partitioned across designated worker nodes. The transformations are not performed immediately, as multiple subsequent transformations may be efficiently performed or joined during run-time. Instead, they are only performed when *actions* are called in the code. Some examples of actions are *count* and *collect*, where the data must logically be combined to compute an answer for the action. RDDs can manually be persisted into memory (with a call to a *persist* function), but by default they can spill into disk.

RDDs also perform fault tolerance, but not via redundancy. Instead, RDDs keep track of their *lineage*, a graph of all transformations applied to it up to any given point. This way, if a Spark worker node that holds part of an RDD fails, the partition of the RDD from that node can be recalculated without having to re-run the entire program to that point.

2.3 Hadoop

The Hadoop Distributed File System (HDFS) is the file system component of Hadoop that is similar but different for reasons due to performance from the UNIX file system [7]. HDFS stores file system metadata and application data separately, on a *NameNode* and *DataNodes*, respectfully.

2.3.1 NameNode. The *NameNode* maintain the mapping of file blocks to DataNodes (the physical location of file data) [7]. This mapping is an hierarchical structure spread throughout multiple data nodes. It is replicated on at least 3 independent DataNodes. The client (HDFS) reads files by first asking the NameNode for where it can find required DataNodes and assemble the requested file. The NameNode will report back the physically closest DataNode(s) that hold the file. When HDFS writes a file, three replicas are assigned and data is written in a pipelined fashion.

2.3.2 DataNodes. DataNodes are comprised of two actual files: a data file and a metadata file. The data file holds all the actual (partitioned) file data of a particular file, while the metadata file includes information like general metadata and checksums for each data file. Some of this data includes a namespace ID and version number, which are checked during startup of HDFS. When created, DataNodes register with the NameNode. DataNodes send *block reports* every hour that identify replica DataNodes. DataNodes also send *heartbeat* messages every three seconds to confirm the node is operational. If a node does not send a heartbeat in ten minutes it is designated as out of operation by the NameNode.

2.4 PageRank

PageRank is an algorithm developed by researchers at Stanford [6] that ranks web pages based on some objective level of importance. The algorithm calculates a random web surfer's probability of arriving at any particular web page, using criteria such as the probability of stopping clicking and the amount of inward and outward links to and from different pages. The algorithm is iterative and has the interesting property of converging after around 50 iterations [6], where PageRank values stop changing significantly.

3 DESIGN

Both Apache Spark and the HDFS provide some form of fault tolerance, but silent data corruption can still occur despite these mechanisms. The first deliverable of this project is to provide a framework to simulate silent data corruption in Spark applications. For the sake of time and simplicity, I chose to create such a mechanism in Python using Spark (PySpark). This framework can be designed as a function that will perform an SDC on a value based on a given probability.

3.1 Fault Injector

To clarify, for this project and for the sake of generality, a simulated SDC translates to a single bit flip in some data. This definitely does not encapsulate all forms of SDC, but for the sake of time and simplicity, a bit flip on a primitive value is the extent of SDC in this project.

The way that this can be performed is during a transformation operation performed on a Spark RDD. Since the data is partitioned, any lambda function called inside of a map transformation is run on separate worker

nodes on particular partitions of data, and it is here that we can simulate SDC on a per-node scale. Note that this implies a uniform probability of SDC occurring over all nodes, not any particular individual node.

Further, the only way to manually simulate SDC is via some point of execution in this lambda function called in a transformation operation. Instead of SDC being a probability distribution over time, this interpretation of SDC changes to a probability distribution over the number of execution calls at the specified point of execution.

Then the implementation of this idea is straightforward - for some calculation in Python of any primitive type, we can simulate SDC by forcefully performing a bit flip after the calculation. The two necessary values needed to perform this are the value to simulate SDC on and the probability it occurs. The value can be of any primitive type (e.g. float, integer, boolean) and the probability be some decimal number from 0 to 1.

3.2 Analysis of Variable SDC on PageRank

The second goal of this project was to explore the effects of SDC on an application like PageRank. For this project, I sought to explore the difference of correctness, means and standard deviations of PageRank results based off the independent variables of the number of iterations of PageRank and the probability of an SDC occurring. The SDC can be simulated with the PySpark fault injector described above.

The results of running PageRank with SDC to be compared are the correctness (accuracy) of the PageRank values that can be captured via metrics like relative error and average deviation, which will be explicitly defined in the next section.

4 IMPLEMENTATION

4.1 Fault Injector

The fault injector was developed in Python for PySpark applications. Here is the code.

```
# PySpark Fault Injector by Sam Williams, 2022

import random
import struct

def _is_primitive(value):
    primitive = ((int, 'i'), (float, 'd'), (bool, '?'))
    return [(p,v) for (p,v) in primitive if isinstance(value, p)]

def SIMULATE_SDC(value, probability):

    # Check to see if value is a primitive
    primitive = _is_primitive(value)

    # Condition for SDC occurring (changing value)
    if random.random() <= probability and len(primitive) > 0:
        # 1. Get the original type and formatting code
        (t,code) = primitive[0]
        # 2. Convert to byte array
        bin_array = bytearray(struct.pack(code, value))
        # 3. Get the size
        sz = len(bin_array)
        # 4. Choose a bit to flip
```

```
        bit_to_flip = random.randrange(8*sz)
        # 5. Find that bit and flip it
        for i in range(sz):
            b = bin_array[i]
            if 8*(i+1) > bit_to_flip:
                # Fancy bit arithmetic
                bin_array[i] = b & ~(0b1 << (bit_to_flip % 8))
                # Convert it back to its type
                return struct.unpack(code, bin_array)[0]

    # Otherwise return original value (no SDC)
    return value


```

The function is straightforward as designed, it takes a value and determines if SDC should occur based on the provided probability using a random number between 0 and 1, and then performs manual SDC by choosing an arbitrary bit to flip and performing some basic bit arithmetic to do so.

First, however, the value is checked to be a primitive with the `_is_primitive` function. If it is, the corresponding entry with both the type and a formatting code is returned. This formatting code is used for the Python struct's[4] decomposition and re-composition of the primitive to a byte array.

4.1.1 Usage. The fault injector is added to a PySpark application Python file via a bash script, where the user provides the input PySpark application file name as an input and an output file name for the resulting file. In the PySpark application, all the user has to do is wrap whatever primitive they wish to stimulate SDC on with `SIMULATE_SDC(value, probability)`, where `value` is the original primitive expression that is evaluated and `probability` is the probability that SDC will occur.

4.1.2 Building. For the sake of simplicity, the way that the framework code is provided for the API call described above is via a pre-pend operation done to an input PySpark application file. This is accomplished by a bash script that takes an input file and an output file and pre-pends the above code onto the top of the input file, and stores this in the output file. Here is the bash script.

```
# In this file, I implement the fault injector for PySpark.
# Sam Williams, 2022

# Get args
while getopts :i:o: flag
do
    case "${flag}" in
        i) infilename=${OPTARG};;
        o) outfilename=${OPTARG};;
    esac
done

# Check args
if [ -z "$infilename" ] || [ -z "$outfilename" ]
then
```

```
  echo "Usage: '$0' -i <infilename> -o <outfilename>"
  exit 85
fi

# Check if the file exists
if [ ! -f "$infilename" ]; then
    echo "Error: the file '$infilename' does not exist."
    exit 1
fi

# Create updated injected file
echo "<PYTHON FAULT INJECTOR CODE>" > "$outfilename"


# Read the rest of the input file
while IFS= read -r line; do
  printf '%s\n' "$line" >> "$outfilename"
done < "$infilename"

```

This bash script checks for both the input and output file arguments and exits if they are not provided. It then ensures the input file exists. If it does, then the fault injector framework code is injected into the output file, followed by the original file content.

4.2 Analysis of Variable SDC on PageRank

The way that the analysis of PageRank was performed was surprisingly simple. The analysis was performed on Apache Spark's example PageRank PySpark application, provided in the default installation of Spark.

4.2.1 Integration of Fault Injector with PageRank. As mentioned in the design section, in order to simulate SDC, a point of entry in a transformation lambda function should be chosen. For this implementation of PageRank, I chose this to be in the `computeContribs` function. The actual change was quite simple. The result of this function before the SDC was injected was a tuple for each provided URL, the result being the URL and its rank divided by the number of URLS it has. This second term was the one I decided to simulate SDC on, since it is both in an RDD transformation function and vital to the calculation of PageRank.

The change was to wrap this quotient with a call to `SIMULATE_SDC` and pass a probability as the second parameter. After this, the PageRank application file was passed to the builder bash script (to inject the framework code) and then was runnable on Spark.

```
def computeContribs(urls: ResultIterable[str], rank: float) -> Iterable[Tuple[str, float]]:
    """Calculates URL contributions to the rank of other URLs."""
    num_urls = len(urls)
    for url in urls:
        yield (url, SIMULATE_SDC(rank / num_urls, <probability>))
```

However, this only worked with hard-coded probabilities. Ultimately, I had to add an argument to the PageRank application that would be the probability of SDC to use for the entirety of the PageRank algorithm, which was minimal work. Additionally, I removed text output so that only the PageRank values were printed to standard out and hid all logging output Spark provides by default to standard out.

4.2.2 Problem Instances. The instance (of pages) I chose to work with for the PageRank application were a collection of 4 pages: A, B, C and D. They were in a file appropriately formatted for the PageRank application as follows.

```
# From Page ID      # To Page ID
A                   B
A                   C
B                   C
C                   A
D                   C
```

It is important to note that the Spark implementation of PageRank does not provide a PageRank score for node D, as it has no references. Thus, only the PageRanks of pages A, B and C are considered in this analysis.

The PySpark application used in this experiment was implemented by modifying the PageRank PySpark application to run 3 nested for loops: one for the number of PageRank iterations, one for each probability, and fifty trials for each. For the sake of brevity and because the implementation is simply wrapping the existing code with for loops, the corresponding code is omitted from this paper.

5 EVALUATION

For the evaluation, 2 Ubuntu 22.04.1 LTS virtual machines simulating a cluster in a VirtualBox NatNetwork that were running Apache Spark 3.3.0 with Hadoop 3.3.4 were used. The instances of running PageRank with SDC were performed 50 times each with the following independent variables:

(1) The number of iterations of PageRank (10, 25, 50)
(2) The probability of SDC occurring (0.001, 0.01, 0.1)

The following dependent variables were measured to quantify accuracy for each PageRank score:

(1) Relative error: | average - actual | / average
(2) Average Deviation: (sum of | actual - measured |) / number of trials

The relative errors and average deviations for each PageRank score were averaged for all pages. For example, the relative error was calculated via the following steps:

(1) Find the actual PageRank value for i PageRank iterations with 0% probability of SDC occurring.
(2) Calculate the average PageRank value for each page given i and probability of SDC p.
(3) Calculate the relative error for each page's PageRank value for given i and p (formula given above).
(4) Report the average relative error for all pages' PageRank values.

The output of the modified PageRank application described in the implementation section was stored in a log file and then parsed using the methodology described above, outputting the data values in Tables 1 and 2, which were then plotted to figures 1 and 2, respectively.

Table 1. Probability of SDC occurring vs Relative Error (Avg. All Pages)

# of PageRank Iterations	Probability of SDC	Relative Error
10	0.001	0.00000
10	0.01	0.00706
10	0.1	0.06169
25	0.001	0.01040
25	0.01	0.01040
25	0.1	0.06554
50	0.001	0.00000
50	0.01	0.00612
50	0.1	0.06350

Table 2. Probability of SDC occurring vs Average Deviation (Avg. All Pages)

# of PageRank Iterations	Probability of SDC	Average Deviation
10	0.001	0.00000
10	0.01	0.00725
10	0.1	0.06549
25	0.001	0.00308
25	0.01	0.01099
25	0.1	0.06012
50	0.001	0.00000
50	0.01	0.00595
50	0.1	0.06245

Figure 1. Probability of SDC occurring vs Relative Error (Table 1)

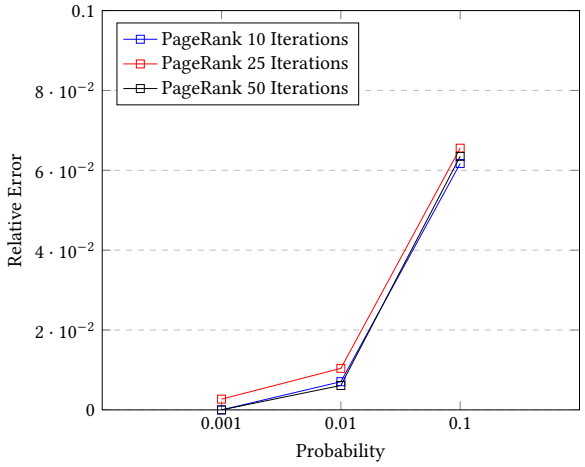

Figure 2. Probability of SDC occurring vs Average Deviation (Table 2)

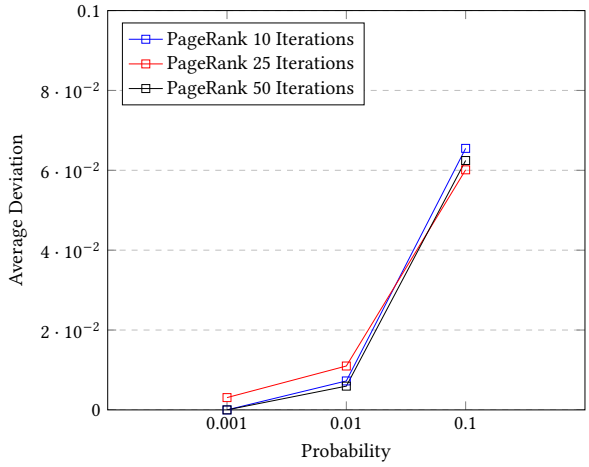

Figure 1 shows the effect the probability of silent data corruption occurring has on the average relative error experienced by the PageRank application running 10, 25 and 50 iterations. As expected, there is exponential

growth in relative error as the probability increases by 10 times. The data from Table 1 indicates that there is no significant difference in relative error experienced by any of the three PageRank iteration counts tested. This indicates that using more PageRank iterations does not translate to better application correctness given a probability of SDC occurring.

Figure 2 shows the effect the probability of silent data corruption occurring has on the average average deviation experienced by all pages' PageRank values running for 10, 20 and 50 iterations. Similar to the data collected for average relative error, there is exponential growth in the average deviation of PageRank scores as the probability increases 10 fold, and there is no significant difference in the average deviation of PageRank scores experienced by any of the three PageRank iteration counts tested. This also supports the claim that using more PageRank iterations does not translate to better application correctness given a probability of SDC occurring.

Overall, the data collected for relative error and average deviation is quite similar. A takeaway supported by the data is that correctness of PageRank affected by SDC is not improved by increasing the number of iterations. As a reminder, it is claimed that PageRank values converge at around 50 iterations [6]. Interestingly, there is slightly higher relative error and average deviation experienced during 25 iterations for low probability of SDC occurring, over the 10 and 50 iteration tests. It is not clear why this occurs, and future work could explore this phenomenon by performing tests with more intermediate iteration counts.

6 CONCLUSION

In this project I created a fault injector for PySpark applications that provides the ability to simulate silent data corruption (SDC). The API requires a singular call to method `SIMULATE_SDC` on a data value with a parameter indicating the probability of the SDC occurring. Any PySpark application can call this API method in an RTT transformation lambda function, and can run after being submit to a bash script program to inject the framework code.

When an example PageRank application was tested on Spark and Hadoop with variable iteration counts and probabilities of SDC, results implicated that the number of PageRank iterations does not affect application correctness given a probability of SDC occurring. The data used to back these claims were averaged PageRank values collected over 50 iterations. However, the error and experienced by PageRank running 25 iterations was slightly higher than the 10 and 50 iteration tests for low probability of SDC occurring, which could be examined further in future work.

Future work for this project is to convert the fault injector code to an official Python Module instead of pre-pending the code to a user code file. Another obvious next step is to expand the fault injector framework into other Spark-supported languages like Java, Scala and R. For future replications of this experiment, a PageRank input with significantly more pages should be analyzed for accuracy. Additionally, parallelization should be explored for this larger dataset, as each call to the PageRank application can be run independently of one another.

ACKNOWLEDGMENTS

Special thanks to Dr. Dimitrios Nikolopoulos at Virginia Tech.

REFERENCES

[1] David F. Bacon. 2022. Detection and Prevention of Silent Data Corruption in an Exabyte-scale Database System. In *The 18th IEEE Workshop on Silicon Errors in Logic – System Effects.*

[2] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. 2021. Silent Data Corruptions at Scale. https://doi.org/10.48550/ARXIV.2102.11245

[3] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. 2012. Detection and correction of silent data corruption for large-scale high-performance computing. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* 1–12. https://doi.org/10.1109/SC.2012.49

[4] Python Software Foundation. [n. d.]. struct — Interpret bytes as packed binary data. https://docs.python.org/3/library/struct.html

[5] Jiho Kim, Soonhee Kwon, Jaesang Noh, and Dong-Joon Shin. 2022. Construction of Cyclic Redundancy Check Codes for SDDC Decoding in DRAM Systems. *IEEE Transactions on Circuits and Systems II: Express Briefs* (2022), 1–1. https://doi.org/10.1109/TCSII.2022.3175066

[6] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66. Stanford InfoLab. http://ilpubs.stanford.edu:8090/422/ Previous number = SIDL-WP-1999-0120.

[7] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–10. https://doi.org/10.1109/MSST.2010.5496972

[8] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (oct 2016), 56–65. https://doi.org/10.1145/2934664

Received 26 December 2022