Silent Data Corruptions at Scale

Harish Dattatraya Dixit
Facebook, Inc.
hdd@fb.com

Sneha Pendharkar
Facebook, Inc.
spendharkar@fb.com

Matt Beadon
Facebook, Inc.
mbeadon@fb.com

Chris Mason
Facebook, Inc.
clm@fb.com

Tejasvi Chakravarthy
Facebook, Inc.
teju@fb.com

Bharath Muthiah
Facebook, Inc.
bharathm@fb.com

Sriram Sankar
Facebook Inc.
sriramsankar@fb.com

ABSTRACT

Silent Data Corruption (SDC) can have negative impact on large-scale infrastructure services. SDCs are not captured by error reporting mechanisms within a Central Processing Unit (CPU) and hence are not traceable at the hardware level. However, the data corruptions propagate across the stack and manifest as application-level problems. These types of errors can result in data loss and can require months of debug engineering time.

In this paper, we describe common defect types observed in silicon manufacturing that leads to SDCs. We discuss a real-world example of silent data corruption within a datacenter application. We provide the debug flow followed to root-cause and triage faulty instructions within a CPU using a case study, as an illustration on how to debug this class of errors. We provide a high-level overview of the mitigations to reduce the risk of silent data corruptions within a large production fleet.

In our large-scale infrastructure, we have run a vast library of silent error test scenarios across hundreds of thousands of machines in our fleet. This has resulted in hundreds of CPUs detected for these errors, showing that SDCs are a systemic issue across generations. We have monitored SDCs for a period longer than 18 months. Based on this experience, we determine that reducing silent data corruptions requires not only hardware resiliency and production detection mechanisms, but also robust fault-tolerant software architectures.

KEYWORDS

silent data errors; data corruption; system reliability; hardware reliability; bitflips

1 INTRODUCTION

Facebook infrastructure serves numerous applications like Facebook, Whatsapp, Instagram and Messenger. This infrastructure consists of hundreds of thousands of servers distributed across global datacenters. Each server is made up of many fundamental components like Motherboard, Central Processing Units (CPU), Dual In-line Memory Modules (DIMMs), Graphics Processing Units (GPU), Network Interface Cards (NICs), Hard Disk Drives (HDDs), Flash Drives and interconnect modules. The key unit that brings all these components together is the CPU. It manages the devices, schedules transactions to each of them efficiently and performs billions of computations every second. These computations power applications for image processing, video processing, database queries, machine learning inferences, ranking and recommendation systems. However, it is our observation that computations are not always *accurate*. In some cases, the CPU can perform computations incorrectly. For example, when you perform 2x3, the CPU may give a result of 5 instead of 6 silently under certain microarchitectural conditions, without an indication of the miscomputation in system event or error logs. As a result, a service utilizing the CPU is potentially unaware of the computational accuracy and keeps consuming the incorrect values in the application. This paper predominantly focuses on scenarios where datacenter CPUs exhibit such silent data corruption. We dive deep into a real-world application-level impact of a corruption, the processes used in debugging such corruption, and conclude with detection and mitigation strategies for silent data corruptions. While we present one case study, we have observed several scenarios, data paths and architectural blocks where SDCs manifest, and hence it is a systemic problem that the industry should tackle collectively.

Prior work [11], [24], [28], [14], [15], [18] within this domain focused on soft errors due to radiation or synthetic fault injection. In contrast, we observe that silent data corruptions are not limited to soft errors due to radiation and environmental effects with probabilistic models. Silent data corruptions can occur due to device characteristics and are repeatable at scale. We observe that these failures are reproducible and not transient. Techniques like Error Correction Code (ECC) are beneficial for reducing the error rates in SRAM. However not all the blocks within a datacenter CPU have similar datapath protection. Moreover, CPU SDCs are evaluated to be a one in a million occurrence within fault injection studies. We observe that CPU SDCs are orders of magnitude higher than soft-error based FIT simulations. CPU SDCs occur at a higher rate due to minimal error correction within functional blocks. With increased silicon density and technology scaling [31], [13], we believe that academic researchers and industry should invest in methods to counter these issues.

Facebook infrastructure initiated investigations into silent data corruptions in 2018. In the past 3 years, we have completed analysis of multiple detection strategies and the performance cost associated. For brevity, this paper does not include details on the performance vs cost tradeoff evaluation. A follow up study would dive deep into the details. In this paper, we provide a case study with an application example of the corruption and are not using any fault injection mechanisms. This corruption represents one of the hundreds of CPUs we have identified with real silent data corruption through our detection techniques.

The rest of the paper is structured as follows: Section 2 provides an overview of related work within this domain. Section 3 walks through the different defect categories in silicon design and manufacturing. Section 4 details a real-world application example of silent data corruption and propagation of corruptions across the stack. Section 5 lists the best practices for root-causing silent data corruptions at scale, and walks through the debugging for the application in the case study. Section 6, concludes the debug findings and revisits application failure with a deeper understanding of the CPU defect. Section 7 provides a high level overview of fleet detection mechanisms that can be implemented to mitigate the risk of silent errors. Section 8 provides a high level overview of software fault tolerant mechanisms for bitflips and data corruptions.

2 RELATED WORK

Previous work within the silent error domain studies the impact of soft errors due to radiation [11], and how environmental factors can lead to soft errors within the system. The study provides error rate observations for a non ECC protected SRAM. This is calculated using a Soft Error Rate (SER) from radiation resulting in an estimated 50000 FIT (Failure-In-Time: One FIT is equivalent to one failure in 1 billion device hours). Hence they recommend using ECC which reduces the error rate by 1000x for SRAMs.

Experiments with bit-flip injection mechanisms in floating point units [18] have shown the theoretical impact of bitflips within processors. Bitflip injection mechanisms have also been used to compare the performance of processors under benchmarks with synthetic injection and radiation induced bitflips [15]. A 2012 study on silent data corruption in a HPC cluster with 96 nodes [21] evaluated the impact of soft errors using fault injector and correcting the corruptions with focus on Message Passing Interface (MPI) Protocols. Within the fault injection study, the fault injector ran with a corruption frequency of 1 in 5 million messages to ensure a relatively high likelihood for an injection. Faster corruption frequency of 1 in 2.5 million messages was also included to evaluate the impact of higher occurrence rates on MPI workloads.

Another set of studies evaluate the risk and mitigation strategies of soft error induced faults within microprocessors. A study from ARM [24] evaluates the vulnerability assessment of soft-errors on ARM Cortex R5 CPUs by breaking down the percentage of sequential logic vulnerable to soft errors which propagate to output ports. In a collaboration study between Intel and University of Michigan [28] radiation induced soft errors are identified to not reflect a permanent failure. The study captures the essential metrics required for quantifying soft errors, evaluating Failure-In-Time (FIT) and techniques to reduce the soft error rate using process technology, circuit, and architectural solutions. A similar study from IBM targets 114 SDC FIT for Power4 systems [14]. All these studies evaluate errors as transient or soft indicating the radiation dependent nature of the error.

ECC reduces the error rate for SRAMs but all the datapaths within datacenter CPUs are not protected by ECC. In addition, the FIT models for CPU also derive from soft error probabilities to evaluate robustness, vulnerability assessments and fault tolerance in the above studies. Since datacenter SDCs are observed to be at higher orders of magnitude, it is valuable for us to explore best practices to debug, detect and mitigate SDCs at scale.

3 DEFECT CATEGORIES

Each datacenter CPU contains billions of transistors which are switching constantly. These transistors are devices made of chemical compositions predominantly of silicon with p-type and n-type impurities. A CPU is designed to meet the desired computing requirements while keeping within the power, thermal and spatial constraints for the chip. Once the design is signed off, a layout for the chip is prepared where billions of logic gates are placed to minimize electrical noise, crosstalk, boost signal distribution and stability. Finally, after validation of all the functional, architectural, and physical requirements, the chip is taped-out as part of the chip development process. After the manufacturing process, the designed chips are then subject to test patterns for expected functional behavior, quality control and eventually shipped to all the computing customers worldwide.

3.1 Device Errors

Within the manufacturing and design process there are opportunities for defects to manifest. It is possible that the design has corner case scenarios. For example, a block which manages the cache controller under a particular power state can have functional limitations. This can result in the device being stuck or manifest functional errors. During placement and routing of blocks within the CPU, there could be uncertainty in the arrival time for signals, which can then lead to an erroneous bit-flip. One example of such failure is a timing path error. While manufacturing, it is also probable that all the transistors are not etched reliably, and all of them do not have the same peak-operating voltage or power thresholds. This can lead to variations in device characteristics and results in manufacturing errors [27], [16].

3.2 Early Life Failures

Some of the early life failures are identified during manufacturing tests, these failures negatively impact the yield of the process. A few of the devices are healthy enough to pass the manufacturing test pattern but exhibit failure symptoms only after they have been in the field serving workloads. Depending on the type of electrical weakness within the transistor, a fault may manifest within the first weeks, months or any time before the end of the expected device life [10], [17]. These failures are classified as early life failures.

3.3 Degradation

It is also possible for the devices to get weaker with usage. A computational block used frequently can show wear and tear, and degrade faster than the other parts of the CPU. These are uncommon in comparison to early life failures but are still observed within the industry. An example of this can be seen in another device used in servers - *Rowhammer attacks* for DDR4 memory components [23]. Devices incorporate error correction mechanisms like Error Correction Codes (ECC) to protect against degradation within the device. Degradation based failures can have negative impact as the aging is not uniform across different chips that fall under this failure category.

3.4 End-of-Life Wear-out

When the device has been in the field serving workloads for a while, beyond their rated life, the entire silicon starts exhibiting wear-out [26], [20], [8]. This is observed in most components and is classified as silicon wear-out within the bathtub curve modeling of failures. This is also typically the duration for which the failure analysis support or firmware support exists for CPUs.

All the four failure modes described above have the potential to lead to SDC within a fleet of machines. It is statistically more likely to encounter silent data corruption with increasing CPU population. It is our observation that increased density and wider datapaths increase the probability of silent errors. This is not limited to CPUs and is applicable to special function accelerators and other devices with wide datapaths. In the next section, we analyze how these errors propagate across the stack and cause application-level manifestations. We present ways to debug them at scale and discuss detection practices at different abstraction levels.

4 APPLICATION LEVEL IMPACT OF SILENT CORRUPTIONS

Facebook infrastructure is made up of hundreds of thousands of servers and has billions of users accessing our applications. With billions of users accessing the Facebook family of applications, the infrastructure receives billions of requests per day. With billions of user queries, image uploads, and media content, the processing required for these applications needs to be fast, reliable, and secure. We utilize fundamental concepts within distributed systems to partition our applications and optimize each of the said partitions. A typical application can require anywhere between tens of machines to hundreds of thousands of machines based on the complexity, resource profile and computing needs of the application. One such partition is our querying infrastructure. This querying infrastructure is used to fetch and execute SQL and SQL like queries (Presto, Hive, Spark) [5], [6] across multiple datasets.

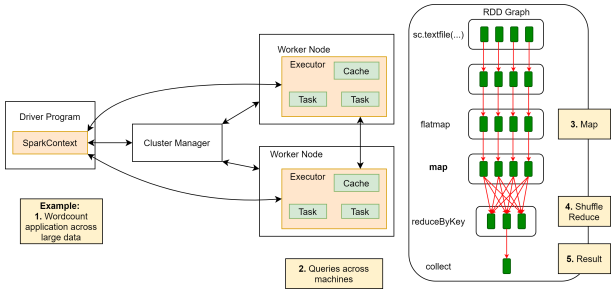

Figure 1: High Level Spark Architecture

4.1 Spark

Figure 1 [19] describes a typical architecture of a spark cluster. Spark is a widely known distributed processing framework which works based on the concept of Resilient Distributed Datasets (RDDs) each of which can be run in parallel. The results for a large data processing application are produced after several key steps. At a high level, a mapping function first maps the data blocks. This is followed by a reduction operation which aggregates the results across multiple RDDs. The result is presented in the collect phase after reduction.

For example, a Wordcount application, trying to count the number of occurrences of each word within a large file would execute in the following way. The large file would be split into multiple RDDs. The RDDs are assigned to worker nodes, these worker nodes compute the word-count for a subset of the dataset. Results from each node are aggregated together in the shuffle reduce stage. Finally, an output table of each word and its associated occurrence count is provided to the user. In a large infrastructure environment like Facebook, these applications run millions of such computations every day.

4.2 FB Compression Application

Like wordcount, compression is a technique which is used to reduce the storage footprint of datastores and can make use of the spark architecture. There are multiple algorithms for compression. In this paper we will not be going into details of the algorithms. Interested readers can review the following papers for details and comparison of compression algorithms [30], [12], [25]. Files are usually compressed when they are not being read and decompressed when a request is made for reading the file. In a large infrastructure, millions of compression and decompression operations are performed every day. In this example, we are mainly focusing on the decompression aspect of files. We have a database, where the files are compressed and stored within a data store. Upon request, multiple sets of these files are sent to the decompression pipeline. Before a decompression is performed, file size is checked to see if the file size is greater than 0. A valid compressed file with contents would have a non-zero size. Figure 2 shows the manifestation of corruptions and interlink to the database pictorially.

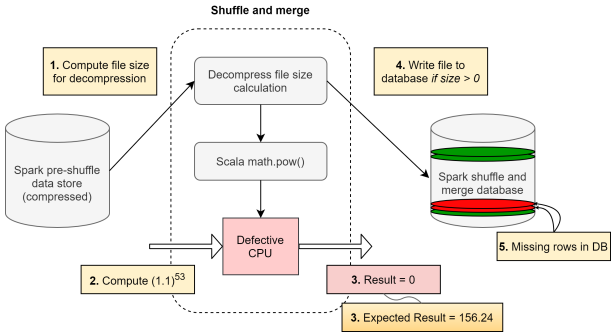

Figure 2: Application level silent data corruption

In one such computation, when the file size was being computed, a file with a valid file size was provided as input to the decompression algorithm, within the decompression pipeline. The algorithm invoked the power function provided by the Scala library (Scala: A programming language used for Spark) [7]. Interestingly, the Scala function returned a **0** size value for a file which was known to have a non-zero decompressed file size. Since the result of the

file size computation is now 0, the file was not written into the decompressed output database.

Imagine the same computation being performed millions of times per day. This meant for some random scenarios, when the file size was non-zero, the decompression activity was never performed. As a result, the database had missing files. The missing files subsequently propagate to the application. An application keeping a list of key value store mappings for compressed files immediately observes that files that were compressed are no longer recoverable. This chain of dependencies causes the application to fail. Eventually the querying infrastructure reports critical data loss after decompression. The problem's complexity is magnified as this manifested occasionally when the user scheduled the same workload on a cluster of machines. This meant the patterns to reproduce and debug were non-deterministic.

5 DEBUGGING SILENT DATA CORRUPTIONS AT SCALE

With concerted debugging efforts and triage by multiple engineering teams, logging was enabled across all the individual worker machines at every step. This helped narrow down the host responsible for this issue. The host had clean system event logs and clean kernel logs. From a system health monitoring perspective, the machine showed no symptoms of failure. The machine sporadically produced corrupt results which returned zero when the expected results were non-zero.

The reproducer at a multi-machine querying infrastructure level was then reduced to a single machine workload. From the single machine workload, we identified that the failures were truly sporadic in nature. The workload was identified to be multi-threaded, and upon single threading the workload, the failure was no longer sporadic but consistent for a *certain subset of data values* on one particular core of the machine. The sporadic nature associated with multi-threading was eliminated but the sporadic nature associated with the data values persisted. After a few iterations, it became obvious that the computation of

$$Int(1.1^{53}) = 0$$

as an input to the *math.pow* function in Scala would always produce a result of 0 on Core 59 of the CPU. However, if the computation was attempted with a different input value set

$$Int(1.1^{52}) = 142$$

the result was accurate.

The next step in the process was to gain a deeper understanding of the scenarios the corruptions manifest in. Any other variants associated with this silent data corruption also require investigation. To confirm the data dependency of the issue, we ran multiple iterations on Core 59. Following shows an example of 3 iterations where 2 of the computations produce faulty results repeatedly.

Core pinned Scala workload

```
[root@hostname ~]#
for x in {0..2}; do taskset -c 59 ./bitflip_repro.sh; done
# Int(1.1^{53}), Int(1.1^{68}), Int(1.1^{78})
Iteration 1: 0, 0, 1692
Iteration 2: 0, 0, 1692
Iteration 3: 0, 0, 1692
```

The data dependency is clearly established for the defect. In this example, core 59 is faulty. Ideally when workloads are faulty, the workload can be stepped through GNU Project debugger (GDB) [4] and reverse engineered. The instruction data could be compared to a reference computation by stepping through instructions. This step-through process, while time-consuming, enables debugging of silent errors. However, Scala is a language whose workloads cannot be stepped through in GDB. Scala is compatible to run Java Byte Code in a Java Virtual Machine (JVM). Java Byte Code (JBC) [3] is compiled by a Just-In-Time (JIT) compiler.

5.1 Tools

We need to perform language conversion while keeping reproducer consistency to triage the root-cause. In this example, we traverse from Scala language reproducer to Java reproducer to JIT compiled JBC to Assembly to triage the instruction level root-cause and enable the reproducer code. Unlike C and C++, Just-In-Time (JIT) compiled code is not compiled ahead of time. However, to debug a silent error, we cannot proceed forward without understanding which machine level instructions are executed. We either need an ahead-of-time compiler for Java and Scala or we need a probe, which upon execution of the JIT code, provides the list of instructions executed.

5.1.1 Example Scala to Java Byte Code.

The first step to get to assembly is to convert the reproducer from Scala to Java. There are more resources to aid this conversion. We can use the Scala compiler (scalac) to obtain the Java Class routines for the source code. To obtain the Scala compiled Java Byte Code, we modified the Scala script to a Scala compiler friendly reproducer code.

```
[root@hostname ~]# scalac Bitflip.scala
# This generates the intercompatible scala/java class files
# This can be read as Java Byte Code.
[root@hostname ~]# javap -c -v Bitflip\$.class
```

5.1.2 GCJ.

GCJ [1] was an open source ahead-of-time compiler which could convert JBC to blobs of object files and binary. This binary can be used within GDB to debug. However, the tool development has been deprecated since 2008, and CentOS deprecated the tool in 2010. Without an *ahead-of-time* compiler, it is challenging to perform the static conversion of Java Byte Code to assembly.

5.1.3 HotSpot.

Java provides options to use *+PrintAssembly* to act as a probe and print assembly of the executed code with the use of HotSpot Profiling. To support *+PrintAssembly*, there are 2 requirements,

- **Virtual machine with support for hotspot profiler:** This can be identified for an example machine using the following command. An output providing HotSpot confirms that the virtual machine enables profiling. Version numbers shown

here are example versions and are not representative of any deployment.

```
$> java -version
java version "A.B.C_DEF"
Java(TM) SE Runtime Environment (build G.H.I_JKL-MNO)
Java HotSpot(TM) 64-Bit Server VM (build PQ.RST-UVW, mixed mode)
# This means the VM can be profiled.
```

- **Library for profiling:** Hotspot is a performance profiler used to analyze hot spots for a program. These hotspots are optimized for high performance execution with minimal overhead for the less-performance critical code. The profiler enables the option for *PrintAssembly* [2], and can print the assembly compiled by JIT. These assembly instructions subsequently enable us to root cause and triage the failing instruction.

After enabling the profiler, we obtain the assembly that the code executes (JIT + Hotspot output assembly). Our first version of the assembly was 430K lines. With our assembly, we can debug the silent error. The Scala *math.pow* functions are identified within the 430K line assembly. We parse the 430K line assembly to optimize the reproducer. However, the disassembly does not output the sequence of executed instructions but rather lists the methods used in the call stack. The sequencing can be unclear. To obtain a reproducer, we need to sanitize, reverse engineer with a smaller assembly code. From this raw assembly, we can understand the sequence of instructions sent to the CPU and root-cause the faulty instruction by following the best practices to debug silent errors.

5.2 Best Practices for Silent Error Debug

A few guidelines while reverse engineering the printed assembly code. While these guidelines are derived from this example, they can be leveraged for debugging similar silent data corruptions.

- **Absolute address references:** Leaving absolute addresses to jump to within the code while optimizing for a reproducer will lead to segmentation faults. Instead of managing all the memory locations, it is preferred to eliminate the absolute address reference if that section of assembly is found to have no dependency on the reproducibility.
- **Unintended branches:** If unintended branch and jump calls are left unmapped, the code crashes with segmentation faults and undefined code branches. This introduces more variability within the function. It is advisable to limit variability when attempting for a deterministic bitflip reproducer.
- **External Library References:** Identify which instructions invoke a call outside the current code path to external libraries. With the goal of a minimal reproducer, it is preferred to not have external library dependencies.
- **Compiler Optimization:** High performance code features multi-pass compiler optimizations. Observing optimization to mathematical equations can help in understanding the critical assembly required for the reproducer. Optimizations may not be intuitive while stepping through assembly instructions.
- **Stub and Redundant Instructions:** It is preferred to eliminate redundant and stub instructions. Stubs are used by Scala for book-keeping and are not relevant for debugging the failing instruction. Stub instructions do not interfere with functionality outside of the Scala execution context.
- **Input/Output registers:** For any bitflip reproducer we need to identify the data input and result registers for the critical instructions. After identification, additional instructions must be added to provide user inputs and obtain results. This enables a stable reproducer code and enables identification of data dependency for the silent data corruption.
- **Managing Stack Frames:** Standalone assemble reproducers require stack frames to be appropriately managed. Managing transactions into the stack frame to prevent buffer overflow or underflow is critical for stability. Without stack frames, reproducer code cannot manage stack-based requests or function calls.
- **Memory-offset references:** Registers typically use memory offsets within instructions. The offsets must be initialized appropriately. If offsets are not calculated and initialized, we will encounter segmentation faults or reproducer corruption due to uninitialized data.
- **Special Function Units:** We need to monitor transactions to special function units (like ALU, DSP, FPU, AVX etc) as they bring in approximations. In addition, special function units utilize varied bit widths, special function registers and stack architecture.
- **Main Frames:** A standalone reproducer cannot be complete without appropriate main frames and function frames. This makes the code executable.

In this section, we are purely focusing on the best practices for silent error debugging, and not on the knowledge prerequisites about CPU architectures or GDB internals.

- We are skipping over the hardware architecture and implementation details for all the CPU sub-blocks. Details associated with the status flags, differences between special function stacks and normal integer stack, instruction truncation and handshakes between different precision bit-width and operand types are skipped. All of these are key to identify the steps within a CPU and are widely documented in published research.
- We are skipping over all the steps within GDB, and the methods to print, step through commands, scripting through different stacks, registers, memory addresses as these are documented widely.

After reverse engineering, identifying the handshake between hardware blocks and dependency graphs for assembly, we can arrive at a simpler reproducer. Here are some interesting observations from the assembly that were obtained for this example.

- For squaring a number, the scala compiler implements a fast optimization using look-up tables.
- *math.pow* function is in-lined in the power function, even though PrintAssembly prints them separately.
- Scala *math.pow* computes powers using the formula -

$$x^y = 2^{y*log_2 x}$$

We step-through instructions in GDB. During the step-through process, instruction operands, memory and register states, and

instruction outputs are examined for corruption. As a result of this process, we obtain the faulty instruction within the defective CPU.

5.3 Assembly Level Test Case

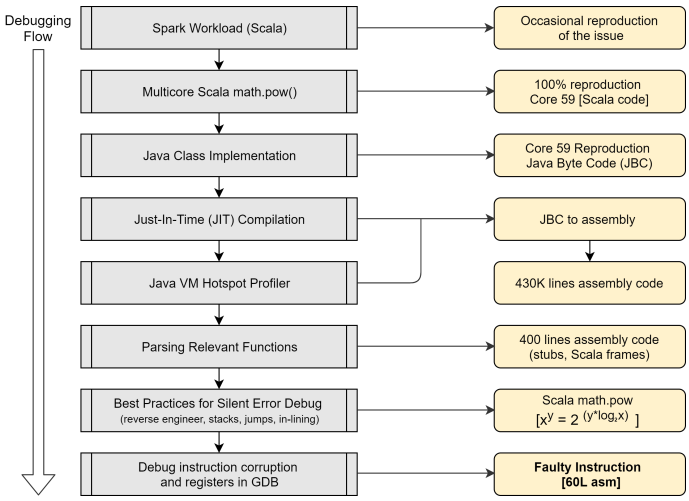

Figure 3: High Level Debug Flow

Once the reproducer is obtained in assembly language, we optimize the assembly for efficiency. The assembly code accurately reproducing the defect is reduced to a 60-line assembly level reproducer. We started with a 430K line reproducer and narrowed it down to 60 lines. Figure 3 provides a high level debug flow followed for root-causing silent errors.

6 REVISITING APPLICATION FAILURES

Note that that all the machines operating the application do not have any logs or system level health information indicating this failure mode. We identified cases of corruption impacting computations involving non-zero operands and results. For example, the following incorrect computations were performed on the defective CPU. We identified that the computation affected positive and negative powers for specific data values. In some cases, the result was non-zero when it should have been zero. We noticed incorrect values with varying degrees of precision.

Example errors:

$$Int[(1.1)^3] = 0, expected = 1.$$
$$Int[(1.1)^{107}] = 32809, expected = 26854.$$
$$Int[(1.1)^{-3}] = 1, expected = 0.$$

As a result, an application could have decompressed files of incorrect size and are incorrectly truncated without an End-Of-File (EoF) terminator. This leads to dangling file nodes, missing data, and no traceability of a corruption within an application. The intrinsic data dependencies on the core as well as the data inputs make the corruptions close to impossible to detect and root-cause without a targeted reproducer. This is challenging, especially in a scenario where a fleet has hundreds of thousands of machines performing a few million computations every second. We identified additional machines with the targeted reproducer. We integrated our lessons from the reproducer into detection mechanisms within the fleet. In addition, the best practices identified for silent error debugging enable faster root-cause and sensitivity analysis for similar errors within the fleet.

We initiated efforts in estimating the business impact due to SDCs by quantifying the scale and criticality of the problem to our infrastructure. Given the silent nature of these errors, evaluating the scale of the problem was challenging at first. Initially the calculations for defective-parts-per-million predictions, debug engineering time allocations and business impact were based on heuristics and smaller datasets. With data collection and analysis in the past 18 months, we arrived at empirical values and ranges for each of the above.

6.1 Hardware approaches to counter SDCs

We observe that silent data corruptions are not limited to rare one in a million occurrences within a large-scale infrastructure. These errors are systemic and are not as well understood as the other failure modes like Machine Check Exceptions. There are several studies evaluating the techniques to reduce soft error rate within processors [33], [29], we can extend these techniques to repeatable SDCs which can occur at a higher rate. We can mitigate the exposure of applications to silent errors by using different strategies.

- **Protected Datapaths:** Augmenting blocks within the device to have increased datapath protection using algorithms similar to Error Correcting Codes (ECC) can increase resiliency of the device.
- **Specialized Screening:** Dedicated screens and test patterns within the manufacturing flow to detect silent errors. Testing with randomized data streams can increase the probability of hit rate within manufacturing testing.
- **Understanding @Scale Behavior:** Close partnership with the customers using devices at scale to understand and evaluate the impact of silent errors. It is beneficial to study occurrence rates, time to failure in production, dependency on frequency, voltage, and environmental conditions to obtain insights into manifestations of SDCs.
- **Architectural priority:** With increased density, wider datapaths and technology scaling; we are more likely to observe silent data corruptions moving forward. Prioritizing protection against silent data corruption within our architectural choices can enable future semiconductor devices to be more resilient.

The strategies described above are not limited to CPUs and can be extended to Application Specific Integrated Circuits (ASIC) and devices with wider data paths and unprotected logic.

7 DETECTION MECHANISMS

To detect errors of this type in the fleet, we need workloads which execute specific types of computations. We then compare the results of these computations with known reference values to ensure that the results are accurate. Silent corruptions tend to be data dependent making it difficult to predict their occurrence in the fleet. Given that any downtime for testing in a production fleet is an efficiency loss, this can be achieved in 3 different ways:

7.1 Opportunistic

Opportunistically utilize machines in maintenance states and perform instruction level accuracy validation with randomized data inputs. The challenge here is that the coverage of the fleet is based on how frequently machines fall into these opportunistic states. In a large fleet, we do not expect large percentages of machines to be in these states, however there are transition states (provisioning, service setup etc) that can be used opportunistically.

7.2 Periodic

Implement a scheduler which periodically monitors machines for silent error coverage and then schedules machines based on a periodic timer (for example: 15 days) for testing. Here the overhead is high as the machine is forced to an out of production status to perform testing at a specified schedule.

7.3 Production Friendly

Tests can be optimized to be minimal in size and run-time. This can enable test instructions to be executed in parallel with the workloads on the machine. The result is sent to a collector to notify a pass or fail status for the machine. This method requires close coordination with the workload to not have any adverse impact on the production workload.

8 SOFTWARE FAULT TOLERANT MECHANISMS

To deal with silent errors, we need to rethink the robustness of infrastructure software design philosophies and software abstractions.

8.1 Redundancy

A better way to prevent application-level failures is to implement software level redundancy and periodically verify that the data being computed is accurate at multiple checkpoints. This is a tried and tested method implemented in space research [32], aircraft [22] and automobiles [9]. It is important to consider the cost of accurate computation while adopting these approaches to large-scale data center infrastructure. The cost of redundancy has a direct effect on resources, more redundant the architecture, the larger the duplicate resource pool requirements. However, this provides probabilistic fault tolerance to the application.

8.2 Fault Tolerant Libraries

Adding fault tolerance into well-known open-source libraries like PyTorch would greatly aid the applications to prevent exposure to silent data corruptions. Building algorithmic fault tolerance adds additional overhead on the application. This can be implemented with negligible drop in performance. This effort would need a close handshake between the hardware silent error research community and the software library community.

Facebook infrastructure has implemented multiple variants of the above hardware detection and software fault tolerant techniques in the past 18 months. Quantification of benefits and costs for each of the methods described above has helped the infrastructure to be reliable for the Facebook family of apps. A subsequent publication will go into statistical detail on trade-offs across detection strategies and coverage scenarios for detection mechanisms and fault tolerant software libraries.

9 CONCLUSIONS

Silent data corruptions are **real** phenomena in datacenter applications running at scale. We present an example here which illustrates one of the many scenarios that we encounter with these data dependent, reclusive and hard to debug errors. Understanding these corruptions helps us gain insights into the silicon device characteristics; through intricate instruction flows and their interactions with compilers and software architectures. Multiple strategies of detection and mitigation exist, with each contributing additional cost and complexity into a large-scale datacenter infrastructure. A better understanding of these corruptions has helped us evolve our software architecture to be more fault tolerant and resilient. Together these strategies allow us to mitigate the costs of data corruption at Facebook's scale.

Acknowledgement The authors would like to thank Manish Modi, Vijay Rao, T.S. Khurana, Aslan Bakirov, Melita Mihaljevic, Kushal Thakkar, Nishant Yadav, Aravind Anbudurai, Jason Liang, Jianyu Huang, Sihuan Li, Jongsoo Park and other infrastructure engineers for their inputs in the implementation of solutions and valuable technical suggestions.

REFERENCES

[1] 2007. *GCJ: The GNU Compiler for Java - GNU Project - Free Software Foundation (FSF).* https://web.archive.org/web/20070509055923/http://gcc.gnu.org/java/
[2] 2013. *PrintAssembly - HotSpot - OpenJDK Wiki.* https://wiki.openjdk.java.net/display/HotSpot/PrintAssembly
[3] 2019. *Java Programming/Byte Code - Wikibooks, open books for an open world.* https://en.wikibooks.org/wiki/Java_Programming/Byte_Code
[4] 2020. *GDB: The GNU Project Debugger.* https://www.gnu.org/software/gdb/
[5] 2021. *MySQL :: MySQL Documentation.* https://dev.mysql.com/doc/
[6] 2021. *Overview - Spark 3.0.1 Documentation.* https://spark.apache.org/docs/latest/
[7] 2021. *Scala Documentation.* https://docs.scala-lang.org/?_ga=2.201016622.1205038718.1605503218-1722664999.1605503218
[8] Mridul Agarwal, Bipul C Paul, Ming Zhang, and Subhasish Mitra. 2007. Circuit failure prediction and its application to transistor aging. In *25th IEEE VLSI Test Symposium (VTS'07).* IEEE, 277–286.
[9] Pete Bannon, Ganesh Venkataramanan, Debjit Das Sarma, and Emil Talpes. 2019. Computer and Redundancy Solution for the Full Self-Driving Computer. 1–22. https://doi.org/10.1109/HOTCHIPS.2019.8875645
[10] T. S. Barnett, A. D. Singh, and V. P. Nelson. 2003. Extending integrated-circuit yield-models to estimate early-life reliability. *IEEE Transactions on Reliability* 52, 3 (2003), 296–300. https://doi.org/10.1109/TR.2003.816418
[11] R. C. Baumann. 2005. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability* 5, 3 (2005), 305–316. https://doi.org/10.1109/TDMR.2005.853449
[12] Arup Kumar Bhattacharjee, Tanumon Bej, and Saheb Agarwal. 2013. Comparison study of lossless data compression algorithms for text data. *IOSR Journal of Computer Engineering (IOSR-JCE)* 11, 6 (2013), 15–19.
[13] M. T. Bohr and I. A. Young. 2017. CMOS Scaling Trends and Beyond. *IEEE Micro* 37, 6 (2017), 20–29. https://doi.org/10.1109/MM.2017.4241347
[14] D. Bossen. 2002. CMOS Soft Errors and Server Design - IRPS. Tutorial Notes - Reliability Fundamentals. (2002).
[15] G. C. Cardarilli, F. Kaddour, A. Leandri, M. Ottavi, S. Pontarelli, and R. Velazco. 2002. Bit flip injection in processor-based architectures: a case study. In *Proceedings of the Eighth IEEE International On-Line Testing Workshop (IOLTW 2002).* 117–127. https://doi.org/10.1109/OLT.2002.1030194
[16] C. Constantinescu. 2008. Intermittent faults and effects on reliability of integrated circuits. In *2008 Annual Reliability and Maintainability Symposium.* 370–374. https://doi.org/10.1109/RAMS.2008.4925824
[17] M. S. Cooper. 2005. Investigation of Arrhenius acceleration factor for integrated circuit early life failure region with several failure mechanisms. *IEEE Transactions*

on Components and Packaging Technologies 28, 3 (2005), 561–563. https://doi.org/10.1109/TCAPT.2005.848581

[18] James Elliott, Frank Mueller, Frank Stoyanov, and Clayton Webster. 2013. *Quantifying the impact of single bit flips on floating point arithmetic.* Technical Report. North Carolina State University. Dept. of Computer Science.

[19] EPCC. 2019. *Spark Cluster Overview.* https://events.prace-ri.eu/event/850/sessions/2616/attachments/955/1528/Spark_Cluster.pdf

[20] R. Fernandez, J. Martin-Martinez, R. Rodriguez, M. Nafria, and X. H. Aymerich. 2008. Gate Oxide Wear-Out and Breakdown Effects on the Performance of Analog and Digital Circuits. *IEEE Transactions on Electron Devices* 55, 4 (2008), 997–1004. https://doi.org/10.1109/TED.2008.917334

[21] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. 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

[22] Paul M. Frank. 1990. Fault diagnosis in dynamic systems using analytical and knowledge-based redundancy: A survey and some new results. *Automatica* 26, 3 (1990), 459 – 474. https://doi.org/10.1016/0005-1098(90)90018-D

[23] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TRRespass: Exploiting the Many Sides of Target Row Refresh. arXiv:2004.01807 [cs.CR]

[24] X. Iturbe, B. Venu, and E. Ozer. 2016. Soft error vulnerability assessment of the real-time safety-related ARM Cortex-R5 CPU. In *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 91–96. https://doi.org/10.1109/DFT.2016.7684076

[25] SR Kodituwakku and US Amarasinghe. 2010. Comparison of lossless data compression algorithms for text data. *Indian journal of computer science and engineering* 1, 4 (2010), 416–425.

[26] C. Liu, E. Schneider, M. Kampmann, S. Hellebrand, and H. Wunderlich. 2018. Extending Aging Monitors for Early Life and Wear-Out Failure Prevention. In *2018 IEEE 27th Asian Test Symposium (ATS)*. 92–97. https://doi.org/10.1109/ATS.2018.00028

[27] E. J. McCluskey and Chao-Wen Tseng. 2000. Stuck-fault tests vs. actual defects. In *Proceedings International Test Conference 2000 (IEEE Cat. No.00CH37159)*. 336–342. https://doi.org/10.1109/TEST.2000.894222

[28] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. 2005. The soft error problem: an architectural perspective. In *11th International Symposium on High-Performance Computer Architecture*. 243–247. https://doi.org/10.1109/HPCA.2005.37

[29] N. Oh, P. P. Shirvani, and E. J. McCluskey. 2002. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability* 51, 1 (2002), 63–75. https://doi.org/10.1109/24.994913

[30] Khalid Sayood. 2017. *Introduction to data compression.* Morgan Kaufmann.

[31] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. 2002. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings International Conference on Dependable Systems and Networks*. 389–398. https://doi.org/10.1109/DSN.2002.1028924

[32] Joel R. Sklaroff. 1976. Redundancy management technique for space shuttle computers. *IBM Journal of Research and Development* 20, 1 (1976), 20–28.

[33] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. 2004. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.* 264–275. https://doi.org/10.1109/ISCA.2004.1310780