Silent Data Corruption: Detection, Mitigation and Machine Learning

William Daugherty-Miller
Department of Operations Research and Information Engineering
Cornell Tech
New York, NY 10044
wld37@cornell.edu

Abstract

Silent Data Corruption(SDC) is becoming an increasing problem that leads to the loss of data or incorrect results. It can appear in systems of all sizes, however large-scale systems, such as databases or high-performance computing are particularly susceptible. Due to the silent nature of the data defect, SDC is a major factor in data integrity issues, system crashes, financial losses, and even potential safety hazards in critical systems. Various strategies have been proposed to detect and mitigate SDC, including checksums, redundant storage, and error-correcting codes. This paper aims to describe, discuss, contrast and critique various methods of SDC Mitigation/Detection and use cases. Overall, the development of effective SDC detection and mitigation strategies is crucial for ensuring data integrity and the reliability of large-scale systems.

1 Introduction

Silent Data Corruption (SDC), characterized by undetected errors occurring during computational outcomes without system. These errors result in failures or alerts, that pose a significant hurdle to the data integrity in the era of large-scale computational systems, machine learning hardware and machine learning models.(2) While the mistake might seem benign in nature, such as $1 + 1 = 3$ this can have major repercussions in high-stakes industries/applications such as healthcare, autonomous vehicles, and large-scale data processing. Below we can seen an example of this SDC in the context of the company Meta (5).

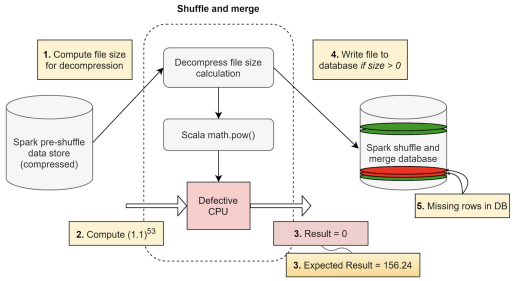

Figure 1: Example of SDC at Meta

As stated, this paper provides a survey of various papers regarding SDC as it applies to machine learning and hardware systems. More specifically, this survey will include the use of artificial neural networks for online error detection, correction of SDC in convolutional neural networks, and strategies for large-scale high-performance computing amongst other strategies.

2 Causes of SDC

Before launching into discussion of various strategies, we should first discuss how SDC occurs. We classically describe SDC as coming as hardware faults in CPUs or PCIe lanes, software bugs, and network issues. These areas are particularly at risk due to the constant data transference or environmental factors. Environmental here include the alpha particles from cosmic rays or heat-induced failures.

Miniaturization of computer chip manufacturing processes is another factor which increases their susceptibility to external influences such as there is increased EMI interference, signal integrity issues and power consumption/heat dissipation issues. (12). Adding onto these factors, we observe that in large-scale high-performance computing and Exabyte-scale database systems face found at major tech companies or servers this can be magnified as there are more chances for the system to go awry. The scale is another issue as well given that these systems compound SDC detection and correction difficulty, rendering traditional error-checking methods impractical due to performance and resource considerations (6; 1).

In short, the growth of large-scale machine learning systems introduces additional complexity, as silent errors in training data can result in biased models and inaccurate predictions with grave real-world implications (2).

3 Evaluation and Analysis of SDC Mitigation Techniques

Various techniques have been proposed to mitigate SDC, each with its strengths and limitations. Below are three examined approaches:

3.1 FlipBack: Automatic Targeted Protection against Silent Data Corruption

FlipBack (11) is an approach designed to address the increasing issue of SDC in computers due to the diminishing size of transistors. At its core it is a software-based method which focuses on control data and field data, differentiating the protection offered based on their properties.

To accomplish this, FlipBack uses compile-time analysis and program markup to identify critical data affecting control flow. Then, it deploys a runtime-guided replication technique that selectively replicates computation, enhancing the system's resilience to SDCs. The method further employs compiler slicing pass to minimize recomputation time and memory overhead, and it also uses a runtime system component to handle local checkpoint/recovery, recomputation, and verification. To safeguard transient calculations from bit flips, FlipBack incorporates selective instruction duplication, thus preventing severe consequences like program crashes or SDCs. Shown below are the two main algorthims outlined in the paper responsible for the fault based tolerance.

In (11) FlipBack's performance was assessed using LLFI(9), a fault injection tool designed to simulate bit flip induced soft errors in parallel programs, with the analysis involving two proxy applications (Miniaero and Particle-in-cell) and a micro-benchmark (Stencil3D) in thousands of failure injection experiments. The data demonstrated FlipBack's efficacy in detecting and rectifying the majority of bit flips that would have otherwise led to inaccurate results. Notably, FlipBack ensured all bit flips were captured and an automatic local recovery was enacted, enabling seamless program execution devoid of crashes or hangs, with the performance overhead decreasing as the parallel efficiency of the program declined due to the idle time utilization. When compared to traditional checkpoint/restart strategies, FlipBack proved to be a more efficient and scalable solution for managing soft errors. Traditional approaches may be effective for infrequent soft errors, but their efficiency diminishes as the error rate and application scale rise. In contrast, FlipBack exhibited less than 10% overhead for the tested applications and offered protection against bit flips leading to incorrect results, a feat unattainable with the checkpoint restart strategy. Additionally, FlipBack's runtime guided replication can detect SDCs in memory and bit flips in store instructions, which represent a unique point of failure for software-based instruction duplication.

```
Input:
f: the targeted function to perform slicing on
c: set of control variables
Output: slices: the program slice for recomputation
// search for slicing criterions
foreach Instruction I in f do
    if Defs(I) ⊂ c or I sends messages then
        criterions.push(I);
    end
end
while !criterions.empty() do
    I←criterions.top(); criterions.pop();
    if !I.processed() then
        slices.push(I);
        // data flow analysis
        foreach Values I′ in Uses(I) do
            foreach Instruction I″ in Defs(I′) do
                if I″ may lead to I then
                    criterions.push(I″);
                end
            end
        end
        // control flow analysis
        foreach BasicBlock B that may lead to I do
            criterions.push(B.getTerminator());
        end
    end
end
```

Figure 2: **Algorithm 1:** Slicing pass to find recomputation region

```
Input: o: the original full fledged chare
s: the shadow chare
// RTS receives a message M for o
checkpointControl(o);
checkpointControl(s);
restart←true;
while restart do
    // buffering outgoing messages
    o.invoke(M); s.invoke(M);
    if compareControl(o, s) and compareMsgs(o, s) then
        restart←false;
        sendMsgs(o); deleteMsgs(s);
    end
    else
        restartControl(o); restartControl(s);
    end
end
```

Figure 3: **Algorithm 2:** Workflow in RTS

3.2 Recovery Algorithm for Convolutional Neural Networks

The next technique in question is a recovery algorithm for Convolutional Neural Networks. This technique was developed ot counteract SDC in synaptic storage (13). It highlights the resource-intensity of CNNs, which has been mitigated in part by the rise of GPUs and distributed computing, but soft errors, leading to SDC, remain a significant challenge. To address this, the paper proposes a software-based recovery algorithm that uses Association Rule Mining (ARM) techniques, specifically the FP-Growth Algorithm, to generate rules for identifying and recovering corrupted bits of memory cells from the weights of a pre-trained CNN. This rule-mining approach takes into account the significant bits and their association for recovery, thus enhancing the robustness of the CNN in volatile conditions.

In Figure 4, the accuracy of an AlexNet model is plotted on the y-axis against the percentage of soft errors on the x-axis. Soft errors are calculated by randomly selecting one of the 32 bits for each weight in the network and determining the percentage of errors based on the total number of weights.

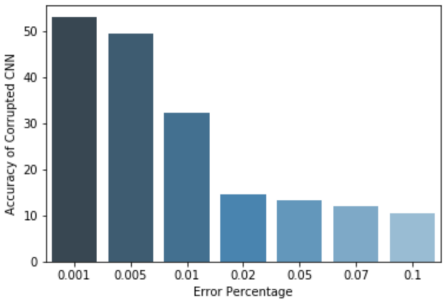

Figure 4: Accuracy of CNN after corruption vs. percentage of error

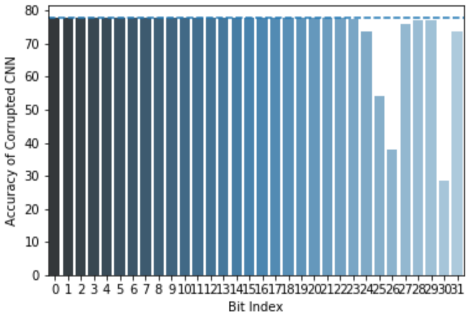

Figure 5: Accuracy of CNN after corruption (random layer) vs. bit index

The results indicate a sharp decline in accuracy as the percentage of soft errors increases, reaching a saturation point at 0.05. It is important to note that the initial accuracy of the trained model is 77.22%. Subsequent sections explore how this behavior changes when isolating specific layers or bit indices.

On the other hand, Figure 5, reveals insights into the impact of Soft Data Corruption (SDC) on the accuracy of the CNN with respect to different bits. Before bit 25, there is minimal influence observed. To investigate this further, a controlled experiment was conducted by injecting SDC into the last seven bits (25th - 31st). Each iteration involved randomly selecting a bit and weight to assess the resulting accuracy.

This approach builds a recovery model by representing the weights of the pre-trained CNN as a matrix of binary values and identifies the most significant bits to prioritize the recovery of corrupted bits. The authors use AlexNet CNN trained on the CIFAR10 dataset for this demonstration, although the process is extendable to other CNNs. In building the recovery model, the proposed method incorporates a stochastic approach to generate association rules from binary bit representations, calculating the likelihood of a bit being corrupted. They also analyze the impact of selective hardware-dependent recovery models on the classification accuracy by identifying certain bits that are more prone to errors.

The authors tested this recovery algorithm by conducting sensitivity analysis and error injection on a pre-trained AlexNet to understand the impact of soft errors on classification results. The results demonstrated that while accuracy did decrease with increasing levels of SDC, the recovery model was effective even at high corruption rates.

3.3 nZDC: A Compiler Technique for Near Zero Silent Data Corruption:

nZDC(3), or near Zero SDC, is an advanced compiler technique that minimizes soft errors in processor designs. It overcomes the limitations of the SWIFT technique (a purely software technique), which leaves around 18% of the committed instructions in MiBench benchmarks for an ARM-V8 architecture unprotected. nZDC introduces a "checking load instruction" and implements load instruction duplication, effectively protecting stores and memory read instructions, respectively.

It also features a robust control flow checking (CFC) mechanism, guarding all control flow(CF) components, including operands of compare instructions, pipeline registers, conditional registers, and branch instructions.

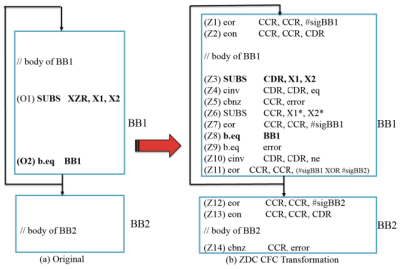

Figure 6: nZDC CFC transformation

The paper found that effectiveness of nZDC against soft errors surpasses that of SWIFT, without incurring any additional performance overhead. In fault injection tests that involved injecting 72,000 faults into various components of the processor, nZDC drastically reduced the failure percentage of most unprotected microarchitectural components. Specifically, for the Load Store Queue (LSQ) component, a significant soft error vulnerability point, nZDC achieved a 0% failure rate. Additionally, it achieved a zero SDC rate for nZDC-protected programs, compared to higher rates for SWIFT. Performance evaluations on an ARM processor showed comparable execution time overheads for both nZDC and SWIFT, at 224% and 213% respectively. However, nZDC was found to completely close register file vulnerable intervals, thereby significantly reducing the failure rate.

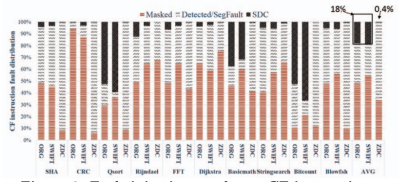

Figure 7: Fault injection results on CF instruction

In conclusion, while these techniques show promise in mitigating SDC, they also underscore the complexity and multifaceted nature of the challenge. Future research in this area should continue to explore a diverse range of strategies, from system-level protections to algorithmic approaches, while also considering the trade-offs in performance, cost, and scalability.

4 Comparison of SDC Mitigation Techniques:

In mitigating SDC, the unique approaches provided by the three discussed papers offer insights tailored to various contexts and system types. The specifics of the application largely influence the most suitable approach.

FlipBack, as introduced in (11), offers a promising solution against SDC by directing bit flips to less critical program sections. While this mechanism reduces the SDC rate in benchmark applications, its wider applicability, system impacts, and resource demands need further investigation.

The recovery algorithm for CNNs in (13) effectively reduces accuracy degradation due to weight corruption. As a robust solution specific to CNN-based image processing tasks, its scalability due to the overhead of maintaining and comparing backup weights remains an open question.

The nZDC technique in (3), a compiler-level approach, significantly reduces SDC rates with minimal performance overhead. However, the complex process of accurately identifying vulnerable instruc-

tions, especially in larger and diverse software programs, underscores the need for sophisticated vulnerability analysis tools.

Comparing these techniques reveals that a one-size-fits-all solution for SDC mitigation does not exist. The chosen method depends on the application specifics, available resources, and system architecture. Furthermore, empirical validation is necessary to understand the actual potential and trade-offs of these approaches in real-world systems.

5 Critiques of SDC Research:

While Flipback (11) proposes an automatic protection mechanism against SDC. Despite its efficacy in reducing SDC probability, the paper doesn't adequately explore the cost-efficiency and potential performance impact of the mechanism, leaving room for further investigation on the balance between mitigation effect and operational impact.

Conversely the paper by narayan, (10) provides valuable insights into the financial costs of SDC detection, but largely overlooks system performance, user experience, and other non-monetary cost impacts.

This is not a comprehensive critiques of the SDC research read but rather is aimed at highlighting what future contributions can be made and where it might be interesting to build off the previously discussed papers.

6 Case Studies of SDC Impact

As SDCs are prevalent across multiple domains, this section aims to provide a more detailed exploration of a notable case study underlining the real-world impacts of SDCs.

6.1 High-Performance Computing

Mentioned above, High-performance computing (HPC) systems are incredibly sensitive to SDCs, given their reliance on accurate and efficient computational outcomes. In a pioneering study involving 48,000 processors in the Los Alamos National Laboratory's HPC system, researchers discovered a far higher occurrence rate of soft errors than previously assumed (7). Interestingly, soft errors did not lead to system-wide failures but subtly corrupted the computational results. Consequently, the reliability of the entire system was called into question, particularly given the high stakes associated with HPC operations, such as climate modelling or nuclear simulations.

However, the detection and correction of SDCs in these large-scale HPC systems is a challenging task due to their size and complexity (1). The report discussed the application of checksum and replication-based SDC detection methods in HPC, providing insights into the challenges associated with large-scale error detection and correction.

7 Challenges in Detecting SDC

The detection of SDCs brings unique challenges due to their silent nature and lack of conspicuous system crashes. This section provides a detailed examination of these challenges.

7.1 Lack of Immediate System Failure

SDC presents unique detection challenges due to its characteristic stealth, altering computational results without causing immediate system crashes or malfunctions (4). Thus, without effective detection mechanisms, these errors can silently persist and escalate.

Despite the daunting nature of SDC detection, methods like SVM-based identification of SDC-vulnerable instructions have been utilized, to address these challenges with a combination of partial fault injection and SVM classification. This approach harnesses key program features to train the classifier, enhancing SDC detection efficacy (16).

Moreover, PVInsiden leverages feature engineering to identify instructions with a higher probability of causing SDCs, such as connector instructions ("connins") and comparison instructions ("cmpins"). Through fault injection experiments, the susceptibility of each register to soft errors is assessed. This novel approach not only significantly reduces the cost of fault injection by 65% but also holds promising potential for transforming SDC detection methodologies.

7.2 Cost and Complexity of Detection

Another challenge with SDC detection is the overhead associated with it. Detection mechanisms can be resource-intensive, negatively impacting system performance and increasing energy consumption (10). Furthermore, certain SDCs may escape detection due to their subtle impact and the complexity of the processes involved.

Additionally, a novel approach to reduce SDC rates is provided by nZDC, a compiler technique that offers near-zero SDC (3). This technique highlights the need for innovative solutions to address the challenges associated with detecting and mitigating SDCs.

7.3 SDCs in Large-scale Systems

As previously mentioned, the scale and complexity of contemporary computational systems introduce additional challenges for SDC detection. Large-scale systems like Exabyte-scale database systems pose unique difficulties due to the volume of data and the number of operations carried out (1). This necessitates detection and prevention mechanisms designed to cope with the scale of these systems without imposing a significant performance overhead.

In the context of parallel computing, Sirius, a method using Neural Network Based Probabilistic Assertions, has been proposed for SDC detection (14). This approach underscores the importance of leveraging advanced techniques such as machine learning for efficient and accurate SDC detection in large-scale and complex systems.

Sirius is designed to detect SDC in parallel programs by leveraging neural networks and the concept of spatial and temporal locality in critical variables. The tool captures the values of selected variables during runtime, considering their spatial location and time-step iteration numbers. By analyzing the spatial and temporal patterns, Sirius generates probabilistic assertions that act as indicators of silent data errors. The evaluations of Sirius involved benchmarking studies on two parallel benchmark applications, LULESH and CoMD, where assertions generated by the neural network model were tested against positive and negative test samples. The results demonstrated the superior performance of Sirius compared to existing techniques, achieving a detection rate of 98% with a false positive rate of less than 0.02, compared to the previous technique with a false positive rate of 0.06.

7.4 SDCs in Neural Networks

In addition to general computational systems, specialized systems like artificial neural networks also face challenges in SDC detection due to their unique computational paradigms (15). Traditional error detection mechanisms might not be directly applicable or efficient for these systems, requiring customized solutions. The study provides an example of such a tailored solution, where a novel online error detection technique is proposed for ANNs. This stresses the necessity for error detection mechanisms specifically designed for neural networks to cope with the growing use of such systems in various applications.

Overall, the inherent challenges in detecting SDCs necessitate a concerted effort to develop efficient and reliable detection and mitigation strategies. This includes research into specialized solutions for different computational paradigms and the creation of robust, general-purpose detection mechanisms that can be applied across a wide array of systems.

8 Future Directions

While substantial progress has been made in SDC mitigation techniques, research needs to continue in several key areas. First, techniques must improve the balance between mitigation effectiveness and system performance, minimizing computational overhead and performance degradation. Scalability

of solutions is essential. A robust, sophisticated set of vulnerability analysis tools should be developed to identify susceptible code areas in varied software programs. Moreover, it's essential to incorporate SDC mitigation into machine learning architectures to safeguard the growing prevalence of these systems in different sectors.

The importance of SDC mitigation continues to increase as we become more reliant on computational systems. Future research should not only enhance existing techniques but also innovate in order to confront SDC effectively. This might be a daunting task, but the continuous effort in this field is significantly relevant to society, aiming for a future with reliable and trusted computational systems. The implications of SDC can be far-reaching, potentially catastrophic in sectors like machine learning, and therefore the call for focused research and development in SDC mitigation is both timely and crucial.

9 Acknowledgements

I would like to acknowledge the help and comments I received from both Dr. Mohamed Abdelfattah and Yash Akhauri. Additionally, I would like to acknowledge the use of ChatGPT and other generative models in the creation of this paper.

References

[1] Bacon, D. F. (n.d.). Detection and Prevention of Silent Data Corruption in an Exabyte-scale Database System.

[2] Constantinescu, C., Parulkar, I., Harper, R., & Michalak, S. (2008). Silent Data Corruption, 2014; Myth or reality? 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN), 108–109. https://doi.org/10.1109/DSN.2008.4630077

[3] Didehban, M., & Shrivastava, A. (2016). nZDC: A compiler technique for near zero silent data corruption. Proceedings of the 53rd Annual Design Automation Conference, 1–6. https://doi.org/10.1145/2897937.2898054

[4] Dixit, H. D., Boyle, L., Vunnam, G., Pendharkar, S., Beadon, M., & Sankar, S. (2022). Detecting silent data corruptions in the wild (arXiv:2203.08989). arXiv. http://arxiv.org/abs/2203.08989

[5] Dixit, H. D., Pendharkar, S., Beadon, M., Mason, C., Chakravarthy, T., Muthiah, B., Sankar, S. (2021). Silent Data Corruptions at Scale. arXiv preprint arXiv:2102.11245.

[6] Fiala, D., Mueller, F., Engelmann, C., Ferreira, K., Brightwell, R., & Riesen, R. (n.d.). Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing.

[7] Hari, S. K. S., Rech, P., Tsai, T., Stephenson, M., Zulfiqar, A., Sullivan, M., Shirvani, P., Racunas, P., Emer, J., & Keckler, S. W. (2020). Estimating Silent Data Corruption Rates Using a Two-Level Model (arXiv:2005.01445). arXiv. http://arxiv.org/abs/2005.01445

[8] Hochschild, P. H., Turner, P., Mogul, J. C., Govindaraju, R., Ranganathan, P., Culler, D. E., & Vahdat, A. (2021). Cores that don't count. Proceedings of the Workshop on Hot Topics in Operating Systems, 9–16. https://doi.org/10.1145/3458336.3465297

[9] Lu, Q., Farahani, M., Wei, J., Thomas, A., Pattabiraman, K. (2015). LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults. In 2015 IEEE International Conference on Software Quality, Reliability and Security (pp. 11-16). doi: 10.1109/QRS.2015.13

[10] Narayan, S., Chandy, J. A., Lang, S., Carns, P., & Ross, R. (2009). Uncovering errors: The cost of detecting silent data corruption. Proceedings of the 4th Annual Workshop on Petascale Data Storage, 37–41. https://doi.org/10.1145/1713072.1713083

[11] Ni, X., & Kale, L. V. (2016). FlipBack: Automatic Targeted Protection against Silent Data Corruption. SC16: International Conference for High Performance Computing, Networking, Storage and Analysis, 335–346. https://doi.org/10.1109/SC.2016.28

[12] Qutub, S., Geissler, F., Peng, Y., Grafe, R., Paulitsch, M., Hinz, G., Knoll, A. (2022). Hardware faults that matter: Understanding and Estimating the safety impact of hardware faults on object detection DNNs (Vol. 13414, pp. 298˘318). https://doi.org/10.1007/978-3-031-14835-4$_2$0

[13] Roy, A., & Ludwig, S. A. (2020). Recovery algorithm to correct silent data corruption of synaptic storage in convolutional neural networks. International Journal of Hybrid Intelligent Systems, 16(3), 177–187. https://doi.org/10.3233/HIS-200278

[14] Thomas, T. E., Bhattad, A. J., Mitra, S., & Bagchi, S. (2016). Sirius: Neural Network Based Probabilistic Assertions for Detecting Silent Data Corruption in Parallel Programs. 2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS), 41–50. https://doi.org/10.1109/SRDS.2016.016

[15] Vassiliadis, V., Parasyris, K., Antonopoulos, C. D., Lalis, S., & Bellas, N. (2021). Artificial neural networks for online error detection (arXiv:2111.13908). arXiv. http://arxiv.org/abs/2111.13908

[16] Yang, N., & Wang, Y. (2019). Identify Silent Data Corruption Vulnerable Instructions Using SVM. IEEE Access, 7, 40210–40219. https://doi.org/10.1109/ACCESS.2019.2905842