SEVI: Silent Data Corruption of Vector Instructions in Hyper-Scale Datacenters

Yixuan Mei
yixuanm@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Shreya Varshini
shreyavarshini@meta.com
Meta Platforms Inc.
Menlo Park, CA, USA

Harish Dixit
hdd@meta.com
Meta Platforms Inc.
Menlo Park, CA, USA

Sriram Sankar
sriramsankar@meta.com
Meta Platforms Inc.
Menlo Park, CA, USA

K. V. Rashmi
rvinayak@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Abstract

Silent Data Corruption (SDC) poses a reliability threat in modern datacenters. These insidious errors evade detections and propagate incorrect results throughout the system. Companies including Google, Meta, and Alibaba have reported SDC incidents affecting their production. In this paper, we present the first comprehensive instruction- and application-level analysis of vector instruction SDCs in hyper-scale datacenters using a two-stage approach. We perform over 78 trillion test rounds in more than 14 billion CPU seconds. Our observations reveal undocumented SDC patterns that provide insights into possible underlying causes and inspire new mitigation strategies. Based on these findings, we propose a low-overhead SDC detection mechanism leveraging in-application algorithm-based fault tolerance. Our method achieves 88% to 100% SDC machine detection rate with a time overhead of only 1.35% even for modestly sized inputs.

CCS Concepts: **• Computer systems organization → Reliability**; **• Hardware → Testing with distributed and parallel systems**.

Keywords: Silent Data Corruption; Datacenter Reliability; Algorithm-Based Fault Tolerance; Vector Instructions; Hardware Faults

ACM Reference Format:
Yixuan Mei, Shreya Varshini, Harish Dixit, Sriram Sankar, and K. V. Rashmi. 2026. SEVI: Silent Data Corruption of Vector Instructions in Hyper-Scale Datacenters. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 16 pages.
https://doi.org/10.1145/3779212.3790217

This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '26, Pittsburgh, PA, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790217

1 Introduction

Modern silicon has achieved remarkable performance improvements through increased clock frequencies and higher core counts, enabled by advances in semiconductor manufacturing technology. However, the growing design complexity and increasing transistor densities have made these processors increasingly susceptible to hardware faults [10, 28, 33]. These hardware faults arise from multiple sources, including design flaws, manufacturing defects, unstable operating conditions, age-related degradation [12, 24] and cosmic radiation [52]. Based on their symptoms, the hardware faults can be categorized into: (1) fail-noisy faults, which trigger immediate hardware exceptions or software errors, and (2) fail-silent faults, which evade detection mechanisms and propagate incorrect results through the system [12, 24, 47]. The latter category, known as Silent Data Corruption (SDC), poses a particularly severe threat to system reliability.

SDC is notoriously difficult to identify and diagnose [4, 11, 27, 38]. The probability of an SDC occurring on a single machine is very low. However, at datacenter scale the aggregate fault rate cannot be ignored [12, 24, 47].

Existing research extensively studied possible causes of SDC through injection of synthetic faults in simulation, for both CPUs [8, 16, 38] and accelerators [21–23]. Some works have also studied SDC in datacenters with a focus on case studies [12, 24] or general software impacts [47]. Despite those efforts, SDC remains a persistent and challenging problem in modern datacenters, with hyperscalers including Google [24], Meta [12] and Alibaba [47] reporting SDC incidents affecting their production workloads. The severity of this threat has prompted industry leaders to establish SDC research initiatives through the Open Compute Project [35].

In this paper, we present the first comprehensive evaluation of vector instruction SDCs in hyper-scale datacenters, at both instruction- and application-levels. We also propose a low-overhead SDC detection mechanism using in-application algorithm-based fault tolerance.

Why vector instructions? Our study focuses on SDC in vector instructions due to their fundamental importance across

numerous applications, their high SDC rate, and the noticeable gap in existing research on this topic. Vector instructions are widely used in compression [18, 30, 31], cryptography [14, 17], media processing [40, 43], scientific computing [2, 42, 51], numerical libraries [3, 20], and CPU backend of deep learning frameworks [1, 6, 15, 39]. They account for ~80% of compute instructions for AI workloads, and ~50% of the residency in hyper-scale workloads across ads, recommendation, database, cache and storage. Prior work [8] shows that vector instructions cause *several orders of magnitude more SDCs* compared to other instruction types through their micro-architecture-level modeling. However, previous works [16, 24, 47] only report existence of SDC in vector instructions, without systematic analysis of its patterns and implications — a gap we want to fill in this paper.

Our study methodology (Sec. 2). We present the first comprehensive evaluation of vector instruction SDCs in hyperscale datacenters using a two-stage approach through our SEVI[1] framework. In SEVI, we first deploy automated fleet-scale SDC tests in the infrastructure to identify potential SDC machines ("SDC Suspects") during routine datacenter operations. We monitored the production fleet housing millions of machines for multiple years. During this period, we identified over 2,500 SDC Suspects. Then, we run long-duration instruction and application level testing on these machines to systematically analyze SDC patterns and their impacts.

Instruction-level SDC analysis (Sec. 3). We perform over 78 trillion test rounds in over 14 billion CPU seconds using the SEVI framework. Our observations reveal new SDC patterns that provide insights into possible SDC causes and inspire new mitigation strategies. They also validate previous findings on real-world SDC software impacts [4, 38, 47, 48]. Compared to prior work [47], we significantly expand both the scale of machines extensively tested (2500s vs. 27) and the test volume (78 trillion vs. tens of millions). We list a few key findings below, others can be found in Sec. 3: (1) SDCs in Fused Multiply-Add (FMA) instructions are most prevalent, accounting for over 92% of observed SDCs; (2) Some SDCs occur with very low frequencies ($<10^{-5}$), requiring long test time to detect; (3) Contrary to prior research, SDC in floating-point operations can affect any output bit, causing high accuracy loss; (4) Memory access instructions have two SDC fault modes: incorrect offset reads (76%) and corrupted reads (24%); (5) Vector instruction SDCs show spatial locality, only affecting a single physical core per processor, implying that selective core disabling can be effective. Also, 98.5% of vector SDCs only affect a single vector lane.

Application-level impact (Sec. 4). To further understand the impacts of vector instruction SDCs on applications, we study the SDC patterns in matrix multiplication (matmul). This choice is motivated by two key factors. First, matmul is widely used in many data center applications, including database [41, 44], machine learning [1, 6, 15, 39], scientific computing [45, 46], and signal processing [54]. Understanding the SDC patterns in matmul and developing low-overhead detection mechanisms would have significant implications across a wide spectrum of applications. Secondly, matmul's heavy reliance on FMA instructions makes it particularly susceptible to SDCs. This vulnerability makes matmul an ideal candidate for investigating SDC susceptibility in real-world applications and developing robust detection methods that complement existing SDC detection mechanisms.

[1]SEVI stands for **S**ilent **E**rror in **V**ector **I**nstructions.

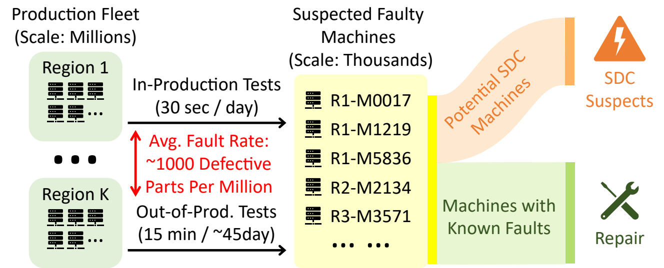


Figure 1. Fleet testing mechanism.

Low-overhead SDC detection (Sec. 5). Current SDC detection frameworks mostly rely on specialized SDC test cases. To detect SDCs with very low frequency, such an approach requires long test times. From workload perspective, the time spent running these tests represents pure overheads. To maintain test coverage while reducing this time overhead for highly prevalent[2] vector (especially FMA) instruction SDCs, we propose to transform certain production workloads into SDC detection canaries via in-application algorithm-based fault tolerance. This technique provides error detection capability using low-overhead parity computation. We implement our detection mechanism in matmul, allowing any application using matmul to detect SDCs. This approach enables detection to occur concurrently with normal operation. It complements existing fleet tests by allowing them to focus on other less prevalent SDCs. Our method effectively detects 88% to 100% SDC machines during our real machine evaluation, with a time overhead of only 1.35% even for modestly sized matrices, such as 1024×1024 matrices.

Our contributions in this paper include:

- The first comprehensive evaluation of SDC in vector instructions in hyper-scale datacenters, which significantly expands both the scale of machines extensively tested (2500s vs. 27) and the volume of test rounds executed (78 trillion vs. tens of millions) compared to prior work.
- Identification of new instruction- and application-level SDC patterns that provide insights into possible SDC causes and inspire new mitigation strategies.

[2]Throughout the paper, we use the term "prevalent" to refer to the number of machines affected by a specific SDC type, not how frequently the SDC manifests during execution.

- Showing the efficacy of using an ABFT-based approach that runs detection alongside normal workloads for low-overhead vector SDC detection.
- An instruction-level SDC test framework (SEVI) that can also be used to study SDC in other instruction sets.

Our open-source implementation is available at https://github.com/Thesys-lab/SEVI-ASPLOS26.

2 Study Methodology

2.1 Study Method Overview

We conduct this study in Meta's global hyper-scale datacenters comprising multiple millions of machines. These machines are equipped with common datacenter CPUs from leading semiconductor manufacturers, spanning seven recent architecture generations. They run 16 workload families including AI (training and inference), shared services, security, storage, database, caches, and gaming applications. Meta's infrastructure is one of the largest deployments of contemporary server architectures in production.

To study the vector instruction SDC patterns in detail while maintaining feasibility at these large scales, we adopt a two-stage approach through our SEVI framework. In SEVI, we first run automated SDC tests (e.g. vendor tests and production code snippets) across the fleet to identify potential SDC machines ("SDC Suspects") during routine datacenter operations (Sec. 2.2). These machines remain in the production fleet, serving workloads and receiving updates identical to other production machines. However, they are virtually flagged to be reservable for long-duration testing. Over multiple years, we have identified more than 2,500 SDC Suspects in the hyper-scale fleet. We then perform extensive vector instruction SDC tests on those production machines to analyze the SDC patterns and impacts (Sec. 2.3).

2.2 Identification of SDC Suspects in the Fleet

As illustrated in Fig. 1, the fleet employs two concurrent SDC testing mechanisms (*Out-of-Production Tests* and *In-Production Tests*) to detect suspects of *all types of SDCs* related to cloud workloads. Both mechanisms use standard silicon vendor test suites and code snippets from *all 16 production workload families* for SDC testing. These test cases cover diverse instructions related to arithmetic, vectors, concurrency, control flow, data movement, floating-point operations, transactional memory, encoding functions, etc..

Out-of-production testing runs during scheduled maintenance windows, with each machine undergoing ∼15 minutes of testing every 45 days. In-production testing provides always-on lightweight coverage by co-locating millisecond-scale test rounds with production workloads, accumulating ∼30 seconds of testing per machine daily. Both mechanisms are integrated into the infrastructure and automatically apply to the entire fleet. To our knowledge, we have the highest vector testing time allocation relative to workload time among hyper-scale SDC research.

Faulty machines that match known non-SDC fault types are routed to repair. Other faulty machines suspected of any SDC failure are virtually flagged as SDC Suspects, which makes them reservable for long-duration testing. Note that all SDC Suspects continue to serve production workloads and receive the same software and kernel updates as the rest of the fleet. This ensures that our testing results represent production scenarios. Over time, as we identify new non-SDC fault types, SDC Suspects exhibiting only those non-SDC faults will be repaired and unflagged.

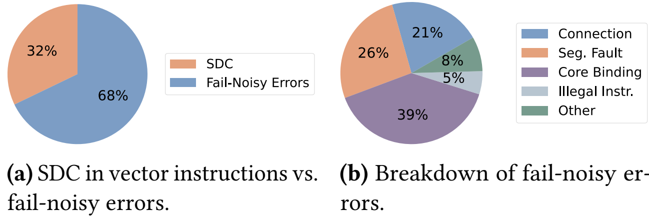
Over multiple years, we have identified over 2,500 SDC Suspects. Testing these SDC Suspects provides a feasible and reliable proxy for understanding fleet-wide SDC patterns. Despite the rigorous fleet testing method, the stealthy nature of SDC means some machines with very low-frequency SDCs may still evade detection, a fundamental limitation of any testing approach. Statistically, these rare evasions are unlikely to significantly alter our findings or conclusions about SDC behavior across the fleet.

2.3 Long-Duration Testing on All SDC Suspects

We conduct comprehensive testing on *all SDC Suspects* to examine how SDC manifests in vector instructions, analyzing at both instruction and application levels.

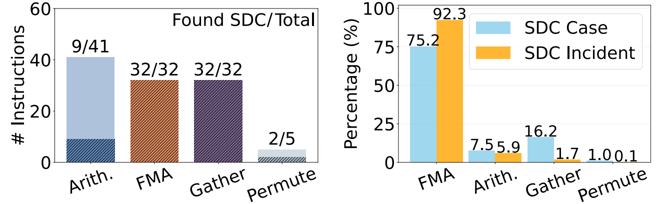
Instruction-level testing. We develop a test framework featuring detailed system status logging and instruction-level isolation. Each test case examines a single instruction, providing precise accountability for observed behaviors. Our current test suite comprises 246 test cases that exhaustively cover all instructions for Advanced Vector Extensions 2 (AVX2), Fused Multiply-Add 3 (FMA3), and Bit Manipulation Instruction sets (BMI1 and BMI2). In our experiments, we execute each test case 1 million rounds on each logical core of each machine. Each round takes a different input and compares the vector output with a reference scalar output. Since these computations utilize different hardware units, it is highly improbable that hardware faults would lead to identical incorrect outputs, providing confidence on the correctness of our testing method. To evaluate the reproducibility of SDC occurrences, we conduct the same experiments twice. Finally, we perform a third test pass using uniform inputs across all vector lanes to investigate potential correlations between SDC and specific lane positions. In total, we run 78 trillion test rounds in 14 billion CPU seconds.

Application-level testing. We implement the matmul-based application-level test suite using NumPy [20] in Python. The underlying implementation heavily relies on vector instructions (especially FMA instructions) and is representative of matmul workloads in production. To capture diverse input patterns in production environments, we test across two



(a) SDC in vector instructions vs. fail-noisy errors. **(b)** Breakdown of fail-noisy errors.

Figure 2. Error breakdown.



(a) Number of instructions that found SDC in each type. **(b)** Breakdown of SDC cases and incidents by instruction type.

Figure 3. Instructions that found SDC and their prevalence.

input value distributions and three matrix sizes. We evaluate bounded inputs [-1,1] simulating normalized data and unbounded inputs accepting any valid floating-point value. Additionally, we evaluate three matrix dimension bounds (10, 25, and 100), randomly selecting actual dimensions within these bounds for each test iteration to simulate varying computational loads. In our experiments, we execute 100K test rounds per machine core for each configuration. For the largest dimension bound (100), we perform 10K rounds per machine to accommodate time constraints. In total, we run 43 billion test rounds in 2.5 billion CPU seconds.

2.4 Benefits of Two-Stage Study Method

The two-stage study method via SEVI allows us to perform testing at much larger scales. The long-duration test reservation on SDC Suspects creates a dedicated environment for comprehensive machine testing. The test time per machine is equivalent to what would be experienced during 21 years[3] of normal fleet operation. It also dramatically expands our testing capacity compared to prior work [47], increasing both the scale of machines extensively tested (2500s vs. 27) and the volume of test rounds executed (78 trillion vs. tens of millions). The SDC Suspects flagging mechanism ensures that the machine composition, configuration stack and operating condition we test on are the same as production. Thus, conclusions drawn from testing SDC Suspects can generalize reliably to the hyper-scale production fleet.

3 Instruction-Level SDC Analysis

Terminology: Each test case is specialized to evaluate SDC in a single vector instruction, as described in Sec. 2.3. An *SDC case* refers to the detection of SDC by a test case on a specific machine core. An *SDC incident* refers to a result mismatch during one test round. One SDC case may encompass multiple SDC incidents. Unless otherwise specified, "SDCs" refers to vector SDCs henceforth.

Observation 1: *Overall, we observe 28 million SDC incidents in 400 SDC cases in vector instructions. The fleet-level machine vector SDC rate is approximately 0.072‱.*

All SDC cases have been rigorously validated through multiple independent channels, including automated fleet testing, expert analysis, and confirmation by semiconductor vendors. Our fleet-level SDC detection rate aligns with the findings reported by prior work [47] (0.348‱ during regular testing with approximately 40% in vector instructions). This concordance indicates that our two-stage testing method robustly captures the vector instruction SDCs in the fleet.

Additionally, we observe fail-noisy errors that cause immediate test framework crashes on 38 machines. These fail-noisy errors include unexpected segmentation faults in multiprocessing environments, illegal instruction reported for supported instructions, etc. These errors are not limited to vector instructions. They can result from hardware faults in any CPU component, including cache or transactional memory. While fail-noisy errors require case-by-case manual analysis for root cause identification, they are easily detectable since they abort programs immediately and thus fall outside our paper's focus on SDC. Fig. 2a shows the vector instruction SDC rate versus other fail-noisy errors. Fig. 2b shows a breakdown of the fail-noisy errors we observed.

3.1 General SDC Patterns

Observation 2: *The SDC incidents we observe span four instruction types, among which SDCs in fused multiply-add instructions are most prevalent.*

During our evaluation, 75 of the 246 test cases detected SDC incidents. Those test cases span four instruction types: arithmetic, fused multiply-add (FMA), gather, and permute. Fig. 3a shows the number of instructions that found SDC in each instruction type (only the types that found SDC are plotted to save space). We further analyze the total SDC occurrences of each instruction type and show the results in Fig. 3b. We find that SDCs in FMA instructions are most prevalent, accounting for over 75% of the SDC cases and over 92% of the SDC incidents. Finally, we analyze the number of SDC machines and cores for each instruction and list the top-20 SDC-causing instructions in Fig. 4. We find that 19 out of the 20 instructions are FMA instructions.

The following two are the likely primary factors contributing to the frequent SDC occurrences in FMA instructions. First, FMA operations utilize hardware multiplier units that contain 30× more logic gates than simpler components such as adders, and these multipliers typically lack hardware-level

[3] We test each machine for ~42 hours. In production, each machine receives ~15 minutes of testing within a 45 day period.

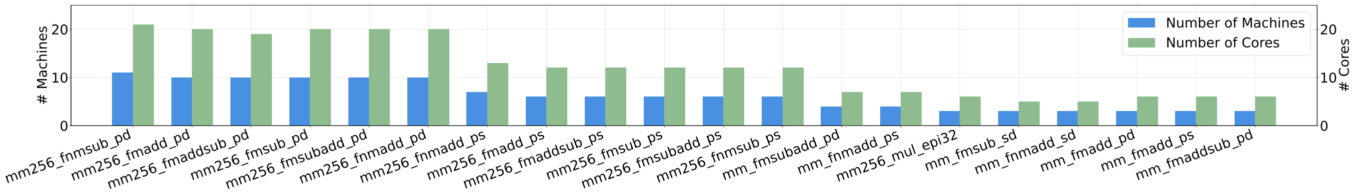


Figure 4. Top-20 SDC-causing vector instructions based on number of failed machines/cores.

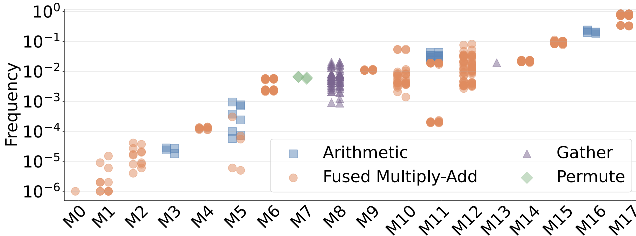


Figure 5. Distribution of SDC detection frequencies across affected machines. X-axis: machines (cores from the same machine shown in adjacent columns). Y-axis: SDC frequency. Each data point represents an SDC-detecting test case. We color the data points based on the instruction tested by the SDC-detecting test case. One machine core may detect SDC in multiple instructions, thus having multiple data points.

protection [8]. This high gate count likely increases their vulnerability to failures. Second, the hardware floating-point vector multiplication units are shared across 32 different instructions. This intensive utilization accelerates component wear, likely leading to a higher SDC probability.

Implications: The prevalence of SDC in FMA instructions presents a reliability challenge for data center operations: these instructions are extensively utilized across numerous data center applications, many of which require computational accuracy (e.g. scientific computing, database acceleration, etc.). This vulnerability is further exacerbated because FMA instructions typically operate within computation pathways rather than control logic, increasing the likelihood that corrupted results propagate to the output without triggering system crashes or detectable errors. However, the widespread deployment of FMA instructions also offers an opportunity for detection: implementing high-coverage SDC detection in a frequently executed representative workload (a "canary" workload) can effectively protect all FMA-utilizing applications, as detection of corruption would enable prompt removal of faulty hardware from the fleet. This observation motivates the in-application Algorithm-Based Fault Tolerance (ABFT) method in Sec. 5.

Observation 3: *SDC occurrence patterns exhibit significant variation across machines. However, within each machine, faulty instructions show consistent SDC frequencies possibly due to single defective hardware unit per core.*

Fig. 5 shows the distribution of SDC detection frequencies across all machines we found SDC on. On different machines, we observe significant variations in both the number of test cases detecting SDC and their detection frequencies. While some machines detect SDC in only a small number of the 246 instruction-level test cases, others identify it in up to 32 cases. The average detection frequency spans several orders of magnitude, ranging from one in a million rounds to nearly all of the rounds. Within individual machines, the detection frequency typically remains consistent within one to two orders of magnitude. Upon analysis of the hardware components associated with faulty instructions, we hypothesize that each defective core only exhibits a single point of failure in one specific hardware unit (e.g. vector multiplication, shuffle, or memory units). This explains the consistent SDC frequencies observed across different faulty instructions on the same defective core, as these instructions, despite their functional differences, ultimately utilize the same compromised hardware component. The spatial locality of SDC we observe in Obs. 13 also supports this hypothesis.

In cases where a machine has two failing cores, they demonstrate similar patterns, exhibiting similar sets of SDC-detecting tests and detection rates. This is because both cores correspond to the same physical core. We will discuss the hardware patterns further in Sec. 3.3.

Implications: To optimize testing efficiency while preserving coverage, SDC tests should prioritize comprehensive hardware unit coverage rather than exhaustive instruction coverage. The data gathered on hardware unit error rates can provide valuable insights for future architecture development by identifying vulnerabilities in current designs.

Observation 4: *Permute and gather instructions demonstrate consistent SDC detection frequencies. Meanwhile, among the two types of FMA instructions, 128-bit ones only show SDC cases with relatively high frequency, while 256-bit ones show additional low-frequency SDC cases.*

Fig. 6 shows the distribution of SDC detection frequencies across different test cases. For permute and gather instructions, the SDC detection frequencies remain relatively consistent across test cases. However, FMA instructions exhibit a notable pattern correlated with bit width. Compared with their 128-bit counterparts, 256-bit FMA instructions detect SDC across a broader range of machines, with the additional machines showing very low detection frequency ($< 10^{-5}$).

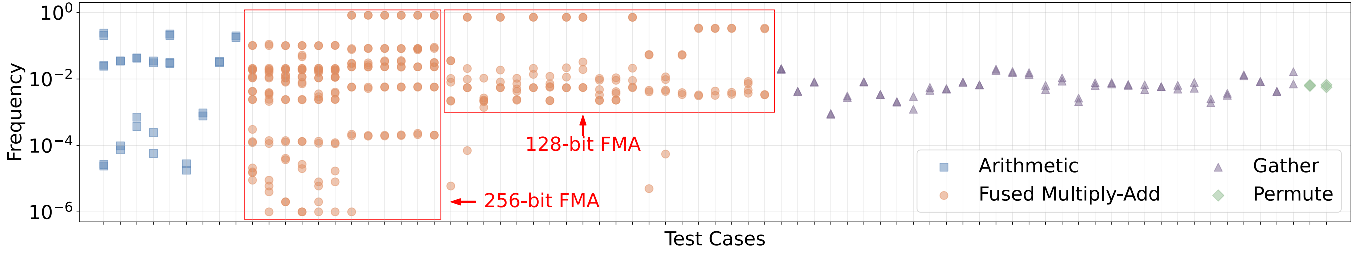
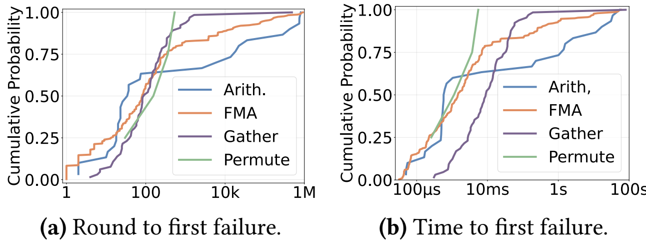


Figure 6. Distribution of SDC detection frequencies across test cases. The x-axis shows test cases, while the y-axis indicates detection frequency. Each data point represents a processor core that detected SDC incidents for the corresponding test case.



(a) Round to first failure. **(b)** Time to first failure.

Figure 7. Cumulative distribution of round to first failure and time to first failure for the SDC cases.

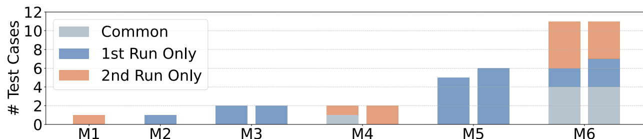


Figure 8. Number of test cases detecting SDC in two runs with identical setups. Cores from the same machine are shown in adjacent bars. Only machine cores exhibiting inconsistent SDC detecting test cases between runs are displayed.

There are two likely architectural causes for this phenomenon. First, 256-bit FMA instructions activate more transistors during execution due to its wider data path, increasing the potential sources of SDCs. Second, 256-bit FMAs consume 2× more power than their 128-bit counterparts. The higher power consumption leads to larger current fluctuations when switching between scalar and FMA instruction execution. Such fluctuations are more likely to induce voltage variations that exceed safe operating margins [26, 53], potentially causing computation errors.

Implications: To detect these and even lower frequency SDC cases in FMA instructions, conventional isolated testing would introduce prohibitive time overheads. This motivates us to turn production workloads into tests to increase test coverage while minimizing additional testing costs (Sec. 5).

Observation 5: *For more than 80% of the SDC cases, time to first failure is shorter than 1 second.*

Fig. 7 shows the cumulative distribution of round to first failure and time to first failure for the SDC cases in each instruction type. For more than 80% of SDC cases, the first SDC incident is found within 10K rounds (1 second).

Implications: For clusters without SDC detection mechanisms, implementing even basic detection would yield substantial benefits with minimal time overhead. However, the remaining 20% of SDC cases represent a long tail that requires continuous refinement of detection mechanisms.

Observation 6: *Test-case-level reproducibility is determined by SDC frequency rather than instruction type.*

To evaluate the reproducibility of SDC, we conduct two complete test suite runs using identical setups. Our analysis reveals that among the 19 machines exhibiting SDC in either run, 6 show inconsistent SDC detecting test cases between runs. Fig. 8 shows the distribution of SDC-detecting test cases across both runs for these 6 machines. We find that the majority of inconsistent test cases detect SDC at very low frequency (10^{-5} to 10^{-6}), with only a few exceptions that detect SDC with frequency up to 10^{-3}. These exceptions primarily involve gather instructions, whose behavior depends on cache states uncontrollable at software level. We will elaborate on this in the next observation. We observe non-reproducible test cases from all instruction types.

Implications: Since SDC frequency is the major factor here, further increasing the number of test rounds will allow stable detection of those low-frequency SDCs.

Observation 7: *Input-level reproducibility is determined by instruction type. Instructions involving memory access have a poor input-level reproducibility due to cache states.*

Following the last observation, we further analyze SDC reproducibility when executing each instruction on the same hardware and inputs. Fig. 9 shows the reproduction rates across different test cases. A reproduction rate of n% indicates that SDC can be consistently reproduced on n% of all SDC-inducing inputs. Our results show that the reproduction rate for Arithmetic, FMA, and permute instructions exhibits significant variation across different machines (almost 0% to 90%). In contrast, gather instructions consistently exhibit near-zero reproduction rates. This phenomenon occurs because gather instructions involve memory access, which

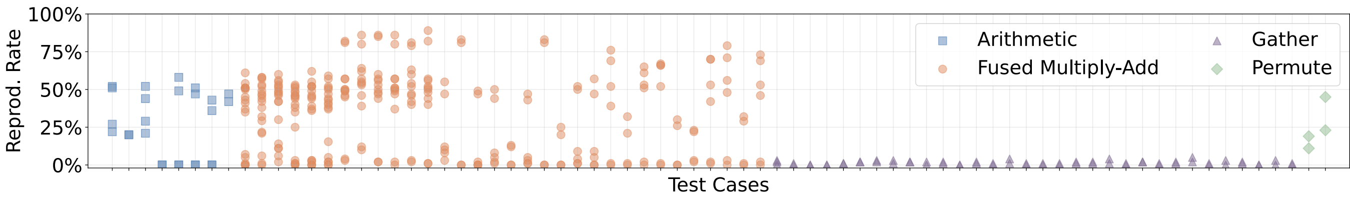


Figure 9. Reproduction rate of the test cases with identical inputs. Each data point is a processor core that detects SDC with the corresponding test case. We show test cases for different operation types in different colors.

highly depends on cache states. Between different runs, maintaining identical cache states is virtually impossible from the software level, as these states are influenced by numerous factors including OS scheduling decisions, memory allocation patterns, and other concurrent test processes. This inherent variability in cache behavior extends to production workloads as well, making SDC in instructions involving memory access fundamentally difficult to reproduce in general.

Implications: Our analysis provides the first quantified measurement of SDC reproduction rates at the instruction level, which refines the broad observations of SDC reproducibility in prior work [12, 24, 47]. The generally low reproduction rate implies that a single successful test execution does not guarantee the absence of SDCs; rather, it necessitates continuous testing across the fleet to effectively identify and isolate machines with SDC.

3.2 Instruction Type Specific Patterns

Observation 8: *For FMA instructions, we identify a significant input pattern in 37% of the SDC cases, potentially attributable to a common hardware vulnerability. Contrary to prior work, we observe three distinct output bit patterns, indicating that SDC can affect any output bits including exponent bits, rather than just mantissa bits. High accuracy loss is probable.*

Our analysis identifies a significant pattern in input bit values across 113 of the 301 SDC cases (37%) involving FMA operations, observed across 7 distinct machines. In these SDC cases, specific bit positions exhibit a strong statistical bias, with probabilities exceeding 0.8 for a value of 0. Fig. 10 (left) shows an example of this pattern. The CPUs on these machines span multiple architectures from two different manufacturers, indicating that this pattern might arise from a common design/manufacture vulnerability. Further examination of these biased bits reveal different characteristics between FP32 and FP64 FMA operations, as shown in Fig. 10 (right). In FP32 operations, the biased bits predominantly appear in the mantissa of input values. In contrast, FP64 operations exhibit this pattern primarily in the exponent bits of inputs. This divergence likely stems from the different hardware implementations for the same functional unit in FP32 and FP64. Despite the potential architectural differences, our data shows that FP32 and FP64 FMA instructions show similar number of SDC cases (126 versus 175).

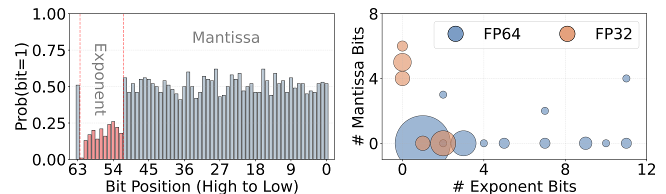
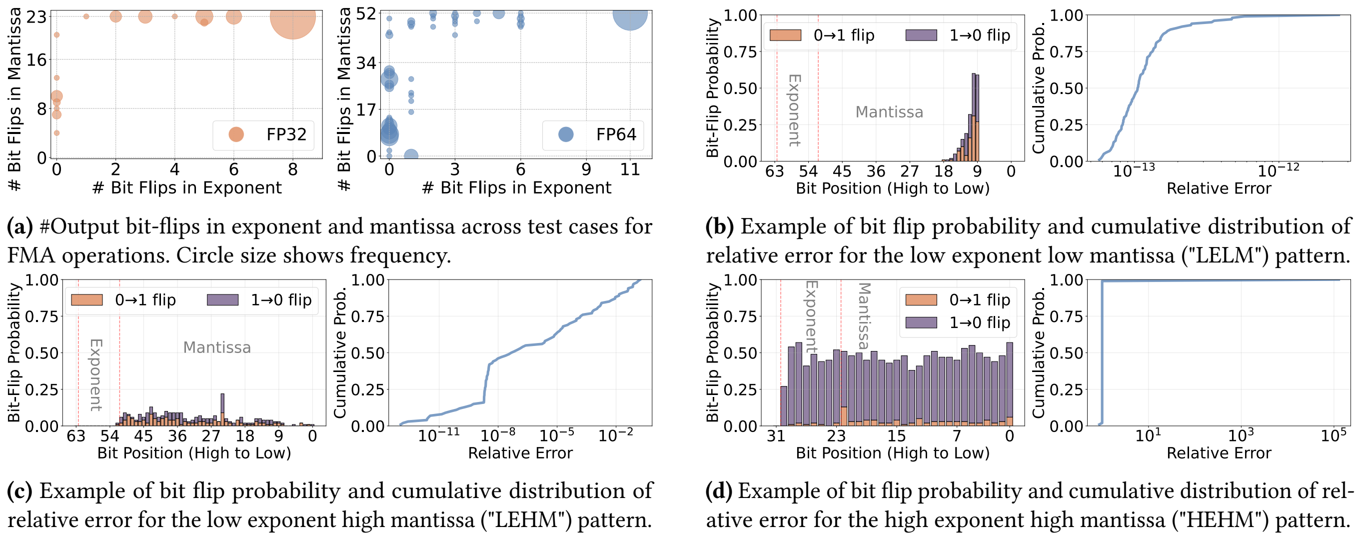


Figure 10. Left: An example of bias in input bit values. **Right:** Distribution of the number of biased exponent and mantissa bits in the inputs across SDC cases. The circle size in the figure indicates occurrence frequency.

We further analyze the bit-flip distributions in FMA operation outputs. Out of the 301 SDC cases, 65 cases on two machines show sign bit flips, causing significant relative error. For exponent and mantissa bits, Fig. 11a shows the correlation between the number of bit-flips in these two fields. For our analysis, if the number of flipped bit positions in all SDC incidents is less than one-quarter of the total number of bits in a field, we classify the SDC case as showing low number of flipped bits in the field. Based on this criteria, we observe three distinct bit-flip patterns (with examples in Fig. 11): (1) the "LELM" pattern (Low Exponent, Low Mantissa): 62 SDC cases, 6 machines (Fig. 11b), (2) the "LEHM" pattern (Low Exponent, High Mantissa): 61 SDC cases, 8 machines (Fig. 11c), and (3) the "HEHM" pattern (High Exponent, High Mantissa): 178 test cases, 5 machines (Fig. 11d). The first two patterns, characterized by low exponent bit-flips, result in only minor accuracy losses. In contrast, the "HEHM" pattern led to high accuracy loss. From machine perspective, 7 of the 13 machines consistently display a single pattern across all their SDC cases. The remaining 6 machines alternated between two adjacent patterns ("LELM" and "LEHM", or "LEHM" and "HEHM").

Implications: The output bit-flip patterns we observe challenge the conventional understanding that SDC in floating-point computations predominantly affects mantissa bits and has minimal impact on accuracy [47]. Our findings demonstrate that bit-flips may also occur in exponent bits, introducing significant magnitude errors to computation results. This poses a substantial reliability threat to applications utilizing FMA instructions. This vulnerability highlights the need for efficient SDC detection mechanisms that can provide good coverage for FMA operations, like the one proposed in Sec. 5.



(a) #Output bit-flips in exponent and mantissa across test cases for FMA operations. Circle size shows frequency.

(b) Example of bit flip probability and cumulative distribution of relative error for the low exponent low mantissa ("LELM") pattern.

(c) Example of bit flip probability and cumulative distribution of relative error for the low exponent high mantissa ("LEHM") pattern.

(d) Example of bit flip probability and cumulative distribution of relative error for the high exponent high mantissa ("HEHM") pattern.

Figure 11. Numerical patterns of FMA operation outputs.

Observation 9: *There are two distinct fault modes in SDC in memory access instructions: they are either caused by incorrect offset reads (76%) or corrupted reads (24%).*

Our analysis reveals two distinct fault modes in memory access instructions (e.g. gather instructions). In our experiments, we allocate a 256-byte memory space for instruction reads and record all contents within this space. The predominant fault mode (76% of SDC incidents) involves instructions reading valid data from incorrect offsets, indicating errors in the offset/address calculation logic. The remaining 24% of incidents involve instructions reading invalid data, either because they access memory outside the allocated space or because data becomes corrupted in cache or CPU internal data paths. Notably, we observe one SDC case where a gather instruction consistently reads 0 in a particular vector lane. Finally, across all observed cases, even when instructions accessed invalid addresses, no system crashes occurred—unlike typical software-level invalid access exceptions.

Implications: Unlike SDC in cache/registers, these faulty memory access instructions predominantly read valid data but from incorrect memory offsets, introducing subtle errors that are challenging to locate in production. Such errors are even harder to reproduce due to the cache state dependencies we previously discussed. Therefore, robust fleet testing mechanisms should incorporate specific detection strategies for memory access anomalies, particularly focusing on access pattern validation to identify offset-related corruptions.

Observation 10: *Unlike FMA instructions, SDCs in vector arithmetic instructions exhibit no significant input bit patterns. The output bit-flip patterns are also highly machine-specific.*

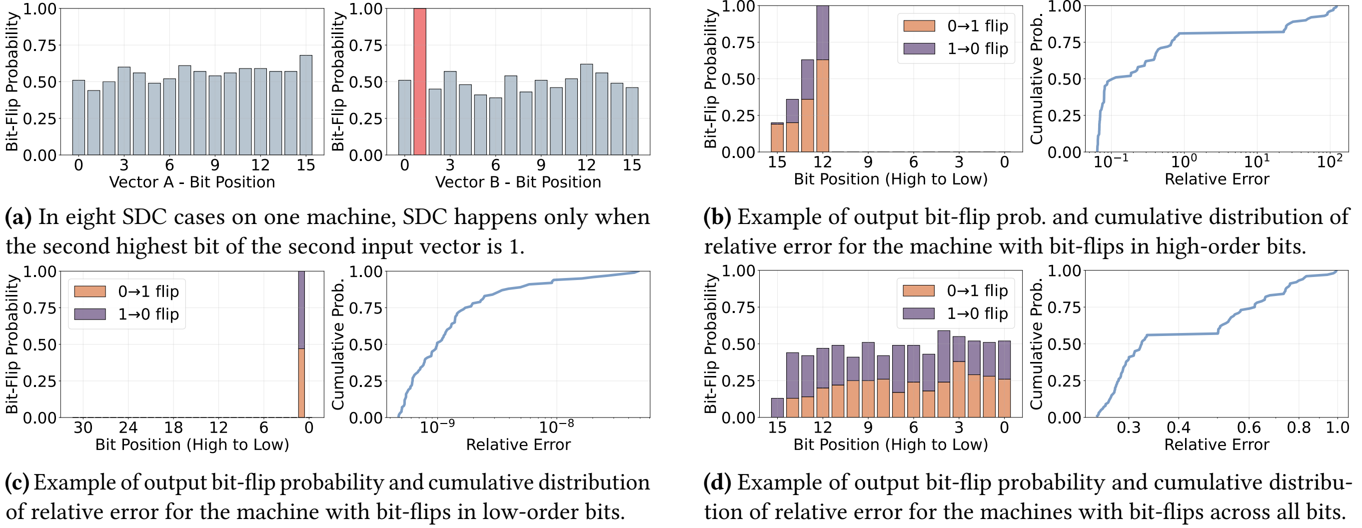
Unlike FMA instructions, our analysis reveals that the majority of SDC cases (22 out of 30) involving vector arithmetic instructions exhibit no statistically significant bias in their input bit patterns. However, we observe a notable exception on one machine where 8 SDC cases related to INT16 and UINT16 multiplication operations demonstrate a consistent pattern (Fig. 12a): in every SDC incident within this subset, the second highest bit of the second input vector is invariably 1. Unfortunately, without access to the proprietary hardware design, we cannot determine the underlying hardware mechanism responsible for this distinctive pattern.

For the bit-flips in the output, our analysis indicates that the patterns are highly machine-specific. Within each faulty machine, SDC cases exhibit remarkably consistent patterns. Specifically, one machine mostly experiences bit-flips in the high-order bits (Fig. 12b), another machine exhibits bit-flips exclusively in low-order bits (Fig. 12c), and two other machines shows bit-flips across all bit positions (Fig. 12d).

3.3 Hardware Patterns

Observation 11: *We observe vector instruction SDCs on 7 of the 17 CPU models, and 4 of the 7 architectures. The SDC rates of architectures range from 4‰ to 24‰ within SDC Suspects.*

Over multiple years, the fleet testing mechanisms have identified SDC Suspects across 17 CPU models and 7 CPU architectures. For proprietary reasons, and in line with the practices in prior work [47], we are unable to disclose specific vendor or architecture names in the paper. Instead, we identify these architectures using anonymized labels and detail their key specifications in Table 1. Figure 13 (left) shows the architectural distribution of SDC Suspects. There are a limited number of machines with A4, A5, and A7 architectures because these are very recent architectures that have not operated long in the production fleet. During our extensive instruction-level evaluation, we found vector instruction SDCs on 7 of the 17 CPU models, spanning 4 of the 7 architectures. Fig. 13 (right) shows the SDC rates across all



(a) In eight SDC cases on one machine, SDC happens only when the second highest bit of the second input vector is 1.

(b) Example of output bit-flip prob. and cumulative distribution of relative error for the machine with bit-flips in high-order bits.

(c) Example of output bit-flip probability and cumulative distribution of relative error for the machine with bit-flips in low-order bits.

(d) Example of output bit-flip probability and cumulative distribution of relative error for the machines with bit-flips across all bits.

Figure 12. Numerical patterns of arithmetic operations.

Table 1. Key specifications of the CPU architectures that we detected SDC on. The core ranges correspond to the specific CPU models in which errors were observed.

Arch Label	Release Date	Core Range
A1	Q4'15 - Q1'16	4 ~ 16
A2	Q3'17 - Q1'18	18 ~ 28
A3	Q4'20	18 ~ 26
A4	Q2'21	24 ~ 40
A5	Q1'23	30 ~ 56
A6	Q4'21	36
A7	Q4'22	88 ~ 96

architectures with in SDC Suspects, which ranges from 4‰ to 24‰. We do not show fleet-level hardware SDC rates here due to their sensitive nature as proprietary company data.

Observation 12: *For vector instructions, SDC typically manifests in a single physical core per processor, affecting both logical cores with similar patterns.*

Across all 18 machines exhibiting vector instruction SDCs, we observe that SDC consistently affects only one physical core per machine. In 16 of these machines (89%), SDC manifests on both logical cores associated with the affected physical core, while the remaining 2 machines (11%) show SDC on only one logical core. Notably, our analysis reveals that in 97% of the observed SDC cases, SDC occurs across both logical cores with nearly identical frequencies. This pattern aligns with the typical architectural design where vector instructions predominantly utilize hardware components that are not shared between physical cores (such as multipliers and arithmetic units). Consequently, hardware faults remain highly localized within the specific core rather than propagating across the whole processor.

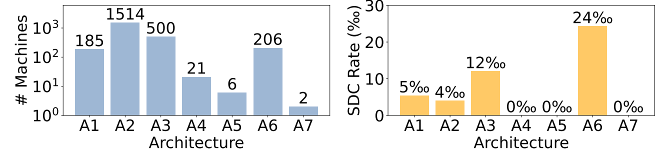


Figure 13. Left: #SDC Suspects by architecture. **Right:** Vector SDC rate by architecture within SDC Suspects.

Implications: Rather than decommissioning an entire processor containing potentially hundreds of cores when SDC manifests in vector instructions, datacenter operators can implement a more resource-efficient approach by selectively disabling only the identified faulty physical core. This targeted mitigation strategy allows continued utilization of the remaining functional cores for non-safety-specific workloads, significantly improving hardware economics while maintaining appropriate reliability boundaries.

Observation 13: *Vector instruction SDCs exhibit strong spatial locality: 98.5% of SDC incidents only affect a single vector lane. For the remaining 1.5% of SDC incidents that affect multiple lanes, 96% only affect adjacent vector lanes.*

We further analyze the physical distribution of the affected vector lanes. Fig. 14 shows the relationship between SDC frequency and the average number of affected vector lanes per SDC incident. Our analysis reveals that even when all lanes process identical inputs, 74% of SDC cases exhibit SDC confined to a single lane. More precisely, across the 28 million observed SDC incidents, 98.5% demonstrate SDC isolated to a single vector lane. Within the remaining 1.5% of incidents where multiple lanes experience SDC, 96% display a pattern where only adjacent lanes are affected. This result indicates a strong spatial locality for the affected lanes. An architectural explanation is that each vector lane possesses

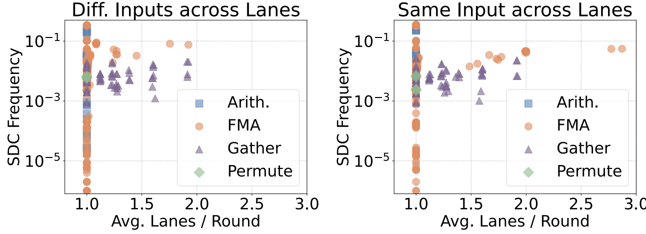


Figure 14. Average number of affected lanes per SDC incident versus SDC frequency. Each data point is a SDC case.

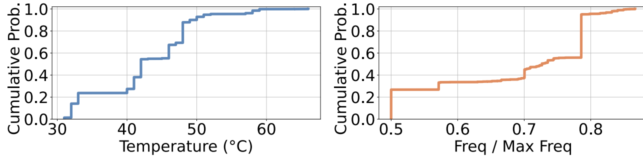


Figure 15. Left: Cumulative distribution of CPU temperature at SDC incidents. **Right:** Cumulative distribution of normalized CPU frequency ratio at SDC incidents.

its own dedicated set of functional units and faults typically manifest as localized impacts on these units, resulting in the observed spatial confinement of errors.

Implications: For applications requiring high reliability, a potential option is to implement selective computation redundancy across non-adjacent vector lanes, enabling efficient SDC detection while still leveraging partial vector instruction performance benefits. However, for structured workloads like matrix multiplication, our method in Sec. 5 can provide high detection with much lower time overheads.

Observation 14: *The majority of SDC incidents occur when CPUs are operating within normal temperature ranges and well below maximum frequency thresholds.*

To normalize CPU frequency, we calculate a *frequency ratio* by dividing the current CPU frequency by the CPU's maximum frequency. Fig. 15 shows the cumulative distribution of both CPU temperature and frequency ratio.

For SDC incidents occurring on the same machine, we observe consistent CPU temperature and frequency. All recorded SDC incidents occur at CPU temperatures below the maximum operational threshold, with a substantial margin before reaching thermal throttling levels. Also, in 95% of cases, the CPU operates at less than 80% of its maximum frequency during SDC incidents. Finally, we confirm a notable correlation between SDC detection frequency and CPU temperature, which is also reported by prior work [47]. As Fig. 16 shows, when CPU temperature increases, we begin to observe the emergence of SDC cases with lower detection frequencies. A linear fit of logarithmic SDC frequency against CPU temperature yields a Pearson correlation coefficient of -0.392 with a statistically significant p-value of 5.68×10^{-12}.

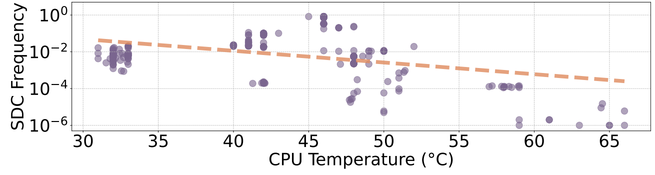


Figure 16. SDC detection frequency versus average CPU temperature for the SDC cases we observe.

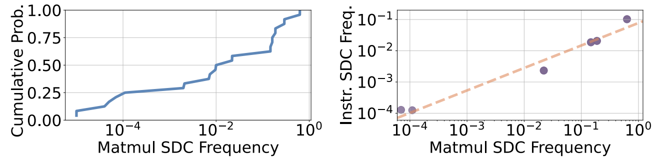


Figure 17. Left: Cumulative distribution of SDC frequency in matmul. **Right:** SDC frequency in matmul versus that in FMA instructions. Each data point is a machine core.

Implications: SDC testing in production should allocate additional test cycles to machines operating at higher temperature. Besides, implementing temperature control mechanisms like the one suggested by [47] can help reduce low-frequency SDCs induced by higher temperature.

4 Application-Level SDC Impacts

Observation 15: *Our framework detects SDC in matmul on 12 machines (24 SDC cases) over 43 billion test rounds. Many root-causes to SDC in FMA. The fleet-level SDC rate is ∼0.048‱.*

The total number of observed SDC incidents in matmul reaches 292K. Fig. 17 (left) shows the cumulative distribution of SDC frequency for the faulty machine cores. Notably, we find that 10 of the 24 SDC cases in matmul occur on machine cores that also exhibit SDC in FMA instructions. Fig. 17 (right) shows the correlation between SDC frequency in matmul and the average SDC frequency of FMA instructions on these affected cores. The relationship demonstrates a strong linear correlation, with a Pearson correlation coefficient of 0.979. This strong correlation suggests that SDC in FMA instructions are likely the root cause of these matmul SDC cases. The remaining 14 matmul SDC cases mostly exhibit very low SDC frequency. Since matmul experiments execute substantially more FMA instructions than instruction-level tests, these SDC cases likely result from very low-frequency FMA SDCs undetected by our instruction-level tests.

Observation 16: *Time to first failure is smaller than 8 seconds (2400 rounds) for 75% of the SDC cases we observe in matmul.*

This aligns with Obs. 5 in Sec. 3. We show the cumulative distribution of time and round to first failure in Fig. 18.

Observation 17: *Machine cores with higher SDC frequency shows more wrong results per SDC incident. Relative error can reach up to 10^{240} due to bit-flips in exponents.*

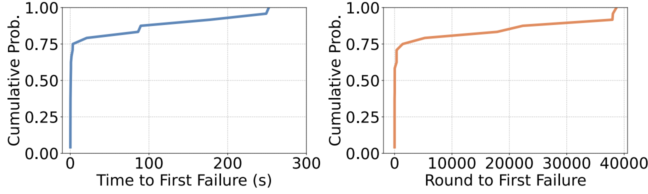


Figure 18. Left: Cumulative distribution of time to first failure. **Right:** Cumulative distribution of round to first failure.

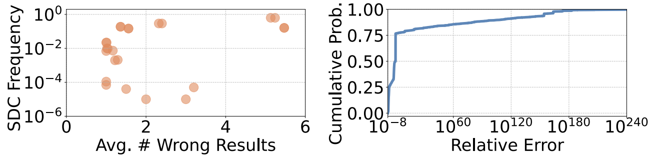


Figure 19. Left: Average number of wrong results per SDC incidence versus SDC frequency of the core. **Right:** Cumulative distribution of relative errors.

We analyze the number of wrong elements in the result matrix for each SDC incident. Fig. 19 (left) shows the relationship between a machine core's matmul SDC frequency and the average number of incorrect results per SDC incident. Result shows that machine cores with higher SDC frequencies generally produce more incorrect results per SDC incident. Fig. 19 (right) presents the cumulative distribution of relative errors for these incorrect results. Our analysis indicates that 75% of the incorrect computations yield relatively minor deviations, with relative errors below 1. However, the remaining 25% exhibit substantially larger relative errors, reaching magnitudes of up to 10^{240}. These extreme deviations can be attributed to bit-flips occurring in the exponent bits during FMA operations (Obs. 8).

Observation 18: *SDC in matmul has good reproducibility, especially on machine cores with high SDC frequency.*

We run the same experiment twice to evaluate the reproducibility. Among the 24 machine cores that detect SDC in either runs, 22 of them consistently detect SDC across both runs at similar frequencies. The remaining two cores detect SDC only in the first run, attributable to their notably low SDC frequency of 10^{-5}. We further examine the correlation between SDC frequency and input-level reproduction rate. As shown in Fig. 20 (left), machine cores with higher SDC frequencies demonstrate higher reproduction rates when processing identical inputs. This relationship is supported by a strong Pearson correlation coefficient of 0.816. Furthermore, Fig. 20 (right) shows that approximately 50% of the machines achieve reproduction rates exceeding 80%.

Observation 19: *Different input value ranges may result in up to 245× difference in SDC frequency on the same core. Increasing input matrix dimension also increases #SDCs observed.*

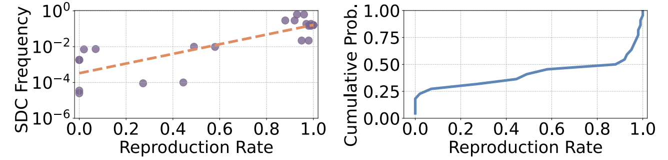


Figure 20. Left: SDC frequency versus reproduction rate. **Right:** Cumulative distribution of reproduction rate.

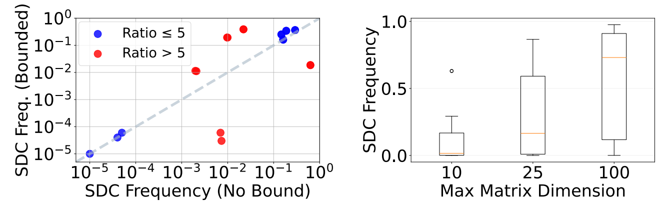


Figure 21. Left: SDC frequency when the value of input matrix elements are bounded to $(-1, 1)$ or unbounded. Each data point is a machine core. **Right:** SDC frequency distribution for tests with different max matrix dimensions.

Our results shows substantial variations in SDC frequency across different machine cores when processing different input value ranges. As shown in Fig. 21 (left), ten machine cores exhibit SDC frequency ratios exceeding 5× (with the maximum observed ratio reaching as high as 245×). This significant variation likely stems from instruction-level sensitivity to specific input bit patterns. Furthermore, Fig.21 (right) demonstrates how SDC frequency correlates with matrix dimensions. Tests involving larger matrix sizes consistently exhibit more rounds with SDC occurrences, which is expected given the increased volume of computations performed in each round.

Observation 20: *SDC in matmul only affects a single physical core and both logical cores associated with it. Higher core temperature induces SDC cases with lower detection frequency.*

This aligns with Obs. 12 and 14 in Sec. 3.

5 SDC Detection via In-Application ABFT

Prior research [8] has shown that vector instructions induce SDCs at rates orders of magnitude higher than other instruction types. In Sec. 3 and 4, our analysis further reveals that SDCs in FMA instructions are most prevalent among all vector instructions, constituting 92% of the observed SDC incidents (Obs. 2). These SDCs can substantially impact computational accuracy (Obs. 8 and 17) without triggering system crashes during execution, thus remaining undetected until affecting final results. Consequently, SDC in vector (especially FMA) instructions presents a significant reliability challenge for the wide range of datacenter applications that heavily depend on these operations.

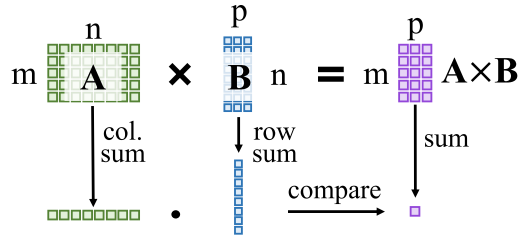


Figure 22. Illustration of our SDC detection mechanism.

These SDCs may occur with extremely low frequency (Obs. 4), making detection particularly challenging. Traditional fleet testing mechanisms require numerous specialized SDC tests, necessitating long test durations. From workload perspective, the time invested in testing represents pure overheads. To maintain comparable or even better test coverage while reducing this time overhead, we propose transforming production workloads into SDC detection canaries, enabling detection to occur concurrently with normal operation. To achieve this goal efficiently, we leverage Algorithm-Based Fault Tolerance (ABFT) [25], which provide error detection capabilities with low-overhead checksum computation.

Our method complements rather than replaces existing fleet testing frameworks. It efficiently identifies the most prevalent vector (especially FMA) instruction SDCs, allowing conventional fleet testing mechanisms to concentrate on other potential failure modes. We focus on SDC detection because once identified, the faulty core/processor can be decommissioned to prevent more SDCs.

ABFT has long been used for protecting scientific computing applications where accuracy is key [5, 9, 13, 49] and more recently in neural network workloads [19, 29, 36]. However, to our knowledge, this work is the first to propose integrating ABFT directly into datacenter application workflows for detecting SDC-inducing cores and the first to validate this method on hardware with real SDC defects, moving beyond fault-injection-based evaluations. Experimental validation on real SDC errors on hardware is essential because error patterns significantly affect detection effectiveness of ABFT techniques. Our results on real SDC errors occurring in vector instructions hardware demonstrate that: (1) ABFT can achieve low-overhead SDC detection in practice, and (2) using only a single row and column checksum vector is highly effective for SDC detection.

5.1 Design of In-Application ABFT

Algorithm-Based Fault Tolerance [25] (ABFT) is a kind of fault detection technique that leverages mathematical invariants in specially designed computational structures for fault detection. It offers much lower compute overhead compared with replication-based methods, which perform same computation twice and compare the results to check errors. This approach has been widely adopted for fault detection in various domains [7, 19, 29, 36, 49]. We adopt ABFT in matrix-multiplication (matmul) to turn any workload using it into SDC detection canaries. We implement the detection mechanism in matmul for two reasons. First, matmul is a basic building block for a wide range of datacenter applications, allowing a wide coverage of our detection mechanism. Second, matmul heavily utilizes FMA and other vector instructions, which are most SDC-prevalent based on Obs. 2 in Sec. 3 and prior work [8].

Implementing detection within matmul safeguards the broader system by using these pervasive operations as canaries to identify faulty hardware. However, ABFT principles are not limited to matmul; they can be extended to other workloads (e.g., FFT) through algorithm-specific adaptations, as demonstrated by TurboFFT [50].

Fig. 22 shows how the detection mechanism works. For a matrix multiplication operation $A \times B$, it first computes a row checksum vector of A and a column checksum vector of B. Then, it takes the dot product of these checksum vectors to produce a final scalar checksum, which is mathematically equivalent to the sum of all elements in the resultant matrix $A \times B$. It leverage this mathematical equivalence as an invariant property for SDC detection. We assume SDC may affect either or both computations (matmul and checksum). However, given that SDC's behavior is highly input dependent, it is unlikely that we get the same wrong checksum from both sides due to SDC. Therefore, once an SDC occurs during computation, it is highly likely that we will be able to detect it from the mismatched checksums.

As we will show in Sec. 5.3.2, the time overhead of the proposed ABFT-based SDC detection method is very low. Let m, n, p be the input matrix dimensions. It only requires $O(mn+np)$ addition computation, which is much lower than the $O(mnp)$ time complexity of the matrix multiplication itself.

For large matrices, detection can be more challenging due to lower relative error in checksums. We recommend implementing ABFT at tile level and keeping tile size < 100. The checksum vectors can be shared across tiles to keep the time complexity identical to a global implementation.

5.2 Evaluation Setup

We evaluate our method on all SDC Suspects, using the same setups as application-level testing in Sec. 2.3 (two input value distributions and three matrix sizes). These diverse setups represent diverse production workload characteristics. For detection coverage experiments, we partition the result matrix into four tiles (arranged in a 2×2 grid) and verify the checksum of each tile individually. For performance overhead, we assess both tiled and global checksum schemes, observing nearly identical results.

While the evaluation was not integrated into production applications due to longer integration timelines, the core libraries which are used in production applications are tested

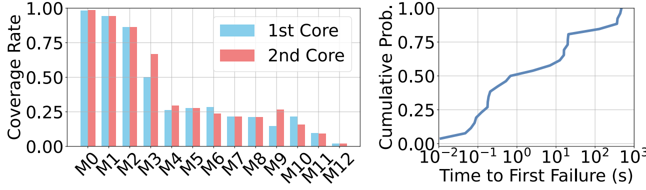


Figure 23. Left: SDC incident coverage rate on each machine. **Right:** Cumulative distribution of time to first failure.

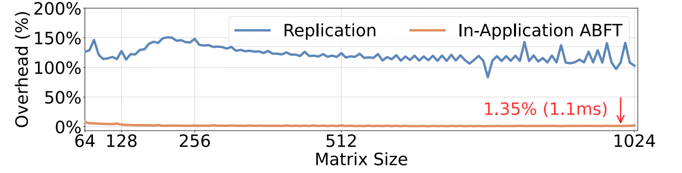


Figure 24. Time overhead comparison.

on real production machines, thereby showcasing effectiveness in production environments.

In order to detect when SDC occurs, we implement a replication-based verification method as our ground truth. This approach duplicates matmul operations using scalar computation (implemented via for-loops). Since scalar computation uses a different set of hardware units than vectorized computation, comparing the outputs provides a reliable mechanism for identifying when SDC occurs. Note that if an SDC occurs exclusively within the scalar verification path, our framework classifies this as a missed detection by the ABFT mechanism. Consequently, the SDC detection rate reported in this paper represents a conservative lower bound. This deviation is negligible, however, as prior work [8] demonstrates that scalar units are significantly less susceptible to SDCs than vector units. We define the machine coverage by dividing the number of machine cores that detected SDC with our ABFT method by the total number of machine cores that detected SDC.

5.3 Evaluation Results

Results shows that our in-application ABFT method achieves high SDC machine coverage with low time overheads.

5.3.1 Coverage Results. When using matrices with max dimension 10 and unrestricted floating-point values, our method achieves *complete machine coverage*. It also successfully identifies all 24 machine cores that exhibited SDC in Sec. 4. Fig. 23 (left) shows the SDC incident detection rates across machines. The average SDC incident detection rate, weighted by the number of SDC incidents per machine, is ~60%. While some SDC incidents escaped detection due to minimal relative errors in their results, this limitation does not impact the method's effectiveness. As shown in Fig. 19 (right), approximately 25% of SDC incidents produce relative errors larger than 1, enabling us to successfully identify all affected machine cores. Fig. 23 (right) shows the cumulative distribution of time to first SDC detected by our method. For over 80% of the machine cores, the first SDC is detected in 21 seconds.

Impact of matrix dimensions. Larger matrices reduce the relative error of SDC in checksums, challenging detection. Despite this, our method maintains strong coverage even with larger matrices. With a max dimension of 25, our method achieves a machine coverage of 94%, while a max dimension of 100 yields 88% coverage. For workloads involving very large matrices, we recommend applying tiling to reduce the effective matrix dimension, thereby maintaining high detection sensitivity (albeit with slightly higher overheads).

Impact of value ranges. When constraining input values to the range of 0-1, our method achieves coverage of 21 out of 23 machines (91%), while detecting ~99% SDC incidents across all detected machines. The high SDC incident coverage rate stems from the bounded input values, which result in larger relative errors when SDC occurs in the computation. This finding has significant practical implications, as most real-world applications operate with bounded inputs.

5.3.2 Time overhead of In-Application ABFT. We measure the execution times of $n \times n$ square matrix multiplication both with and without ABFT. Our evaluation ranges n from 10 to 1024 and reports the average time across 1000 rounds. For comparison, we implement a replication-based baseline in C++ that duplicates computation across two vector lanes for SDC detection. This approach leverages the observation that simultaneous failures across different vector lanes occur with low probability (Obs. 13). To ensure fair comparison, our ABFT method is also implemented in C++.

For matrix sizes 10, 25 and 100, which were used in coverage results above, the time overhead is 0.04μs (11%), 0.22μs (11%) and 3.04μs (3%), respectively. Fig. 24 shows the time overhead of our method and the baseline with n ranging from 64 to 1024. Result shows that the time overhead of our method is orders of magnitude lower in all cases. It is as low as 1.35% when n approaches 1024. The baseline's time overhead fluctuates possibly because of compiler optimizations that are more effective in certain ranges.

5.4 Deployment in Production

For direct production deployment, we suggest implementing our method at the math library level, making it application-agnostic: any application using matrix multiplication will automatically gain SDC detection capabilities. They can serve as SDC detection canaries while sharing cluster resources with non-matmul applications. The detection mechanism can be selectively enabled through library linking decisions. To achieve coverage equivalent to existing fleet testing frameworks, our approach is estimated to incur *only a few seconds* of time overhead daily. Our method complements existing

fleet testing infrastructure, allowing those frameworks to focus on other fault modes with their limited test time.

6 Related Work

SDC in the wild. SDC has emerged as a significant concern in production datacenters, with companies including Google [4, 24], Meta [11, 12, 38], and Alibaba [47, 48] documenting its impact. Prior research has addressed SDC through three main approaches: case studies of specific SDC incidents [12, 24], broad analyses of SDC's system and software-wise impacts [4, 34, 38, 47, 48], and deployment of specialized detection techniques for production environments [11]. Our work presents the first comprehensive analysis of vector instruction SDCs in hyper-scale datacenters.

SDC analysis through fault injection. Prior work has extensively employed fault injection techniques and use synthetic faults to study SDC. FIdelity [21] provides an efficient fault injection framework for deep learning accelerators. SpotSDC [33] develops a visualization system to facilitate fault injection analysis. He et al. [22, 23] perform fault injection to understand SDC's effect on deep neural network training workloads. There are also studies [8, 16, 37] that use fault injection to study SDC in CPU architectures. Our analysis of real-world cluster data validates fault-injection research and also reveals new undocumented patterns.

SDC detection. Prior research explores different SDC detection techniques. Harpocrates [27] develops a framework that automatically generates concise functional test programs for SDC detection. F_Radish [52] optimizes assertion-based testing by eliminating redundant checks. Dixit et al. [11] focus on using specialized tests to detect SDC in production. Our in-application ABFT method turns datacenter workloads into SDC detection canaries, enabling low-overhead SDC detection.

ABFT. Algorithm-Based Fault Tolerance (ABFT) [25] has long been proposed for protecting scientific computing applications where accuracy is key [5, 9, 13, 49]. It has also been proposed to protect neural network workloads [19, 29, 36], FFT workloads [50] and other applications [32]. To the best of our knowledge, our paper is the first to propose integrating ABFT directly into datacenter application workflows for detecting SDC-inducing cores and the first to validate this method on hardware with real SDC defects, moving beyond fault-injection-based evaluations in prior work.

7 Conclusion

In this paper, we present the first comprehensive instruction- and application-level analysis of vector instruction SDCs in hyper-scale datacenters using a two-stage approach with our SEVI framework. Our observations provide insights into possible SDC causes and inspire new mitigation strategies. We also provide a low-overhead SDC detection method via in-application ABFT and validate it on real SDC devices.

Acknowledgment

We thank the anonymous reviewers and our shepherd, Professor James Tuck, for their valuable feedback. We thank CQ Tang, Manish Modi, Vijay Rao and other infrastructure engineers at Meta for their support. This work was supported in part by a Sloan Foundation Faculty Fellowship and funding from the Open Compute Project.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[2] Dashti Ali, Aras Asaad, Maria-Jose Jimenez, Vidit Nanda, Eduardo Paluzo-Hidalgo, and Manuel Soriano-Trigueros. 2023. A Survey of Vectorization Methods in Topological Data Analysis . *IEEE Transactions on Pattern Analysis & Machine Intelligence* 45, 12 (Dec. 2023), 14069–14080. doi:10.1109/TPAMI.2023.3308391

[3] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' guide (third ed.).* Society for Industrial and Applied Mathematics, USA.

[4] Rich Bonderson. 2021. Training in turmoil: Silent data corruption in systems at scale. In *International Test Conference Silicon Lifecycle Management Workshop*. IEEE, USA, 1–36.

[5] George Bosilca, Rémi Delmas, Jack Dongarra, and Julien Langou. 2009. Algorithm-based fault tolerance applied to high performance computing. *J. Parallel and Distrib. Comput.* 69, 4 (2009), 410–416.

[6] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. JAX: composable transformations of Python+NumPy programs. http://github.com/jax-ml/jax

[7] Claus Braun, Sebastian Halder, and Hans Joachim Wunderlich. 2014. A-ABFT: Autonomous algorithm-based fault tolerance for matrix multiplications on graphics processing units. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, IEEE, USA, 443–454.

[8] Odysseas Chatzopoulos, Nikos Karystinos, George Papadimitriou, Dimitris Gizopoulos, Harish Dattatraya Dixit, and Sriram Sankar. 2025. Veritas – Demystifying Silent Data Corruptions: {mu}Arch-Level Modeling and Fleet Data of Modern x86 CPUs. In *2025 IEEE International Symposium on High-Performance Computer Architecture*. IEEE, USA, 1–14.

[9] Zizhong Chen and Jack Dongarra. 2008. Algorithm-Based Fault Tolerance for Fail-Stop Failures. *IEEE Transactions on Parallel and Distributed Systems* 19, 12 (2008), 1628–1641. doi:10.1109/TPDS.2008.58

[10] Eric Cheng, Hyungmin Cho, Shahrzad Mirkhani, Lukasz Szafaryn, Jacob Abraham, Pradip Bose, Chen-Yong Cher, Klas Lilja, Kevin Skadron, Mircea Stan, and Subhasish Mitra. 2025. CLEAR Cross-Layer Resilience: A Retrospective. *IEEE Design & Test* 42, 3 (2025), 74–85. doi:10.1109/MDAT.2024.3483028

[11] Harish Dattatraya Dixit, Laura Boyle, Gautham Vunnam, Sneha Pendharkar, Matt Beadon, and Sriram Sankar. 2022. Detecting silent data

corruptions in the wild. arXiv:2203.08989 [cs.AR] https://arxiv.org/abs/2203.08989

[12] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. 2021. Silent Data Corruptions at Scale. arXiv:2102.11245 [cs.AR] https://arxiv.org/abs/2102.11245

[13] Jack Dongarra, Thomas Herault, and Yves Robert. 2015. Fault tolerance techniques for high-performance computing. In *Fault-tolerance techniques for high-performance computing*. Springer, USA, 3–85.

[14] Nir Drucker, Shay Gueron, and Vlad Krasnov. 2019. Making AES great again: the forthcoming vectorized AES instruction. In *16th International Conference on Information Technology-New Generations (ITNG 2019)*. Springer, USA, 37–41.

[15] ggml-org. 2023. llama.cpp. https://github.com/ggml-org/llama.cpp GitHub repository.

[16] Dimitris Gizopoulos, George Papadimitriou, Odysseas Chatzopoulos, Nikos Karystinos, Harish D Dixit, and Sriram Sankar. 2024. Silent data corruptions in computing systems: Early predictions and large-scale measurements. In *2024 IEEE European Test Symposium (ETS)*. IEEE, USA, 1–10.

[17] Martin Goll and Shay Gueron. 2014. Vectorization on ChaCha stream cipher. In *2014 11th International Conference on Information Technology: New Generations*. IEEE, USA, 612–615.

[18] Dirk Habich, Patrick Damme, Annett Ungethüm, and Wolfgang Lehner. 2018. Make Larger Vector Register Sizes New Challenges? Lessons Learned from the Area of Vectorized Lightweight Compression Algorithms. In *Proceedings of the Workshop on Testing Database Systems* (Houston, TX, USA) *(DBTest '18)*. Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages. doi:10.1145/3209950.3209957

[19] Siva Kumar Sastry Hari, Michael B Sullivan, Timothy Tsai, and Stephen W Keckler. 2021. Making convolutions resilient via algorithm-based error detection techniques. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2546–2558.

[20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. doi:10.1038/s41586-020-2649-2

[21] Yi He, Prasanna Balaprakash, and Yanjing Li. 2020. Fidelity: Efficient resilience analysis framework for deep learning accelerators. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, USA, 270–281.

[22] Yi He, Mike Hutton, Steven Chan, Robert De Gruijl, Rama Govindaraju, Nishant Patil, and Yanjing Li. 2023. Understanding and mitigating hardware failures in deep learning training systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. IEEE, USA, 1–16.

[23] Yi He and Yanjing Li. 2023. Understanding Permanent Hardware Failures in Deep Learning Training Accelerator Systems. In *2023 IEEE European Test Symposium (ETS)*. IEEE, USA, 1–6.

[24] Peter H Hochschild, Paul Turner, Jeffrey C Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E Culler, and Amin Vahdat. 2021. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. ACM, New York, NY, USA, 9–16.

[25] Kuang-Hua Huang and Jacob A Abraham. 1984. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers* 100, 6 (1984), 518–528.

[26] Russ Joseph, David Brooks, and Margaret Martonosi. 2003. Control techniques to eliminate voltage emergencies in high performance processors. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings*. IEEE, IEEE, USA, 79–90.

[27] Nikos Karystinos, Odysseas Chatzopoulos, George-Marios Fragkoulis, George Papadimitriou, Dimitris Gizopoulos, and Sudhanva Gurumurthi. 2024. Harpocrates: Breaking the silence of cpu faults through hardware-in-the-loop program generation. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, USA, 516–531.

[28] Kasem Khalil, Omar Eldash, Ashok Kumar, and Magdy Bayoumi. 2020. Machine learning-based approach for hardware faults prediction. *IEEE Transactions on Circuits and Systems I: Regular Papers* 67, 11 (2020), 3880–3892.

[29] Jack Kosaian and KV Rashmi. 2021. Arithmetic-intensity-guided fault tolerance for neural network inference on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, USA, 1–15.

[30] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 311–326. doi:10.1145/2882903.2882925

[31] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29.

[32] Sihuan Li, Hongbo Li, Xin Liang, Jieyang Chen, Elisabeth Giem, Kaiming Ouyang, Kai Zhao, Sheng Di, Franck Cappello, and Zizhong Chen. 2019. FT-iSort: efficient fault tolerance for introsort. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 71, 17 pages. doi:10.1145/3295500.3356195

[33] Zhimin Li, Harshitha Menon, Dan Maljovec, Yarden Livnat, Shusen Liu, Kathryn Mohror, Peer-Timo Bremer, and Valerio Pascucci. 2020. Spotsdc: Revealing the silent data corruption propagation in high-performance computing systems. *IEEE Transactions on Visualization and Computer Graphics* 27, 10 (2020), 3938–3952.

[34] Jeffrey Ma, Hengzhi Pei, Leonard Lausen, and George Karypis. 2025. Understanding Silent Data Corruption in LLM Training. arXiv:2502.12340 [cs.LG] https://arxiv.org/abs/2502.12340

[35] Open Compute Project. 2024. Computing's Hidden Menace: The OCP Takes Action Against Silent Data Corruption (SDC). https://www.opencompute.org/blog/computings-hidden-menace-the-ocp-takes-action-against-silent-data-corruption-sdc.

[36] Elbruz Ozen and Alex Orailoglu. 2019. Sanity-check: Boosting the reliability of safety-critical deep neural network applications. In *2019 IEEE 28th Asian Test Symposium (ATS)*. IEEE, USA, 7–75.

[37] George Papadimitriou and Dimitris Gizopoulos. 2023. Silent data corruptions: Microarchitectural perspectives. *IEEE Trans. Comput.* 72, 11 (2023), 3072–3085.

[38] George Papadimitriou, Dimitris Gizopoulos, Harish Dattatraya Dixit, and Sriram Sankar. 2023. Silent data corruptions: The stealthy saboteurs of digital integrity. In *2023 IEEE 29th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, USA, 1–7.

[39] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA, 1–12.

[40] Gang Ren, Peng Wu, and David Padua. 2005. An empirical study on the vectorization of multimedia applications for multimedia extensions. In

19th IEEE International Parallel and Distributed Processing Symposium. IEEE, USA, 10–pp.

[41] J Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. 2007. Collaborative filtering recommender systems. In *The adaptive web: methods and strategies of web personalization*. Springer, USA, 291–324.

[42] BM Shabanov, AA Rybakov, and SS Shumilin. 2019. Vectorization of high-performance scientific calculations using AVX-512 intruction set. *Lobachevskii Journal of Mathematics* 40 (2019), 580–598.

[43] Narasimhan Sreraman and Ramaswamy Govindarajan. 2000. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming* 28 (2000), 363–400.

[44] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: high performance graph analytics made productive. *Proc. VLDB Endow.* 8, 11 (July 2015), 1214–1225. doi:10.14778/2809974.2809983

[45] Barna Szabó and Ivo Babuška. 2021. *Finite element analysis: Method, verification and validation*. John Wiley & Sons, USA.

[46] Jiyuan Tu, Guan Heng Yeoh, Chaoqun Liu, and Yao Tao. 2023. *Computational fluid dynamics: a practical approach*. Elsevier, The Netherlands.

[47] Shaobu Wang, Guangyan Zhang, Junyu Wei, Yang Wang, Jiesheng Wu, and Qingchao Luo. 2023. Understanding Silent Data Corruptions in a Large Production CPU Population. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 216–230. doi:10.1145/3600006.3613149

[48] Shaobu Wang, Guangyan Zhang, Junyu Wei, Yang Wang, Jiesheng Wu, and Qingchao Luo. 2024. Understanding Silent Data Corruption in Processors for Mitigating its Effects. *ACM Trans. Archit. Code Optim.* 21, 4, Article 84 (Nov. 2024), 27 pages. doi:10.1145/3690825

[49] Panruo Wu, Qiang Guan, Nathan DeBardeleben, Sean Blanchard, Dingwen Tao, Xin Liang, Jieyang Chen, and Zizhong Chen. 2016. Towards Practical Algorithm Based Fault Tolerance in Dense Linear Algebra. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing* (Kyoto, Japan) *(HPDC '16)*. Association for Computing Machinery, New York, NY, USA, 31–42. doi:10.1145/2907294.2907315

[50] Shixun Wu, Yujia Zhai, Jinyang Liu, Jiajun Huang, Zizhe Jian, Huangliang Dai, Sheng Di, Franck Cappello, and Zizhong Chen. 2025. TurboFFT: Co-Designed High-Performance and Fault-Tolerant Fast Fourier Transform on GPUs. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Las Vegas, NV, USA) *(PPoPP '25)*. Association for Computing Machinery, New York, NY, USA, 70–84. doi:10.1145/3710848.3710853

[51] Chang Xia, Haijun Wang, Anqi Zhang, and Wenting Zhang. 2018. A high-performance cellular automata model for urban simulation based on vectorization and parallel computing technology. *International Journal of Geographical Information Science* 32, 2 (2018), 399–424.

[52] Na Yang and Yun Wang. 2020. F_Radish: Enhancing Silent Data Corruption Detection for Aerospace-Based Computing. *Electronics* 10, 1 (2020), 61.

[53] Jia Zhao, Basab Datta, Wayne Burleson, and Russell Tessier. 2010. Thermal-aware voltage droop compensation for multi-core architectures. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI*. IEEE, USA, 335–340.

[54] Yicong Zhou, Weijia Cao, Licheng Liu, Sos Agaian, and CL Philip Chen. 2015. Fast Fourier transform using matrix decomposition. *Information Sciences* 291 (2015), 172–183.