Understanding Silent Data Corruptions in a Large Production CPU Population

Shaobu Wang, Tsinghua University
Guangyan Zhang[*], Tsinghua University
Junyu Wei, Tsinghua University
Yang Wang, The Ohio State University
Jiesheng Wu, Alibaba Cloud
Qingchao Luo, Alibaba Cloud

Abstract

Silent Data Corruption (SDC) in processors can lead to various application-level issues, such as incorrect calculations and even data loss. Since traditional techniques are not effective in detecting processor SDCs, it is very hard to address problems caused by SDCs. For the same reason, knowledge about SDCs in the wild is limited.

In this paper, we conduct an extensive study on SDCs in a large production CPU population, encompassing over one million processors. In addition to collecting overall statistics, we perform a detailed study to understand 1) whether certain processor features are particularly vulnerable and their potential impacts on applications; 2) the reproducibility of SDCs and the triggering conditions (e.g., temperature) of those less reproducible SDCs; and 3) the challenges and opportunities to mitigate SDCs.

Inspired by the above observations, we design an efficient SDC mitigation approach called Farron, which relies on prioritized testing to detect highly reproducible SDCs and temperature control to mitigate less reproducible SDCs. Our experimental results indicate that Farron can achieve lower overall overhead with better coverage of SDCs, compared to the baseline used in Alibaba Cloud.

CCS Concepts: • **Computer systems organization → Reliability**; *Maintainability and maintenance*; *Processors and memory architectures.*

Keywords: processor, silent data corruption, reliability, fault tolerance

[*]Corresponding Author: Guangyan Zhang (gyzh@tsinghua.edu.cn).
Shaobu Wang, Guangyan Zhang, and Junyu Wei are with the Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China.

ACM Reference Format:
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 *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3600006.3613149

1 Introduction

In recent years, CPU technology has achieved rapid development with higher clock frequency and more cores attached in one processor. A classical assumption is that processors work as designed and produce reliable computation results [38]. However, processor faults do occur in production environments [1, 46–48, 56]. Both the growing complexity of modern CPUs and the increasing scale of cloud infrastructures have increased the risk of processor faults.

These processor faults can lead to application-level errors, which fall into two categories. One class of errors causes crashes or exceptions promptly. The other class of errors can introduce undesired data (e.g., incorrect calculation results or even data loss) without being detected immediately. We call the second class of errors caused by processor faults as *"silent data corruptions"*, acronymized as *SDCs*.

CPU SDCs occur at a low but non-negligible frequency in production. For example, a few processors in Alibaba Cloud occasionally gave wrong checksum calculation results. Such incorrect information misled the cloud application to conclude that request data was corrupted and thus triggered repeated requests frequently, which impaired the overall performance. SDCs also occur in Google Cloud [48]: a small subset of their processors gave wrong results when executing some rarely-used instructions. These errors made a large-scale data-analysis application give wrong answers. Meta also notices SDCs [46]: one machine occasionally misjudged the file size to be zero due to wrong calculation, and caused a database to lose files.

Data corruptions in storage and memory systems are well-known to be dangerous and hard to detect and diagnose. CPU SDCs are even more notorious, because the basic technique to detect data corruptions in storage and memory systems can hardly be applied to CPU instructions (e.g., how to know that a computational instruction gives a wrong result?). As a result, Meta shows SDCs can require months of engineering time to debug [46], and Alibaba Cloud once took several weeks to debug a SDC.

Researchers have conducted studies on SDCs [4, 11, 15, 16, 33, 46–48, 50, 54], which fall into three categories: 1) theoretically stating the existence of SDCs but not studying concrete errors [8, 11, 16]; 2) studying the impacts and tolerance techniques of SDCs, through artificial fault injection instead of naturally produced SDCs [10, 11, 23, 32, 36, 57]; 3) providing brief data about CPU SDC cases in real world but lacking detailed measurement and analysis about failure rate, software symptoms, occurrence patterns, etc [46–48, 50–52]. To this end, *the current knowledge about SDCs in the wild is limited*.

In this paper, we *investigate SDCs in a large production CPU population*, encompassing over one million CPUs from hundreds of clusters in 28 data centers across 14 countries. To the best of our knowledge, this is the first work to quantitatively assess and systematically analyze CPU SDC phenomena in a large-scale production environment. The main contributions of this paper can be summarized as follows:

- *Assessing SDCs in a large-scale production environment*: We have conducted SDC testing on over one million processors over 32 months. In addition to providing the overall statistics of failure rate, we further analyze how micro-architecture, the timing of testing, etc., affect the failure rate and whether these failures affect a single core or all cores within the processor.
- *Investigating software impacts of SDCs*: Through this study, we identify vulnerable features in the processors (e.g., cache consistency, floating-point computation, and vector computation), and find that, not surprisingly, workloads that extensively engage such vulnerable features are likely to be affected. Moreover, we reveal the deficiency of existing failure models. For example, we find for floating point calculations, SDCs are more likely to cause bitflips in the fraction part, which only cause a small loss of accuracy due to floating point encoding [17] and thus render existing accuracy-based detection techniques less effective [33].
- *Analyzing reproducibility of SDCs*: We find that, on the one hand, some SDCs are highly reproducible, which means, without proper mitigation, they will manifest frequently in production: this is confirmed by Alibaba Cloud's investigation of production errors caused by SDCs. On the other hand, some SDCs can only be triggered under specific conditions (e.g., temperature, workload stress). Such observations motivate our exploration of new SDC mitigation strategies, as discussed next.
- *Assessing current mitigation practices for SDCs*: We identify a substantial design space for improving existing SDC testing by prioritizing testcases and developing testcases focused on multi-threading scenarios. Moreover, we discuss the challenges that SDCs present to fault-tolerance techniques.
- *Proposing a concrete approach for SDC mitigation.* Inspired by the above observations, we propose Farron, an efficient SDC mitigation approach. It relies on regular testing to identify those highly reproducible SDCs, and improves the efficiency of this approach with testcase prioritization; it controls the temperature of processors to minimize the occurrences of those less reproducible SDCs; it can further make a trade-off between these two approaches by assigning longer testing time if a processor has to work under a high temperature for long. Our evaluation shows that Farron can protect applications from CPU SDCs, with both higher SDC detection rate and lower overhead compared to the baseline approach.

2 Motivation and Methodology

2.1 Target System

We study SDCs in Alibaba Cloud, which involves hundreds of clusters deployed in 28 data centers worldwide. Alibaba Cloud has provided a stable working environment for hardware, with a strong focus on cooling, power distribution, cable management and environmental monitoring, and environment variations are controlled to be minimal.

Our study includes over one million processors deployed since 2017. These processors are supplied by a well-known international chip manufacturer. These processors cover a wide range of micro-architectures in recent years, apply the advanced lithographic technology, and widely use the multi-core technology. We believe our processors are able to represent the international mainstream.

2.2 SDC Examples in Production

Over time, Alibaba Cloud has occasionally observed servers with a higher error rate compared to others. After extensive debugging to identify the root cause, we find the problems are due to processor defects. Here we present some examples.

In one case, a storage application frequently reported checksum mismatch of the user data. After weeks of debugging, we found that one processor in the fleet was faulty and a *checksum-calculation related instruction* on the processor gave wrong result intermittently.

In the second case, we also observed checksum mismatches, but our debugging revealed a different cause: A client thread packed data and its checksum into a buffer, which was then shared with a daemon thread. Due to *defective cache coherence*, the daemon thread sometimes got inconsistent data, incurring checksum mismatches.

In another case, a program sometimes triggered assertion failures. We later found this is because the application used a hash map to manage its metadata, and *defective hashing calculation* in a faulty processor affected its metadata service.

These cases, particularly the concerns that some errors may not be detected by checksums, and similar reports from industry [46, 48] have motivated us to conduct this study.

2.3 Toolchain

We use a toolchain provided by the chip manufacturer, which is designed to detect SDCs related to cloud workloads. The toolchain includes 633 testcases and a framework. The framework drives these testcases and checks for the occurrence of SDCs. According to a user's specification, the framework selects the testcases to be performed and controls their execution order, resource allocation (such as CPU time and concurrency) during testing, etc.

Testcases are programs that simulate cloud workloads, carefully crafted with consideration of both software behaviors and hardware features. Most testcases focus on individual processor features, such as floating point calculation, branch prediction, cache, interconnect between cores, etc. The complexity of these testcases vary significantly: 1) Some execute a specific instruction within a loop. 2) Some call functions in libraries. 3) Some invoke application logics.

Moreover, we also try other toolchains designed for SDC detection like OpenDCDiag [60] as supplementary and reach the same observations in our study.

In our study, this toolchain serves two roles. First, it provides an authoritative method to test processor functions. Second, it acts as an impacted workload simulator when conducting in-depth study on faulty processors. To facilitate testing and analysis, we have designed additional tools, which will be discussed in the corresponding sections.

Despite the toolchain provided by the manufacturer, it is often not easy to determine whether a failed test is due to a CPU SDC or other reasons (e.g., memory error), especially considering some failures are not reproducible. Therefore, if we cannot determine the root cause with a reasonable amount of effort, we will send the suspected processor back to the manufacturer. All the faulty processors reported in this paper have been confirmed by the manufacturer.

On the other hand, like any testing techniques, the toolchain may not be complete to cover all SDCs. We did find SDCs that cannot be detected by this toolchain, after extensive testing and debugging. Therefore, this work should be considered a best-effort approach to detect and understand SDCs: both false negatives and false positives are possible.

2.4 Study Process and Approaches

We carry out large-scale tests in order to find faulty processors in Alibaba Cloud, both before production and during production. As shown in Figure 1, pre-production testing is carried out 1) after factory delivery (after manufactured chip is shipped to Alibaba Cloud), 2) after datacenter delivery, and 3) after system re-installation (before a machine goes into production, it needs to install a new system for its service). Then in production, machines will be regularly tested in groups. Testing for each group lasts about 2 weeks, and testing for the whole fleet needs months.

In these large-scale tests, we execute all the testcases in the toolchain sequentially, and each testcase is allocated

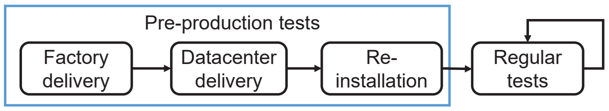


Figure 1. Test timing in our fleet.

Factory	Datacenter	Re-install	Regular	**Total**
0.776‱	0.18‱	2.306‱	0.348‱	3.61‱

Table 1. Failure rate of different test timings.

with equal test duration specified by the administrator. We started such tests since January 2021, and so far, we have found hundreds of faulty processors.

Among these faulty processors, we have conducted extensive experiments on 27 of them for a more detailed analysis (the others have been returned to the manufacturer). To be concrete, we have run tens of millions of tests and collected more than ten thousand SDC records from these tests to understand their potential impacts on applications and their reproducibility.

3 SDCs in the Wild

3.1 Brief Overview of Test Results

Observation 1. *In overall, 3.61‱ of the CPUs are identified to cause SDCs in our study.*

Our results are consistent with but are more precise than those reported by Google ("the order of a few mercurial cores per several thousand machines" [48]) and Meta ("hundreds of CPUs detected for SDCs in hundreds of thousands of machines" [46]). Google's and Meta's decisions to not disclose exact numbers may be for business considerations. This observation substantiates the notion that SDCs represent a pervasive issue rather than a "black swan" event, especially in the cluster with a significant number of processors.

Observation 2. *The failure rates observed during the pre-production testing period and the regular testing period amount to 3.262‱ and 0.348‱, respectively.*

As shown in Table 1, 3.262‱ of CPUs are detected to cause SDCs in pre-production tests, accounting for a significant proportion (90.36%) of all faulty processors we have identified. This means pre-production testing is indispensable since it prevents many faulty processors from entering our production environment. 0.348‱ of CPUs are detected to cause SDCs in regular testing. These faulty processors have passed pre-production tests and some have even passed several rounds of regular tests.

However, despite all SDC tests, we still encounter SDC issues that affect Alibaba Cloud services as discussed in Section 2.2. This can be attributed to the window between regular SDC tests and the non-determinism of reproducing SDCs. Addressing this issue is challenging, as it is not feasible to perform regular SDC tests frequently. As a result, services

Arch	M1	M2	M3	M4	M5	M6	M7	M8	M9	**avg**
Failure rate	4.619‱	0.352‱	2.649‱	0.082‱	0.759‱	3.251‱	1.599‱	9.29‱	4.646‱	3.61‱

Table 2. Failure rate of different micro-architectures.

requiring high reliability may need to take SDC tolerance into consideration.

Observation 3. *SDCs have been identified across all micro-architectures present within Alibaba Cloud fleet. The failure rate does not decrease with newer chips.*

We have found faulty processors in every micro-architecture we have, indicating SDC is a general problem for modern processors. We have tested hundreds of thousands of samples for each of these micro-architectures. As shown in Table 2, the failure rates of different micro-architectures range from 0.082‱ to 9.29‱.

In our tests, the failure rate does not decrease with newer chips. This phenomenon can be attributed to multiple factors. The testing ability may increase with the processor development, but the difficulty of testing also increases, as features and circuits become more complex with the processor development. In fact, due to the complex micro-architecture diagram, comprehensive testing for the chip has prohibitive costs and makes faulty processors escaping from high-volume manufacturing testing a fact of life [58]. Moreover, different micro-architectures have different degrees of maturity, which also affects their failure rates.

Despite extensive testing and the advancements in chip development, online services continue to be exposed to potential risks stemming from SDCs. This highlights the essential requirement of SDC-tolerant systems for cloud vendors to enhance the reliability of their services.

3.2 Zooming in on Faulty Processors

Table 3 shows the hardware details and error information of a subset of our faulty processors as examples. We make the following observation by studying these faulty processors:

Observation 4. *A single processor fault may exert its influence on an individual physical core or encompass all cores within the processor.*

In about half of the faulty processors, there exists only one defective physical core. This is probably because in these faulty processors, the defects occur in the components that can only be used by a single physical core, like arithmetic units. Note that multiple hardware threads, also known as logical cores, can share a single physical core. In most cases, all the logical cores sharing the same defective physical core are affected and they fail the same testcases with a similar frequency.

In the other half of the faulty processors, defects impact all physical cores. Some are probably due to the fact that the defects occur in the components shared by all cores, like CPU cache. However, we do observe cases that a defect impacts the same non-shared component of every core (e.g., MIX1 and MIX2 in Table 3). These cores fail the same testcases but at a different frequency. The difference can be up to several orders of magnitude under the same test setting, making some of the defective cores difficult to be detected. We presume this phenomenon may come from defects in chip design and manufacturing. The proportion of processors with multiple defective cores in our study (i.e., about half) is significantly higher than what has been reported by Google [50], where a single processor with multiple defective cores is considered a low-probability event. We hypothesize that such difference is primarily due to the fact that we use a different toolchain, and our toolchain appears to have better detection capabilities for coherency problems among cores.

Large companies decommission the whole faulty processor or isolate the whole machine no matter which of its cores are identified as faulty [48, 52]. This practice is reasonable given the relatively low failure rate. However, it could be worthwhile to investigate the feasibility of continuing to utilize the unaffected cores within a faulty processor [56].

4 Software Symptoms of SDCs

4.1 Impacted Workloads

Observation 5. *SDCs exhibit a substantial prevalence in particular workloads, exposing five vulnerable features, namely arithmetic logic computation, vector operations, floating point calculation, cache coherency and transactional memory.*

This observation can be explained by two contradictory theories: On the one hand, it is possible that, compared to other features, these features are indeed more vulnerable due to their complexity (e.g., cache is known to occupy a big portion of the chip area [57]). On the other hand, it is also possible that other features are equally or even more vulnerable, but since operating systems and applications make use of other features heavily, a fault in other features will cause a crash instead of a SDC [57]. Either way, this observation suggests that a developer can focus on a limited set of features when considering SDC related issues.

Figure 2 shows the proportion of faulty processors per feature. Note that the sum of these proportions is bigger than 1. This is because a defect can occur on shared or integrated components of multiple features and thus some processors can encounter errors among multiple features. For example, we find Processor MIX1 has wrong execution results in both vector operations and complicated floating-point calculation, and we blame this problem on the combination of FPU functionalities with vector units. Another example is CNST1, which fails to guarantee the consistency in both cache and transactional memory.

CPU id	arch	age(Y)	#pcore	#err	SDC type	impacted workloads	impacted datatypes
MIX1	M2	1.75	16	25	computation	matrix calculation; checksum calculation; string manipulation; large integer arithmetic;	int32; unint32; float32; float64; byte; bin16; bin32;
MIX2	M2	0.92	16	24	computation	matrix calculation; checksum calculation; bit operations; hashing;	int16; int32; unint32; float32; float64; bit; byte; bin16; bin32;
SIMD1	M2	2.33	1	5	computation	matrix calculation;	float32
SIMD2	M5	0.50	1	1	computation	matrix calculation;	float64;
FPU1	M5	0.58	1	3	computation	floating-point computing; mathematical function;	float64; float64x;
FPU2	M5	1.83	1	3	computation	floating-point computing; mathematical function;	float64; float64x;
FPU3	M3	3.08	1	2	computation	floating-point computing;	float64;
FPU4	M6	1.62	1	1	computation	floating-point computing;	float64;
CNST1	M2	0.92	1	9	consistency	multi-thread lock; transactional memory	–
CNST2	M3	1.08	24	8	consistency	transactional memory;	–

Table 3. Hardware details and error information of a subset of our faulty processors. #pcore denotes the number of defective physical cores on the faulty processor; #err is the number of failed testcases on the faulty processor.

We further categorize SDCs with defective features into two types: computation and consistency. SDCs with computation type are due to defective arithmetic operations, including arithmetic logic computation, vector operations and floating point calculation. SDCs with consistency type are due to defective features related to consistency guarantee, such as cache coherency and transactional memory. We distinguish these two types for two reasons: First, they require different testing strategies since SDCs with consistency type can only be detected with multi-threaded tests. Second, we observe that, if one processor has multiple defective features, they always belong to one type. Among the 27 faulty processors we have tested extensively, 19 processors have computation defects and the remaining ones have consistency defects.

Since each testcase is designed to mimic a real-world workload, we can further speculate potential impacts on real-world workloads, as sampled in Table 3. For example, Processor FPU1 produces incorrect results on a specific floating-point calculation operation, which is used by a library widely used in HPC applications. The wide impacts of certain SDCs are due to the wide usage of these defective features.

We have tried to further pinpoint which instructions are problematic, which turns out to be a challenging task. For some of these errors, the toolchain preserves the context and points out the incorrect instructions. For example, in SIMD1, the toolchain reports that a vector instruction that performs multiplication and addition operations simultaneously gives wrong results. The others, however, need manual investigation, but we meet the classic problem that, since these errors are often hard to reproduce, it is unclear where to modify the testcases to print more information. Therefore, we turn to a statistical approach: we instrument the toolchain to catch the number of times each type of instruction is executed during each testcase via Pin [9]. This method helps us narrow down the scope of suspected instructions. Take cases in Table 3 as examples: we find one instruction, which uses the floating-point calculation feature to calculate a complex math function (arctangent), is a suspect in FPU1 and FPU2, because all the testcases using this instruction could reproduce SDCs and all the other testcases can pass. In another example, we find instructions responsible for managing the transactional region a suspect in CNST2.

However, not all errors have obvious suspected instructions. The SDCs in CNST1 causes cache coherence issues and we fail to locate the suspected instructions. This is reasonable since cache coherence mechanisms are mostly hidden from a program so a program often does not invoke a specific instruction for cache coherence.

It should be noted that not all testcases executing a defective instruction will generate errors. For example, in MIX1, we find a defective instruction is used in seven testcases, but only two of them generate errors. We study the triggering conditions in details in Observation 10.

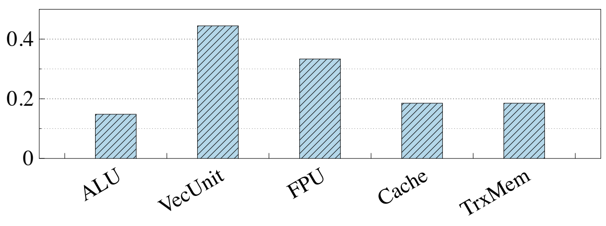


Figure 2. The proportion of processors with a faulty feature.

4.2 Error Breakdown in Mis-calculated Data

We further study the properties of computation SDCs to understand the influence of defective features on the workload results. We exclude consistency SDCs from this investigation since they don't have a deterministic pattern.

Observation 6. *SDCs have been confirmed to affect operations on all tested data types, including integers, single- and double-precision floating-point numbers, bytes, and more. Notably, operations related to floating-point numbers demonstrate heightened vulnerability to SDCs.*

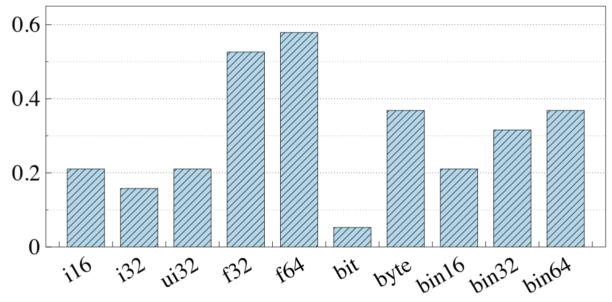


Figure 3. Proportion of faulty processors with a certain affected operation datatype.

Figure 3 shows the proportion of faulty processors involving each operation datatype. We find all datatypes under tests are impacted by SDCs, and floating-point datatypes involve more faulty processors than other datatypes. We find two reasons attribute to this issue: Many different vulnerable features are related to floating-point calculation, including vector operations with floating-point datatypes and specific floating-point calculation. Some floating-point operations, such as trigonometric functions, are complex, which increases the difficulty on the design and test of relevant processor features.

Observation 7. *For floating-point numbers, bitflips predominantly occur in the fraction part, resulting in minor precision losses.*

As for computation SDCs, we investigate which bits are different between the expected result and the actual result, which is also known as bitflip phenomenon. Figure 4(a)-4(d) shows the bitflips of different numerical data types. We find that it is rare that bitflips occur in the most significant bits. Note this bitflip pattern does not apply to non-numerical data, in which all the positions have comparable amount of bitflips (Figure 5).

Furthermore, we find nearly half (51.08%) of bitflips are changed from zero to one, which means there is no tendency of bitflip direction in general. However, a tendency exists in some corner cases. For example, as for 16-bit integer data statistics in MIX1, 72.27% of bitflips are from zero to one.

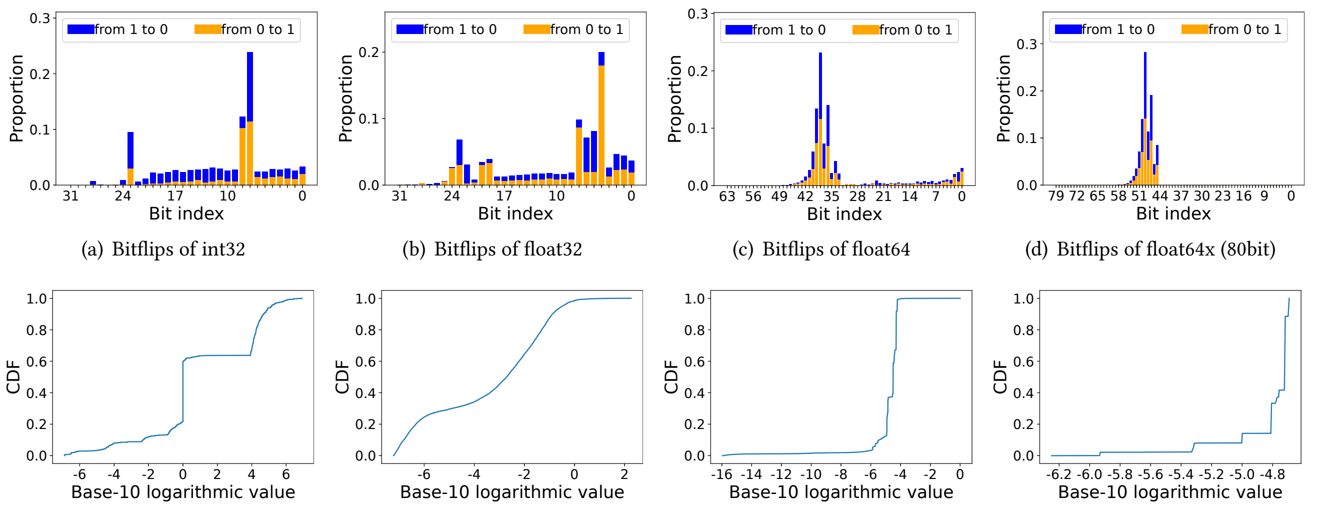
The impact of an SDC on a numerical data depends on the type of the data. In floating point encoding standard (i.e., IEEE-754 [17]), the bits are divided into three parts: sign, exponent, and fraction. A bitflip usually hits the fraction part, probably due to two reasons. First, as for floating-point numbers, the computation logic of the fraction part is more complex than that of the exponent part, making the fraction part more vulnerable. Second, we observe a concentration of many bitflips in the middle of the data and a gradual decrease towards the ends, which follow related research about failure distribution on registers [57]. Given the relatively long fraction part in the data, most of the bitflips tend to occur in this part. Because IEEE-754 assumes an implicit leading 1 in the fraction, the relative precision loss caused by one bitflip in fraction only depends on the position of the bit but does not depend on the value of the number. Other datatypes do not have this property. For example, for an integer, if its value is small, then a bitflip in a less significant bit can still cause a significant precision loss.

We show the relative precision losses between expected data and actual data in Figure 4(e)- 4(h). Since the bitflips we observed mostly occur in the fraction bits, the precision losses of floating-point datatypes are small. For example, all of the precision losses on extended double precision (80bit) floating-point numbers are less than 0.002%. 99.9% of the precision losses on double precision (64bit) floating-point numbers are less than 0.02%. 80.25% of the precision losses on single precision (32bit) floating-point numbers are less than 5%. On the other side, 40.2% of the precision losses on 32-bit integer data are bigger than 100%.

Observation 8. *As for a specific failed setting (i.e., a combination of a testcase and a processor), bitflips tend to manifest at fixed position(s) within the number representations.*

We define a bitflip pattern as a set of positions where bitflips occur with whatever inputs in a given setting, i.e., a combination of a testcase and a processor. To find bitflip patterns, we use a mask, i.e., the exclusive-or value of the expected result and the actual result, to represent the bitflip positions. If more than 5% of the SDC records of a setting have the same mask, we regard this mask as a bitflip pattern. A setting could have multiple bitflip patterns in our observations. We suspect it is because the multiple instructions in the testcase are impacted by the defect and these instructions fail to stably reproduce errors (i.e., in one run of a testcase, some of them fail but others succeed), which causes different combinations of error instructions to generate different bitflip patterns. Figure 6 shows the proportion of SDC records with bitflip patterns in some settings. One potential explanation for bitflip patterns is that the hardware defect of specific faulty processor causes deterministic influence and thus these bitflips tend to occur at fixed position(s).

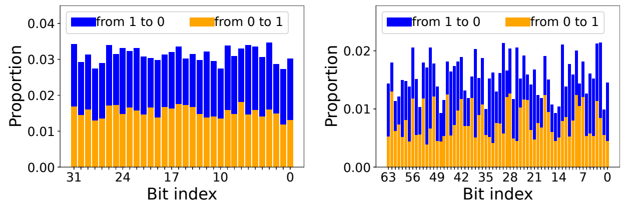
We further analyze the number of flipped bits within SDCs belonging to some bitflip pattern across different data types in Figure 7. As shown in this figure, in most cases, only one



(a) Bitflips of int32 (b) Bitflips of float32 (c) Bitflips of float64 (d) Bitflips of float64x (80bit)

(e) Precision losses of int32 (f) Precision losses of float32 (g) Precision losses of float64 (h) Precision losses of float64x (80bit)

Figure 4. Bitflips and precision losses of data with different numerical types.



(a) Bitflips of bin32 (b) Bitflips of bin64

Figure 5. Bitflips of different non-numerical types.

bitflips, but there is also a considerable number of SDCs with two or even more flipped bits.

Deficiencies on current failure models. Current failure models, such as models based on irradiation, often assume that every bitflip on every position is independent and identically distributed (IID) [26]. Another assumption made by current SDC failure models is that multiple flipped bits are unlikely events [8]. Our observations challenge current SDC failure models, and suggest areas for improvement:

- *Location preference*: Our study has shown that bitflips tend not to occur in the most significant bits in some datatypes (Observation 7).
- *Correlation between bitflips*: Our study has revealed that there exists a correlation between bitflips, causing some SDCs to have multiple bits flipped (Observation 8).

Further research is necessary to study the application implication of this model. Although floating-point numbers seems to incur less accuracy loss, some cases, such as finance data management and autonomous vehicles, advocate more reliable data [54]. It may also be possible to promote data reliability by designing encoding standards in consideration of these bitflip patterns.

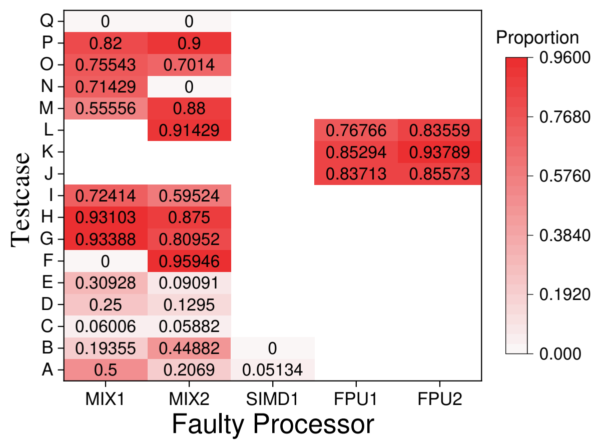


Figure 6. Proportion of SDCs with some bitflip patterns.

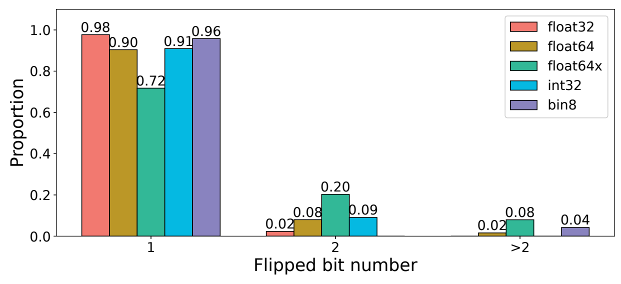


Figure 7. Proportion of the number of flipped bits in SDCs with bitflip patterns.

5 Reproducibility of SDCs

After we identify that a CPU can fail in a testcase, we repeatedly run the failed testcase to understand the reproducibility of the problem. Since the length (i.e., number of loops) of each testcase is configurable and the chance of triggering an error depends on the length of the testcase, we use occurrence frequency, which is defined as the number of errors per minute, to quantitatively measure reproducibility. Since the occurrence frequency depends on both the CPU and the workload (i.e., testcase), we record the occurrence frequency per setting.

Observation 9. *Some SDCs are highly reproducible, resulting in large impact on applications.*

We find that SDC occurrence frequency varies significantly across different settings, from as low as 0.01 times per minute to as high as hundreds of times per minute. In 51.2% of the settings, the occurrence frequency is higher than once per minute.

The high reproducibility of certain SDCs and the fact that existing systems are not designed to tolerate SDCs mean that these SDCs can manifest quickly and repeatedly in production, which is confirmed by our case studies as discussed in Section 2.2. For example, a service in Alibaba Cloud falsely reported 26 invalid-data errors in approximately 4.5 hours because of one faulty processor, which impacted the system performance. This suggests that, although the failure rate of processors is low, SDCs could potentially have a large impact, especially if they are not detected promptly.

Observation 10. *Among those less reproducible SDCs, temperature serves as an important SDC triggering condition. In some settings, the occurrence frequency of SDCs demonstrates exponential growth in response to increasing temperatures. Furthermore, the occurrence frequency is associated with the minimum triggering temperature across different faulty processors and workloads.*

It is well-known that temperature impacts the functioning of semiconductors [13, 22]. Processors have allowable range for their working temperature, and datacenters strive to minimize temperature influence through cooling systems. However, we observe that even when temperature remains within the allowable range during workload execution, the rising temperature can still increase the occurrence frequency of SDCs.

We investigate the quantitative relationship between SDC occurrence frequency and temperature. We monitor the processor temperature during testcase execution by reading cooling device monitor data from system kernel file. Some settings can naturally reach a temperature that is close to the upper bound of the processor's working temperature, which allows us to collect adequate testcase execution information with different temperatures. Some settings cannot reach a high temperature naturally. For these settings, before testing, we use stress toolchains (e.g., Linux "stress" cmd tool) to preheat the processor to the desired temperature.

By taking the base-10 logarithm value of the SDC occurrence frequency, we find that this value has a linear dependence on core temperature, based on the least square method, on six out of our 27 processors.

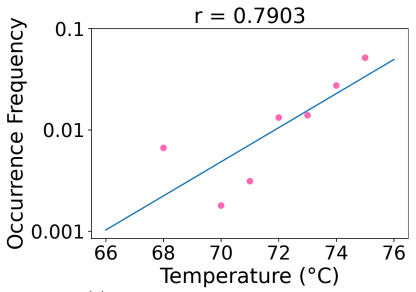
Figure 8(a)- 8(c) display this relation for some faulty processors, and their Pearson correlation coefficients are bigger than 0.75, which confirm the exponential correlation between temperature and SDC occurrence frequency.

Furthermore, we observe that in some settings, SDCs only occur when the temperature exceeds some threshold. For example, we observe all the SDC records with testcase C on MIX1 are generated with their temperature above 59℃, which is much higher than its idle temperature (about 45℃), but is still within the normal range. Tests below this temperature threshold have been extensively conducted for several days, but cannot reproduce errors.

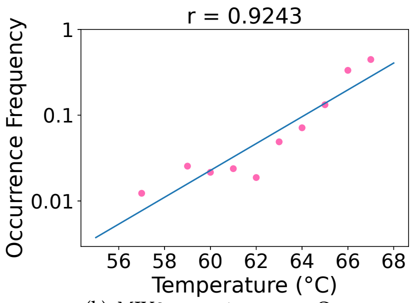
In our large-scale tests, we experience several counterintuitive cases, which we later find to be caused by temperature issues:

- *Other core behaviors*: We observe one defective core only produces errors when other cores are busy, with its occurrence frequency increasing as the number of busy cores increases. It is surprising because the defective component is not shared between cores. Upon further investigation, we discover that the cores share cooling devices, which results in the defective core reaching a higher temperature when other cores are busy.
- *Remaining heat*: We observe one faulty processor generates errors depending on the test order. For example, errors in testcase Y occur when testcase X is executed prior to testcase Y, and fail to occur with reversed test order. We later discover that testcase X exerts significant stress on the processor and produces considerable amount of heat, resulting in testcase Y being tested at a temperature that is difficult to attain when solely executing testcase Y.
- *Toolchain update.* We observe that after updating to use a higher version of the detection toolchain, the occurrence frequency of some SDCs in a faulty processor decreased, which was surprising as the update did not modify the logic of the testcases and we had not changed any other test configuration. Further investigation revealed that the updated toolchain uses a more efficient framework, which reduced the heat generated.

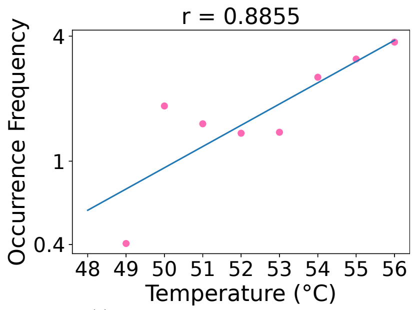
Besides temperature, there also exist other triggering conditions. Recall that we have observed that many testcases do not exhibit errors even when they utilize instructions identified as defective or suspected as discussed in Section 4.1. Our run-time instrument study further reveals that *instruction usage stress* is one of the reasons behind this observation. Failed testcases use this defective instruction several orders of magnitude more frequently than other testcases, highlighting the impact of instruction usage stress on error occurrence. Since temperature is highly correlated with stress, we use



(a) MIX1, pcore0, testcase C



(b) MIX2, pcore1, testcase C



(c) FPU2, pcore8, testcase L

Figure 8. SDC occurrence frequency (*log scale*) variation with temperature.

the following method to separate their effects: we use stress toolchain on some cores that are not under test while execute test workloads on target cores. In this experiment, since the heat is mainly produced by stress toolchain and dissipated by cooling devices, the tested workload has little impact on temperature. With this approach, we can increase CPU utilization in the faulty processor, with temperature almost unchanged, and we observe a higher occurrence frequency of SDCs with a high CPU utilization.

SDC Mitigation using multiple strategies. We further explore the design space to mitigate SDCs by combining multiple strategies in a coordinated and complementary manner. Figure 9 illustrates the relationship between the minimum triggering temperature for SDCs and their occurrence frequency under the minimum triggering temperature. We perform a linear fit between the logarithmic values of occurrence frequency and the values of minimum triggering temperature, yielding a Pearson correlation coefficient of -0.8272, which indicates a relatively strong correlation.

Motivated by this figure, we classify SDCs into two types based on the occurrence frequency and minimum triggering temperature: apparent and tricky. Different types of SDCs are suitable for different mitigation strategies. “Apparent” SDCs can be detected near idle temperature and exhibit high occurrence frequency, making them suitable for SDC tests.

On the other hand, “tricky” SDCs have higher minimum triggering temperature than “apparent” SDCs and tend to have relatively low occurrence frequency. For these SDCs, relying solely on SDC testing would require maintaining processors at high temperatures for a long time, which can be detrimental to processor health. Even worse, since we don’t know whether a CPU has such tricky SDCs in the first place, we will need to apply such long high-temperature tests to all CPUs, which is inefficient. Instead of testing, we propose to control the CPU temperature at run time to mitigate this type of SDCs. We can control the temperature by either controlling the cooling devices [7] or by limiting the CPU utilization of the workloads (called “workload backoff” in the rest of this paper). The former has no impact on application performance, but unfortunately it is not widely applicable in Alibaba Cloud yet, so this work explores the latter. Workload backoff can also reduce instruction usage stress, known as another triggering condition. Section 7.1 presents how to apply this idea in detail, in particular how to adaptively adjust the temperature threshold and test duration.

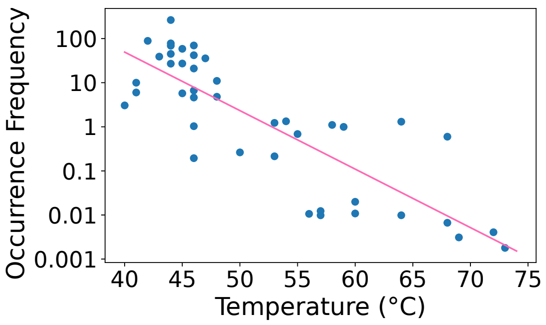


Figure 9. Relation between occurrence frequency (*log scale*) and minimum triggering temperature of different SDCs. Each point in the figure stands for a SDC setting.

6 Performance of Existing Strategies

6.1 Proactive SDC Testing

Many cloud vendors, such as Alibaba Cloud, Meta [52] and Google [50], conduct SDC tests to remove faulty processors before SDC generation. However, SDC testing can be inefficient without guidance.

Observation 11. *In a production environment with tens of thousands of CPUs, 560 out of the 633 testcases have not detected any errors.*

Unfortunately, we only have detailed test logs for a subset of the CPUs we have tested (for others, we only know whether the CPU is identified as faulty or not), but we believe they can shed light on how to improve test efficiency.

In this production environment, although we allocate the same test resources to all testcases, 560 of the 633 testcases fail to detect any faults. Moreover, if we further look at each CPU micro-architecture, the number of ineffective testcases per micro-architecture is even higher. This is reasonable since companies usually buy a specific type of CPUs in a batch, and a specific type or batch of CPUs may be vulnerable in the same way. This motivates our following proposal to prioritize tests, considering companies like chip manufacturers and cloud vendors may have a large amount of history data to guide testing: in pre-production tests, since test resources are adequate, every testcase can be fully tested; in regular tests, during which test resources are limited, we can give longer duration to testcases that have found SDCs in either pre-production tests or earlier regular tests.

On the other hand, there exist cases where our toolchain fails to detect faults. We observe that these faulty processors only manifest SDCs under some complex multi-thread conflict scenarios that are difficult to be covered with existing testcases. We believe these issues will be addressed in the future with more comprehensive and powerful testcases contributed by both academia and industry. Since our toolchain is not publicly available, for those who are interested in this field, we recommend OpenDCDiag [60] since we have validated that it can reach the same observations as our toolchain. Another similar tool is SiliFuzz [50], but we haven't got the chance to try it.

6.2 SDC Detection and Tolerance

Unlike SDC testing, which performs proactive detection before SDC generation, there exist many approaches to detect SDCs after their generation. This section discusses how our observations challenge these approaches.

Observation 12. *The effectiveness of existing fault tolerance techniques is diminished when confronted with CPU SDCs.*

Checksum and parity. End-to-end checksums are widely used to detect data corruptions and verify data integrity in the datapath [20, 27, 47]. For example, Checksum calculation algorithms, such as Cyclic Redundancy Check (CRC) and hashing, derive the data to a smaller summary, which can be used to check the integrity of the original data. Error Correcting Code (ECC) can detect and correct errors in processor cache and registers by leveraging parity bits [14, 59]; Erasure Coding (EC) techniques apply parity information to recover transferred or stored data when they are lost [2, 24].

However, we observe these techniques are often ineffective to detect CPU SDCs due to multiple reasons: 1) EC is primarily used to recover lost data, but not used to detect corrupted data. 2) ECC and CRC assume the data is correct when computing the parity and afterwards can detect bitflips in either the data or the parity bits, but CPU SDCs may generate a wrong result before parity is computed and in this case these techniques may generate a parity that matches with the already corrupted data. 3) Even if the corruption happens after parity is generated, standard ECC can correct only single bitflip errors and detect two bitflip errors, but our study shows multiple bitflip errors are possible (Observation 8).

Even worse, some of these checksum algorithms engage vulnerable features heavily, which means they are more vulnerable to CPU SDCs. For example, both EC and CRC heavily involve vector operations [21, 61], which is one of the vulnerable features (Observation 5), to accelerate computation. For EC, this is particularly dangerous since EC itself does not have the ability to detect corruptions, and thus a corrupted data block may be used to construct a lost data block, causing the corruption to propagate.

Redundancy. Some works apply redundancy to detect and tolerate SDCs [3, 6, 11, 18, 19, 23, 25, 27, 55]: they execute the same logic on multiple replicas and compare their results to detect and even correct errors. The redundancy can also be implemented by the hardware, such as using the DCLS (dual-core lockstep) technique [39]. However, considering the low failure rate of CPUs, such kind of techniques are too costly to be applied to every application, though they may be suitable for a small number of critical applications.

Prediction. Some works use machine learning models to predict the appearances of SDCs [29–31, 40–42, 49]. Part of them predict a range for the result and assert a silent error when the real result is out of this range [29–31]. However, real SDCs may have minor precision losses (Observation 7), making it challenging for these methods to determine a narrow range for detecting SDCs. On the other hand, whether such minor losses are acceptable is a topic requiring further investigation.

On the other hand, our study shows some new opportunities to detect and tolerate CPU SDCs: Considering only a small number of features or instructions are vulnerable, can we design techniques targeting those vulnerable features? Considering temperature is a key factor, can we control the temperature to mitigate SDCs? Considering bitflips have location preference, can we design better coding techniques? The next section explores some of these ideas.

7 Farron: An Efficient SDC Mitigation Tool

To illustrate how our observations assist in SDC mitigation, we propose a concrete strategy called Farron by improving Alibaba Cloud strategies based on observations aforementioned. Farron is able to protect applications from CPU SDCs with low overhead and high testing efficiency.

Baseline. Existing strategies used by Alibaba Cloud mitigate SDC impacts by conducting proactive SDC testing, which helps prevent SDC impacts by identifying and removing faulty processors before they generate SDCs. In summary, SDC tests are conducted both in pre-production and every three months during production, and in every round of tests, all testcases are executed sequentially and allocated with equal testing resources. As for one processor whose core(s) are detected as defective, Alibaba Cloud deprecates the entire processor.

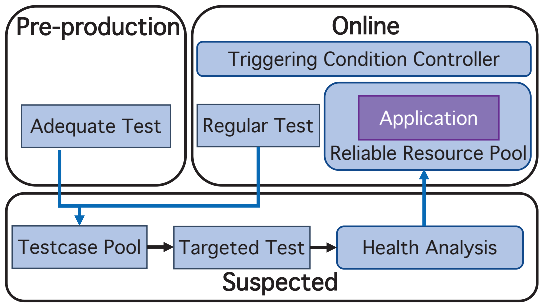


Figure 10. Workflow of Farron.

7.1 Design

Due to the limitation of SDC testing, Farron uses temperature controls as a complement to SDC testing, based on our insight from Observation 10. To determine when to activate temperature controls and when to apply SDC testing, Farron establishes a temperature boundary, which is adaptive to actual run-time conditions of the application. Farron further performs efficiency-focused SDC tests, especially in regular SDC tests. Moreover, Farron employs the fine-grained processor decommission (Observation 4) and maintains a reliable resource pool to manage unaffected cores [56].

Figure 10 illustrates the Farron workflow, which operates in three states: pre-production, online, and suspected. SDC tests with adequate resources will be performed during the pre-production state. During the online state, user application is executed on cores that have been proven reliable through SDC testing and operates under the triggering condition controlled by Farron. Regular SDC tests are conducted in this state for long-term protection. In the event that SDC tests fail, Farron performs in-depth SDC tests targeted at the suspected processor, and adjusts the reliable resources based on the analysis of test results.

Adaptive temperature boundary. As mentioned in Observation 10, different SDCs can be divided by a temperature boundary, which decides when to perform triggering condition controls and how long SDC testing needs to execute. The primary challenge Farron faces is how to determine the boundary. If the boundary is set too high, long SDC testing duration is required to guarantee reliability under high temperature (Observation 10). Conversely, if the boundary is set too low, some triggering condition controls, such as workload backoff, will be frequently activated, leading to impacts on application performance.

Farron assigns the highest priority to application performance, thereby minimizing the frequent use of workload backoff. To accomplish this, Farron differentiates the temperature boundary for cooling device operation and workload backoff, and makes the boundary for workload backoff adaptive. Farron employs a window to track recent temperature monitoring records, raising the temperature boundary for workload backoff if more than a half of temperature records within the window exceed current boundary, indicating that the temperature is within normal working range for the application in the given situation. By iteratively increasing the temperature threshold, Farron autonomously learns the standard working temperature, thereby preventing the excessive use of workload backoff. If less than half of the temperature records exceed current boundary, workload backoff will be triggered, until the temperature is below the boundary.

Farron further adjusts regular test duration based on this adaptive temperature boundary, adhering to the patterns outlined in Observation 10 (i.e. lower temperature boundary condition will be allocated less test duration).

Efficiency-focused SDC testing. Due to the constraints of online test resources, regular tests are conducted with an emphasis on testing efficiency. However, given the limited guidance available for SDC tests, achieving efficiency in existing testing procedures proves challenging (Observation 11). Farron seeks to enhance SDC testing efficiency by drawing on insights related to testcase prioritization, targeted features and testing environments (Observation 5, 10 and 11).

We designate targeted features and priorities for testcases, establishing three distinct priority levels: basic, active, suspected. The "basic" priority is assigned to testcases that, despite being designed for a particular feature, fail to detect faults in our large-scale tests. The "active" priority is designated for testcases with proven track records of successfully identifying defective features. Lastly, the "suspected" priority is only assigned to testcases that have detected errors on the core(s) of the current processor.

Farron mainly allocates testing resources to testcases whose targeted feature is utilized by the protected application, focusing on those marked as "suspected" (if any) and "active". Remaining testcases are tested in a best-effort mode, ensuring a comprehensive but efficient testing approach.

Additionally, we place a strong emphasis on the testing environment. Farron initiates the testing by running burn-in workloads and tests every core in a processor simultaneously to increase core temperature while testing (Observation 10). We believe this testing method can cover the application execution temperature, since testcases in the toolchain are stressful and effectively generate heat.

Fine-grained processor decommission. Identifying all defective cores in a faulty processor can prove difficult, as some defects may be challenging to detect (Observation 4). Initially, Farron accumulates testcases with the "suspected" priority by performing adequate testing on the cores identified with defects. By conducting adequate SDC tests targeted on these "suspected" testcases, Farron can efficiently validate the function of the remaining cores (Observation 4). If more than two cores within a processor are found defective, Farron deprecates the entire processor in line with the pattern presented in Observation 4. Conversely, Farron masks that particular defective core and continues utilizing the other cores as normal.

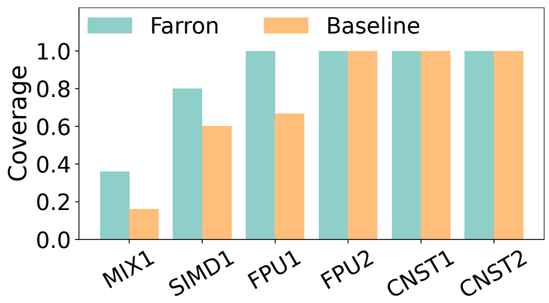


Figure 11. Regular testing coverage for faulty processors.

7.2 Evaluation

We implement and evaluate Farron on our faulty processors, and measure Farron's efficiency and overhead.

Figure 11 shows the coverage of SDCs in one round of tests, which is defined as the ratio of detected errors to the total known errors in the faulty processor. As shown in the figure, the coverage of Farron is higher than the baseline. In terms of overhead, the average one-round regular test duration of Farron is 1.02 hours, whereas in baseline, it is 10.55 hours. Both improvements stem from the prioritization strategy, which gives more resources to testcases that are likely to find errors.

Note that in some processors, there exist cases that are difficult to cover in one round of tests, since these errors need both high temperature and long-term testing. Farron mitigates them with temperature controls. We simulate workloads affected by these errors using our toolchain for hours and find these workloads do not trigger SDCs with the protection of Farron. During the procedure, Farron's workload backoff was triggered 0.864 seconds per hour on average, keeping the temperature under 59℃. Owing to the adaptive temperature boundary, the workload backoff strategy is triggered infrequently, resulting in minimal performance impact.

Table 4 presents the overhead of Farron and the baseline on different faulty processors. For Farron, the overhead includes the testing overhead and the temperature control overhead. The testing overhead is equal to the duration of one round of test over three months since regular tests are performed every three months. The temperature control overhead is equal to the backoff duration over the total duration of the simulation. The baseline only includes testing overhead. Note that for Farron, the testing overhead can vary across CPUs due to its adaptive choice of tesetcases to run and adaptive balance of testing duration and temperature control threshold.

8 Related Work

Section 6 has discussed SDC testing, detection, and tolerance in detail, so this section discussed other related works.

SDC analysis. Prior works have studied silent errors caused mainly by radiation rays, which are transient and hard to capture [10–12, 44]. Some cloud service providers have noticed SDCs caused by faulty processors in recent years [46–48, 50–52]. For example, Meta provides a case study on a SDC debugging process in the production environment [46]. Our work is a systematic study on such processor-caused SDCs in the production environment. Besides CPUs, SDCs can be produced by other system components, such as disks, memory, and TPUs [10, 33, 44, 54]. Silent errors can also be produced by software, like bugs in corner cases [47, 53].

Overhead:	Farron			Baseline
	Test	Control	Total	Test
MIX1	0.051%	0.049%	0.100%	0.488%
SIMD1	0.115%	0.031%	0.145%	
FPU1	0.017%	0	0.017%	
FPU2	0.017%	0	0.017%	
CNST1	0.033%	0.013%	0.046%	
CNST2	0.027%	0	0.027%	

Table 4. Farron overhead for different faulty processors.

SDC triggering conditions and fault injection. Environmental factors, e.g., temperature and humidity, can influence the working of electronic devices [13, 22, 28, 34, 37, 43], and we confirm that core temperature is one of triggering conditions. Modern systems use fault tolerance techniques to prevent the impact of SDCs to applications [27, 35, 45, 47, 51]. To evaluate the reliability and performance of these systems, fault injection is widely used. Some injectors use neutron beam to create SDCs according to the irradiation model [5, 44, 54], and others use the simulator or specific experimental devices to inject synthetic faults [4, 10, 32, 36]. Our observations can help improve the injector designs so as to better evaluate the solutions to SDCs in production environments.

9 Conclusion

In this research paper, we undertake a comprehensive investigation of CPU SDC phenomena with measurement and analysis in a large production environment. Our research involves multiple perspectives, including fleet maintenance, software symptoms, occurrence patterns and current practices on SDCs. We further present 12 observations, elucidating their implications on systems. Subsequently, we propose a concrete mitigation approach named Farron, which illustrates how to leverage our study to improve existing mitigation strategies.

Acknowledgements

We thank all reviewers for their insightful comments, and especially our shepherd, Baris Kasikci, for his guidance during our camera-ready preparation. We also thank Jun Yang, Ying Hu and Guangren Chai from Alibaba Cloud for helping us perform some detection experiments. This work was supported by the National Natural Science Foundation of China under Grant 62025203.

References

[1] Ravishankar K. Iyer, David J Rossetti, and Mei-Chen Hsueh. "Measurement and modeling of computer reliability as affected by system activity". In: *ACM Transactions on Computer Systems (TOCS)* 4.3 (1986), pp. 214–237.

[2] Luigi Rizzo. "Effective erasure codes for reliable computer communication protocols". In: *ACM SIGCOMM computer communication review* 27.2 (1997), pp. 24–36.

[3] Castro et al. "Practical byzantine fault tolerance". In: *OSDI*. Vol. 99. 1999. 1999, pp. 173–186.

[4] Phillipe Cheynet et al. "Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors". In: *IEEE Transactions on Nuclear Science* 47.6 (2000), pp. 2231–2236.

[5] Tanay Karnik et al. "Scaling trends of cosmic ray induced soft errors in static latches beyond 0.18/spl mu". In: *2001 Symposium on VLSI Circuits. Digest of Technical Papers (IEEE Cat. No. 01CH37185)*. IEEE. 2001, pp. 61–62.

[6] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system". In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 2003, pp. 29–43.

[7] Intel Hewlett-Packard. *Microsoft, Phoenix, and Toshiba. Advanced configuration and power interface specification*. 2004.

[8] Robert C Baumann. "Radiation-induced soft errors in advanced semiconductor technologies". In: *IEEE Transactions on Device and materials reliability* 5.3 (2005), pp. 305–316.

[9] Chi-Keung Luk et al. "Pin: building customized program analysis tools with dynamic instrumentation". In: *Acm sigplan notices* 40.6 (2005), pp. 190–200.

[10] Sarah E Michalak et al. "Predicting the number of fatal soft errors in Los Alamos National Laboratory's ASC Q supercomputer". In: *IEEE Transactions on Device and Materials Reliability* 5.3 (2005), pp. 329–335.

[11] Shubhendu S Mukherjee, Joel Emer, and Steven K Reinhardt. "The soft error problem: An architectural perspective". In: *11th International Symposium on High-Performance Computer Architecture*. IEEE. 2005, pp. 243–247.

[12] Heather Quinn and Paul Graham. "Terrestrial-based radiation upsets: A cautionary tale". In: *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*. IEEE. 2005, pp. 193–202.

[13] Eric Humenay, David Tarjan, and Kevin Skadron. *Impact of parameter variations on multi-core chips*. Tech. rep. VIRGINIA UNIV CHARLOTTESVILLE DEPT OF COMPUTER SCIENCE, 2006.

[14] Pablo Montesinos, Wei Liu, and Josep Torrellas. "Using register lifetime predictions to protect register files against soft errors". In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE. 2007, pp. 286–296.

[15] Lakshmi N Bairavasundaram et al. "An analysis of data corruption in the storage stack". In: *ACM Transactions on Storage (TOS)* 4.3 (2008), pp. 1–28.

[16] Cristian Constantinescu et al. "Silent data corruption Myth or reality?" In: *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE. 2008, pp. 108–109.

[17] Peter Markstein. "The new IEEE-754 standard for floating point arithmetic". In: *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2008.

[18] Allen Clement et al. "Upright Cluster Services". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 277–290. ISBN: 9781605587523. DOI: 10.1145/1629575.1629602. URL: https://doi.org/10.1145/1629575.1629602.

[19] Bernadette Charron-Bost, Fernando Pedone, and André Schiper. "Replication". In: *LNCS* 5959 (2010), pp. 19–40.

[20] Yupu Zhang et al. "End-to-end Data Integrity for File Systems: A ZFS Case Study." In: *FAST*. 2010, pp. 29–42.

[21] Parth Deshmukh, Sean Maginnis, and Josh Chandler. "Jerasure 2.0". In: (2011).

[22] David Solomon Wolpert. *Managing temperature effects in nanoscale adaptive systems*. University of Rochester, 2011.

[23] David Fiala et al. "Detection and correction of silent data corruption for large-scale high-performance computing". In: *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2012, pp. 1–12.

[24] Cheng Huang et al. "Erasure coding in windows azure storage". In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 2012, pp. 15–26.

[25] Manos Kapritsos et al. "All about Eve: Execute-Verify Replication for Multi-Core Servers". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 237–250. ISBN: 9781931971966.

[26] James Elliott et al. *Quantifying the impact of single bit flips on floating point arithmetic*. Tech. rep. North Carolina State University. Dept. of Computer Science, 2013.

[27] Yang Wang et al. "Robustness in the Salus scalable block store". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 2013, pp. 357–370.

[28] Christian Zorn and Nando Kaminski. “Temperature humidity bias (THB) testing on IGBT modules at high bias levels”. In: *CIPS 2014; 8th International Conference on Integrated Power Electronics Systems*. VDE. 2014, pp. 1–7.

[29] Leonardo Bautista-Gomez and Franck Cappello. “Exploiting spatial smoothness in HPC applications to detect silent data corruption”. In: *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. IEEE. 2015, pp. 128–133.

[30] Eduardo Berrocal et al. “Lightweight silent data corruption detection based on runtime data analysis for HPC applications”. In: *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. 2015, pp. 275–278.

[31] Sheng Di, Eduardo Berrocal, and Franck Cappello. “An efficient silent data corruption detection method with error-feedback control and even sampling for HPC applications”. In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE. 2015, pp. 271–280.

[32] Qiang Guan et al. “Empirical studies of the soft error susceptibility of sorting algorithms to statistical fault injection”. In: *Proceedings of the 5th Workshop on Fault Tolerance for HPC at eXtreme Scale*. 2015, pp. 35–40.

[33] Thomas Herault and Yves Robert. *Fault-tolerance techniques for high-performance computing*. Springer, 2015.

[34] Dieter K Schroder. *Semiconductor material and device characterization*. John Wiley & Sons, 2015.

[35] Bo Fang et al. “Sdc is in the eye of the beholder: A survey and preliminary study”. In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE. 2016, pp. 72–76.

[36] Xabier Iturbe, Balaji Venu, and Emre Ozer. “Soft error vulnerability assessment of the real-time safety-related ARM Cortex-R5 CPU”. In: *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE. 2016, pp. 91–96.

[37] Vinod Kumar Khanna. *Extreme-temperature and harsh-environment electronics*. IOP Publishing Limited Bristol, 2017.

[38] Haryadi S Gunawi et al. “Fail-slow at scale: Evidence of hardware performance faults in large production systems”. In: *ACM Transactions on Storage (TOS)* 14.3 (2018), pp. 1–26.

[39] Ádria Barros de Oliveira et al. “Lockstep dual-core ARM A9: Implementation and resilience analysis under heavy ion-induced soft errors”. In: *IEEE Transactions on Nuclear Science* 65.8 (2018), pp. 1783–1790.

[40] Cheng Liu et al. “SDC-causing error detection based on lightweight vulnerability prediction”. In: *Asian Conference on Machine Learning*. PMLR. 2019, pp. 1049–1064.

[41] Na Yang and Yun Wang. “Identify silent data corruption vulnerable instructions using SVM”. In: *IEEE Access* 7 (2019), pp. 40210–40219.

[42] Na Yang and Yun Wang. “Predicting the silent data corruption vulnerability of instructions in programs”. In: *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE. 2019, pp. 862–869.

[43] Davide Cimmino and Sergio Ferrero. “High-voltage temperature humidity bias test (HV-THB): Overview of current test methodologies and reliability performances”. In: *Electronics* 9.11 (2020), p. 1884.

[44] Lucas Matana Luza et al. “Effects of thermal neutron irradiation on a Self-Refresh DRAM”. In: *2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. IEEE. 2020, pp. 1–6.

[45] Dario Mamone et al. “On the analysis of real-time operating system reliability in embedded systems”. In: *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE. 2020, pp. 1–6.

[46] Harish Dattatraya Dixit et al. “Silent Data Corruptions at Scale”. In: *arXiv preprint arXiv:2102.11245* (2021).

[47] Siying Dong et al. “Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience”. In: *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 2021, pp. 33–49.

[48] Peter H Hochschild et al. “Cores that don’t count”. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. 2021, pp. 9–16.

[49] Sihuan Li et al. “Resilient error-bounded lossy compressor for data transfer”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–14.

[50] Kostya Serebryany et al. “SiliFuzz: Fuzzing CPUs by proxy”. In: *arXiv preprint arXiv:2110.11519* (2021).

[51] David F Bacon. “Detection and Prevention of Silent Data Corruption in an Exabyte-scale Database System”. In: *Workshop on Silicon Errors in Logic – System Effects, IEEE* (2022).

[52] Harish Dattatraya Dixit et al. “Detecting silent data corruptions in the wild”. In: *arXiv:2203.08989* (2022).

[53] Chang Lou, Yuzhuo Jing, and Peng Huang. “Demystifying and Checking Silent Semantic Violations in Large Distributed Systems”. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 91–107.

[54] Rubens Luiz Rech et al. “Reliability of google’s tensor processing units for embedded applications”. In:

2022 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE. 2022, pp. 376–381.

[55] Gefei Zuo et al. “Tolerate Silent Data Errors with Coded Computation”. In: *DISCC 2022 1ST WORKSHOP ON DATA INTEGRITY AND SECURE CLOUD COMPUTING*. 2022.

[56] Jialun Lyu et al. “Hyrax:{Fail-in-Place} Server Operation in Cloud Platforms”. In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 2023, pp. 287–304.

[57] George Papadimitriou and Dimitris Gizopoulos. “Silent Data Corruptions: Microarchitectural Perspectives”. In: *IEEE Transactions on Computers* (2023).

[58] Adit Singh et al. “Silent Data Errors: Sources, Detection, and Modeling”. In: *2023 IEEE 41st VLSI Test Symposium (VTS)*. IEEE. 2023, pp. 1–12.

[59] Intel Corporation. *ECC for L2 Cache Data Memory*. https://www.intel.com/content/www/us/en/docs/programmable/683360/18-0/ecc-for-l2-cache-data-memory.html.

[60] Intel Corporation. *OpenDCDiag*. https://github.com/opendcdiag/opendcdiag.

[61] Intel Corporation. *Optimizing storage solutions using the intel® intelligent storage acceleration library*. https://software.intel.com/en-us/articles/optimizing-storage-solutions-using-the-intel-intelligent-storage-acceleration-library.