Understanding Silent Data Corruption in Processors for Mitigating its Effects

SHAOBU WANG, Tsinghua University, Beijing, China
GUANGYAN ZHANG*, Tsinghua University, Beijing, China
JUNYU WEI, Tsinghua University, Beijing, China
YANG WANG, The Ohio State University, Columbus, United States
JIESHENG WU, Alibaba Cloud, Hangzhou, China
QINGCHAO LUO, Alibaba Cloud, Hangzhou, China

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 these errors, it is very hard to address problems caused by SDCs in processors. For the same reason, knowledge about these SDCs in the wild is limited.

In this paper, we conduct an extensive study on CPU 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 CPU SDCs and the triggering conditions (e.g., temperature) of those less reproducible SDCs; and 3) the challenges to mitigate and handle CPU SDCs.

We further investigate the implications which our observations obtained from the above researches have, on the SDC fault models, SDC mitigation strategies and the future research fields. In addition, we design an efficient SDC mitigation approach called Farron, which uses prioritized testing to detect highly reproducible SDCs and temperature control to mitigate less reproducible SDCs. Our experimental results indicate that Farron can achieve better coverage of CPU SDCs with lower overall overhead, compared to the baseline used in Alibaba Cloud. This demonstrates that our observations are able to assist in SDC mitigation.

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

Additional Key Words and Phrases: Processor, silent data corruption, reliability, fault tolerance

This article is an extended version of our paper appeared in SOSP '23 [61]. Over 40% of content in this paper is new, including SDC occurrence patterns over the processor lifetime, root causes of SDCs, the phenomena accompanying SDCs, the bitflip distribution, challenges on SDC handling, implications of our observations, more statistics data from newly found faulty processors, as well as details requested during the conference.

*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.

Authors' Contact Information: Shaobu Wang, Tsinghua University, Beijing, China; e-mail: wsb21@mails.tsinghua.edu.cn; Guangyan Zhang, Tsinghua University, Beijing, Beijing, China; e-mail: gyzh@tsinghua.edu.cn; Junyu Wei, Tsinghua University, Beijing, Beijing, China; e-mail: wei-jy19@mails.tsinghua.edu.cn; Yang Wang, The Ohio State University, Columbus, Ohio, United States; e-mail: wang.7564@osu.edu; Jiesheng Wu, Alibaba Cloud, Hangzhou, Zhejiang, China; e-mail: jiesheng.wu@alibaba-inc.com; Qingchao Luo, Alibaba Cloud, Hangzhou, Zhejiang, China; e-mail: chongti.lqc@alibaba-inc.com.

ACM 1544-3973/2024/9-ART
https://doi.org/10.1145/3690825

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 [29]. However, processor faults do occur in production environments [23, 24, 32, 37, 44]. 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. The second class of errors caused by processor faults is also known as *"silent data corruption"*, abbreviated as CPU SDC or SDC [23, 32].

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. CPU SDCs also occur in Google Cloud [32]: 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 CPU SDCs [23]: 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 CPU SDCs can require months of engineering time to debug [23], and Alibaba Cloud once took several weeks to debug a CPU SDC issue.

Researchers have conducted studies on CPU SDCs [3, 11, 14, 23, 24, 30, 32, 50, 55, 57], which fall into three categories: 1) theoretically stating the existence of CPU SDCs but not studying concrete errors [4, 14]; 2) studying the impacts and tolerance techniques of SDCs, through artificial fault injection instead of naturally produced SDCs [26, 28, 36, 46, 50, 52]; 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 [2, 22–24, 32, 57]. To this end, *the current knowledge about CPU SDCs in the wild is limited.*

In this paper, we *investigate CPU 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 CPU SDCs in a large-scale production environment*: We have conducted testing for CPU SDCs 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, the processor lifetime, etc., affect the failure rate and whether these failures affect a single core or all cores within the processor.
- *Investigating software impacts of CPU 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, for floating point calculations, we find that CPU SDCs are more likely to cause bitflips in the fractional part, which only cause a small loss of accuracy due to floating point encoding [45] and thus render existing accuracy-based detection techniques less effective [30].
- *Analyzing reproducibility of CPU 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 failures caused by CPU SDCs. On the other hand, some CPU 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 and SDC handling.
- *Investigating implications of CPU SDCs*. Inspired by our observations, we further provide 9 implications on fault models, mitigation strategies and future research fields.
- *Proposing a concrete approach for SDC mitigation*. Based on the above observations, we propose an efficient SDC mitigation approach called Farron. 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.

The rest of this paper is organized as follows: Section 2 describes our methodologies for the study, including the targeted system, the toolchain for SDC detection, study methodologies as well as the real-world CPU SDC cases that motivate us to conduct study on CPU SDCs. Section 3 describes the general existence pattern of CPU SDCs, including the prevalence of processor faults and their breakdown from multiple perspectives, such as micro-architecture and lifetime. Section 4 describes the software symptoms of CPU SDCs, including the impacted workloads and the corruption patterns in the data. Section 5 describes the reproducibility of CPU SDCs and the triggering conditions of errors, which provide the opportunities to mitigate CPU SDCs. Section 6 describes the limitation and the underlying causes of existing strategies against CPU SDCs. Section 7 describes the implications of our findings to address CPU SDCs, including the advancement directions of fault models, fault-tolerant strategies, and other directions for future research. Section 8 proposes a concrete approach named Farron to illustrate how to leverage our observations to enhance SDC mitigation. Section 9 concludes this paper.

2 Motivation and Methodology

2.1 Target System

We study CPU 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, and cable management, 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 CPU 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 [23, 32] have motivated us to conduct this study.

2.3 Toolchain

We use a toolchain provided by the chip manufacturer, which is designed to detect CPU SDCs related to cloud workloads. The toolchain includes 633 testcases and a framework. The framework drives these testcases and checks for the occurrence of CPU 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 [17] 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 chip manufacturer.

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

Since our toolchain is not publicly available, for those who are interested in this field, we recommend OpenDCDiag [17] since we have validated that it can reach the same observations as our toolchain. Another similar tool is SiliFuzz [57], but we haven't got the chance to try it.

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.

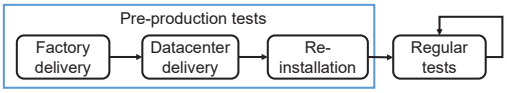


Fig. 1. Test timings in our fleet.

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

3 CPU SDCs in the Wild

3.1 Brief Overview of Test Results

Observation 1. *In general, 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" [32]) and Meta ("hundreds of CPUs detected for SDCs in hundreds of thousands of machines" [23]). Google's and Meta's decisions to not disclose exact numbers may be for business considerations. This observation substantiates the notion that CPU 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 processors 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 into 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.

Factory	Datacenter	Re-install	Regular	**Total**
0.78‰	0.18‰	2.31‰	0.35‰	3.61‰

Table 1. Failure rate of different test timings.

However, despite all SDC tests, we still encounter CPU 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 CPU SDCs. Addressing this issue is challenging, as it is not feasible to perform regular SDC tests frequently. As a result, services requiring high reliability may need to take CPU SDC tolerance into consideration.

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

Arch	M1	M2	M3	M4	M5	M6	M7	M8	M9	**avg**
Rate	4.62‰	0.35‰	2.65‰	0.08‰	0.76‰	3.25‰	1.60‰	9.29‰	4.65‰	3.61‰

Table 2. Failure rate of different micro-architectures M1-9 denotes anonymous micro-architectures

We have found faulty processors in every micro-architecture we have, indicating CPU 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‰. It is worth mentioning that we also find faults with micro-architectures different from those listed in Table 2, but these faults cannot reach a statistical conclusion due to their limited scale.

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.

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;
MIX3	M3	–	24	94	computation	matrix calculation; checksum calculation; floating-point computing; encryption;	float32; float64; float64x int8; int32; int64; bit; byte;
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. Details of a subset of our faulty processors , each line represents an individual faulty processor (#pcore denotes the number of defective physical cores on the faulty processor; #err is the number of failed testcases on the faulty processor).

Despite extensive testing and the advancements in chip development, online services continue to be exposed to potential risks stemming from CPU 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 belong to 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.

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 with the same allocation of testing resources, 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 [57], 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.

Observation 5. *Processor faults that occur before the third year of deployment contribute the majority of CPU SDCs in the production environment, revealing two necessary root causes of CPU SDCs: device faults (e.g., logic bugs and manufacturing defects), and in-field faults (e.g., transistor weakness and wearing out).*

We record the detection age of each faulty processor, namely the duration between the deployment of the processor and the identification of the fault. Faulty processors found during factory-delivery tests or datacenter-delivery tests are assigned a detection age of zero, as their faults occur before the deployment. We collect 80 faulty processors to demonstrate the detection age in the working life of processors, with 50 of them are identified during re-installation tests. Figure 2 shows the cumulative distribution of faulty processors over their lifetimes.

We suspect faults that occur before the deployment of the processor are device faults, such as logic bugs and manufacturing defects [40, 58]. These device faults cannot be eliminated absolutely before entering the production since modern processors have billions of transistors and various working conditions, posing a challenge to comprehensive testing [16]. For example, under a lower working voltage configuration, the circuit paths in the processors may become more sensitive to noise and consequently become less reliable, which introduces corner cases that fail to be stably detected by the manufacturing testing [59].

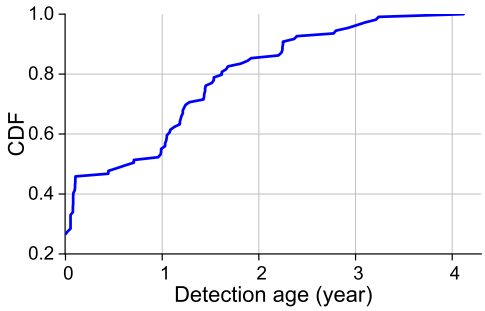


Fig. 2. CDF of faulty processors.

The detection age of other faults spreads across the whole working life of processors. These faults occur after processors being deployed, which are also known as in-field faults [23]. For example, bias temperature instability (BTI) gradually degrades the reliability of transistors under the workload stress [39]. We observe that 93.75% of these in-field faults occur in the first three years of deployment. This trend follows the utilization pattern of Alibaba Cloud: Processors mainly carry on workloads in the first three years, and from then on, workloads on processors gradually decrease. We also observe that many faulty processors can be detected in the first few weeks after their deployment, which align with existing researches [64]. Besides, we observe that few processor faults are detected after the fourth year. This is because processors in the production environment are gradually decommissioned as they reach the end-of-life phase (i.e., about five years after the deployment), leading to a reduced incidence of detected faulty processors. For the same reason, faults occurring in the end-of-life phase have a minor effect on production services.

The discrimination of device faults and in-field faults helps explain the phenomenon that the re-installation test holds the highest failure rate among test timings, as shown in Table 1. Firstly, since checks after factory delivery and datacenter delivery are undertaken before processors being deployed, they mainly detect device faults. However, they can detect a limited number of processor faults since chip manufacturers have conducted the similar tests. Secondly, re-installation tests and regular tests are undertaken after processors being deployed, which means they can capture in-field fault. Since re-installation tests are conducted offline and have more testing resources than online regular tests, re-installation tests become more effective than regular tests.

4 Software Symptoms of CPU SDCs

4.1 Impacted Workloads

Observation 6. *CPU 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 [52]). 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 CPU SDC [52]. Alternatively, defects of other features can be easily

identified during HVM testing of the chip manufacturer, and consequently can be eliminated before our testing. Either way, this observation suggests that a developer can focus on a limited set of features when considering issues related to CPU SDCs.

Figure 4 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.

We further categorize CPU 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 30 faulty processors we have tested extensively, 21 processors produce SDCs with computation type and the remaining ones produce SDCs with consistency type.

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 CPU 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 [43]. 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 13.

Observation 7. *CPU SDCs do not exist in isolation — processors that produce SDCs can lead to crashes simultaneously.*

In some faulty processors, crashes co-occur with SDCs. In other words, some faulty processors can manifest SDCs and, at times, may also experience crashes.

For example, in MIX3, we collect 7205 errors and 16.09% of them are crashes. Figure 3 shows the breakdown of part affected testcases in MIX3. We observe that the proportion of crashes varies across different testcases.

Testcase C exhibits the highest crash proportion (59.57%) among all affected testcases. This testcase applies the float64 data type to intensive computational stress. 91.29% of these crashes are caused by the aborts, where violations of certain checks or assertions in applications or operating systems trigger the system signal that terminates the testing process. Other crashes have multiple reasons, including time out, segmentation fault and illegal instruction.

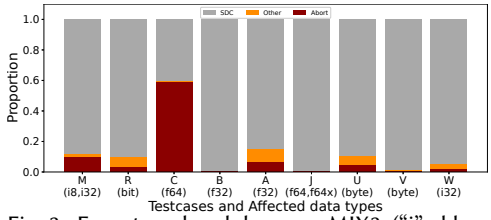


Fig. 3. Error type breakdown on MIX3. ("i" abbreviates "int" and "f" abbreviates "float")

We further conduct a case study to figure out these crashes. In one of CPU SDC cases, we identify a hash instruction intermittently giving wrong results, while we also find the operating system kernel takes advantage of this instruction, and this fault will lead to both system crashes and data corruptions.

In fact, a single micro-architecture can exhibit hundreds of logic bugs after its release, and chip manufacturing cannot avoid defects. Many of these processor faults can lead to unpredictable behaviors, including crashes and data corruptions [16]. A single fault introduces different errors among applications — some applications manifest SDCs, while others experience crashes. An application may not always experience one of unpredictable behaviors with each execution. Consequently, in the production environment, crashes and CPU SDCs occur in a mixed style.

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 do not have a deterministic pattern.

Observation 8. *CPU SDCs have been confirmed to affect operations on all tested data types, including integers, floating-point numbers, bytes, and more. Notably, operations related to floating-point numbers demonstrate heightened vulnerability to CPU SDCs.*

Table 3 shows the impacted data types in a subset of our faulty processors. We find these impacted data types cover a wide range, including both numeric data types (e.g., integer, unsigned integer, single- and double precision floating-point numbers, extended double precision floating-point numbers (float64x)), and non-numerical data types (ranging from 1bit to quad word (64bit)). Figure 5 shows the proportion of faulty processors involving each data type. We find all data types under tests are impacted by CPU SDCs, and floating-point data types involve more faulty processors than other data types. We find two reasons attribute to this issue: Many different vulnerable features are related to floating-point calculation, including vector operations with floating-point data types 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.

In some faulty processors, the CPU SDCs involve almost all the aforementioned types, such as MIX1 and MIX2. This is because the operations related to their defective features involve multiple data types. In other faulty processors, CPU SDCs only involve one fixed data type. For example, FPU3 only involves double-precision floating-point numbers, and SIMD1 only involves single-precision floating-point numbers.

Observation 9. *Different processor faults lead to distinct bitflip distributions, which can be categorized as four types: center-gathered, end-gathered, specific-bits, and uniform.*

To investigate the influence of processor faults on corrupted data, we analyze the bitflip distribution for each setting (i.e., a combination of a testcase and a faulty processor). Figure 6 present a sampling of the bitflip distribution across some settings, each involving hundreds of CPU SDC records. We observe the variation in

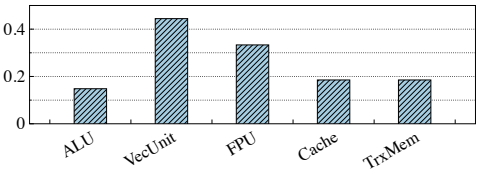


Fig. 4. Proportion of faulty processors with a certain faulty feature.

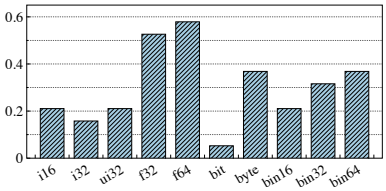
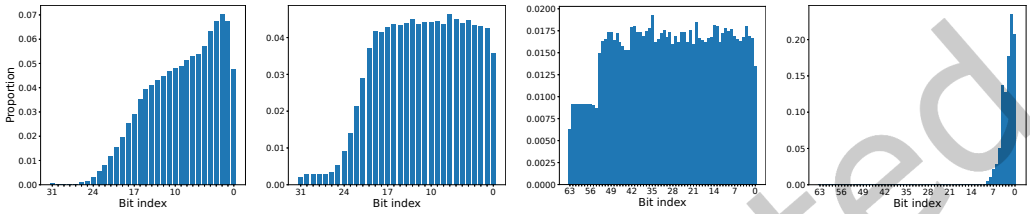
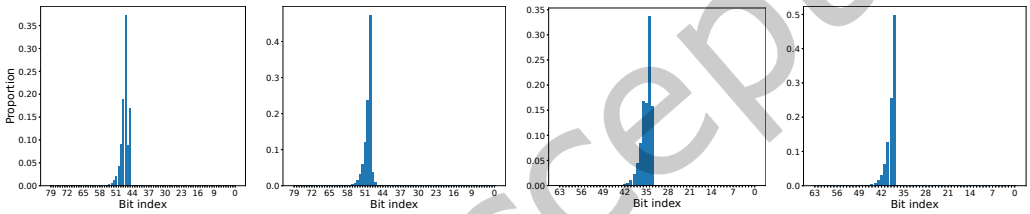


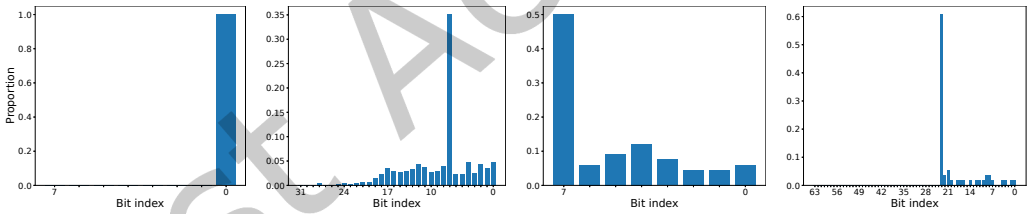
Fig. 5. Proportion of faulty processors with a certain affected operation data type.



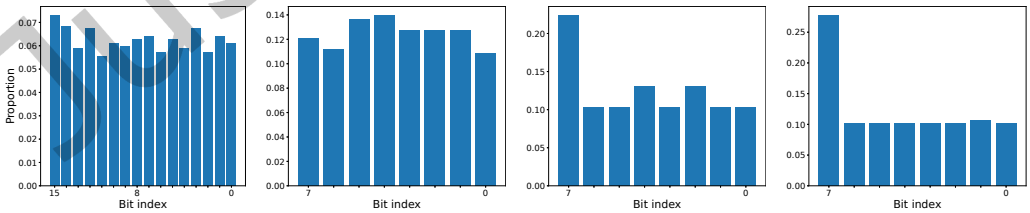
(a) SIMD1, Testcase A (b) SIMD1, Testcase B (c) MIX3, Testcase C (d) SIMD2, Testcase K



(e) FPU1, Testcase J (f) FPU2, Testcase J (g) FPU1, Testcase L (h) FPU2, Testcase K



(i) MIX3, Testcase M (j) MIX1, Testcase O (k) MIX2, Testcase I (l) MIX2, Testcase L



(m) MIX2, Testcase R (n) MIX3, Testcase I (o) MIX1, Testcase G (p) MIX1, Testcase H

Fig. 6. Bitflips of different settings.

bitflip distributions across different settings, which indicates that CPU SDCs are related to multiple types of processor faults. Certain distributions confirm previous bitflip distribution of processor faults. Besides, software

factors can also affect the bitflip distribution in some cases. We summarize four typical bitflip distributions and analyze the reasons behind them.

End-gathered. In Figures 6(a)-6(d), we observe that bitflips tend to occur at lower significant bits. This pattern can be explained by two factors. Firstly, this pattern aligns with previous studies of the timing errors [59], where bitflips are more prevalent in lower significant bits and gradually decrease towards higher significant bits, as illustrated in Figures 6(a) and 6(d). Secondly, differences in processing complexity also contribute to this phenomenon. For example, according to the IEEE-754 floating point encoding standard [45], lower significant bits belong to fraction part which has a higher processing complexity than sign and exponent parts. A case of double floating-point numbers in Figure 6(c) further reveals bits of the same type (either sign, exponent or fraction) tend to share a similar bitflip likelihood.

Center-gathered. In Figures 6(e)-6(h), we observe bitflips gather in the central region of the data and decrease gradually towards two sides. This pattern aligns with previous studies on failure distribution in registers [52], which shows a higher failure potential for bits in the central region.

Specific-bits. In Figures 6(i)-6(l), we observe that bitflips predominantly occur on specific bits. This could potentially be attributed to the specific manufacturing defects, which can only affect a specific bit [40].

Uniform. In Figures 6(m)-6(p), we observe that bitflips nearly uniformly distribute across all bits. This phenomenon can be attributed to three factors. Firstly, as shown in Figure 6(m), some workloads, such as hashing and checksum computations, can produce significantly different outputs even for minor variations in input or intermediate data. Consequently, all bits of outputs are simultaneously affected by a single processor fault in these workloads, incurring the uniform distribution. Secondly, as shown in Figure 6(n), the processor fault can incur all bits of the outputs to either zero or one. Given the random inputs, each bit has a similar likelihood to flip. Thirdly, some instructions set the same value to each bit, which leads to a uniform distribution. For example, as samples shown in Figures 6(o) and 6(p), an instruction, which is designed to compare data from two strings, is affected by the CPU SDC. Under some conditions, it sets all bits within a byte to the same value, which leads a uniformly distributed bitflips.

We also observe the correlation between bitflip pattern and faulty processor as well as the testcase by only changing one of them. Some bitflip distributions is only related to the faulty processor, regardless of the testcases. For example, Figures 6(a) and 6(b) share the same processor and have a similar bitflip pattern, which also prove such bitflip distributions are only attributed to processor fault rather than software factors. As for the bitflip distributions with the same testcase but different faulty processors, multiple factors can impact the results. For example, Figures 6(e) and 6(f) involve different faulty processors but share the same testcase while they have the similar bitflip distributions. We later find their faulty processors have the same micro-architectures, and the similarity of their bitflip distributions indicate a general challenge of this micro-architectures. In Figures 6(n) and 6(k), bitflip distributions are completely different, revealing a single processor feature may be susceptible to multiple types of processor faults.

Observation 10. *Some bitflips are position-correlated, i.e. in some settings different inputs manifest bitflips at several fixed positions.*

For CPU SDCs with specific-bits distribution, it is conceivable that the positions of bitflips remain fixed. We also observe such phenomenon, i.e. multiple records with completely different input share the same bitflip positions, in CPU SDCs of other types of bitflip distributions, which motivates us to further explore these position-correlated bitflips.

We define the position-correlated bitflips as the phenomenon that bitflip(s) occur at a fixed set of positions with whatever inputs in a given setting. For example, in Figure 6(i), all SDCs under the given setting have one bitflip that occur at the least significant bit, and we regard this as an instance of position-correlated bitflips. One potential explanation for position-correlated bitflips is that the hardware defect of specific faulty processor causes

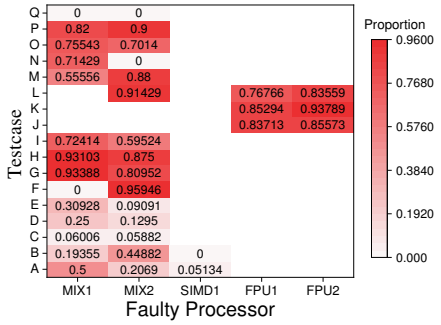


Fig. 7. Proportion of SDCs with bitflip patterns.

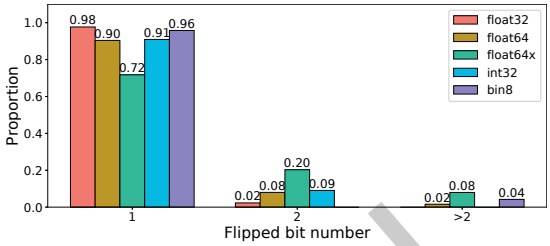


Fig. 8. Proportion of the number of flipped bits in SDCs with bitflip patterns.

deterministic influence and thus these bitflips tend to occur at fixed position(s). To explore position-correlated bitflips, we use the 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 an instance of position-correlated bitflips.

A setting could have multiple instances of position-correlated bitflips 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 instances of position-correlated bitflips. Figure 7 shows the proportion of SDC records with position-correlated bitflips in some settings. We observe the phenomenon of position-correlated bitflips is not limited in settings with specific-bit distribution. For example, CPU SDCs under the setting of testcase J and FPU2 follow the center-gathered distribution while 85.57% of them belong to instances of position-correlated bitflips. We also observe 59.46% settings in this figure have more than half their SDCs belong to instances of position-correlated bitflips. Such observation reveals that the phenomenon of position-correlated bitflips can serve as a "side channel" of CPU SDCs, namely if multiple errors in the system were logged with the same bitflip positions, the fleet of the system would be suspected to contain a CPU SDC.

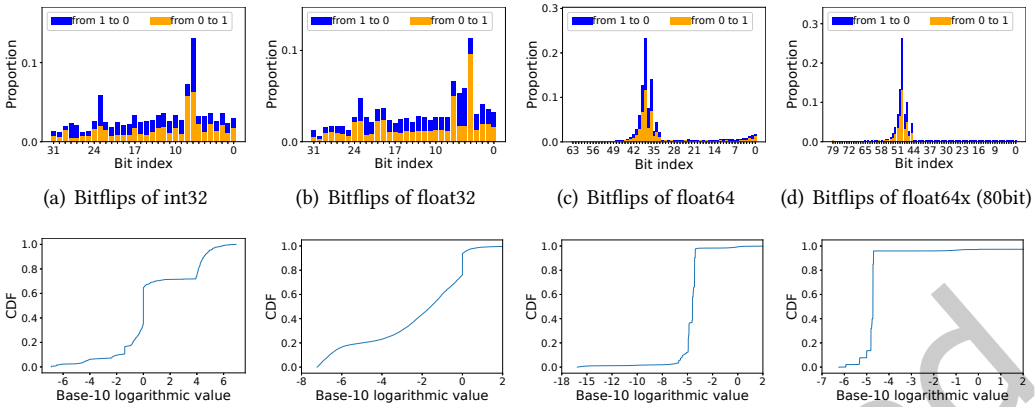
We further analyze the number of flipped bits within SDCs belonging to some instances of position-correlated bitflips across different data types in Figure 8. As shown in this figure, in most cases, only one bitflip, but there is also a considerable number of SDCs with two or even more flipped bits. Our analysis does not include CPU SDCs that do not belong to any bitflip patterns, as many of these SDCs have been propagated, where a single SDC record contains multiple bitflips are becoming increasingly common.

Observation 11. *For floating-point numbers, bitflips predominantly occur in the fractional part, resulting in minor precision losses.*

Given the fact that some data types follow specific encoding standards, the role of each bit differs, which consequently leads to specific influence on the value of these data types, especially floating-point data types.

Figures 9(a)-9(d) shows the bitflips of different numerical data types. We find that it is rare that bitflips occur in the most significant bits. This bitflip pattern does not apply to non-numerical data, where all positions have a comparable proportion of bitflips (Figures 10(a)and 10(b)).

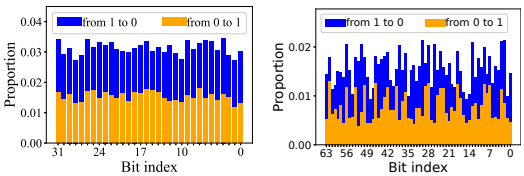
Furthermore, we find that nearly half (51.08%) of bitflips are changed from zero to one, which means there is no tendency of bitflip direction in general. However, a tendency exists in some corner cases. For example, in 16-bit integer data within MIX1, the statistics of bitflips are skewed, with 72.27%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



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

(e) Precision loss of int32 (f) Precision loss of float32 (g) Precision loss of float64 (h) Precision loss of float64x

Fig. 9. Bitflips and precision losses of data with numerical data types.



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

Fig. 10. Bitflips and precision losses of data with non-numerical data types.

The impact of a CPU SDC on a numerical data depends on the type of the data. A bitflip usually hits the fractional part, probably due to two reasons. Firstly, as for floating-point numbers, the computation logic of the fractional part is more complex than that of the exponent part, making the fractional part more vulnerable. Secondly, given the relatively long fractional part in the data, bitflips that follow the center-gathered or end-gathered distribution 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 data types 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 Figures 9(e)- 9(h). Notably, few of CPU SDC instances result in a NaN (Not a Number) corruption, which is regarded as 100% precision losses. Since the bitflips we observed mostly occur in the fraction bits, the precision losses of floating-point data types are small. For example, 95% of the precision losses on extended double precision (80bit) floating-point numbers are less than 0.002%. 97.5% of the precision losses on double precision (64bit) floating-point numbers are less than 0.01%. half of the precision losses on single precision (32bit) floating-point numbers are less than 2.7%. On the other side, 35.37% of the precision losses on 32-bit integer data are bigger than 100%. This phenomenon (i.e., data corruptions of floating-point numbers have minor precision losses) can be more extreme when we only regard to a single faulty processor. For example, in each faulty processor where bitflips of extended double precision floating-point numbers follow the center-gathered distribution, all the precision losses on this data type are less than 0.002%.

5 Reproducibility of CPU 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 12. *Some CPU SDCs are highly reproducible, resulting in large impact on applications.*

We find that occurrence frequency of CPU SDCs 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 CPU SDCs mean that these SDCs can manifest quickly and repeatedly in production, which is confirmed by our case studies in production environment: 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, CPU SDCs could potentially have a large impact, especially if they are not detected promptly.

Observation 13. *Among those less reproducible SDCs, temperature serves as an important SDC triggering condition. In some settings, the occurrence frequency of CPU 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 [35, 65]. 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 CPU 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.

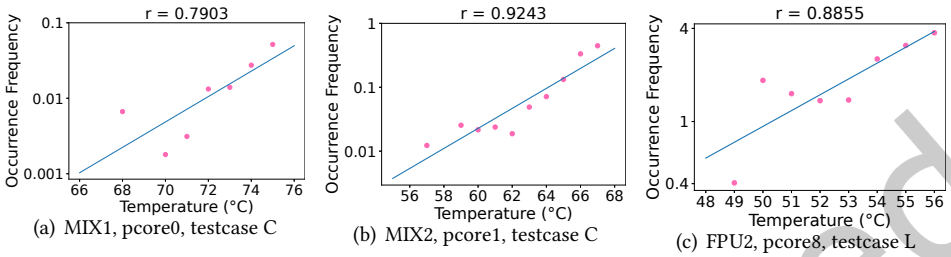
Figures 11(a)- 11(c) display this relation for some faulty processors, and their Pearson correlation coefficients are bigger than 0.75, which confirm the exponential correlation between temperature and SDC occurrence frequency.

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 counter-intuitive cases 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.



(a) MIX1, pcore0, testcase C (b) MIX2, pcore1, testcase C (c) FPU2, pcore8, testcase L

Fig. 11. SDC occurrence frequency (*log scale*) variation with temperature.

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 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. And the occurrence frequency of the error decreases when we reduce CPU utilization (with adequate test duration). We presume this stress impacts the electronic working environment of some transistors, such as temperature and voltage, which in turn impacts their functions [58, 65]. Consequently, we consider workload operations that intensively utilize vulnerable features as susceptible.

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 12 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 do not know whether a CPU has such tricky SDCs in the first place, we will need to apply such long high-temperature testing to all CPUs, which is inefficient. Additionally, we observe there are faulty processors where all SDCs require relatively high temperatures. In these cases, SDC testing becomes especially inefficient since it is difficult to identify the processor as faulty in the first place.

Instead of testing, we propose to control the CPU temperature at run time to mitigate "tricky" SDCs. Related works indicate that the workload temperature on a server is lower than a specific threshold in production environments, which is far below the maximum working temperature of the processors [48, 60]. In another words, workloads do not meet the triggering conditions of some "tricky" SDCs, reducing the need for testing of specific CPU SDCs. As for the occasional conditions with high-temperature, these SDCs can be mitigated by controlling the temperature. We can control the CPU temperature by either controlling the cooling devices [31] 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 8.1 presents how to apply this idea in detail, in particular how to adaptively adjust the temperature threshold and test duration.

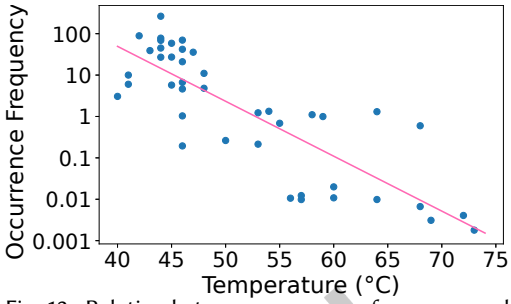


Fig. 12. Relation between occurrence frequency and minimum triggering temperature of different SDCs. Each point in the figure stands for a SDC setting.

6 Performance of Existing Strategies

6.1 Proactive SDC Testing

Many cloud vendors, such as Alibaba Cloud, Meta [22] and Google [57], conduct SDC tests to remove faulty processors before SDC generation. However, testing for CPU SDCs can be inefficient without guidance.

Observation 14. *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. We verify that about several tens of testcases can detect faults with more testing resource, which exceed the sustainability of production environments, but still remain failed to verify the testing ability of other hundreds of testcases. 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.

Additionally, determining the best testcase for detection is challenging. Since detection performance varies across different faulty processors, a testcase can be effective in one processor but ineffective in another. For instance, while testcase C outperforms testcase M in one micro-architecture, the opposite holds true in another.

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.

6.2 SDC Detection, Tolerance, and Handling

Unlike SDC testing, which performs proactive detection before SDC generation, there exist many approaches to detect corruptions after error generation.

Observation 15. *The effectiveness of existing fault tolerance techniques is diminished when confronted with CPU SDCs.*

Checksum and parity. End-to-end checks are widely used to detect data corruptions and verify data integrity in the datapath [24, 53, 62, 68]. 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. End-to-end check techniques are widely applied in software workloads [24, 62, 68], and they are equally applicable in firmware programs [53]. Erasure Coding (EC) techniques are software techniques that apply parity information to recover transferred or stored data when they are lost [33, 56]. Error Correcting Code (ECC) can detect and correct errors in processor cache and registers by leveraging parity bits [15, 47]. These ECC techniques are integrated in hardware circuits of processors, thereby becoming transparent to software operations.

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 used in processor can correct only single bitflip errors and detect two bitflip errors, but our study shows multiple bitflip errors are possible (Observation 10).

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 [18, 20], which is one of the vulnerable features (Observation 6), 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 corruptions occur in the either computational process or stored data [9, 10, 13, 26, 27, 38, 50, 62, 69]: they execute the same logic on multiple replicas and compare their results to detect and even correct errors. Redundancy techniques are widely applied in software workloads [26, 69], and they can also be implemented by the hardware, such as the DCLS (dual-core lockstep) technique [19]. Redundancy techniques can tolerate CPU SDCs. 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.

Ad-hoc. Several strategies are designed for protecting specific operations, such as algorithm-based fault tolerance (ABFT) and residue codes [1, 6, 8, 12, 34, 49]. The core idea of these strategies is to utilize certain relation within data (e.g., linear dependence) which should hold during the execution of operations. Upon an error occur, they can detect the error by checking the relation within data. Such techniques can be applied in both software [8, 34] and hardware [1].

Such ad-hoc techniques can dramatically reduce detection overhead, especially compared with redundancy techniques. However, their practical application is limited by the issue of "non-universality". For example, residue codes are designed for integer computation, and fail to protect floating-point computation [49]. Additionally, most of ABFT techniques are designed for algorithms used in high-performance computing (HPC) scenarios, such as ordinary differential equations and bitonic sort, and there remains a necessity to extend protection to the other operations susceptible to CPU SDCs.

Prediction. Some works use machine learning models to predict the appearances of corruptions [5, 7, 21, 41, 42, 66, 67]. These techniques are usually deployed in software workloads. Part of them predict a range for the result and assert a silent error when the real result is out of this range [5, 7, 21]. However, CPU SDCs may have minor precision losses (Observation 11), making it challenging for these methods to determine a narrow range for detecting CPU 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? Section 8 explores some of these ideas.

Observation 16. *Due to the inefficient diagnosis and handling strategies, CPU SDCs introduce significant human costs and the risk of irreparable failures.*

Like the samples mentioned in Section 2.2, some production services have set some strategies for data integrity verification, which enables systems to detect some CPU SDCs before being exposed to the user. In these samples, however, CPU SDCs still trouble services. This motivates us to investigate challenges on SDC diagnosis and handling.

Due to the silent nature, intermittent occurrence and insufficient knowledge when facing SDCs, it is challenging to debug a SDC problem. Alibaba Cloud spent weeks of engineering time for diagnosing a single SDC issue, and so did other cloud service vendors like Meta [23]. Even with a comprehensive understanding about the influence of SDCs on specific applications after encountering repeated instances of similar SDC issues, the debugging process continues to rely on manual identification skills of experienced engineers. To save engineering costs, there is still a demand for automated SDC diagnosis techniques and SDC-related information reporting techniques.

Additionally, existing systems also need to improve the SDC handling process. Given that the root cause of a SDC issue is not pinpointed immediately, the faulty processor will continuously generate SDCs and affect the system within a time interval. Consequently, at the beginning of the troubleshooting and recovery, systems will encounter multiple SDCs. These SDCs, with spatial and temporal locality, have the potential to exceed the protection ability of fault tolerance strategies, and thus have a widespread impact. In order to shorten the time interval of SDC influence, efficient troubleshooting strategies, especially responsive to repeat offenders (i.e., components that frequently correlate with errors), are imperative.

7 Implications

CPU SDCs have become a non-negligible challenge for systems. This section discusses their impacts on fault models, optimizations of strategies against CPU SDCs, and future research fields to better address SDC-related problems.

7.1 Optimizations of Fault Models

Fault models are widely used to analyze potential consequences, design related strategies, and build fault injectors for simulation experiments. When assessing the occurrence of SDCs in processors, existing fault models predominantly focus on the SDCs caused by exogenous factors (e.g., irradiation), but fail to consider CPU SDCs, which are caused by endogenous factors (i.e., hardware faults of processors) [46, 50, 51, 54]. Due to the significant difference between SDCs caused by exogenous factors and CPU SDCs, these fault models fail to accurately describe the errors, and their shortcomings result in the lack of efficient strategies. Following this, we investigate challenges on existing fault models through our observations, and explore opportunities for constructing an advanced model.

CPU SDC is not just a "Black Swan" event. Plenty of CPU SDC cases occur in the production environment, encompassing various testing timings, various micro-architectures and the whole lifetime of the processor (Observation 2, 3 and 5), with a non-negligible failure rate and occurrence frequency (Observation 1 and 12). However, in existing fault models, SDC that impacts the processor computing is an extremely unlikely event, with the MTTF (mean time to failure) as 1000 years [50]. As a result, CPU SDC is no longer limited to particular scenarios, such as space applications and critical applications, but becomes a general problem, especially in the scenarios of addressing scalability challenges.

Patterns in corrupted data. It is often assumed in SDCs caused by exogenous factors that each bit position has an equal probability of flipping [25], i.e., the distribution of SDCs caused by exogenous factors follow a uniform distribution. However, bitflips of CPU SDCs can be governed by other types of distribution besides the uniform one and can be position-correlated (Observation 9 and 10). This further has significant implications for systems, including the value of corrupted data and the effectiveness of fault tolerance techniques (Observation 11 and 15). As a result, an advanced fault model of CPU SDCs should consider various types of bitflip distributions and SDCs with the same bitflip position(s).

Significant occurrence variance. The occurrence of CPU SDCs is intermittent, but not transient or permanent like assumptions in existing fault models [40]. However, due to the significant variance among different settings (Observation 12), CPU SDCs face challenges of both transient errors and permanent errors: Less reproducible SDCs act like transient errors, where the poor reproducibility makes identifying the root cause a challenge. Highly reproducible SDCs act like permanent errors, where systems must handle these SDCs before restarting, even when systems can tolerate multiple such errors. As a result, an advanced fault model should consider the full scope of reproducibility in CPU SDCs. For example, a fault injector for CPU SDCs should offer a wide range for the configuration of the occurrence frequency.

7.2 Optimizations of Strategies against CPU SDCs

In Section 6, we investigate the inadequacies of existing strategies against CPU SDCs. In this section, we further present some optimization directions based on our observations.

Operation-specific fault tolerance. The susceptibility to CPU SDCs varies across different operations. Operations that extensively use vulnerable features and produce significant heat are more susceptible than those rarely use such features (Observation 6 and 13). To enhance efficiency of fault tolerance techniques, it is practical to employ different strategies to different operations according to their susceptibility. For example, employing redundancy strategies for highly vulnerable operations and employing testing strategies for the rest can reduce the overall overhead of system protection. Additionally, considering that compared with SDCs caused by exogenous factors, only specific operations are susceptible to CPU SDCs (Observation 6), the need for universal strategies diminishes. Consequently, it is appropriate to design ad-hoc strategies to these susceptible operations, particularly those that are frequently utilized in systems.

Utilizing unaffected cores in faulty processors. Defects of a faulty processor can manifest in a subset of processor cores (Observation 4). Large companies decommission the whole faulty processor or isolate the whole machine no matter which of its cores is identified as faulty [22, 32]. This practice is reasonable given the relatively low failure rate in the current cluster maintenance. However, this practice also leads to the waste of unaffected cores in faulty processors. With the increasing maintenance difficulty of a cluster, such as underwater datacenters and zero-maintenance storage systems, it is worthwhile to investigate the feasibility of continuing to utilize the unaffected cores within a faulty processor [44]. Furthermore, utilizing part of the processors can lead to heterogeneous servers, and thus scheduling strategies are imperative to achieve load balance.

Highlighting the abnormal behaviors in the systems. We observe that some phenomena occur alongside CPU SDCs, potentially serving as "side channels" for detecting CPU SDCs. For example, some CPU SDCs are co-occurred with crashes or have the same bitflip positions (Observation 7 and 10). A practical approach to leverage these side channels is to analyze system logs: If a processor in the system frequently associates with crashes, it could also produce SDCs. Similarly, if multiple errors in the systems have the same bitflip positions, it may indicate a faulty processor with the potential to produce SDCs.

7.3 Future Research Fields

Beyond the scope of our investigation in this paper, several research areas worth future efforts. We highlight the necessity of these fields in light of our observations.

SDC in processing units. Many hardware devices, such as graphics processing unit (GPU), tensor processing unit (TPU), and data processing unit (DPU), have similar manufacturing processes and logic designs to the CPU. Consequently, they share reliability challenges with the CPU. In other words, SDCs can also occur in them and SDCs in different processing units can have similar patterns. Compared with CPU, these processing units have more instances of the vulnerable features and vulnerable datatypes. For example, one GPU can have thousands of cores for floating-point calculations, and floating-point numbers are the most vulnerable one of datatypes. DPU is designed for data-centric operations such as encryption, which is proven to be susceptible to CPU SDCs. These processing units have been extensively engaged in clusters, and tend to scale up in response to popular applications like machine learning. Additionally, they are also becoming more complex. The scale-up situation and increasing complexity highlight the necessity of research for their SDC issues. For example, testing toolchains are essential for these processing units.

Evaluating the vulnerability of features. Vulnerable features identified in our study are limited to the micro-architectures used in Alibaba Cloud and testing ability of the toolchain. As the technology advances, more processor features will emerge in the future. As a result, methodologies to evaluate the feature vulnerability are imperative. Such methodologies aid in the construction of effective testcases and design of efficient testing toolchains. Our study can be used to verify evaluation methodologies, since we have pointed out some vulnerable features and non-vulnerable features. We also assume that some factors, such as the complexity of hardware components (Observation 6), should be considered.

Influence in specific scenarios. There still lack detailed studies for the influence of CPU SDCs in specific scenarios, such as machine learning and high-performance computing (HPC). It is possible that CPU SDCs have minor influence in such scenarios. For example, since floating-point number calculation tends to incur less accuracy losses, quantization technology [63] used in machine learning applications can effectively tolerate many of these minor precision losses. Besides, some algorithms used in HPC, such as Jacobi method, can tolerate these minor precision losses through several computation iterations [25]. This indicates that although floating-point numbers are most vulnerable to SDCs, the actual impacts may not be large. However, it is also possible that CPU SDCs can still lead to catastrophes in these scenarios. Firstly, faulty processors repeatedly manifest SDCs, which pollute the whole data stream rather than certain stored data. Secondly, SDCs occur at specific processes can lead to critical errors (i.e., errors that make models give wrong answers) [55]. In the further, it is beneficial to study the accurate influence of CPU SDCs in specific scenarios.

8 Improving SDC Mitigation Strategies with Our Observations

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 impacts of CPU SDCs 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.

8.1 Design

Due to the limitation of SDC testing, Farron uses temperature controls as a complement to SDC testing, based on our insight from Observation 13. 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 and maintains a reliable resource pool to manage unaffected cores [44].

Figure 13 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.

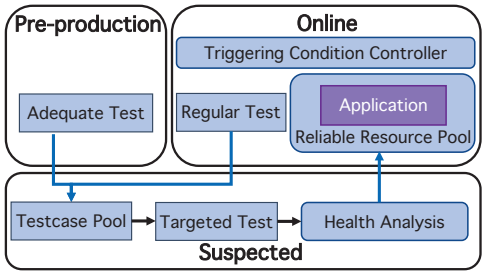


Fig. 13. Workflow of Farron.

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 13, 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 testing duration is required to guarantee reliability under high temperature. Conversely, if the boundary is set too low, some triggering condition controls 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. As illustrated in Algorithm 1, 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 learns the standard working temperature, thereby preventing the excessive use of workload backoff. when the temperature records exceed current boundary, workload backoff will be triggered to protect the application.

Farron further adjusts regular test duration based on this adaptive temperature boundary, adhering to the patterns outlined in Observation 13 (i.e. lower temperature boundary condition will be allocated less test duration).

Algorithm 1: Adaptively adjust temperature boundary

```
1  window[] ← fill_with_temperature(idle_temperature, window_len − 1) // Initialize record window
2  boundary ← idle_temperature // Initialize safe temperature
3  while targeted program still running do
4      window[].enqueue(now_temperature())
5      if window.count(> boundary) ≥ window_len/2 then
6          boundary ← window.filter(> boundary).min() // Update safe temperature
7      if window.tail() > boundary then
8          start_workload_backoff() // Trigger workload backoff
9          while now_temperature() > boundary do
10             sleep(backoff_interval)
11         stop_workload_backoff()
12     sleep(record_interval)
13     window.dequeue() // Ready for next temperature record
```

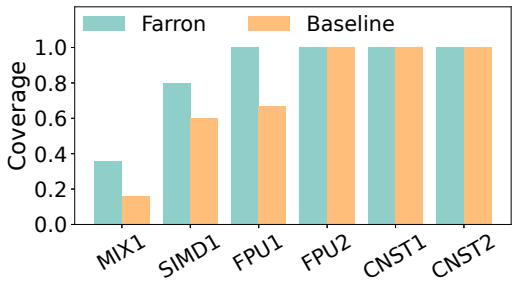


Fig. 14. Regular testing coverage for faulty processors.

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.

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. Farron seeks to enhance SDC testing efficiency by drawing on insights related to testcase prioritization, targeted features and testing environments (Observation 6, 13 and 14).

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. We believe this testing method can cover the application execution temperature, since testcases in the toolchain are stressful and effectively generate heat (Observation 13).

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. 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.

8.2 Evaluation

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

Figure 14 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 more effective testcases.

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. We find Farron can effectively control the run-time temperature of workloads below the boundary and these workloads do not trigger SDCs with the

protection of Farron. For example, in MIX1, temperature of the server can be controlled below 60°C with diverse workloads, while we observe some SDCs on this processor needs more than 70°C to trigger. During the procedure, Farron's workload backoff was triggered 0.864 seconds per hour on average, keeping the temperature under 59°C. 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.

9 Conclusion

In this 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 present 16 observations, and further elucidate their implications on fault models, strategy optimizations, and future research fields. Subsequently, we propose a concrete mitigation approach named Farron, which illustrates how to leverage our study to improve SDC mitigation strategies.

Acknowledgements

We thank all reviewers for their insightful comments and helpful suggestions. We also thank Jun Yang, Ying Hu and Guangren Chai from Alibaba Cloud for helping us perform some experiments. This work was supported by the National Natural Science Foundation of China under Grant 62025203.

References

[1] Algirdas Avizienis. 1973. Arithmetic algorithms for error-coded operands. *IEEE Trans. Comput.* 100, 6 (1973), 567–572.
[2] David F Bacon. 2022. Detection and Prevention of Silent Data Corruption in an Exabyte-scale Database System. *Workshop on Silicon Errors in Logic – System Effects, IEEE* (2022).
[3] Lakshmi N Bairavasundaram, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Garth R Goodson, and Bianca Schroeder. 2008. An analysis of data corruption in the storage stack. *ACM Transactions on Storage (TOS)* 4, 3 (2008), 1–28.
[4] Robert C Baumann. 2005. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and materials reliability* 5, 3 (2005), 305–316.
[5] Leonardo Bautista-Gomez and Franck Cappello. 2015. 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, 128–133.
[6] Austin R Benson, Sven Schmit, and Robert Schreiber. 2015. Silent error detection in numerical time-stepping schemes. *The International Journal of High Performance Computing Applications* 29, 4 (2015), 403–421.
[7] Eduardo Berrocal, Leonardo Bautista-Gomez, Sheng Di, Zhiling Lan, and Franck Cappello. 2015. 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*. 275–278.
[8] Edson T Camargo and Elias P Duarte. 2021. An Algorithm-Based Fault Tolerance Strategy for the Bitonic Sort Parallel Algorithm. In *2021 10th Latin-American Symposium on Dependable Computing (LADC)*. IEEE, 1–10.
[9] Castro et al. 1999. Practical byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.
[10] Bernadette Charron-Bost, Fernando Pedone, and André Schiper. 2010. Replication. *LNCS* 5959 (2010), 19–40.
[11] Phillipe Cheynet, Bogdan Nicolescu, Raoul Velazco, Maurizio Rebaudengo, M Sonza Reorda, and Massimo Violante. 2000. Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. *IEEE Transactions on Nuclear Science* 47, 6 (2000), 2231–2236.
[12] R Chien. 1964. On linear residue codes for burst-error correction. *IEEE Transactions on Information Theory* 10, 2 (1964), 127–133.

[13] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. 2009. Upright Cluster Services. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 277–290. https://doi.org/10.1145/1629575.1629602
[14] Cristian Constantinescu, Ishwar Parulkar, Rick Harper, and Sarah Michalak. 2008. Silent data corruption Myth or reality?. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 108–109.
[15] Intel Corporation. [n. d.]. 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.
[16] Intel Corporation. [n. d.]. Intel Skylake/Kaby Lake processors: broken hyper-threading. https://lists.debian.org/debian-devel/2017/06/msg00308.html.
[17] Intel Corporation. [n. d.]. OpenDCDiag. https://github.com/opendcdiag/opendcdiag.
[18] Intel Corporation. [n. d.]. 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.
[19] Ádria Barros de Oliveira, Gennaro Severino Rodrigues, Fernanda Lima Kastensmidt, Nemitala Added, Eduardo LA Macchione, Vitor AP Aguiar, Nilberto H Medina, and Marcilei AG Silveira. 2018. Lockstep dual-core ARM A9: Implementation and resilience analysis under heavy ion-induced soft errors. *IEEE Transactions on Nuclear Science* 65, 8 (2018), 1783–1790.
[20] Parth Deshmukh, Sean Maginnis, and Josh Chandler. 2011. Jerasure 2.0. (2011).
[21] Sheng Di, Eduardo Berrocal, and Franck Cappello. 2015. 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, 271–280.
[22] Harish Dattatraya Dixit and Others. 2022. Detecting silent data corruptions in the wild. *arXiv:2203.08989* (2022).
[23] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. 2021. Silent Data Corruptions at Scale. *arXiv preprint arXiv:2102.11245* (2021).
[24] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. 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)*. 33–49.
[25] James Elliott, Frank Mueller, Frank Stoyanov, and Clayton Webster. 2013. *Quantifying the impact of single bit flips on floating point arithmetic*. Technical Report. North Carolina State University. Dept. of Computer Science.
[26] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. 2012. Detection and correction of silent data corruption for large-scale high-performance computing. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.
[27] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 29–43.
[28] Qiang Guan, Nathan DeBardeleben, Sean Blanchard, and Song Fu. 2015. 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*. 35–40.
[29] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. 2018. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage (TOS)* 14, 3 (2018), 1–26.
[30] Thomas Herault and Yves Robert. 2015. *Fault-tolerance techniques for high-performance computing*. Springer.
[31] Intel Hewlett-Packard. 2004. Microsoft, Phoenix, and Toshiba. Advanced configuration and power interface specification.
[32] 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*. 9–16.
[33] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure coding in windows azure storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 15–26.
[34] Kuang-Hua Huang and Jacob A Abraham. 1984. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers* 100, 6 (1984), 518–528.
[35] Eric Humenay, David Tarjan, and Kevin Skadron. 2006. *Impact of parameter variations on multi-core chips*. Technical Report. VIRGINIA UNIV CHARLOTTESVILLE DEPT OF COMPUTER SCIENCE.
[36] Xabier Iturbe, Balaji Venu, and Emre Ozer. 2016. Soft error vulnerability assessment of the real-time safety-related ARM Cortex-R5 CPU. In *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 91–96.
[37] Ravishankar K. Iyer, David J Rossetti, and Mei-Chen Hsueh. 1986. Measurement and modeling of computer reliability as affected by system activity. *ACM Transactions on Computer Systems (TOCS)* 4, 3 (1986), 214–237.
[38] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. 2012. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 237–250.
[39] Florian Klemme and Hussam Amrouch. 2021. Machine learning for circuit aging estimation under workload dependency. In *2021 IEEE International Test Conference (ITC)*. IEEE, 37–46.

[40] Sandip Kundu, Sujit T Zachariah, Sanjay Sengupta, and Rajesh Galivanche. 2001. Test challenges in nanometer technologies. *Journal of Electronic Testing* 17 (2001), 209–218.

[41] Sihuan Li, Sheng Di, Kai Zhao, Xin Liang, Zizhong Chen, and Franck Cappello. 2021. Resilient error-bounded lossy compressor for data transfer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[42] Cheng Liu, Jingjing Gu, Zujia Yan, Fuzhen Zhuang, and Yunyun Wang. 2019. SDC-causing error detection based on lightweight vulnerability prediction. In *Asian Conference on Machine Learning*. PMLR, 1049–1064.

[43] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[44] Jialun Lyu, Marisa You, Celine Irvene, Mark Jung, Tyler Narmore, Jacob Shapiro, Luke Marshall, Savyasachi Samal, Ioannis Manousakis, Lisa Hsu, et al. 2023. Hyrax:{Fail-in-Place} Server Operation in Cloud Platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 287–304.

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

[46] Sarah E Michalak, Kevin W Harris, Nicolas W Hengartner, Bruce E Takala, and Stephen A Wender. 2005. Predicting the number of fatal soft errors in Los Alamos National Laboratory's ASC Q supercomputer. *IEEE Transactions on Device and Materials Reliability* 5, 3 (2005), 329–335.

[47] Pablo Montesinos, Wei Liu, and Josep Torrellas. 2007. 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, 286–296.

[48] Justin D Moore, Jeffrey S Chase, Parthasarathy Ranganathan, and Ratnesh K Sharma. 2005. Making Scheduling" Cool": Temperature-Aware Workload Placement in Data Centers.. In *USENIX annual technical conference, General Track*. 61–75.

[49] Shubu Mukherjee. 2011. *Architecture design for soft errors*. Morgan Kaufmann.

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

[51] George Papadimitriou and Dimitris Gizopoulos. 2021. Characterizing soft error vulnerability of cpus across compiler optimizations and microarchitectures. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 113–124.

[52] George Papadimitriou and Dimitris Gizopoulos. 2023. Silent Data Corruptions: Microarchitectural Perspectives. *IEEE Trans. Comput.* (2023).

[53] Martin K Petersen. 2008. Linux data integrity extensions. In *Linux Symposium*, Vol. 4. 5.

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

[55] Rubens Luiz Rech and Others. 2022. Reliability of google's tensor processing units for embedded applications. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 376–381.

[56] Luigi Rizzo. 1997. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM computer communication review* 27, 2 (1997), 24–36.

[57] Kostya Serebryany, Maxim Lifantsev, Konstantin Shtoyk, Doug Kwan, and Peter Hochschild. 2021. SiliFuzz: Fuzzing CPUs by proxy. *arXiv preprint arXiv:2110.11519* (2021).

[58] Adit Singh, Sreejit Chakravarty, George Papadimitriou, and Dimitris Gizopoulos. 2023. Silent Data Errors: Sources, Detection, and Modeling. In *2023 IEEE 41st VLSI Test Symposium (VTS)*. IEEE, 1–12.

[59] Adit D Singh. 2022. Understanding vmin failures for improved testing of timing marginalities. In *2022 IEEE International Test Conference (ITC)*. IEEE, 372–381.

[60] Lizhe Wang, Samee U Khan, and Jai Dayal. 2012. Thermal aware workload placement with task-temperature profiles in a data center. *The Journal of Supercomputing* 61 (2012), 780–803.

[61] 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*. 216–230.

[62] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. 2013. Robustness in the Salus scalable block store. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 357–370.

[63] Zhuang Wang, Haibin Lin, Yibo Zhu, and TS Eugene Ng. 2023. Hi-Speed DNN Training with Espresso: Unleashing the Full Potential of Gradient Compression with Near-Optimal Usage Strategies. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 867–882.

[64] Allan Webber. 2014. Calculating useful lifetimes of embedded processors. *Application report, Texas Instruments* (2014), 1–6.

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

[66] Na Yang and Yun Wang. 2019. Identify silent data corruption vulnerable instructions using SVM. *IEEE Access* 7 (2019), 40210–40219.

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

[68] Yupu Zhang, Abhishek Rajimwale, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2010. End-to-end Data Integrity for File Systems: A ZFS Case Study.. In *FAST*. 29–42.
[69] Gefei Zuo, Jiacheng Ma, Andrew Quinn, and Baris Kasikci. 2022. Tolerate Silent Data Errors with Coded Computation. In *DISCC 2022 1ST WORKSHOP ON DATA INTEGRITY AND SECURE CLOUD COMPUTING*.

Received 10 February 2024; revised 4 August 2024; accepted 20 August 2024