Veritas – Demystifying Silent Data Corruptions: μArch-Level Modeling and Fleet Data of Modern x86 CPUs

Odysseas Chatzopoulos[†] Nikos Karystinos[†] George Papadimitriou[†]
Dimitris Gizopoulos[†] Harish D. Dixit[§] Sriram Sankar[§]

[†]University of Athens, Greece, {od.chatzopoulos | n.karystinos | georgepap | dgizop}@di.uoa.gr
[§]Meta Platforms Inc, Menlo Park, California, {hdd | sriramsankar}@meta.com

***Abstract*—Hyperscalers have reported unexpectedly high numbers of defective CPU chips, with a defect rate of 1 in a 1000, leading to Silent Data Corruptions (SDCs) in their computing fleets. However, there is no public data on the rate of SDC incidents (corrupted program executions) in large fleets, nor nor any detailed information on which CPU units, microarchitectures, or workloads are more likely to generate SDCs due to silicon defects. While CPU array structures have been studied for fault effects, arithmetic units like integer and floating-point units have not been thoroughly analyzed as potential root causes of SDCs. This paper addresses this critical gap by accurately modeling hardware faults in the arithmetic units of modern x86 CPUs and measuring the probability and rates of SDCs. Using a full-system gem5-based fault injector, the paper examines SDC trends across five recent x86 microarchitectures, various arithmetic units, and instruction classes. By integrating real-world defect rates from large-scale datacenter experiments with early-stage modeling and simulation, the paper provides critical insights into SDC incident rates across different systems. This information is essential for guiding hardware-based or software-based fault protection methods and is the paper's primary contribution to minimizing the impact of silent data corruptions in computing.**

I. Introduction

Hyperscalers (Meta [1], Google [2], Alibaba [3], [4]) have reported an unexpectedly large number of CPUs in their fleets that experience Silent Data Corruptions (SDCs) [5]–[7]; about one CPU per thousand. SDCs are program output corruptions due to defective silicon in CPU or other compute chips [8]–[10], and dramatically impact application correctness because they escape the existing error reporting or logging mechanisms at the hardware, system, or application layers. SDCs have a hard-to-measure cost in the community's trust in computing and a considerable impact on engineering and operational expenses in data centers and other large-scale computing domains. The problem has numerous dimensions: it depends on the number of CPUs and systems, their vendor ISAs (Instruction Set Architectures) and microarchitectures, the manufacturing technology, operating conditions, age, and workload profiles (control vs. data intensiveness, data types, etc.).

These disclosures strongly undermine the trustworthiness of modern computing [11]. It is even more critical that these SDCs originate from the core CPU silicon hardware. Although considerable attention has been directed towards assessing the impact of hardware faults within array components [12]–[22], there is a clear gap in assessing the full system effects of faults in other critical parts of CPU hardware: integer and floating-point scalar arithmetic units and their modern vector processing counterparts, which are all foundational pillars of computational efficiency in all types of computing chips.

Fault protection for arithmetic units has always been a challenge due to their importance in determining performance (clock rate). On the contrary, array structures (e.g. caches) can be effectively protected against hardware faults (mainly transients) through error coding techniques that usually have a small impact on performance. Fortunately, arithmetic hardware units are less susceptible to transient faults [23] than on-chip storage elements. However, transient faults in large-scale data centers do not typically result in large SDC rates [1]–[4]. The focal point is now on completely unprotected (or costly to protect) arithmetic hardware units in CPUs and other processing chips such as GPUs and ML/AI accelerators. Data collection from multi-year, large-scale testing on cloud data centers sheds light on the issue of SDCs and is very useful. Still, field data arrives late and brings high-level, coarse-grained information (the number of identified defective chips in a large population). A much needed game changer is a diligent *combination of field data and early prediction of the SDCs behaviors*. Such combination can deliver knowledge and insights about the *expected SDC rates across different μarchs, hardware units, workloads* and is an important contribution to design chips and systems with significantly reduced SDC rates at acceptable costs for mitigation.

Silicon defects in the field is a fact of life: manufacturing test escapes, variability, aging and wear-out contribute to such defects. Cost-effective chip design (from vendors) and system design (hyperscalers) against silicon defects in CPUs can be assisted by addressing the open research questions

TABLE I
Research Questions for SDCs due to Defective CPU Chips; This Paper Contributes to the Last Four.

Research Question	**Answered ?**
Percentage of SDC-generating CPUs	Partially [1]–[4]
SDC incidents rate	*Open*
Rank SDC-prone μarches	*Open*
Rank SDC-prone arithmetic structures	*Open* (Hints in [3], [4])
Rank SDC-prone instruction classes	*Open*

that Table I shows. Such information will eventually guide vendors (architects, designers), researchers, and hyperscalers in devising system-level fault protection methods and minimizing the impact of SDC incidents in computing at scale. No absolute numbers are needed at early stages, *only* relative trends among the different options suffice: *which microarchitecture, which hardware units, which instruction classes can minimize the SDC rates?*

In the pursuit of unveiling the truth behind CPU-generated SDCs, we propose a very large scale experiment involving the ability to resemble (through fault injection in gem5 [24], [25]) *hundreds of thousands of defective CPUs* with different microarchitectures. The modeled CPUs, each containing different silicon defects across dozens of hardware structures, will be subjected to diverse software workloads until the end of execution to measure SDC probabilities through fast fault injection. We complement this experiment with the results of another one on real systems: defective parts per million (DPPM) data from Meta's large-scale production fleet. Fleet defectivity data were collected through multi-year fleet testing experiments. The insights unveiled from this endeavor are below:

1) Measurement of the relative SDC incident rates across CPUs of different microarchitectures employed in a large simulated fleet of machines utilizing real field data on actual chip defectivity.
2) Identification of the arithmetic hardware units that are the main contributors to the total SDC rates.[1]
3) Measurement of the relative SDC rates of workloads classes with different instruction mixes.

The two extensive experiments have been carried out independently (to not bias the results of one towards the specifics of the other) and the primary goal is to complement each other. The modeling and simulation experiment obtains *fine-grained* information on how faults injected in the hardware units manifest at the program output as SDCs and translates the collected probabilities into SDC rates using information about the CPU clock rates, their instruction throughput, the size of the hardware units, and the execution times of workloads. This is impossible in the real system experiment alone. On the other hand, data from extensive testing of a fleet of real systems at Meta provide *true defectivity* information from a considerable number of CPU chips, which is only possible with such an experiment. To facilitate a comprehensive evaluation, we leverage five contemporary x86 CPU designs, from different vendors and introduction years, representative of the prevailing architecture in cloud computing today. The same CPUs are employed in both the experiments: in the data center fleet, and the models at the simulation framework.

The importance of combining the findings of the two deliberately independent experiments is fundamental. The fleet experiment (on real CPU chips) operates blindly, sifting through a mix of functional and flawed chips, and identifies a handful of faulty chips amidst thousands without details on the faulty hardware units or affected instruction classes. Still, it is more than 3 orders of magnitude faster than microarchitectural simulation. The fleet experiment executes on CPUs operating at frequencies between 1.6GHz to 3GHz, while simulation can achieve a rate of 600K instructions per second. Simulation on gem5 (cycle-level operating at the microarchitecture level) focuses solely on analyzing the behavior of faulty chips (modeled through injections), making it 1,000 times more efficient than the fleet in terms of defectivity analysis throughput. Therefore, simulation in gem5 brings its fine-grain analysis capabilities: faults are statistically injected in any hardware unit of the CPU (while on actual chips the location of the fault is unknown) and the effects are observed at the output of the program execution.

The combination of the two experiments can be performed with different knobs to evaluate their effect on SDC manifestations. For the demonstration of our approach, we employ the following "instruments":

- Five modern x86 CPU designs. Similar family CPUs are modeled in the simulation-based experiment (using publicly available information) and are employed in the actual fleet (subject to intake and decommissioning variances).
- Millions of fault injections in all arithmetic units that generate SDCs: integer and floating-point scalar units, integer and floating-point vector units.
- Hundreds of thousands of CPUs in a cloud production fleet analyzed for 6 calendar years and billions of CPU hours.
- Permanent fault model (stuck-at) so we provide a *pessimistic upper bound* of the SDC rates (many defects in silicon have a less persistent behavior than the stuck-at fault model).
- A diverse set of workloads to exercise the hardware units during fault injection.
- A rich portfolio of synthetic x86 instruction viruses (stress programs) for the target arithmetic structures to test the actual CPUs in the hyperscaler's fleet.

The outcome of the twin effort to combine the results of the two large scale experiments is the ranking in terms of relative SDC generation rates of the: 1) *five x86 μarchs*, 2) *seventeen major arithmetic hardware units*, 3) *different x86 instruction classes*. In summary the main novel contributions of the paper are enumerated below:

1) For the first time, we enable fast gate-level fault injection in CPU arithmetic units of all types in gem5, with virtually no loss in simulation throughput. Previous works focused on transient faults in arrays.
2) We investigate the impact of such gate-level faults on program execution by running workloads to the end in a full-system simulation and measure SDC probabilities and rates.
3) We conduct large-scale fleet experiments to calculate defectivity data and SDC rates, using custom-designed targeted stress programs.

[1]Alibaba [3], [4] identified, at a very high level, the arithmetic units of CPUs as the primary suspects. We delve into much finer granularity.

4) We combine the fleet data with the simulation results to derive trends and insights across μarchs, hardware units, and instruction classes.

II. Background & Motivation

A. Background: Defects, Faults, Errors

A *fault* in a CPU represents a physical issue (a defect) causing a discrepancy between expected and actual performance, with faults manifesting as *errors*. High-energy particles cause transient faults, while silicon defects lead to intermittent or permanent faults. If a faulty hardware resource is never accessed during program execution, the fault is benign; otherwise, it may lead to Silent Data Corruption (SDC), crashes, or error notifications. SDCs are particularly problematic because they compromise software accuracy and data integrity without warning [26]. Traditionally, CPU Reliability, Availability, and Serviceability (RAS) design has focused on mitigating particle-induced faults, or "soft errors" [27]. However, recent SDC reports from Meta, Google, and Alibaba suggest new fault causes, such as marginal defects that arise under specific conditions like temperature or workload patterns, leading to intermittent faults [26], [28]–[30]. Additionally, device degradation and lifetime reliability are significant in modern processors, causing both intermittent and permanent faults, necessitating a deep understanding of various physical fault sources and their implications [1], [2], [26].

B. Silent Data Corruptions at scale

Silent errors within hardware devices occur when defects manifest in parts of the circuit (in our case a CPU chip), that are not equipped with any checking logic to detect incorrect operations. These errors can range from flipping a single bit of data to executing wrong instructions, impacting large-scale infrastructure services. SDCs escape error reporting and hardware fault tolerance, and they are untraceable at the hardware level but severely affect the application-level, potentially resulting in data loss identified only after lengthy and costly debugging. Various defect types in silicon manufacturing contribute to SDCs, necessitating methodologies and debug flows to identify faulty instructions. Mitigations involve modifying the hardware and implementing detection/testing methodologies across the CPU fleet [31].

To address this, Meta employs two publicly documented approaches: Fleetscanner for out-of-production testing and Ripple for in-production testing [1], [31]. Continual evaluation of infrastructure tradeoffs over six years of production experience optimizes these methods. Extensive testing across hundreds of thousands of machines has detected hundreds of devices that generate SDCs, indicating systemic issues across generations. Monitoring SDCs in the fleet over six years underscores the need for early-stage analysis (before the actual chip design of the next processor generations), hardware resiliency, production detection mechanisms, and fault-tolerant software mechanisms. Mitigating SDCs demands innovation in silicon design, verification, testing, fault modeling, instruction resilience, and software architectures. Therefore, SDC rate evaluations at the early design stages for the most vulnerable hardware units are vital for reliable next-generation chips, which will equip modern fleets of cloud and data center providers [1], [31].

C. Microarchitecture-level SDC analysis: hardware structures

Microarchitecture-level analysis of the effects of hardware faults (transient or of a more persistent nature) offers a good tradeoff between hardware model fidelity and simulation performance that no other abstraction layer can deliver: it allows *fast simulation* (thus fault injection experiments of high statistical significance) for *long workloads until the end* (a mandatory requirement to check if the final output is corrupted) while maintaining a *reasonable modeling accuracy of the underlying hardware*. The specific needs for a particular microarchitecture-level analysis depend on: (a) the hardware units of interest, (b) the fault models and effects under consideration. Table II presents the different classes of CPU hardware units and their properties in the context of the problem of Silent Data Corruptions which is our focus.

Faults in any hardware structure can potentially lead to a silently corrupted output. However, the likelihood of SDC incidents strongly depends on (a) the operation/role of the hardware unit in instruction execution, and (b) the hardware protection scheme it encapsulates. We refer below to "faults" in general but the main focus is on faults with a more persistent nature than transients, i.e., permanent or intermittent faults, since it is commonly agreed that the SDC problem from CPUs is not due to transient faults [1]–[4], [32]. In particular:

- *Instruction caches* – a workload running with a persistent fault in any level of cache that stores instructions is very unlikely to survive to the end and produce a corrupted output. Faults in most fields of an instruction will primarily impact the execution flow or the instruction operands, and thus, lead to a crash [12]. Besides, cache memories are usually equipped with a cost-effective ECC-based protection scheme: parity or SECDED [33]. Therefore, even the few cases of faults in (typically small) parts of an instruction that can affect the data flow (an immediate field) will be corrected and are very unlikely to lead to data corruption.
- *Data caches* – a workload running with a persistent fault in a cache level that stores data of any type may produce a Crash (if data determine the control flow), an SDC (if only the data flow is affected) or the fault may be masked. Besides, the same protection schemes applied

TABLE II
Criticality for SDCs, Usual Protection Provisions. Units in Bold are Likely to Generate SDCs and are Weakly Protected.

Hardware unit class	SDC Likelihood	Protection
Instruction caches	Small	Light to Strong
Data caches	Medium	Light to Strong
Registers, buffers, queues	Small	Weak to Light
Control logic	Small	None to Weak
Integer scalar arithmetic	Medium	None to Weak
Floating point scalar arithmetic	Large	None to Weak
Integer vector arithmetic	Large	None to Weak
Floating point vector arithmetic	Large	None to Weak

in instruction caches may significantly reduce (or even eliminate) the SDC generation likelihood at some cost.

- *Registers, buffers, queues* – although these hardware structures are usually unprotected (due to the impact of any protection on their latency) persistent faults in them are very unlikely to lead to an execution that "survives" to the end and produces a wrong output. Execution crashes or hangs are the naturally expected result in most cases.
- *Control logic* – persistent faults in the (usually unprotected) control logic of a CPU (instruction decoding, scheduling, etc.) are very unlikely to lead to an SDC. They usually severely affect the control flow and lead to a crash.
- *Integer scalar arithmetic* – persistent faults in the integer ALUs of a CPU have a mixed impact on both the control flow (conditions and pointers calculations) and the data flow. Thus, they have some probability of contributing to SDC incidents since they are hardware units that encounter the highest frequency of usage during program execution.
- *Floating-point scalar arithmetic* – unlike their integer counterparts, floating-point arithmetic units are almost exclusively employed in data flow thus are intuitively expected to be large contributors to SDC incidents.
- *Integer vector arithmetic* – unlike its scalar counterpart, the integer vector arithmetic unit is minimally employed in control flow and has a considerable silicon footprint. Thus, it is a suspected large contributor to SDC incidents.
- *Floating-point vector arithmetic* – along the same lines as its sibling integer vector units, they occupy a large silicon portion and participate only in the data flow. Another suspected large contributor to SDC incidents.

The last four cases and rows in Table II (integer and floating-point, scalar and vector structures) are (a) usually *unprotected* in general-purpose CPUs and (b) the effect of faults in them towards the generation of SDCs *has never been analyzed with microarchitecture-level simulation/injection*. In this paper, *we complete the landscape of μarch level analysis of hardware faults for SDC estimations*. We enhance the state-of-the-art μarch simulator gem5 with detailed (gate-level) fault injection capabilities in the key missing components of the CPU: integer and floating-point (both scalar and vector) arithmetic structures. Through permanent faults (which we employ as an upper bound of defect severity) injections at gate-level models embedded in fast microarchitectural simulators (gem5 in our setup), we maintain the main benefit of microarchitecture-level analysis (unlike past approaches which add very large overheads [34]) - its speed which allows long workloads execution to the end (which is mandatory for SDC measurements) - while further improving the fidelity of the hardware modeling.

III. Methodology

We first present in detail how we leverage the gem5 microarchitecture-level simulator to simulate hundreds of thousands of defective chips with faults in the functional arithmetic units of the CPU, observing their effects on actual workloads running end-to-end on top of a Linux operating system. Subsequently, we summarize how we obtain results from a real large-scale data center across several years of monitoring and testing. We then combine the findings from the two large scale experiments.

A. Functional units statistical fault injection in gem5

gem5 [24], [25], [35] is a versatile and widely utilized open-source simulator in the realm of computer architecture research. Its modular framework enables the simulation of various processor designs, from simple in-order cores to complex superscalar out-of-order (OoO) microarchitectures.

In this work, we leverage the gem5 simulator to model five modern x86 out-of-order (OoO) microarchitectures summarized in Table III. While gem5 offers a configurable OoO model for the simulation of the specified x86 chips, its treatment of functional units is rudimentary. Arithmetic unit operations are implemented at a functional level; i.e., the unit's internal operation is not simulated, thereby ignoring all structural information and intrinsic operation details [36].

In contrast to arithmetic units, array components which have been the focus of almost all prior research [12]–[19] are much more faithfully modeled in gem5. Injection is also much easier as bit flips in these arrays are well understood (both their physical causes and incidence rate) and are directly translatable to the gem5 implementation. To conduct Statistical Fault Injection (SFI) on the logic gates of arithmetic units and observe their impact on workload execution, we undertake the following steps:

1) Generate gate-level models of functional units in C++. These models are automatically generated in C++ for easy integration into the gem5 code base.
2) Instrument the C++ gate-level models to enable fault injection on any gate of the modeled functional unit. For the demonstration of the framework we inject permanent

TABLE III
Specifications of the Microarchitectures Examined in This Paper.

	CPU A (year 2014)	**CPU B (year 2020)**	**CPU C (year 2020)**	**CPU D (year 2019)**	**CPU E (year 2015)**
L1 Data/Instruction Cache	32KB 8-way / 32KB 8-way				
L2 Cache	256 KB 8-way	512 KB 8-way	1 MB 16-way	512 KB 8-way	1 MB 16-way
Physical Register File	168 Int; 168 FP	192 Int; 160 FP	180 Int; 168 FP	180 Int; 160 FP	168 Int; 168 FP
LQ/SQ/IQ/ROB Entries	72/42/64/192	44/48/148/256	72/56/97/224	44/48/128/224	72/56/97/224
Scalar Int FUs	4 Add; 1 Mul	5 Add; 1 Mul	4 Add; 1 Mul	4 Add; 1 Mul	4 Add; 1 Mul
Scalar FP FUs	1 Add; 1 Mul	2 Add; 2 Mul	1 Add; 1 Mul	1 Add; 1 Mul	1 Add; 1 Mul
Vector Int FUs	2 Add; 1 Mul	4 Add; 2 Mul	3 Add; 2 Mul	3 Add; 1 Mul	3 Add; 2 Mul
Vector FP FUs	1 Add; 2 Mul	2 Add; 2 Mul	2 Add; 2 Mul	2 Add; 2 Mul	2 Add; 2 Mul

(stuck-at) faults, but the models can be, obviously, instrumented for other fault types.

3) Integrate these gate-level models into gem5 using a smart hybrid approach aimed at *minimizing simulation throughput loss* (which is less than 4 percent even for the most complex units).
4) Create a custom fault injection controller within gem5 to handle fault injection.
5) Utilize a highly optimized fault injection campaign manager designed to execute multiple parallel fault injections, leveraging the full processing speed of the host machine. Parallel fault injection and subsequent optimizations, which will be elaborated on later, significantly enhance the simulation throughput of our fault injection campaigns.

Fig. 1 shows the gem5 OoO CPU model we employ and the hardware units (in color) that we enhance with gate-level structural detail for fault injection. In the following subsections, we explain in detail how we implement the aforementioned steps.

B. Software gate level models of functional units

Functional units, unlike on-chip storage arrays which have been the focus of numerous previous works on microarchitecture-level fault injection, are only functionally simulated in gem5. For instance, the integer adder is implemented simply as a single C++ statement: `result = PSrcReg1 + PSrcReg2;`[2]. To model actual physical faults that can occur in functional units, we need to incorporate their gate-level structure in a minimally intrusive manner in gem5. There are several approaches to address this issue, such as utilizing external commercial gate-level simulators to model the hardware of functional units, either offline (using program traces) or online (halting gem5 simulation every time we need to perform an operation passing through the target functional unit) [34]. However, these techniques severely hinder simulation throughput.

Our approach involves employing software-specified gate-level models of functional units that can be *directly integrated* in gem5 to minimally affect simulation throughput. While gate-level models of functional units could be handwritten, we opt to utilize a highly customizable arithmetic circuit generator called ArithsGen [37] (other similar tools can be used). ArithsGen can produce various arithmetic circuits, including multiple implementations of adders, multipliers, and dividers. While ArithsGen can produce both Verilog and BLIF, we utilize the C++ software models it generates. A snippet from the generated C++ function representing a 64-bit carry-lookahead adder with 4-bit carry-lookahead blocks is shown in Fig. 2. Each line represents a single gate of the adder.

C. Gate level model instrumentation for fault injection

To perform fault injection on the gates of the arithmetic unit models described in the preceding subsection, we need to instrument the generated C++ code to enable the manipulation of gate outputs, thus emulating physical gate-level faults. This is accomplished through a custom Python module that parses the generated code, identifies the software statements representing the circuit gates, and supplements the requisite "wrapper code" to simulate the desired physical fault model. The code snippet of Fig. 3 illustrates the first three gates from the same section of the carry-lookahead adder, as presented in the previous subsection, now instrumented for fault injection using the stuck-at-1 fault model. The parameter that specifies the faulty gate is passed to the `u_cla64_stuck1` function via the injection controller. The Python instrumentation module is highly configurable and can model different underlying faults just by specifying a custom code pass.

D. Integration of software gate level FU models in gem5

Once the gate-level software models of functional units and generated and equipped for fault injection, the subsequent step involves the integration of these models in gem5. This integration occurs at the micro-operation (micro-op) level, since x86 macro-ops are broken into smaller RISC-like operations to facilitate more efficient hardware implementation. gem5's out-of-order CPU model executes these micro-ops within the Issue-Execute-Writeback (IEW) stage. This execution is facilitated by a singular "execute function" automatically generated by a gem5-specific functional description of the micro-op.

We interface with these "execute functions," transmitting all pertinent micro-op data, including inputs and expected correct outputs, to the injection controller. The injection controller then verifies whether the micro-op is scheduled for execution at the target functional unit. If so, it activates the instrumented gate-level model, supplying it with the micro-op's input data

[2]PSrcReg1 and PSrcReg2 are variables representing the input registers of the instruction.

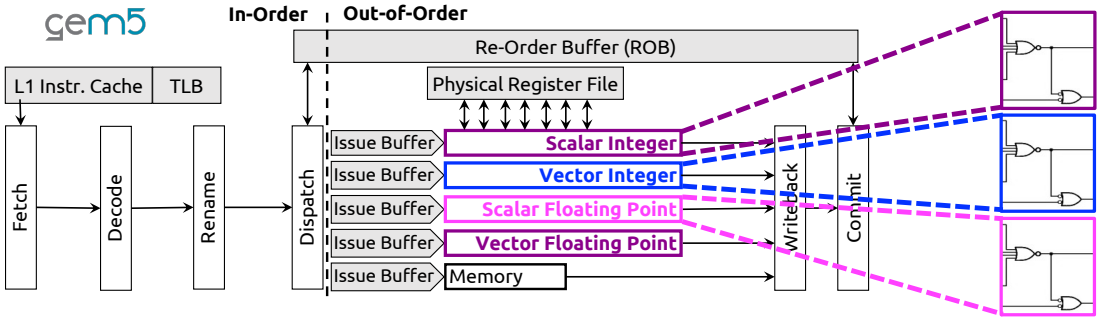

Fig. 1. Gate-level modeling infrastructure for functional units on gem5 simulator.

```
uint64_t u_cla64(uint64_t a, uint64_t b)
...
pg_logic25_or0 = ((a >> 25) & 0x01) | ((b >> 25) & 0x01);
pg_logic25_and0 = ((a >> 25) & 0x01) & ((b >> 25) & 0x01);
pg_logic25_xor0 = ((a >> 25) & 0x01) ^ ((b >> 25) & 0x01);
xor25 = ((pg_logic25_xor0 >> 0) & 0x01) ^ ((or56 >> 0) & 0x01);
and113 = ((or55 >> 0) & 0x01) & ((pg_logic25_or0 >> 0) & 0x01);
and114 = ((and113 >> 0) & 0x01) & ((pg_logic24_or0 >> 0) & 0x01);
...
```

Fig. 2. Code snippet from C++ gate-level model of a 64-bit carry-lookahead adder. Each line represents a single gate of the actual circuit.

and selecting a gate for injection. The resultant output from the simulated faulty functional unit is then compared with the anticipated correct result for logging purposes. Subsequently, the calculated value is returned and utilized by the micro-op.

By selectively invoking the gate-level model only when necessary and integrating it tightly into the gem5 code base (no invocation of external tools), we manage to minimize throughput loss compared to running the vanilla version of gem5. In Fig. 4 we can see the slowdown that the integration of our gate-level models incurs. The slowdown ranges from 0% to 4% and is proportional to the complexity (gate-count) of the modeled unit. In [34] the use of an external gate-level simulator to model stuck-at faults in a simple integer ALU incurs a performance penalty reaching **220%**.

E. Injection campaign workflow

The injection campaign manager is a Python module that automates the process of running fault injection campaigns targeting the functional units of a given microarchitecture. To configure the injection campaign manager we feed it with the target microarchitecture, a list of benchmarks and parameters specifying the number of faults to be injected as well as the underlying fault model. The campaign manager then automatically parses information about the number and type of functional units of the microarchitecture and initiates injection campaigns on these functional units for the specified workloads. To avoid injecting faults to inactive units, during the golden run (single fault-free execution of the workload) stats are collected including functional unit activity (i.e., the number of times the functional unit is used during workload execution). If no activity is observed on a specific functional unit then no injections are carried out and all faults that would have been injected are considered to be masked as faults present in the target unit could never propagate to program output.

An overview of the complete fault injection flow is shown in Fig. 5. The injection campaign configuration ❶ is fed into the injection campaign manager module that generates a fault list ❷ based on the number of faults to be injected and the requested fault model. Subsequently, parallel gem5 simulation

```
uint64_t u_cla64_stuck1(uint64_t a, uint64_t b, uint32_t faulty_gate)
...
pg_logic25_or0 = (faulty_gate == 269)  ?
  1 : ((a >> 25) & 0x01) | ((b >> 25) & 0x01);
pg_logic25_and0 = (faulty_gate == 270) ?
  1 : ((a >> 25) & 0x01) & ((b >> 25) & 0x01);
pg_logic25_xor0 = (faulty_gate == 271) ?
  1 : ((a >> 25) & 0x01) ^ ((b >> 25) & 0x01);
...
```

Fig. 3. Code snippet from C++ gate-level model of a 64-bit carry-lookahead adder now instrumented for gate-level fault injection. Here a stuck-at-1 fault is modeled.

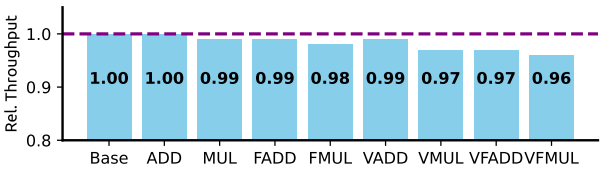

Fig. 4. Relative throughput of gem5 with gate-level models enabled for each functional unit compared to the base version of the gem5 simulator.

instances are spawned ❸ each injecting a fault to a randomly selected gate. When a run is completed, the results of the run are recorded ❹. After all runs finish execution, their results are compared with the golden output to determine the outcome of the program (the fault can either result in an SDC, a Crash, or may be Masked at the software or hardware level). Our focus in this paper is on SDCs since Crashes are by definition detected, allowing for swift identification and replacement of defective chips. In Meta's fleet environment, system crashes occur at least 2 to 3 times more frequently than SDCs.

After the parsing step ❺ is finished, we are provided with P_{SDC} for every examined workload ❽ i.e the probability that a fault in the target functional unit will result in an SDC as well as the error profile ❻ (i.e., how many bits of the functional unit output and at what positions were affected by the injected fault) and the instruction error rates ❼ that the fault resulted to. For the calculation of the SDC rates (number of executions with corrupted output over time) ❾ we use the following equations:

$$SDC_R(u,w,t) = \frac{t}{T_w} \cdot P_{SDC}(u,w) \cdot P_{Faulty}(u) \quad (1)$$

$$SDC_R(u,W,t) = \sum_{w \in W} SDC_R(u,w,\frac{t}{|W|}) \quad (2)$$

$$Core_R(FU,W,t) = \sum_{u \in FU} SDC_R(u,W,t) \quad (3)$$

$$Fleet_R(CPU,c,W,t) = \lambda_{bad} \cdot c \cdot Core_R(FU_{CPU},W,t) \quad (4)$$

- Equation 1 gives the SDC rate stemming from faults in a functional unit u, while the core runs workload w for t seconds. A single run of the workload takes T_w seconds. $P_{Faulty}(u)$ is the probability that unit u is faulty given that the core is defective.
- Equation 2 gives the SDC rate stemming from a faults in functional unit u, while the core runs a set of workloads W round-robin for t seconds in total, $t/|W|$ each.
- Equation 3 gives a defective core's SDC rate including all its functional units FU while the core runs a set of workloads W for t seconds, summing up the SDC rates stemming from every functional unit of the core.
- Equation 4 gives the total SDC rate for a computing fleet consisting of identical cores (CPU) running a set of workloads W for a total time t given the ratio of defective to total cores is λ_{bad} and the total number of cores is c.

F. Modeling assumptions

For this study, we have made the following decisions regarding fault modeling and data presentation:

1) We model permanent (stuck-at) faults at the gate level. While faults in real machines may be less severe (e.g.,

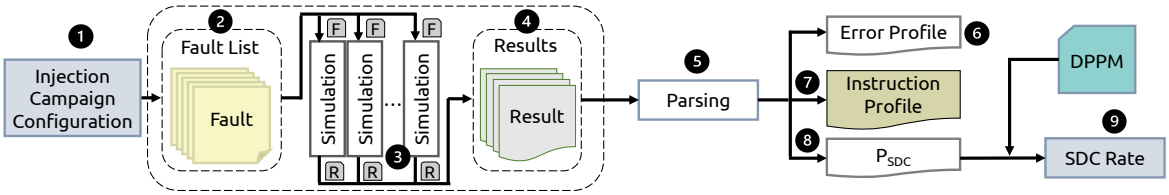

Fig. 5. Fault injection flow for arithmetic units and the final SDC rate for the target unit.

intermittent faults due to marginal defects and small delay defects), we use permanent faults as a *hard upper bound* for the SDC rates we measure. As discussed previously, our infrastructure is highly modular and can be used to model a multitude of fault or error models.

2) We assume that $P_{\text{Faulty}}(u)$ in Equation 1 is determined by the relative area of functional unit u, based on the idea that every gate of the functional units of the core has an equal probability of being affected by a stuck-at (or other type of) fault. While this assumption may not hold for aging-related effects, we avoid using such a model in this study due to the lack of statistics provided by silicon vendors and foundries. Any statistical distribution (including aging-related distribution or delay defects) can be integrated into our infrastructure for the same evaluation of SDC rates.
3) Due to lack of knowledge about $P_{\text{Faulty}}(u)$ and the confidentiality of the real DPPM of chips at scale, we only present relative SDC rates. These results highlight the differences among microarchitectures, hardware units, and instruction classes, abstracting away the unknown parameters of Equations 1- 4. Of course, when the probability of defects in certain hardware units and the chips DPPM numbers are known (by a CPU vendor), absolute SDC rates values can be produced using our modeling and framework.

G. Experimental methodology at scale

Our fleet-wide approach to measuring silent data corruptions (SDCs) at Meta is multifaceted. We utilize a fleet-scale monitoring system to track production workloads and system aspects like utilization, fault occurrence, and fault remediation [38]. This allows us to quantify SDC rates by analyzing fault occurrence and reproduction rates across various configurations, vendors, and software. Our telemetry has been active for *over 6 years*, enabling us to derive fleet-wide SDC rates and understand their prevalence in modern computing environments.

To safeguard the fleet, we employ both in-production and out-of-production testing frameworks with variable runtimes to meet different test objectives. These frameworks include specifically crafted reproducers and bit-level randomization variances across variety of functional blocks to maximize testing effectiveness. Our SDC data is derived from both testing environments and production instances, allowing us to identify potential weaknesses in computational design as well as vulnerabilities specific to particular workloads. This comprehensive approach ensures that we capture a wide range of fault scenarios that may occur in real-world applications. For the data presented in Fig. 8 and Table V, we utilize a series of custom-made tests specifically designed at Meta to stress the arithmetic units of x86 CPUs by ”hammering” them with randomized sequences of input data. These tests are engineered to provoke fault scenarios that might otherwise be difficult to observe under standard operating conditions and workloads.

In addition to these efforts, we conduct an in-depth analysis of the logged data for chips exhibiting SDCs, enabling us to establish links between observed SDC events and specific hardware components. By identifying recurring patterns and pinpointing the root causes of these faults, we can provide targeted recommendations for mitigating risks in future CPU architectures. This holistic approach at Meta’s large-scale fleet not only guides future design strategies but also contributes to enhancing the overall resilience of the systems under test. In Table IV, a high-level summary of the modes of execution of directed reproducers and testing mechanisms is captured, along with the aspects associated with test execution runtime, functional block targets as well as co-location aspects with workloads. Specifically, in this table, fleet coverage cadence (Property #2) refers to how frequently the entire fleet of machines is scanned using the testing framework. For example, if the cadence is set to 60 days, the tests are re-executed on the same set of machines after that period. Each scan aims to identify defects within specific functional blocks types (Property #4), also known as defect families. The testing assumes there is one defect per chip (Property #3), and once a defect is identified in a device, that device is taken out of the fleet, defining the coverage approach. It is essential to note that, among all the available CPU architectures in the fleet (see Property #5), we selected to model a subset of five of them (see Table III) because they have the largest sample sizes, allowing for more reliable analysis and robust conclusions.

We present relative defect rates across x86 microarchitectures and vulnerable instruction families in Sec. IV, based on extensive testing and production observations across hundreds of thousands of CPUs and billions of hours over several years. We address population variations by capturing the latest snapshot of each CPU fault combination and supplementing it with historical data to ensure representative results. This comprehensive approach, involving detailed examinations and statistical analyses, provides a deep understanding of SDC factors, laying the foundation for improved protection mechanisms and system reliability.

We share the relative DPPMs for the CPU architectures and the modeled execution units in discussion across architectures. A device exhibiting corruption once during testing or production execution is considered faulty. These relative DPPMs are used in conjunction with simulation data to arrive at the relative

TABLE IV
FLEET TEST EXECUTION PARAMETERS.

Property	Fleetscanner [39]	Ripple [39]
#1 Test execution runtimes	Order of minutes	Order of milliseconds
#2 Fleet coverage cadence	≈60 days	≈7 days
#3 Percentage of detected defects	≈93%	≈77%
#4 Target number of defect families and functional blocks	≈30	≈5
#5 Number of unique architectures and CPU generations faults detected on	≈10	≈7
#6 Execution Mode	Non-production and non-workload states	Co-located with production workloads

SDC rates presented in Sec. IV (see Figure 10).

IV. RESULTS

We employ a diverse set of 30 workloads for the simulation experiment (which are also very common in the fleet experiment), mainly focusing on data-intensive applications which should better highlight data corruption effects from arithmetic hardware units. We group the workloads into 5 higher-level categories to smooth out single workload anomalies while also making the figures more comprehensible. The 5 non-overlapping categories are the following:

- *Searching & Sorting*: Common sorting and searching algorithms, such as qsort, dijkstra, trie searching, etc. These applications are data-intensive, but also rely on complex control flow.
- *Compression & Hashing*: Standard compression and hashing algorithms, such as zstd, zlib, sha512, crc32, md5sum, etc. These workloads are ubiquitous in modern computing.
- *Image Processing*: Standard image processing algorithms, such as jpeg (compression & decompression), edge detection, smoothing, etc.
- *Floating-Point Heavy*: Linear algebra workloads and numerical algorithms operating on floating-point data. The linear algebra workloads mainly utilize the Eigen library and include: matrix multiplication, singular value decomposition and sparse linear system solving. Numerical algorithms include FFT, polynomial solving, sqrt approximation.
- *Integer Heavy*: Workloads from the previous category operating mainly on integer data types (where possible).

The high dimensionality of the experimental data (five microarchitectures, 12 to 17 FUs[3], 30 workloads) resulted in ≈67,500,000 injection runs corresponding to simulating the workloads on ≈225,000 faulty CPU chips. Specifically, for every functional unit of every μArch we perform 3,000 injections (1,500 gate stuck-at-0, 1,500 gate stuck-at-1 faults) [4] running all 30 workloads on the simulated, faulty chip.

A. SDC rates: relative trends among microarchitectures, hardware units and workload categories

As mentioned in Section III our key metric (and important contribution) is the *SDC rate*, i.e. the number of recorded program runs with corrupted output (SDC) over time. To be able to compare the behavior of different workload categories, microarchitectures, and functional units in Fig. 6 we show the relative SDC rates of each functional unit (x-axes), for each microarchitecture (y-axes) in 5 heatmaps; one for each workload category. The SDC rates are calculated using Equation 2 with arbitrary runtime $t = T$. The results we present are *relative across the five heatmaps* and thus T cancels out. SDC rates are relative to the minimum SDC rate given a value of 1. A cell having a value of 100 means that the corresponding functional unit and microarchitecture running the workloads of the respective category gives 100 times more SDCs per unit of time compared to the reference rate.

In Fig. 6 we observe that the reference value for SDC rates (i.e., the smallest absolute rate) occurs in the integer adders for the *Search and Sort* category. Adders are very frequently used to calculate addresses, loop control variables and determine control flow. Corruptions of adder outputs in *Search and Sort* programs are highly likely to result in an application crash due to the relatively complex memory access patterns and control logic of these applications. Compared to the integer adders, the integer multiplier causes much higher SDC rates (2 orders of magnitude). Multiplier results are far less often used for address calculations or control flow decisions and are more likely to propagate to program output. Note that since the workloads of this category do not utilize FP or vector components, they are not shown as the absolute SDC rate stemming from faults in these components is zero.

Observation #1: Scalar integer adders are the least likely components to cause an SDC. The corrupted values are highly likely to affect the control flow or cause illegal memory accesses resulting in crashes.

The results shown in the *Compression and Hashing* heatmap are comparable to the *Search and Sort* workloads for the scalar integer adders and multiplier. Here, though, the integer adders give out almost 30 times the SDC rate of the latter category, since compression and hashing rely on minimal control flow and run through the data more uniformly. Moreover, intermediate corrupted computations are more likely to affect the program output. The integer multiplier shows an upward trend as well. *Compression and Hashing* workloads also utilize the vector integer adder for more efficient computation, exhibiting high SDC rates for these components, which primarily contribute to the data flow.

When it comes to *Image Processing* workloads, there is a lot more diversity in functional unit utilization. The entire set of FP and integer scalar and vector units have significant

[3]The number of functional units ranges from 12 (CPU A) to 17 (CPU B).

[4]This ensures a confidence interval of 99% with less than 2% error margin [40] when it comes to the estimated SDC probability due to the simulated defect.

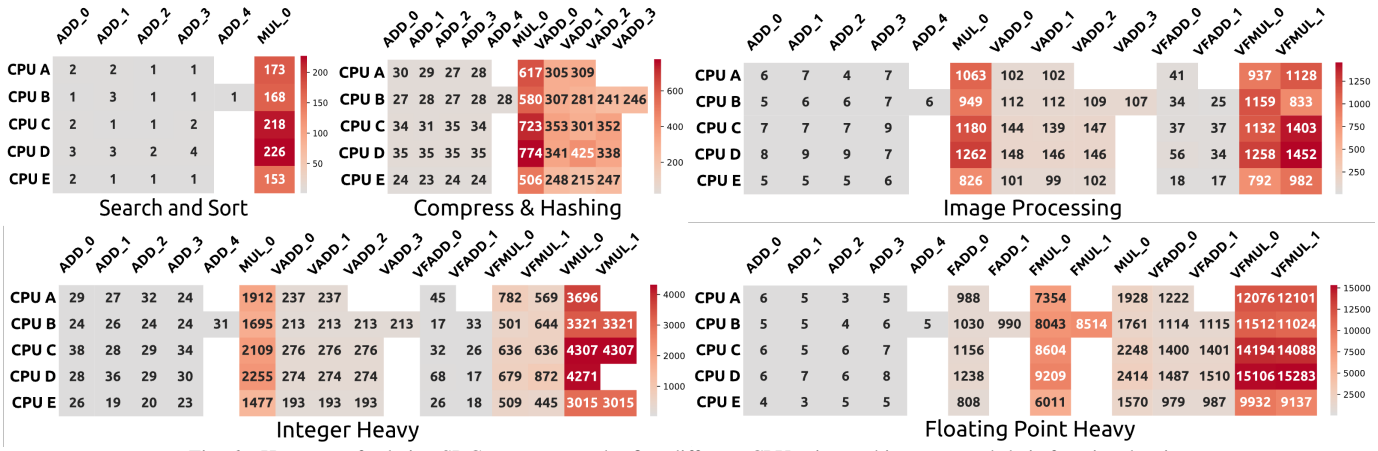

Fig. 6. Heatmap of relative SDC rates across the five different CPU microarchitectures and their functional units.

activity. Integer adders and the integer multiplier follow a familiar pattern, with an even larger SDC rate for the integer multiplier. Among the vector units, the vector floating-point multipliers show the highest vulnerability, with integer vector adders following them. One significant reason for the large difference in SDC rates between adders and multipliers (be it scalar or vector) is their relative gate counts, given our assumption that the probability of a gate being faulty is uniform across all gates. Multipliers are on average 30 times larger than adders, giving a multiplier a 30x higher chance to contain the faulty gate compared to the adder. Comparing the scalar to the vector adders, we observe an order-of-magnitude difference in SDC rates. Vector adders are in principle not used to calculate addresses nor are they involved in control flow decisions. Thus, their erroneous computations are much less likely to cause crashes and much more likely to propagate to program output and corrupt it.

The *Integer Heavy* workloads are the only category using the integer vector multiply units, producing a very high SDC rate. Moreover, within the integer-heavy workloads, the integer adders exhibit similar SDC rates to *Compress & Hashing* workloads while vector integer adders exhibit the highest SDC rates compared to the other five workload categories.

Observation #2: Vector units (integer and FP) cause several orders of magnitude more SDCs. Their results propagate to program output with high probability.

Finally, regarding the *Floating-Point Heavy* workloads, the SDC rates of all FP units skyrocket. The workloads of this category (linear algebra & numerical algorithms) are extremely sensitive to corrupted intermediate results, while these corruptions are very unlikely to cause crashes. For example, during matrix multiplication, one corrupted multiplication result is almost certain to influence the result array and never causes a crash. The result only participates in the data flow of the program. Another point to be drawn out is the following: One might intuitively expect that having 2 units instead of 1 (e.g., 2 FADD & FMUL units in CPU B compared to the other CPUs which have 1) would result in half the SDC rate stemming from each of those components. This is, however, not happening in our experiments: the rates might even be higher in some cases. Even when the operations are split evenly among the two units, there is ample opportunity (due to the permanent nature of the faults) for half of the operations to produce enough erroneous calculation results which propagate to program output, leading to an SDC.

B. *Case study: integer multiplier sensitivity analysis*

Our functional unit modeling methodology is inherently flexible, enabling us to model various gate-level implementations of the functional units we support. To demonstrate this feature, we present data from three different fault injection campaigns on the same underlying microarchitecture, employing three *different gate-level implementations* of the integer multiplier: the Array Multiplier, Wallace Tree Multiplier, and Dadda Multiplier [41], all widely employed in modern CPU designs in different variants.

Figure 7 shows the results of the experiment. We observe only minor variations in the SDC probabilities among the three different multipliers. Despite the design differences among the Array Multiplier, Wallace Tree Multiplier, and Dadda Multiplier, the SDC probabilities when workloads run remain very close to each other. These findings suggest that the choice of multiplier implementation *does not significantly impact the overall reliability of the microarchitecture in the context of SDCs*. This insight is valuable for designers aiming to optimize performance without compromising fault tolerance. More interestingly, this observation enhances confidence in the results we present in Section IV. As we do not have access

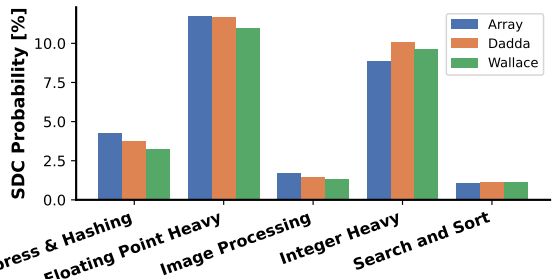

Fig. 7. SDC probability (given a stuck-at fault exists in the multiplier) per workload category for the three multiplier designs.

to any information detailing the exact implementation of the functional units for each x86 microarchitecture we analyze, we use the same gate-level models for each. The results of the sensitivity study in this section suggest that this does not significantly affect the final SDC probability and thus the overall CPU SDC rates we present.

Observation #3: The gate-level design of integer multipliers (Array, Wallace, Dadda) very minimally affects the rates of SDCs due to faults in them.

C. Simulation and fleet data: common patterns

We observe patterns in the relative SDC rates obtained from fleet data and those derived from microarchitecture-level modeling. These patterns can be observed across microarchitectures, matching the multi-year real-world Meta fleet measurements and across workloads having a strong correlation between the modeling results and the fleet observations and ranking for the workload sensitivity to corruptions (see Table VI).

Workload corruption in the fleet is primarily caused by floating-point intensive operations, especially within vector spaces, common in parallel math algorithms, machine learning, cryptography, and large data transfers. Compression and hashing workloads also contribute significantly due to their frequency and data heterogeneity, with corruption in these areas often having a noticeable impact due to sequential dependencies. While image processing applications also experience corruption, its impact is typically benign, such as discolored pixels. Scalar workloads involving floating-point computations also show meaningful corruption, with varying precision and result errors, having a higher impact than image processing.

Due to the colocation, containerization and confidentiality associated with real workloads running on the large-scale fleet, we cannot share data from these directly. However, we present relative SDC rate data from silent data corruption tests designed to heavily utilize different functional unit categories of the CPUs from the fleet systems. In Figure 8, the relative SDC rates for CPUs A, B, C, and E (CPU D is missing due to the lack of statistically significant data in the Meta fleet experiment) across various functional units are shown. While this data is **not directly comparable** to the data in Figure 6 due to the differing nature of the workloads, general trends are consistent. Notably, *vector units, particularly floating-point ones,* generate significantly more SDCs than scalar integer and floating-point units. Adders remain the *weakest* contributors to SDCs, serving as the reference value for the relative data in the heatmap. Although floating-point scalar units (both add and multiply) produce an order of magnitude more SDCs than integer scalar units, they are far surpassed by the vector units in SDC incidence.

Observation #4: Floating point heavy workloads (mostly vectorized) have the highest SDC rates as observed in both the fleet and the simulation experiments.

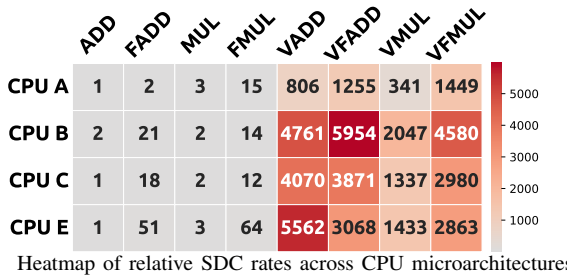

Fig. 8. Heatmap of relative SDC rates across CPU microarchitectures and functional unit categories observed in the fleet. These are obtained through running targeted tests aimed at "hammering" the respective units with stimulus.

D. Total SDC rates: intrinsic differences among the five CPU microarchitectures

Fig. 9 illustrates the total SDC rates for the five microarchitectures analyzed in this paper. To compute the total SDC rate, we employ Equation 4 with arbitrary values of λ_{bad} and $t = T$, aggregating all functional units and workloads. The total SDC rates presented here are again relative, thus rendering λ_{bad} and T inconsequential. We observe that CPU B exhibits the highest SDC Rate, surpassing CPU E - which boasts the smallest SDC rate - by a factor of $1.59\times$. CPU A closely trails CPU E, generating only 3% more SDCs per unit time. CPUs C and D, on the other hand, generate 30% and 43% more erroneous results compared to the baseline, respectively.

Observation #5: Microarchitectural choices have significant effect on SDC rates. Larger counts of the more vulnerable components produce higher SDC incidents rates.

Major contributors to the differences among the five microarchitectures are: 1) the number and types of functional units that each microarchitecture employs, 2) the operating frequency of the CPU core and, 3) the relative performance of the workloads on each microarchitecture. Correlating the data of Fig. 6 and Fig. 9 along with Table III, we infer that microarchitectures with a larger number of the most vulnerable units (scalar/vector FP units and vector integer units) exhibit larger SDC rates (CPU B has the highest number of units from these categories). This explains why the per-component SDC rates of CPU C and D are higher (Fig. 6) yet CPU B is the most vulnerable when aggregating the data of all the arithmetic units (Fig. 9). Implementing hardware-based or software-based fault protection features for these units can significantly reduce the SDC rate.

E. Fleet SDC rates: combining modeling and multi-year observational data

Up to this point, our focus has been on exploring the inherent differences among microarchitectures, demonstrating

Fig. 9. Relative SDC rate of the five microarchitectures. The same DPPM is used across CPUs to only highlight intrinsic differences.

how design choices impact the SDC rates. Initially, we analyzed SDC rates at a granular level, considering functional units and workload categories, then summarized these findings to highlight broader trends across all five microarchitectures.

In the previous sections, we have maintained consistency using uniform Defective Parts Per Million (DPPM) values across the microarchitectures encapsulated within the λ_{bad} coefficient in Equation 4. Now, we shift focus to examine extrinsic factors such as manufacturing quality, process variation, binning, aging, and environmental conditions, on the SDC rates. For this purpose, we draw upon empirical **data gathered over six years within an actual Meta data center setting**. Even though the absolute DPPMs are tracked internally in the fleet, due to the proprietary nature of the dataset, they are not included in the paper. Therefore, we employ relative DPPM data shown in Table V among the modeled microarchitectures to show how factors beyond the microarchitecture play a role in the SDC rates.

Due to the absence of statistically significant fleet data about CPU D within the hyperscaler experiment, our analysis is confined to the four remaining microarchitectures. Adjusting the previously derived SDC rates in Fig. 9 according to the relative DPPMs in Table V we obtain the SDC incidents rates results shown in Fig. 10. We observe that CPU B continues to be the most significant contributor to SDC occurrences. However, there has been a reversal in trends between CPUs A and E, with CPU C now emerging as the second-lowest SDC contributor. This shift demonstrates that intrinsic and extrinsic factors can influence SDC rates differently. Therefore, the comprehensive nature of the data collected from both simulation and fleet testing is essential in understanding SDC rates.

Observation #6: Microarchitecture alone does not determine the relative differences in SDC rates and DPPM needs to be factored in.

F. Instruction classes error rates

Our fault injection infrastructure allows us to compute the error rates of different x86 instructions when simulating a faulty functional unit. The error rate is calculated at the boundary of the functional unit, i.e., we count how many times an instruction causes a wrong computation in the unit by dividing by the total number of unit computations caused by said instruction[5]. In Fig. 11 we show the error rates of the most error-prone instructions having only considered the injection experiments

[5]Note that x86 instructions are broken into micro-ops and thus activate a functional unit multiple times.

TABLE V
LARGE SCALE FLEET EXPERIMENT DATA.

CPU	Rel. DPPM	Main suspect executions
A	0.85	Vector (FP), Vector (Int)
B	0.78	Vector (FP), Vector (Int), Scalar (FP)
C	0.67	Vector (FP), Vector (Int)
D	N/A	*Insignificant fleet data*
E	1	Vector (FP), Vector (Int), Scalar (FP)

Fig. 10. Relative SDC rate across the four CPU microarchitectures (no fleet data for CPU D), adjusted for fleet-experiment measured DPPMs aggregating all hardware units.

that result in SDCs, as we are interested in the instruction failure profiles for the SDC cases. We have grouped the functional units into corresponding types (e.g., *ADD_0 - ADD_4* are grouped in the *ADD* category, etc.) and due to space limitations we show a subset of the results to showcase this aspect of our infrastructure.

When it comes to the *Scalar Integer Adders* we see that instruction error rates in fault injections that result in SDCs are very low, ranging from about 1 wrong calculation in 1M (cmpxchg) to 1 in 31K (add) (note that the heatmap scale is 10^{-4}). Much higher adder instruction error rates result in crashes, because corrupted results are very likely to cause illegal memory accesses or erroneous control flow decisions. *Vector Integer Add* instructions, on the other hand, show much higher error rates (4 orders of magnitude). This is expected, as results from integer vector operations are very likely to end up propagating to program output.

Moving on to floating-point functional units, a faulty *Scalar Floating-Point Adder* results in very high error rates in multiple x87 instructions. These range from 16% to 77%. An interesting observation is that CPU B exhibits significantly lower error rates in x87 instructions. This is because it has double the floating-point adder functional units compared to the rest of the microarchitectures. In an ideal scenario, FADD operations would be similarly distributed (input data & unit scheduling) between the two units; thus we would expect to get a 38.5% error rate (77% divided by 2). The small deviation from this theoretical calculation lies in the scheduling of floating-point instructions that result in slightly different data passing through the two FADD units. Note that, even though FADD instructions exhibit half the error rate given only one of the two adders of CPU B being faulty in the experiment, the one faulty unit still contributes the same SDC rate in the FP heavy workloads as presented in Fig. 6. This is due to very low software masking, meaning that (almost) every computation matters. *Vector Floating-Point* instructions compared to their scalar counterparts exhibit lower error rates. The reason for this disparity is the parallel nature of the vector unit. Using *addps* as an example, four FP additions are performed in parallel. Given a stuck-at fault that affects a gate of the vector FP adder, it is very likely that one of the four results will be affected and not all. The fault could impact more than one result, but since gate-level stuck-at faults usually result in a few localized erroneous bits, this would be statistically improbable.

V. RELATED WORK

The SDC problem in computing systems has garnered significant attention recently [1]–[3], [5]. Previous research has extensively explored faults in CPU array structures (mostly

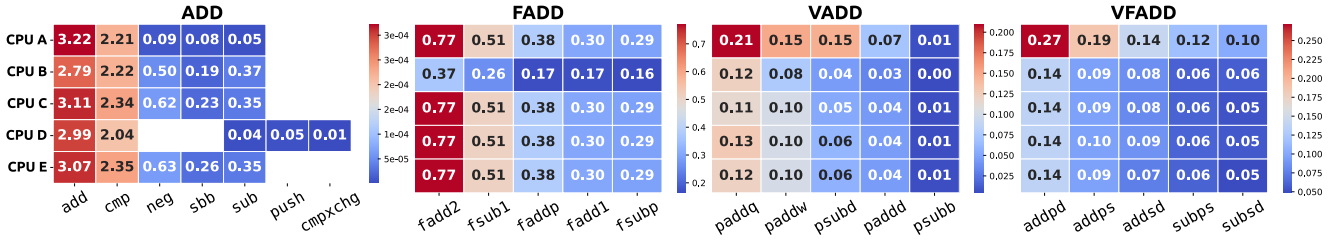

Fig. 11. Instruction error rates given that we have an error in the specific unit.

TABLE VI
SUMMARY OF OBSERVATIONS OBTAINED BY THE SIMULATION INFRASTRUCTURE AND VALIDATED BY FLEET DATA.

Observation	Simulation Data	Fleet Confirmation
#1 Scalar adders	✓	✓
#2 Vector units	✓	✓
#3 Integer multipliers	✓	*inconclusive*
#4 FP operations	✓	✓
#5 uArch sensitivity	✓	*inconclusive/not-feasible*
#6 DPPM factoring	✓	✓

due to transient faults) through microarchitecture-level simulation (e.g., [12], [17], [31], [42]–[47]). CPU array structures nowadays, as explained in Section II-A, are mostly protected and are much less likely to contribute to SDCs. Our focus is on arithmetic hardware units that have virtually no protection and have been correlated with SDCs in recent reports [1]. Limited modeling of faults in arithmetic hardware units (simple ALU and address generation unit) using a microarchitectural simulator has been performed in [34]. In contrast to this study, our approach integrating fast C++ gate-level models inside gem5 incurs virtually no throughput loss with the accuracy of simulating every single gate of each arithmetic unit (scalar and vector, integer and FP components).

Previous studies focused on silent errors induced primarily by transient radiation rays [48]–[50] and noted SDCs caused by faulty processors in cloud service environments [1], [39], [51]. Methods range from using neutron beams based on irradiation models [48], [50], [52] to simulators or specific experimental devices for fault injection [12], [15], [17], [19], [31], [32] or combining both beam experiments and fault injection [16], [18], [53]. The aim is to improve injector designs based on observations to assess SDC solutions in production better. Also, Karystinos *et al.* in [54] proposed a methodology for the generation of functional test programs for the detection of CPU faults for different CPU hardware units and different fault types. In our paper we combine microarchitecture-level modeling with real cloud compute fleet data to provide a more in-depth understanding of the root causes of SDCs, comparing microarchitectures, hardware units, and workload categories.

Many studies in the literature discuss silent data corruptions, but they typically focus on the software layer and how applications are affected by these corruptions without taking the underlying hardware into account. This fails to consider the hardware-induced faults that can silently corrupt the software output [15]. Fang *et al.* in [55] presented a full-software stack study of SDCs, while Guan *et al.* in [56] conducted an empirical study of SDCs vulnerability in sorting algorithms. Li *et al.* in [57] evaluated the impact of software-induced faults on program output. Xu *et al.* in [58] analyzed non-derated faults at the application level and discusses the implications for designing future solutions. These studies are conceptually useful but fail to analyze the entirely different distribution of faults that impact the software level. Hari *et al.* in [58] present low-cost program-level detectors for reducing SDCs. This study focuses on identifying and understanding the program portions, which may cause SDCs.There are also similar studies that focus on microarchitecture level fault occurrences, and thus, consider hardware masking, however, to the best of our knowledge, none of them explore the propagation of hardware-induced faults in arithmetic units and investigate the correlation of them to the SDC distribution. For example, Li *et al.* in [59] study the propagation of permanent faults to the software considering only the integer ALU output, and some memory arrays.

VI. CONCLUSION

This paper fills a critical gap by modeling hardware faults in integer, floating-point, and vector units of modern x86 CPUs, prevalent in hyperscalers' fleets, to practically measure the *SDC incidents rates* due to faults in these top suspect CPU hardware units. We introduce the implementation of a novel gem5-based fault injector for arithmetic hardware units which preserves the simulation speed of gem5. Through a comprehensive microarchitecture-level analysis, we uncover insights not possible in other setups, revealing SDC trends and sensitivities across different x86 microarchitectures, scalar and vector arithmetic units, and instruction classes. By integrating real defective parts per million (DPPM) data from Meta's large-scale fleet experiments, we report on the severity of SDC incidents across microarchitectures, hardware units, and workload classes. This information can guide vendors and hyperscalers in developing system-level fault protection methods, making a significant contribution to minimizing the impact of SDC incidents in computing.

ACKNOWLEDGEMENTS

This work was supported by research gifts from OCP (Open Compute Project), Meta, and AMD, as well as the European Union's Horizon Europe research and innovation programme under grant agreements No 101093062 (Vitamin-V), No 101070238 (NEUROPULS), and No 101097224 (REBECCA). Views and opinions expressed are however, those of the authors only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them. This paper is also funded by HFRI through grant No 16973 (REDESIGN).

References

[1] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, "Silent Data Corruptions at Scale," 2021. [Online]. Available: https://arxiv.org/abs/2102.11245

[2] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores That Don't Count," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 9–16. [Online]. Available: https://doi.org/10.1145/3458336.3465297

[3] S. Wang, G. Zhang, J. Wei, Y. Wang, J. Wu, and Q. Luo, "Understanding silent data corruptions in a large production cpu population," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 216–230. [Online]. Available: https://doi.org/10.1145/3600006.3613149

[4] ——, "Understanding silent data corruption in processors for mitigating its effects," *ACM Trans. Archit. Code Optim.*, Sep. 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3690825

[5] D. Gizopoulos, "Sdcs: A b c," Sep 2024. [Online]. Available: https://www.sigarch.org/sdcs-a-b-c/

[6] S. Gurumurthi, "Emerging fault modes: Challenges and research opportunities," Jul 2023. [Online]. Available: https://www.sigarch.org/emerging-fault-modes-challenges-and-research-opportunities/

[7] T. Macieira, S. Gurumurthy, S. Gurumurthi, A. Haggag, G. Papadimitriou, and D. Gizopoulos, "Silent data corruptions in computing: Understand and quantify," in *2024 IEEE 30th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2024, pp. 1–7.

[8] D. Ma, F. Lin, A. Desmaison, J. Coburn, D. Moore, S. Sankar, and X. Jiao, "Dr. dna: Combating silent data corruptions in deep learning using distribution of neuron activations," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 239–252.

[9] X. Jiao, F. Lin, H. D. Dixit, J. Coburn, D. Moore, and S. Sankar, "Silent data corruptions in ai systems: Evaluation and mitigation."

[10] T. Garrett, S. Roffe, and A. George, "Soft-error characterization and mitigation strategies for edge tensor processing units in space," *IEEE Transactions on Aerospace and Electronic Systems*, 2024.

[11] S. Hesley, "The future of computing depends on you," Keynote at the International Test Conference (ITC), San Diego, CA, USA, 2024, accessed January 17, 2025. [Online]. Available: https://www.itctestweek.org/2024-keynote-visionary-talks/

[12] G. Papadimitriou and D. Gizopoulos, "Silent data corruptions: Microarchitectural perspectives," *IEEE Transactions on Computers*, vol. 72, no. 11, pp. 3072–3085, 2023.

[13] ——, "Characterizing soft error vulnerability of cpus across compiler optimizations and microarchitectures," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, 2021, pp. 113–124.

[14] ——, "Avgi: Microarchitecture-driven, fast and accurate vulnerability assessment," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 935–948.

[15] ——, "Demystifying the system vulnerability stack: Transient fault effects across the layers," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 902–915.

[16] P. R. Bodmann, G. Papadimitriou, R. L. Rech Junior, D. Gizopoulos, and P. Rech, "Soft error effects on arm microprocessors: Early estimations versus chip measurements," *Computer*, vol. 56, no. 7, pp. 4–6, 2023.

[17] G. Papadimitriou and D. Gizopoulos, "Anatomy of on-chip memory hardware fault effects across the layers," *IEEE Transactions on Emerging Topics in Computing*, vol. 11, no. 2, pp. 420–431, 2023.

[18] P. R. Bodmann, G. Papadimitriou, R. L. R. Junior, D. Gizopoulos, and P. Rech, "Soft error effects on arm microprocessors: Early estimations versus chip measurements," *IEEE Transactions on Computers*, vol. 71, no. 10, pp. 2358–2369, 2022.

[19] D. Sartzetakis, G. Papadimitriou, and D. Gizopoulos, "gpufi-4: A microarchitecture-level framework for assessing the cross-layer resilience of nvidia gpus," in *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022, pp. 35–45.

[20] L. Yang, G. Papadimitriou, D. Sartzetakis, A. Jog, E. Smirni, and D. Gizopoulos, "Gpu reliability assessment: Insights across the abstraction layers," in *2024 IEEE International Conference on Cluster Computing (CLUSTER)*, 2024, pp. 1–13.

[21] ——, "Probing weaknesses in gpu reliability assessment: A cross-layer approach," in *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2024, pp. 331–333.

[22] P. Bodmann, G. Papadimitriou, D. Gizopoulos, and P. Rech, "The Impact of SoC Integration and OS Deployment on the Reliability of Arm Processors," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2021, pp. 223–225. [Online]. Available: https://doi.org/10.1109/ISPASS51385.2021.00040

[23] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. Kim, "Robust system design with built-in soft-error resilience," *Computer*, vol. 38, no. 2, pp. 43–52, 2005.

[24] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[25] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian, "The gem5 simulator: Version 20.0+," 2020. [Online]. Available: https://arxiv.org/abs/2007.03152

[26] S. Gurumurthi, V. Sridharan, and S. Gurumurthy, "Emerging fault modes: Challenges and research opportunities," ACM SIGARCH, July 17, 2023. [Online]. Available: https://www.sigarch.org/emerging-fault-modes-challenges-and-research-opportunities/#

[27] C. Weaver, J. Emer, S. Mukherjee, and S. Reinhardt, "Techniques to reduce the soft error rate of a high-performance microprocessor," in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, 2004, pp. 264–275.

[28] L. Peters, "New insights into ic process defectivity," Semiconductor Engineering, November 7, 2023. [Online]. Available: https://semiengineering.com/new-insights-into-ic-process-defectivity/

[29] A. Meixner, "Screening for silent data errors," Semiconductor Engineering, January 10, 2023. [Online]. Available: https://semiengineering.com/screening-for-silent-data-errors/

[30] T. C. I. S. . I. 43.040.10, "Iso/tr 9839:2023, road vehicles, application of predictive maintenance to hardware with iso 26262-5," ISO, August, 2023. [Online]. Available: https://www.iso.org/standard/83605.html

[31] G. Papadimitriou, D. Gizopoulos, H. D. Dixit, and S. Sankar, "Silent data corruptions: The stealthy saboteurs of digital integrity," in *2023 IEEE 29th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2023, pp. 1–7.

[32] A. Singh, S. Chakravarty, G. Papadimitriou, and D. Gizopoulos, "Silent data errors: Sources, detection, and modeling," in *2023 IEEE 41st VLSI Test Symposium (VTS)*, 2023.

[33] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[34] M.-L. Li, P. Ramachandran, U. R. Karpuzcu, S. K. S. Hari, and S. V. Adve, "Accurate microarchitecture-level fault modeling for studying hardware faults," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 2009, pp. 105–116.

[35] "gem5 GitHub Repository," https://github.com/gem5/gem5, accessed: 2023-04-25.

[36] O. Chatzopoulos, G. Papadimitriou, D. Gizopoulos, H. D. Dixit, and S. Sankar, "From gates to sdcs: Understanding fault propagation through the compute stack," in *2025 Design, Automation & Test in Europe Conference & Exhibition*, 2025.

[37] J. Klhufek and V. Mrazek, "Arithsgen: Arithmetics circuit generator for hw accelerators," in *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS '22)*, 2022, p. 4.

[38] F. Lin, M. Beadon, H. D. Dixit, G. Vunnam, A. Desai, and S. Sankar, "Hardware remediation at scale," in *2018 48th Annual IEEE/IFIP Inter-*

national Conference on Dependable Systems and Networks Workshops (DSN-W), 2018, pp. 14–17.

[39] H. D. Dixit, L. Boyle, G. Vunnam, S. Pendharkar, M. Beadon, and S. Sankar, "Detecting silent data corruptions in the wild," 2022.

[40] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation & Test in Europe Conference & Exhibition*, 2009, pp. 502–506.

[41] B. Parhami, *Computer arithmetic: algorithms and hardware designs*. USA: Oxford University Press, Inc., 1999.

[42] O. Chatzopoulos, G. Papadimitriou, V. Karakostas, and D. Gizopoulos, "Gem5-marvel: Microarchitecture-level resilience analysis of heterogeneous soc architectures," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2024, pp. 543–559. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HPCA57654.2024.00047

[43] A. Chatzidimitriou, G. Papadimitriou, C. Gavanas, G. Katsoridas, and D. Gizopoulos, "Multi-bit upsets vulnerability analysis of modern microprocessors," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*, 2019, pp. 119–130.

[44] D. Gizopoulos, G. Papadimitriou, and O. Chatzopoulos, "Estimating the failures and silent errors rates of cpus across isas and microarchitectures," in *2023 IEEE International Test Conference (ITC)*, 2023, pp. 377–382.

[45] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, and D. Gizopoulos, "Accelerated microarchitectural fault injection-based reliability assessment," in *2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, 2015, pp. 47–52.

[46] T. Macieira, S. Gurumurthy, S. Gurumurthi, A. Haggag, G. Papadimitriou, and D. Gizopoulos, "Silent data corruptions in computing: Understand and quantify," in *2024 IEEE 30th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2024, pp. 1–7.

[47] D. Gizopoulos, G. Papadimitriou, O. Chatzopoulos, N. Karystinos, H. D. Dixit, and S. Sankar, "Silent data corruptions in computing systems: Early predictions and large-scale measurements," in *2024 IEEE European Test Symposium (ETS)*, 2024, pp. 1–10.

[48] D. Agiakatsikas, G. Papadimitriou, V. Karakostas, D. Gizopoulos, M. Psarakis, C. Bélanger-Champagne, and E. Blackmore, "Impact of voltage scaling on soft errors susceptibility of multicore server cpus," in *MICRO-56: 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-56. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3613424.3614304

[49] S. Michalak, K. Harris, N. Hengartner, B. Takala, and S. Wender, "Predicting the number of fatal soft errors in los alamos national laboratory's asc q supercomputer," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 329–335, 2005.

[56] Z. Li, H. Menon, D. Maljovec, Y. Livnat, S. Liu, K. Mohror, P.-T. Bremer, and V. Pascucci, "Spotsdc: Revealing the silent data corruption

[50] L. M. Luza, D. Söderström, H. Puchner, R. G. Alía, M. Letiche, A. Bosio, and L. Dilillo, "Effects of thermal neutron irradiation on a self-refresh dram," in *2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2020, pp. 1–6.

[51] D. F. Bacon, "Detection and prevention of silent data corruption in an exabyte-scale database system," in *The 18th IEEE Workshop on Silicon Errors in Logic – System Effects*, 2022.

[52] R. L. Rech and P. Rech, "Reliability of google's tensor processing units for embedded applications," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 376–381.

[53] A. Chatzidimitriou, P. Bodmann, G. Papadimitriou, D. Gizopoulos, and P. Rech, "Demystifying Soft Error Assessment Strategies on ARM CPUs: Microarchitectural Fault Injection vs. Neutron Beam Experiments," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2019, pp. 26–38. [Online]. Available: https://doi.org/10.1109/DSN.2019.00018

[54] N. Karystinos, O. Chatzopoulos, G. Fragkoulis, G. Papadimitriou, G. Gizopoulos, and S. Gurumurthi, "Harpocrates: Breaking the silence of cpu faults through hardware-in-the-loop program generation," in *Proceedings 51st ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2024.

[55] Q. Guan, N. DeBardeleben, S. Blanchard, and S. Fu, "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*, ser. FTXS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 35–40. [Online]. Available: https://doi.org/10.1145/2751504.2751512
propagation in high-performance computing systems," *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, no. 10, pp. 3938–3952, 2021.

[57] X. Xu and M.-L. Li, "Understanding soft error propagation using efficient vulnerability-driven fault injection," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, 2012, pp. 1–12.

[58] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, 2012, pp. 1–12.

[59] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," *SIGPLAN Not.*, vol. 43, no. 3, p. 265–276, mar 2008. [Online]. Available: https://doi.org/10.1145/1353536.1346315