GPU Reliability Assessment: Insights Across the Abstraction Layers

Lishan Yang[§] George Papadimitriou[†] Dimitris Sartzetakis[†]
Adwait Jog[‡] Evgenia Smirni[*] Dimitris Gizopoulos[†]

[§]*Department of Computer Science, George Mason University,* USA, lyang28@gmu.edu
[†]*Dept. of Informatics & Telecomm., University of Athens,* Greece, {georgepap | sartzet | dgizop}@di.uoa.gr
[‡]*Department of Computer Science, University of Virginia,* USA, ajog@virginia.edu
[*]*Department of Computer Science, William & Mary,* USA, esmirni@cs.wm.edu

***Abstract*—Graphics Processing Units (GPUs) are widely deployed and utilized across various computing domains including cloud and high-performance computing. Considering its extensive usage and increasing popularity, ensuring GPU reliability is crucial. Software-based reliability evaluation methodologies, though fast, often neglect the complex hardware details of modern GPU designs. This oversight could lead to misleading measurements and misguided decisions regarding protection strategies. This paper breaks new ground by conducting an in-depth examination of well-established vulnerability assessment methods for modern GPU architectures, from the microarchitecture all the way to the software layers. It highlights divergences between popular software-based vulnerability evaluation methods and the ground truth cross-layer evaluation, which persist even under strong protections like triple modular redundancy. Accurate evaluation requires considering fault distribution from hardware to software. Our comprehensive measurements offer valuable insights into the accurate assessment of GPU reliability.**

***Index Terms*—reliability assessment, GPUs, fault injection**

I. Introduction

Rapid developments in silicon manufacturing have enabled increased performance and improved energy efficiency of current graphics processing units (GPUs) [1]. Nowadays, GPUs are extensively employed in pre-exascale supercomputers for their highly parallel computation throughput to accelerate high-performance computing (HPC) applications [2], [3], which often have strict reliability requirements. The long execution time of these HPC applications also increases the probability of encountering soft errors (hardware transient faults) [3]–[6] that can result in faulty outputs or crashes. Ensuring the reliability of these applications is even more challenging under the growing prevalence of soft errors in advanced manufacturing technologies [7]. The ever-increasing rate of soft errors in newer manufacturing technologies can jeopardize the aggressive evolution of GPUs, which brings additional challenges. For example, since GPU applications are written using the Single-Instruction-Multiple-Threads (SIMT) paradigm, a single transient fault in a bit-cell of a hardware structure can result in multiple data corruptions at the application output [8] or a thread affected by a fault may supply several subsequent parallel threads with corrupted data [9].

Assessing the impact of soft errors on GPU workloads at the early (unprotected) GPU design phase is important for unveiling potentially vulnerable hardware areas that need to be protected. Reliability assessment of a computing system can be realized using different techniques which vary in their design maturity and granularity, the level of accuracy, and the speed of the assessment process [10]. Simulation is a very widely employed method for the assessment of the vulnerability to soft errors, long before the product becomes available to users. GPU reliability evaluations are often performed on models of the actual GPU design using simulators [11]–[16]. Highly detailed and accurate simulation models at the RTL (Register Transfer Level), gate, or transistor level are extremely slow, not scalable, and not feasible. Less detailed models, for example at the microarchitecture level (using cycle-level simulators), are much faster than low-level highly detailed models. Higher-level ISA (Instruction Set Architecture) simulation models, although faster, are even more abstract (hardware-agnostic). Inaccurate reliability assessments can lead to pitfalls and wrong design decisions, finally resulting in more vulnerabilities [17], [18].

Assessing the Architectural Vulnerability Factor (AVF) [19] of each microarchitectural structure of a chip during end-to-end program execution is a comprehensive way to evaluate the vulnerability of the entire system stack to soft errors, from the microarchitecture to the software layers [17]. AVF is the probability that a soft error (e.g., a bit flip) may produce an observable error at the application output. While AVF has been initially proposed for the assessment of reliability in CPUs, it has also been naturally adapted to GPUs [20]–[22]. Typically, application resilience is measured by experimental campaigns based on statistical fault injection (FI) [23] or using analytical methods, such as the Architecturally Correct Execution (ACE) analysis [19]. AVF measurements based on statistical fault injection provide useful and accurate insights for the application reliability profile but come with a limitation: since AVF measurements are based on cycle-level, microarchitecture-detailed simulation, obtaining the AVF of a GPU program is very slow [24].

Software-based vulnerability estimation methods, assuming software-visible origins of hardware bit flips, are significantly faster than full-system hardware measurements, which consider all hardware bits [12]–[14], [25]. The speed difference can be two orders of magnitude or more[1]. These software-

[1]For example, the AVF experiments of this study require 1,258 single-core machine days, compared to the 10 machine days used for the SVF experiments.

level methods derive the Software Vulnerability Factor (SVF, as it is defined in [17]), representing the probability of a fault affecting program execution in a single dynamic instruction. They are commonly used under the assumption that (a) reasonably model the effect of soft errors on the software layer (i.e., the overall resilience) and (b) at least provide correct relative vulnerability comparisons among different workloads. This work challenges these assumptions and demonstrates that neither stands for GPU reliability assessment.

In this paper, we present an unbiased comparison of GPGPU reliability evaluation at different layers. To the best of our knowledge, this is the first study that such a cross-layer analysis has been performed in the GPU domain. We *quantify* and explain the diverging estimation results obtained when assessing the reliability of GPUs at different abstraction layers, specifically at the microarchitecture and the software layers. The contributions of this work are summarized as follows:

- We demonstrate the magnitude of measurement errors introduced by software-level reliability evaluation methods, compared to the ground-truth, cross-layer AVF analysis. To this end, we employ two state-of-the-art, open-source fault injection frameworks that both focus on NVIDIA GPUs: gpuFI-4 [11], [26] and NVBitFI [25], [27], which operate at the microarchitecture level and at the software level, respectively.
- We conduct a case study to measure the effectiveness of a strong software-based protection method, Triple Modular Redundancy (TMR) [28], which aims to eliminate silent data corruptions (SDCs) [29]–[31]. Our case study reveals two major insights: 1) although software-level evaluation (i.e., SVF) confirms that SDCs are effectively eliminated, the cross-layer evaluation (i.e., AVF) shows that some SDCs still remain despite the heavy penalty of protection in terms of performance (and thus, energy consumption [29], [32]–[34]), and 2) while most of the SDCs are eliminated, Detected Unrecoverable Errors (DUEs) instead increase, resulting frequently in higher vulnerability of the heavily protected application compared to the unprotected one.
- We provide insights and reasoning about the sources of assessment error of software-level methods, which eventually lead to diverging results, and explain the reasons that lead to such discrepancies.

II. Experimental Setup

In this study, we employ two open-source fault injection frameworks: gpuFI-4 [11] for microarchitecture-level assessment and NVBitFI [25] for software-level assessment. To ensure fairness, we carefully select closely matched GPUs from gpuFI-4 and NVBitFI-supported sets: Quadro GV100 for microarchitecture-level and Tesla V100 for software-level fault injection. Both GPUs, based on NVIDIA Volta microarchitecture, exhibit highly similar configurations for the considered structures—register files, shared memory, L1 data and texture caches, and L2 caches—meeting the essential criteria for an equitable comparison.

A. Fault Model

We focus on a single-bit flip fault model for our evaluation, anticipating similar outcomes with multi-bit flips. Physical experiments of accelerated beam testing [35] establish that on-chip storage arrays can suffer from multi-bit flips in adjacent areas. In other words, even if a multi-bit flip occurs, the corruption could not occur at two different instruction locations, different threads, or different structures at the same time. Recent studies have shown that this probability is highly proportional to the number of bit flips [36], and single-bit flips contribute the most to the total vulnerability compared to multi-bit faults [36], [37]. Therefore, we do not expect multi-bit fault occurrences to change our observations. Most of the on-chip memory structures are protected through error correction codes (ECC), but with overhead. There are several new proposals for alternative protection schemes aiming at lower performance penalties and/or power consumption [38]–[41]. Reliability evaluation in the early design phases is necessary to decide on the most appropriate protection technique for a new design. Starting with an unprotected GPU design, we aim to gauge the inherent vulnerability of each on-chip structure to inform targeted protection strategies.

Aligned with prior works [17], [37], [42], our experimental approach utilizes statistical fault injection [23]. Each experiment involves injecting a single-bit fault at a random (i.e., uniformly distributed) location. We iterate this process 3,000 times to provide results with 99% confidence intervals and an error margin of approximately $\pm 2.35\%$ [16], [23], [43]. We classify the effect on the program output into the following fault effect classes (typically used in fault injection studies):

- *Masked* outcome happens when the fault does not affect the system or the application in any observable way.
- *Silent Data Corruption (SDC)* occurs when an application completes its execution, yet the output differs from that of the fault-free run.
- *Timeout* occurs when the application does not finish within a certain amount of time.
- *Detected unrecoverable errors (DUEs)* occurs when the execution does not complete because a catastrophic event disturbs it. No output is produced, and it may refer to a kernel or application crash.

B. Microarchitecture-Level Fault Injection

For microarchitecture-level fault injection, we employ the open-source framework gpuFI-4 [11], a state-of-the-art microarchitecture-level reliability assessment framework built on top of the recent GPGPU-Sim 4.0 simulator [44]. We inject faults in the five hardware structures that are supported by gpuFI-4: register files (RF), shared memory (SMEM), L1 data caches (L1D), L1 texture caches (L1T), and L2 caches. For a fair comparison between cross-layer and software-level evaluation methods, we do not consider faults in the L1 instruction cache, since software-level fault injection tools do not consider faults that affect any bit of the instruction format (see details in subsection II-C).

gpuFI-4 provides the cross-layer AVF (Architectural Vulnerability Factor), defined as the probability of a fault in a hardware structure that would result in an error (i.e., the fault is not masked) [19]. In AVF analysis, any bit at the microarchitecture level can be flipped, no matter if it is currently valid (i.e., alive) or not. We follow the well-established AVF methodology to calculate the AVF [17], [19] and describe the detailed calculation below.

We define, as the *failure rate (FR)* of a hardware structure h, the probability of all non-masked faults:

$$FR(h) = Pct(SDC) + Pct(Timeout) + Pct(DUE)$$

where $Pct(x)$ denotes the percentage of component x.

GPGPU-Sim 4.0 simulator does not have a real register file hardware structure as a reference, but it dynamically allocates each register of a thread during the execution (and frees them when a thread finishes). Even if the total number of used registers is known from the beginning, it is not possible to inject a fault against the entire register file, i.e., currently used and unused registers, but only to the registers that are active at a specific cycle. Therefore, to accurately calculate the correct probability (i.e., AVF), which depends on the total number of registers (i.e., the entire population of bits, including currently used and unused bits), we weight the AVF as if we were targeting the entire register file. The same logic applies to the shared memory, with the difference that this memory is allocated per CTA and not per thread.

To overcome this inherent issue of the simulator, in our register file and shared memory analysis, we define a *derating factor (DF)* of a hardware structure h as follows:

$$DF(h) = \frac{size_per_thread(h) \times num_threads}{system_size(h)}$$

The derating factor is only for register files and shared memory, as these two components have the previously discussed simulator issue. DF does not apply to other hardware components like L1 data/texture caches and L2 caches. By using the failure rate (FR) and corresponding derating factor (DF) of each hardware structure h, we can accurately calculate the cross-layer AVF of that structure:

$$AVF(h) = FR(h) \times DF(h)$$

For a fair comparison between microarchitecture and software-level resilience analysis (see subsection II-C), we need to calculate the full AVF of the entire chip, i.e., the consolidated AVF of all hardware structures. To this end, we compute the accurate full GPU chip AVF by weighting all the hardware structures $h_1, h_2, ..., h_n$ by their actual sizes (bit counts), which is a well-established process for accurately delivering the AVF of the entire chip [17]:

$$AVF(all) = \Sigma_{i=1}^{n}[AVF(h_i) \times \frac{size(h_i)}{\Sigma_{j=1}^{n} size(h_j)}]$$

For multi-kernel applications (of different execution times), we first assess the AVF of each kernel separately by injecting faults into each target kernel only. The AVF of the entire application is calculated by weighting the kernel AVF by its number of cycles to reflect its duration:

$$AVF(app) = \Sigma_{i=1}^{k}[AVF(ker_i) \times \frac{num_cycles(ker_i)}{\Sigma_{j=1}^{k} num_cycles(ker_j)}]$$

C. Software-Level Fault Injection

For the software-level fault injection part of this study, we employ the open-source framework NVBitFI [25]. NVBitFI is a state-of-the-art software-level fault injector, officially supported by NVIDIA. It is built on top of NVBit (NVIDIA Binary Instrumentation Tool) [45], which is a dynamic binary instrumentation library built for recent NVIDIA GPUs. NVBit enables the instrumentation of SASS instructions of kernel functions in GPU applications. Because of the nature of the software-level fault injection, faults are only injected into valid (i.e., alive) data. This is exactly how the software-level fault injection measurements are naturally performed, and we follow the same process as it is implemented in NVBitFI tool. NVBitFI provides the Software Vulnerability Factor (SVF). We define the SVF metric as the probability of a fault in a single dynamic instruction that would affect program execution, following its initial definition [17]. SVF expresses the vulnerability factor of faults whose origin is at a software-visible location (i.e., a flipped bit in the destination register) and not a flipped hardware bit at the microarchitecture level. SVF is microarchitecture-independent, in contrast to AVF.

NVBitFI injects faults into the destination registers of executed instructions, thus there is no need to consider any derating factor. The SVF of a kernel can be calculated from the probability of all non-masked faults:

$$\begin{aligned} SVF(ker) &= FR(ker) \\ &= Pct(SDC) + Pct(timeout) + Pct(DUE) \end{aligned}$$

For applications with multiple kernels, we first assess the SVF of each kernel separately by injecting faults into the target kernel. For computing the SVF of the entire application, we weight the kernel SVF with its number of executed instructions, since we assume a uniform fault distribution *across time*:

$$SVF(app) = \Sigma_{i=1}^{k}[SVF(ker_i) \times \frac{num_instructions(ker_i)}{\Sigma_{j=1}^{k} num_instructions(ker_j)}]$$

D. Benchmarks

We carefully select 11 benchmarks (23 kernels) from two very widely used benchmark suites CUDA [46] and Rodinia [47] in reliability studies [15], [48], [49], covering a wide range of applications, especially HPC applications, including image processing, data mining, graph algorithms, linear algebra, bioinformatics, and physics simulations. The variety of these benchmarks eliminates the bias of programming factors such as instruction types and counts, which may affect application resilience.

III. Comparison of AVF and SVF

In this section, we present a detailed comparison between the AVF (i.e., the cross-layer vulnerability) and the SVF (i.e., the software-only, hardware-agnostic vulnerability evaluation)

that unveils the magnitude of estimation error that SVF delivers. Our experimental results include an application-wise comparison and a kernel-wise comparison between AVF and SVF. We profile diverse performance measurements and show that resource utilization can be used as an indicator of some trends. Furthermore, we consider the comparison of other sub-metrics of AVF and SVF, such as AVF-RF (Register File) and SVF, and AVF-Cache and SVF-LD (which is the SVF calculated through injections in Load instructions only). Through these specific comparisons, we bring up awareness of inconsistencies across reliability assessment methodologies and showcase the importance of bridging this cross-layer gap.

A. Comparison between AVF and SVF (Application-wise)

Figure 1 shows the AVF and SVF results of applications studied in this work. Please note the different scales of the vertical axis between the AVF and the SVF graphs: full-system vulnerability absolute values (the bottom graph of Figure 1) are always much smaller than the software-only vulnerability ones (the top graph of Figure 1) because they also consider the full hardware masking effects. Since AVF and SVF assume different origins of faults (for the AVF the origin is any microarchitectural bit; for SVF it is a bit of the destination register of a dynamic instruction), there is no direct comparison of absolute vulnerabilities. The focus here is the *relative trends* (i.e., the vulnerability ranking of two individual applications) and not the comparison of the actual vulnerabilities. This method is in line with other research works in the literature [17].

The differences between AVF and SVF in Figure 1 are dramatic. Trends in SVF and AVF occasionally align and sometimes diverge. In certain cases, SVF and AVF produce entirely contrasting vulnerability estimations, impacting both the overall vulnerability and the severity of specific fault effect classes. We start our discussion with consistent trends. For instance, in Figure 1, consider the pairings of SRADv1 and SRADv2 benchmarks (first two bars) as well as the K-Means and HotSpot benchmarks (third and fourth bars). Both SVF and AVF consistently show that K-Means exhibits notably lower vulnerability compared to HotSpot, just as SRADv1 demonstrates greater reliability than SRADv2. This pattern remains consistent across all three fault effect classes: SDC, timeout, and DUE.

However, there are many pairs of benchmarks that show opposite vulnerability trends between SVF and AVF analysis. Consider the pair of HotSpot and LUD benchmarks (fourth and fifth bars in both graphs of Figure 1). From the cross-layer AVF point of view, LUD is *more* resilient than HotSpot, while based on SVF, LUD is significantly *less* resilient than HotSpot. As another example, for the pair of SCP and VA benchmarks, similar observations occur: SVF shows that SCP is more resilient than VA, while the ground-truth AVF shows the exact opposite trend. Moreover, the vulnerability trend of the SDC fault effect is also different between the cross-layer AVF and software-layer SVF. LUD exhibits significant vulnerabilities considering SDC when measuring SVF (i.e., the SDC probability based on SVF is 74.47%), but from the AVF analysis, the SDC rate is extremely low. This observation is very important, since such diverging SVF evaluations may lead designers to decide and apply a wrong protection scheme in practice. For example, *budgeted protection*, or sometimes called *partial protection*, is a common practice given the expensive cost of protection overhead [41], [50]. The idea is simple, to protect only the most vulnerable components (e.g., applications, kernels, threads, or instructions) in the system. From the SVF analysis, clearly, high protection priority should be given to LUD and VA. However, based on AVF, the most vulnerable applications are SRADv2, HotSpot, and SCP. Therefore, software designers may decide to protect an application (e.g., the most vulnerable application, LUD) against SDCs (which typically occur due to faults in computations and the data flow) by applying a software fortification method [41], [51]. However, since AVF shows that the SDC rate is extremely low, protecting this application from SDCs is unnecessary and the resources are wasted. Even worse, a wrong-decided protection scheme can increase the vulnerability of the application, instead of decreasing it. section IV shows several cases in which wrong-applied protection increases the software vulnerability rather than decreases it.

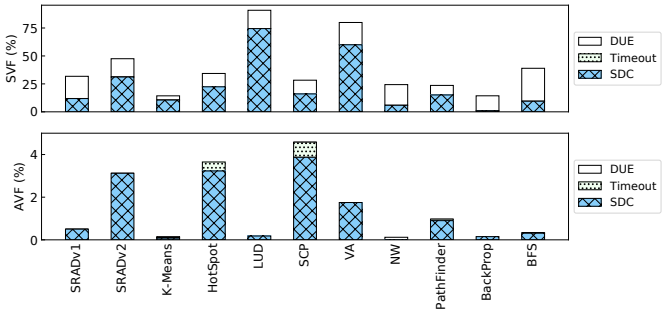


Fig. 1. Application-level comparison: AVF (bottom) and SVF (top).

Overall, there are 32 pairs (i.e., 58%) of applications with consistent trends between AVF and SVF and 23 pairs (i.e., 42%) of applications showing opposite trends between AVF and SVF (see the first row of Table I). Explanations of these trends are presented in subsection III-C.

TABLE I
OPPOSITE TRENDS IN APPLICATION OR KERNEL PAIRS.

	Consistent Trend	Opposite Trend
Application-Level	32 (58%)	23 (42%)
Kernel-Level	144 (57%)	109 (43%)
AVF-RF vs. SVF	32 (58%)	23 (42%)
AVF-Cache vs. SVF-LD	23 (42%)	32 (58%)

B. Comparison between AVF and SVF (Kernel-wise)

Since GPGPU application kernels normally implement independent modules/functions, we also conduct a kernel-wise AVF and SVF comparison for a comprehensive analysis, see Figure 2. Similar to application-wise comparisons, both consistent and opposite vulnerability trends exist at the individual kernel level. For example, K1 and K2 of BackProp show

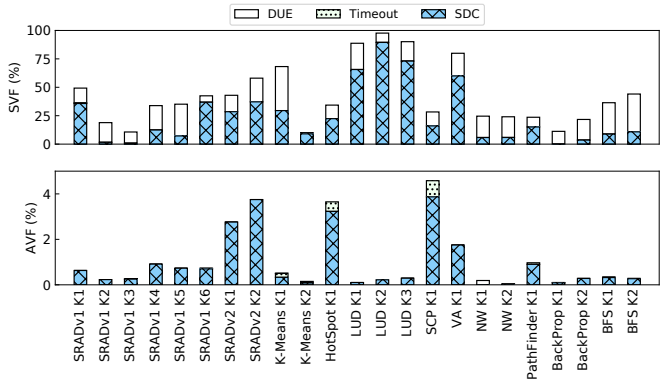


Fig. 2. Kernel-level AVF (bottom) and SVF (top).

consistent trends between AVF and SVF; PathFinder K1 and BackProp K1 also show consistent trends. An example of opposite vulnerability trends is HotSpot K1 and LUD K1: considering AVF, HotSpot K1 is more vulnerable, but based on SVF, HotSpot K1 is significantly less vulnerable than LUD K1. Overall, there are 144 pairs of kernels showing consistent trends and 109 pairs of kernels with opposite trends (see second row of Table I).

Opposite vulnerability trends between AVF and SVF not only occur between different kernels of different applications but also between different kernels of the same application. For example, for the kernels of SRADv1 shown in Figure 2, AVF analysis shows that SRADv1 K4 is the most vulnerable kernel, while according to SVF, SRADv1 K1 is the most vulnerable one. Another example is LUD in Figure 2. While the AVF shows that LUD K3 is the most vulnerable kernel, the SVF shows that the most vulnerable is LUD K2.

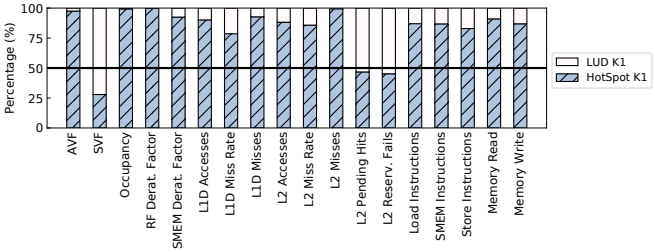
Insight #1: Software and microarchitecture level reliability assessment methods deliver inconsistent relative vulnerabilities of the studied applications and kernels. This observation points to the need for exploration of the relationship of application resilience among these layers.

C. Resource Utilization: An Indicator

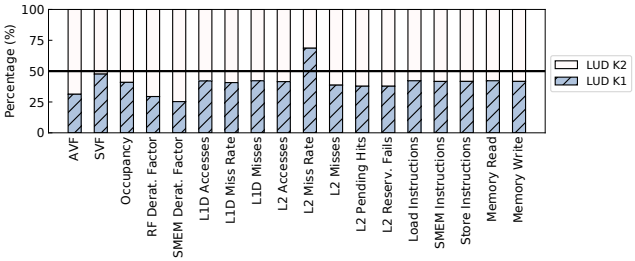
In this section, we present a correlation of vulnerability trends to diverse performance metrics collected by GPGPU-Sim 4.0 [44] during fault-free executions and show that resource utilization serves as an indicator for some resilience trends. Such metrics include (among others) the register file usage, the cache usage, and the total number of instructions. Due to space constraints, we present those metrics that are closely related and contribute the most to the major differences between AVF and SVF. For a fair comparison, each metric is normalized by the sum of the values of the certain metric of two kernels:

$$Norm.Value(Ker_1) = \frac{Value(Ker_1)}{Value(Ker_1) + Value(Ker_2)}$$

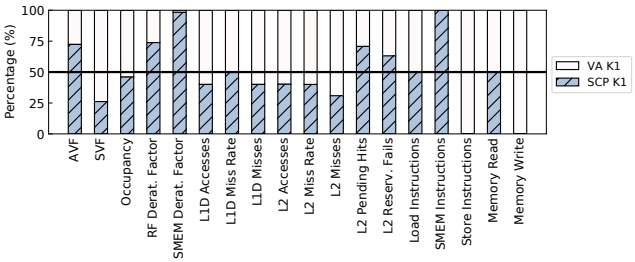
$$Norm.Value(Ker_2) = \frac{Value(Ker_2)}{Value(Ker_1) + Value(Ker_2)}$$



(a) Opposite trend: HotSpot K1 has much higher resource utilization.



(b) Consistent trend: LUD K1 has lower resource utilization, AVF, and SVF.



(c) Opposite trend: resource utilization and resilience of the application.

Fig. 3. AVF, SVF, and performance measurements for each application.

The result value of 50% means that the values of a certain metric (e.g., AVF or SVF) of two kernels are the same. In Figure 3a, the left-most bar shows that the AVF of HotSpot K1 is much higher than LUD K1, but the second left bar (SVF) shows that LUD K1 has a higher SVF, which shows opposite trends between AVF and SVF. Considering other metrics related to resource usage (as shown in the remaining bars of the graph), for most of the metrics, HotSpot K1 has higher resource utilization, indicating that faults injected at the microarchitecture level have a higher chance to propagate to the software level. This observation explains in part the reason for this opposite trend.

Another example considering kernels with consistent trends is shown in Figure 3b. Both AVF and SVF show that LUD K2 is more vulnerable than LUD K1, which suggests that the algorithm and the implementation of LUD K1 are more resilient. In most cases, LUD K2 provides higher resource utilization (i.e., the white-colored portion of bars is larger than the blue-colored portion), therefore both its AVF and SVF are higher than LUD K1.

Most of the applications and application kernels exhibit similar behaviors, but resource utilization cannot be used directly as a proxy. Figure 3c shows another example of opposite trends, but without no clear conclusion of higher or lower resource utilization. In section V we analyze in detail

the sources of assessment errors that contribute to incorrect SVF estimation and conclude that only cross-layer evaluation methods can provide the correct application resilience.

Although resource utilization does not directly determine the relationship between AVF and SVF, it serves as an effective indicator for resilience trends. In cases where a kernel demonstrates lower SVF and decreased resource utilization, its AVF also tends to be lower (as illustrated in Figure 3b). This correlation holds across all 40 kernel pairs exhibiting both reduced SVF and resource usage. This is reasonable because 1) lower SVF indicates higher fault tolerance at the application level; 2) lower utilization suggests a lower probability of faults propagating from hardware to software layers due to a reduced chance of faults occurring in invalid (not alive) data.

Insight #2: Resource utilization of an application or a kernel serves as an indicator for some trends. However, it is unlikely to determine the precise relationship and inconsistencies between AVF and SVF solely through simple resource usage calculations.

D. AVF of Major Hardware Structures and Sub-Comparisons

Software-level fault injection methods only consider faults at a currently used register, i.e., a bit flip occurs in the value of an instruction. Here, to explain the sources of assessment error of the high number of diverging results of SVF, we present fine-grained comparisons: 1) between the AVF of the register file only (i.e., labeled *AVF-RF*) and the SVF and 2) between the AVF of the on-chip memory structures only (i.e., labeled *AVF-cache*) and the SVF of memory operations (i.e., labeled *SVF-LD* and referring to bit flips only to the loaded values from memory). This allows us to compare as close as possible the AVF and SVF by considering the same group of corruptions and not the entire AVF and SVF results. The diverging trends of these comparisons practically render SVF and software-layer measurements incorrect, given that AVF delivers the ground truth measurement.

1) AVF-RF vs. SVF: The comparison of AVF-RF and SVF is shown in Figure 4 (see also the third row of Table I). There are 32 consistent trends and 23 opposite trends across all application pairs. The same number of opposite trends were also observed earlier in Figure 1[2]. It is clearly shown that even if the comparison between AVF and SVF occurs only in the register files, SVF remains a misleading measure of application resilience. The main reason is attributed to the *distribution of faults* in both measurements. On the one hand, AVF considers faults in any currently valid or not valid (i.e., alive or not alive) entry/datum of a hardware structure, which is the ground truth, since a high-energy particle may affect any (valid or not) hardware entry. On the other hand, SVF only considers fault in a certain (alive) value in a register of a dynamic instruction.

[2]The relative trends of AVF and AVF-RF are the same, because register files occupy the largest size in our fault and architecture model, i.e., the register file affects the most the overall AVF due to its size.

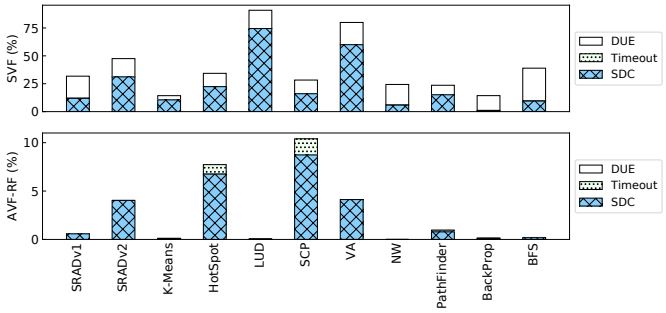


Fig. 4. Comparison of AVF-RF (bottom) and SVF (top).

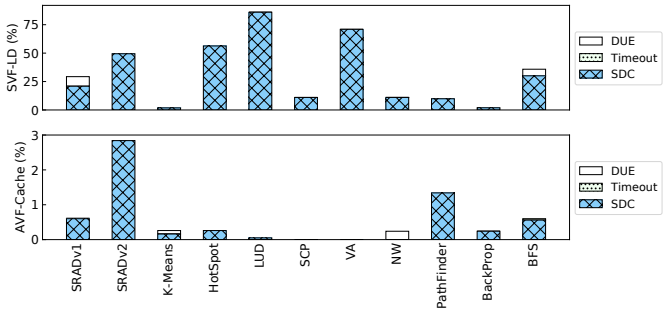


Fig. 5. AVF-Cache (bottom) and SVF-LD (top) comparison.

Insight #3: The distribution of faults is of paramount importance since it can significantly affect the overall estimation and guide the final resilience assessment. SVF may lead to diverging reliability measurement results because only the software-level masking effect is evaluated.

2) AVF-Cache vs. SVF-LD: For the *AVF-Cache* experiments we consider the L1 data caches, the L1 texture caches, and L2 caches. Faults in these hardware components are related to faults on memory operations that are visible (and thus, accessible for fault injection) at the software level. By injecting faults into load instructions, we can obtain memory-related SVF (i.e., *SVF-LD*). Figure 5 shows the comparison between AVF-Cache and SVF-LD. For memory-related operations, AVF and SVF become more erratic compared to the register file-only comparisons. 58% of the total pairs of benchmarks (see also the fourth line of Table I) result in opposite vulnerability trends.

Insight #4: Sub-metrics of AVF and SVF still exhibit consistent and opposite trends. SVF remains a misleading measure of error resilience. Only the full system, cross-layer AVF measurement can deliver accurate comparison among workloads vulnerabilities to faults.

IV. Comparison Between AVF and SVF using a Hardening Method

The final goal of reliability measurement and study is always to fortify applications and systems against hardware faults. In this section, we conduct a case study using both AVF and SVF methodologies to measure the effectiveness

of protection mechanisms. The aspiration herein is that the diverging trends observed in the unprotected system analysis and discussed in the previous section will be less dramatic. We implement a powerful (and thus high-cost) software-level hardening mechanism, Triple Modular Redundancy (TMR) [51], on all application kernels. Both AVF and SVF are used to measure the resilience of the protected application kernels.

A. Hardening Method and Implementation

We start with discussing the choice of hardening method to be examined. Redundancy can be deployed at either the hardware or software level. Hardware redundancy has to be implemented in the simulator used for cross-layer vulnerability assessment (AVF analysis) and on the physical GPU used for software-level fault injection. In short, hardware redundancy on these two deployments needs to be implemented separately, which introduces additional threats to validity. For a fair comparison, we implement TMR at thread level into the application source code, so that the same hardened application is evaluated for its AVF (using the GPGPU-Sim 4.0 [44]) and SVF (using the NVBitFI [25], [27]).

Figure 6 shows the workflow of TMR application hardening. The dark green color in Figure 6 shows the original execution of the application or the kernel without hardening. The figure also illustrates the additional steps for hardening, as enumerated below:

1) *Pre-processing.* Two more copies of input data are added for redundancy.
2) *Kernel Execution.* The number of threads of each application is triplicated, i.e., the same execution is performed three times in total (in parallel).
3) *Post-processing.* From Step 2, three copies of outputs are generated from three identical executions. Majority voting is then used to determine the final (correct) output. If one of the three executions in Step 2 is corrupted and results in incorrect output, the other two copies still have the correct output.

B. Resilience of Hardened Kernels

In this section, we discuss the vulnerability of the hardened application kernels. Figure 7 shows the AVF and SVF of the kernels with and without hardening. For most of the kernels, both AVF and SVF are improved when applying the software-based hardening method, i.e., increased application resilience. However, several kernels show increased vulnerability when applying the software-based hardening method. Specifically, BackProp K2 and SRADv1 K2 show increased AVF compared to the unprotected kernels; BackProp K1, SRADv1 K2, and SRADv1 K3 show increased SVF compared to the unprotected kernels. Here we point out several opposite trends: 1) SRADv1 K3 where the SVF increases but AVF remains at the same levels, 2) BackProp K1 with increasing SVF and decreased AVF, and 3) BackProp K2 where the SVF is slightly decreasing but AVF is much higher after hardening. Clearly, measuring the resilience using the SVF can provide a completely *wrong* indication of improved reliability. Nevertheless, applying TMR

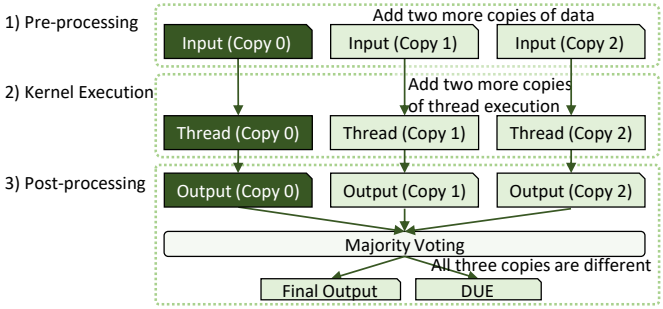


Fig. 6. Triple Modular Redundancy (TMR) workflow.

significantly increases execution time (around $3x$); hence, the application has a higher chance of encountering soft errors. This emphasizes again the importance of correct reliability evaluation and decision.

The purpose of TMR is to correct SDC fault effects. SVF shows that the SDCs (Silent Data Corruptions) are effectively eliminated by TMR. Turning into AVF, we see that there is a considerable number of SDCs even after hardening, see Figure 8. For most kernels, the percentage of SDCs decreases after hardening, but surprisingly this is not the case considering the AVF of SRADv1 K2.

We further elaborate on the faults in some kernels that result in SDCs in AVF after hardening. We note that these faults cannot be detected by any software-based hardening method, because they are hardware-induced faults that cannot be visible to the software. Assume for example that a fault occurs on a cache line that contains data that are part of the application output. If the data of the cache line are not used again by the application (i.e., they are not read again by an instruction), they will be eventually written back to memory without ever being read again by the program flow, and thus, there is no further masking opportunity neither at the hardware nor at the software layer. Since these data are part of the program output, the output will be certainly corrupted (i.e., result in SDC). SVF methods cannot model or evaluate such kinds of faults, since they are unknown to the software layer. This is the reason that while AVF shows the remaining SDCs after hardening, the SVF shows that SDCs are eliminated. This particular phenomenon has been identified in CPUs as well [17].

Figure 9 shows the percentage of timeouts and DUEs with and without hardening. There are very few Timeouts for SVF and very few DUEs for AVF (they are hardly visible in the figure). In most of the kernels, the percentage of DUE outcomes increases, because the resource usage of the application is increasing. For example, the memory usage is triplicated, leading to more "illegal memory accesses" classified as DUE outcomes in software-level fault injection, resulting in increased vulnerability in some kernels.

The detailed breakdown of AVF considering different components with and without hardening for several kernels is shown in Figure 10. Due to space constraints, we only present

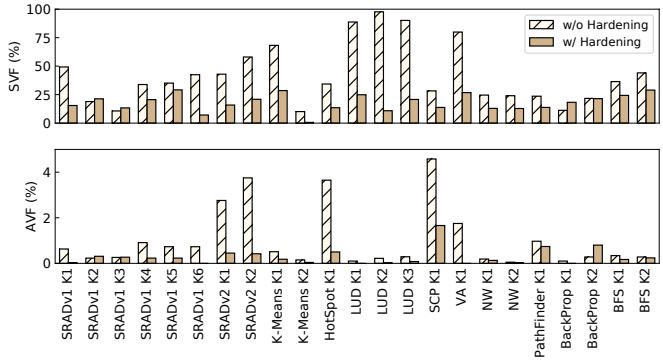


Fig. 7. AVF and SVF of hardened applications.

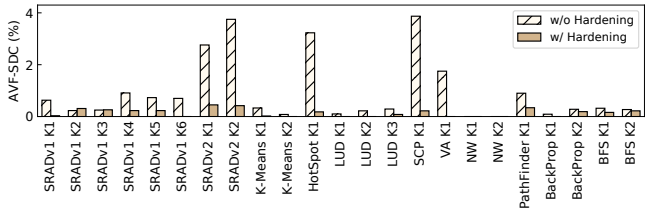


Fig. 8. The percentage of SDC outcomes of AVF for applications with and without hardening.

several representative kernels. For SRADv1 K2, the percentage of SDC outcomes for register files and the shared memory is reduced after hardening, but for the L1 data cache and the L2 cache, there is a higher number of SDC outcomes, which contributes to increased SDCs in the kernel AVF due to the hardening. Note that when applying a software-based hardening method, the reliability characteristics of an application or a kernel are completely different compared to the unhardened one. For example, in SCP K1, the SDC and Timeout in register files and shared memory contribute to the main source of SDC and Timeout in Figure 8 and Figure 9 before hardening; with hardening applied, the main source of SDC and Timeout is from L2 caches. This case illustrates that being agnostic of the underlying hardware structures is a clear

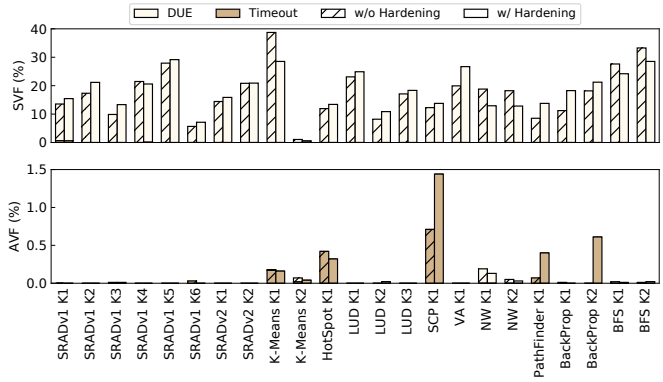
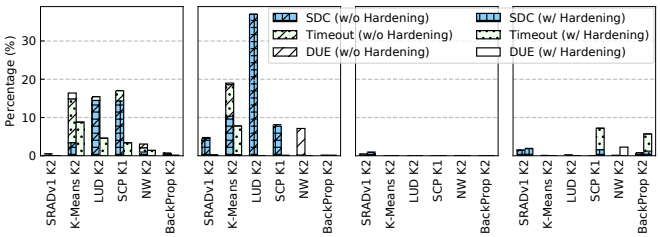


Fig. 9. The percentage of timeout and DUE outcomes of AVF and SVF for applications with and without hardening. Note that there are very few Timeouts for SVF and very few DUEs for AVF (they are hardly visible in the figure).



(a) Register files. (b) Shared memory. (c) L1 data caches. (d) L2 caches.

Fig. 10. AVF of different components before and after hardening.

shortcoming of SVF.

Comparing these four hardware structures, register files and shared memory have an increased probability of getting SDCs without hardening. Therefore, the improvement of TMR mostly falls in register files and shared memory, which confirms its ability to correct SDC outcomes. Figure 10(c) shows that L1 data caches have the smallest vulnerability across all considered hardware structures. In Figure 10(d), we can see that hardening introduces extra vulnerabilities in L2 caches, for example, the increased SDCs in SRADv1 K2 and SCP K1, the timeouts in SCP K1 and BackProp K2, and the DUEs in NW K2. This level of detail in reliability assessment can only be given by cross-layer analysis provided by AVF, but not by SVF.

We are also interested in the effect of soft errors on the control path and data path. Profiling every single executed instruction would introduce tremendous overhead in our analysis. To this end, we profile the number of executed cycles for the hardened kernel and use it as a proxy to track the change of control path. For the data path, the final output is the critical data, so the fault injection outcome, i.e., masked or corrupted, represents the status of the data path. Figure 11 shows the percentage of control-path-affected *masked* runs for each kernel with or without hardening. For most of the kernels, the percentage of control-path-affected masked runs increases after hardening, except for one outlier, SRADv1 K3. This observation shows that hardening is able to correct many control-path-affected runs and maintain the correctness of the data path.

Insight #5: Although software-level evaluation confirms that SDCs are effectively eliminated, the cross-layer evaluation shows that some SDCs remain despite the heavy penalty of protection. While most of the SDCs are eliminated, Detected Unrecoverable Errors (DUEs) instead increase, resulting frequently in higher vulnerability of the heavily protected application compared to the unprotected one.

V. Reasoning about Diverging Results

The endeavor of GPU soft error vulnerability evaluation is very challenging due to the massiveness of the hardware and the highly parallel nature of workloads executed on them. Vulnerability assessment methods considering hardware faults for GPUs focus on one of the fundamental abstraction layers:

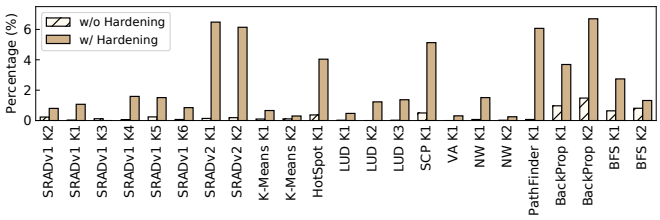


Fig. 11. Control-path affected masked runs for microarchitecture-level fault injection.

either at the microarchitecture or at the software layer. The main difference among the evaluation methods is the presumed origin of the fault: the microarchitecture level starts from the actual hardware bits, while the software level starts even higher from the (destination) register values of a single dynamic (i.e., currently executed) instruction. Apart from the fault origin, the fault injection and simulation methodology followed by any methodology are the same. They both assume a flipped bit as the fault origin (in a certain abstraction layer) and simulate an application to check if the fault affects the eventual output of the program (SDC) or if it affects the program execution before any output is generated (Crash, DUE).

In this section, we discuss two major aspects of soft error vulnerability assessment for GPUs. Firstly, we clarify, for the first time, the difference between microarchitecture-level and software-level fault injection in GPUs. Secondly, we list the most important reasons that, according to our study, higher-level fault injection methods fail to provide correct results for reliability evaluation.

A. Microarchitecture- vs. Software-Level Fault Injection

In the previous sections, we demonstrated that microarchitecture-level and software-level fault injection may result in dramatically different resilience estimations for the same application with respect to both actual values and relative trends. Such discrepancies stem from inherent assumptions of the respective fault injection methodologies.

Microarchitecture-level fault injection assumes that the origin of the fault is any hardware bit. These hardware bits can be residing in either architecturally visible or non-architecturally visible locations. Architecturally visible locations are a subset of the software resources (registers and memory) that are used by a program. For example, since not the entire memory address space is used by an application, only a part of the available memory is used and is visible. In addition, the executed instructions themselves and their operand fields, as well as the data transactions between registers and memory, are architecturally visible.

On the other hand, software-level fault injection assumes that the origin of the fault is any directly addressable resource of the software, which is a subset of architecturally visible resources. For example, software-level fault injection tools (e.g., GPU-Qin [14], SASSIFI [12], and NVBitFI [25]) inject faults only at the values of currently used registers by the program. Such faults can affect the computational results or the temporal memory values (e.g., the fault affects the register value in a load instruction). To the best of our knowledge, there is no software-level fault injection tool that considers faults in opcode or register operands. This means that, by definition, software-level fault injection (and equivalently SVF) only considers a subset of architecturally visible faults.

B. Sources of the Measurement Error of Software-Level Fault Injection

As discussed earlier, SVF provides diverging reliability evaluation results compared to the ground-truth cross-layer AVF. In this section, we analyze the main reasons that SVF fails to provide correct estimation results and propose some potential solutions that could significantly improve SVF estimation.

A major source of the measurement error of software-level fault injection is that it only considers instantaneous faults when a single instruction is executed, but not the effect of multiple accesses of a bit flipped by a transient fault. By flipping a bit in a destination register of an executed instruction, the evaluation fails to assess potential repetitive corruptions of following executed instructions, which depend on this bit flip. For example, assume a hardware-induced fault that affects a single bit of a register, the register R0 in instruction #4 of Figure 12. Instruction #5 and #7 both read from this corrupted register and would be affected by the same fault. This aspect is completely ignored by software-level fault injection studies, but microarchitecture-level fault injection methods cover such a scenario by definition. A potential solution to this problem is to enable fault injection into source registers and augment the software-level fault injection tools with a register reuse analyzer, along with fault injection. The register reuse analyzer can be implemented at the compiler level, and a possibility is to integrate it with the fault injector built on top of LLVM (LLFI-GPU [9]).

Figure 12 shows an example of this process. Assume that a fault is injected in register R0 of instruction #4, as shown in Figure 12. In a typical SVF-based methodology, the fault would affect only this instruction, assuming that an SVF methodology injects in source registers as well. However, that register is getting read again by more instructions (i.e., #5 and #7), and thus, the fault should affect all these instructions. Therefore, a register reuse analyzer could contribute to this limitation of the SVF-based methodologies by replicating the fault in any R0 register of any following instruction that attempts to read from this register, until it is written for the first time. The red circles represent all R0 occurrences that need to be affected by the fault.

Another source of error in software-level fault injection is that it completely fails to consider the hardware (microarchitectural) masking effects. Assume, for example, cache line eviction. Any valid cache line is an architecturally visible resource if the same line is not valid on a higher cache level [52]. However, a cache line eviction, which is a normal microarchitectural operation, can immediately change this condition, and a fault that was initially flagged as software visible can turn into software invisible. Assume, that a hardware-

```
#1  [0x00033c08] S2R    R0, SR_CTAID.X
#2  [0x00033c10] S2R    R3, SR_TID.X
#3  [0x00033c18] IMAD   R4, R0, c[0x0][0x14C], R3
#4  [0x00033c20] ISCADD R3, R0, c[0x0][0x140], 0x2
#5  [0x00033c28] ISCADD R2, R0, c[0x0][0x144], 0x2
#6  [0x00033c30] LD.CG  R3, [R3]
#7  [0x00033c38] ISCADD R0, R0, c[0x0][0x148], 0x2
#8  [0x00033c40] LD.CG  R2, [R2]
#9  [0x00033c48] FADD   R3, R0, R2
#10 [0x00033c50] ST     [R0], R3
```

Fig. 12. An example of the register reuse analyzer. A fault in the register R0 should affect every instruction that attempts to read from this register. The analyzer could replicate the fault of instruction #4 into all R0 occurrences until the R0 is written for the first time.

induced fault occurs in the L1 data cache and that the cache line with the corrupted value is evicted. Eviction only happens on cache lines that are not dirty so that the faulty cache line will be never written back. Thus, a load instruction that loads the data value from a lower memory level will retrieve the correct (i.e., non-corrupted) value.

Insight #6: A microarchitecture-dependent resilience measurement is the only solution.
A fault can be initially considered architecturally visible, but it may eventually turn invisible to the architecture, and thus, to the software. This aspect, by definition, changes the distribution of faults that eventually become architecturally visible. As long as software-level fault injection tools do not consider hardware masking and the distribution of faults that eventually become visible to the software, they fail to provide correct reliability estimation results. Consequently, the only solution for this limitation is a microarchitecture-dependent evaluation. The simulation throughput of cross-layer AVF measurements including the microarchitecture and the software is a clear optimization aspect for tools along these lines.

VI. Future Enhancements

In this section, we emphasize the key areas for improvement, for GPU vulnerability studies aligned with our work:

- **Compute Capability**: Small discrepancies in compute capabilities between GPGPU-Sim 4.0 (used for microarchitecture-level fault injection) and NVBitFI (for software-level fault injection) could marginally affect the findings. Aligning compute capabilities closer could refine absolute numbers. AccelSim [53], a trace-based simulator supporting newer compute capabilities, might offer a potential solution, although its current functionality cannot provide a deterministic output file to determine SDCs.
- **GPGPU-Sim 4.0 Simulator**: gpuFI-4, the chosen microarchitecture-level fault injection tool, may introduce biases due to the GPGPU-Sim 4.0 implementation (since it is not an official Nvidia simulator). Despite this, it remains the most faithful open-source simulator resembling GPU hardware design. While lower-level simulators (like RTL) could refine our findings, their use would significantly increase simulation duration, making such a study with realistic duration benchmarks infeasible.
- **NVBitFI Instructions**: NVBitFI performs fault injection on general-purpose instructions exclusively (i.e., not all the instruction types are supported). While this limitation does not cover all instruction types, it cannot impact our analysis and findings, since general-purpose instructions are prevailed.
- **GPU devices:** In this work, the microarchitecture-level and software-level fault injection experiments are performed on two similar but distinct GPU devices, due to the compatibility restrictions of tools. We carefully select the closest pair of GPUs with the same microarchitecture, and the considered hardware structures are all the same. We acknowledge that this could marginally affect some absolute values, but it cannot impact the final relative trends.
- **SVF in error propagation analysis:** Using a single metric, SVF, is misleading in resilience assessment. However, software-level fault injection may still have its value, for example, conducting fast error propagation analysis across instructions to explore software-based protection techniques. The correctness and possibility of using SVF in error propagation analysis is an interesting angle but out of the scope of this work.

Although there is always space for refinements, GPGPU-Sim 4.0 stands as the predominant open-source microarchitecture-level GPU simulator in recent research. Similarly, gpuFI-4 and NVBitFI are the only open-source microarchitecture-level fault injector and an industry-supported tool, respectively, adding credibility to our research on relative vulnerability analysis in modern GPUs.

VII. Related Work

Application resilience has been measured at different levels. In addition to the cross-layer AVF, Sridharan and Kaeli introduced the concept of Program Vulnerability Factor (PVF), which measures the microarchitecture-independent portion of AVF, by considering only the architecturally-visible faults [54]. Fault injection techniques are applied in the CPU domain at different levels to evaluate CPU application resilience [10], [17], [55]–[62]. Fault injection is commonly used to evaluate the resilience of GPGPU applications as well [12], [14], [16], [41], [48], [63]–[67]. Tselonis *et al.* in [15] proposed GUFI on top of GPGPU-Sim [44], to study the reliability of GPGPU applications.

Neutron beam experiments [68]–[78] are used for resilience assessment. Although these experiments can provide accurate results, they are not always feasible, and it is hard to precisely control fault occurrence and analyze error propagation. Li *et al.* study the error propagation of different kernels in GPGPU applications [9]. Trident [59] analyzes error propagation at different levels to predict the percentage of SDC outputs for the whole application and its instructions. G-SEPM [13] incorporates different machine learning models to achieve

accurate and efficient soft error prediction for GPGPU applications. Comparisons of resilience estimation methodologies have been performed in several CPU studies [17], [79]–[82]. The closest study to the one presented here is done in the CPU domain: Papadimitriou *et al.* consider single-bit faults in different hardware components at the microarchitecture, at the ISA level, and at the software level, and identify pitfalls in CPU reliability evaluation [17]. Our study represents the first extensive study comparing resilience estimation outcomes from microarchitecture-level and software-level fault injection on GPUs, highlighting divergent conclusions regarding application resilience derived from SVF. Moreover, contrasting our results with those of [17], we highlight a considerably greater error magnitude in SVF methods on GPUs, indicating a higher frequency of contradictory vulnerability trends compared to ground-truth AVF, especially due to the underutilization of large register files in GPUs. Consequently, software-level vulnerability methods are more prone to yield inaccurate estimation results for GPU applications compared to CPU applications.

VIII. Conclusion

In this paper, we extensively measured and analyzed transient fault effects on NVIDIA GPUs examining both microarchitecture and software levels. Our key finding emphasizes the discrepancy between software-level vulnerability assessments and their accuracy when compared to microarchitecture-level methods for applications. To delve deeper into this discovery, we conducted a case study evaluating the efficacy of thread triplication — a potent yet costly protection mechanism—using both SVF and accurate cross-layer evaluation (AVF). The insights gained are enlightening: (a) software-level evaluation indicates triplication effectively eliminates SDCs, but cross-layer evaluation reveals the opposite for certain benchmarks — they become more vulnerable despite the heavy protection, and (b) while SDCs are eliminated, the probability of other critical fault effects, like DUEs, impacting application reliability, may significantly increase. We elaborated on why neglecting underlying hardware in software-level fault injection yields divergent outcomes, emphasizing the necessity of microarchitecture-aware evaluations for precise GPU vulnerability assessment. This study highlights the inconsistencies in reliability assessment methodologies and paves the way for addressing this cross-layer gap.

Acknowledgments

This material is based upon work supported by the National Science Foundation (NSF) grants (#2402940 and #2417718) and the Commonwealth Cyber Initiative (CCI) grant (#HC-3Q24-047). This work was performed in part using computing facilities at William & Mary. Jog contributed to this work primarily while he was with William & Mary. For the University of Athens authors, 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 101097224 (REBECCA) and No 101070238 (NEUROPULS). 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.

References

[1] A. Arunkumar, E. Bolotin, D. Nellans, and C.-J. Wu, "Understanding the future of energy efficiency in multi-module GPUs," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 519–532.

[2] J. S. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford *et al.*, "Keeneland: Bringing heterogeneous gpu computing to the computational science community," *Computing in Science & Engineering*, vol. 13, no. 05, pp. 90–95, 2011.

[3] V. Oles, A. Schmedding, G. Ostrouchov, W. Shin, E. Smirni, and C. Engelmann, "Understanding gpu memory corruption at extreme scale: The summit case study," in *Proceedings of the 38th ACM International Conference on Supercomputing*, 2024, pp. 188–200.

[4] D. Tiwari, S. Gupta, G. Gallarno, J. Rogers, and D. Maxwell, "Reliability lessons learned from GPU experience with the Titan supercomputer at Oak Ridge leadership computing facility," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, 2015, pp. 38:1–38:12. [Online]. Available: https://doi.org/10.1145/2807591.2807666

[5] B. Nie, D. Tiwari, S. Gupta, E. Smirni, and J. H. Rogers, "A large-scale study of soft-errors on GPUs in the field," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 519–530.

[6] B. Nie, J. Xue, S. Gupta, C. Engelmann, E. Smirni, and D. Tiwari, "Characterizing temperature, power, and soft-error behaviors in data center systems: Insights, challenges, and opportunities," in *MASCOTS 2017*, 2017, pp. 22–31.

[7] J. Athavale, A. Baldovin, R. Graefe, M. Paulitsch, and R. Rosales, "Ai and reliability trends in safety-critical autonomous systems on ground and air," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2020, pp. 74–77.

[8] P. Rech, T. D. Fairbanks, H. M. Quinn, and L. Carro, "Threads distribution effects on graphics processing units neutron sensitivity," *IEEE Transactions on Nuclear Science*, vol. 60, no. 6, pp. 4220–4225, 2013.

[9] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose, "Understanding error propagation in GPGPU applications," in *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 2016, pp. 240–251.

[10] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 101.

[11] 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.

[12] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "SASSIFI: Evaluating resilience of GPU applications," in *Proceedings of the Workshop on Silicon Errors in Logic-System Effects*, 2015.

[13] H. Yue, X. Wei, G. Li, J. Zhao, N. Jiang, and J. Tan, "G-SEPM: building an accurate and efficient soft error prediction model for GPGPUs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.

[14] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 221–230.

[15] S. Tselonis and D. Gizopoulos, "GUFI: A framework for GPUs reliability assessment," in *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 90–100.

[16] B. Nie, L. Yang, A. Jog, and E. Smirni, "Fault site pruning for practical reliability analysis of GPGPU applications," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 749–761.

[17] G. Papadimitriou and D. Gizopoulos, "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.

[18] ——, "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.

[19] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE, 2003, pp. 29–40.

[20] H. Jeon, M. Wilkening, V. Sridharan, S. Gurumurthi, and G. H. Loh, "Architectural vulnerability modeling and analysis of integrated graphics processors," in *Silicon Errors in Logic – System Effects (SELSE)*, 2013.

[21] N. Farazmand, R. Ubal, and D. Kaeli, "Statistical fault injection-based AVF analysis of a GPU architecture," in *Silicon Errors in Logic – System Effects (SELSE)*, 2012.

[22] J. Tan, N. Goswami, T. Li, and X. Fu, "Analyzing soft-error vulnerability on GPGPU microarchitecture," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, 2011, pp. 226–235.

[23] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2009, pp. 502–506.

[24] C. Avalos Baddouh, M. Khairy, R. N. Green, M. Payer, and T. G. Rogers, "Principal kernel analysis: A tractable methodology to simulate scaled GPU workloads," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 724–737. [Online]. Available: https://doi.org/10.1145/3466752.3480100

[25] T. Tsai, S. K. S. Hari, M. Sullivan, O. Villa, and S. W. Keckler, "NVBitFI: dynamic fault injection for gpus," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 284–291.

[26] (2022) gpuFI-4 github repository. [Online]. Available: https://github.com/caldi-uoa/gpuFI-4

[27] (2021) NVBitFI github repository. [Online]. Available: https://github.com/NVlabs/nvbitfi

[28] J. Chen, S. Li, and Z. Chen, "GPU-ABFT: Optimizing algorithm-based fault tolerance for heterogeneous systems with GPUs," in *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 2016, pp. 1–2.

[29] 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, pp. 1–12.

[30] 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.

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

[32] D. Gizopoulos, G. Papadimitriou, A. Chatzidimitriou, V. J. Reddi, B. Salami, O. S. Unsal, A. C. Kestelman, and J. Leng, "Modern hardware margins: Cpus, gpus, fpgas recent system-level studies," in *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2019, pp. 129–134.

[33] G. Papadimitriou, A. Chatzidimitriou, D. Gizopoulos, V. J. Reddi, J. Leng, B. Salami, O. S. Unsal, and A. C. Kestelman, "Exceeding conservative limits: A consolidated analysis on modern hardware margins," *IEEE Transactions on Device and Materials Reliability*, vol. 20, no. 2, pp. 341–350, 2020.

[34] P. Koutsovasilis, C. D. Antonopoulos, N. Bellas, S. Lalis, G. Papadimitriou, A. Chatzidimitriou, and D. Gizopoulos, "The impact of cpu voltage margins on power-constrained execution," *IEEE Transactions on Sustainable Computing*, vol. 7, no. 1, pp. 221–234, 2022.

[35] E. Ibe, H. Taniguchi, Y. Yahagi, K.-i. Shimbo, and T. Toba, "Impact of scaling on neutron-induced soft error in SRAMs from a 250 nm to a 22 nm design rule," *IEEE Transactions on Electron Devices*, vol. 57, no. 7, pp. 1527–1538, 2010.

[36] 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)*. IEEE, 2019, pp. 119–130.

[37] L. Yang, B. Nie, A. Jog, and E. Smirni, "Practical resilience analysis of GPGPU applications in the presence of single- and multi-bit faults," *IEEE Transactions on Computers*, vol. 70, no. 1, pp. 30–44, 2021.

[38] S. Mittal, H. Wang, A. Jog, and J. S. Vetter, "Design and analysis of soft-error resilience mechanisms for GPU register file," in *2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID)*. IEEE, 2017, pp. 409–414.

[39] M. M. Goncalves, I. P. Lamb, P. Rech, R. M. Brum, and J. R. Azambuja, "Improving selective fault tolerance in GPU register files by relaxing application accuracy," *IEEE Transactions on Nuclear Science*, vol. 67, no. 7, pp. 1573–1580, 2020.

[40] G. Kadam, E. Smirni, and A. Jog, "Data-centric reliability management in GPUs," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 271–283.

[41] L. Yang, B. Nie, A. Jog, and E. Smirni, "Enabling software resilience in GPGPU applications via partial thread protection," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1248–1259.

[42] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (DNN) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[43] L. M. Leemis and S. K. Park, *Discrete-event simulation: A first course*. Pearson Prentice Hall Upper Saddle River, NJ, 2006, pg. 366.

[44] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 163–174.

[45] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 372–383.

[46] NVIDIA. (2011) CUDA C/C++ SDK Code Samples. [Online]. Available: http://developer.nvidia.com/cuda-cc-sdk-code-samples

[47] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[48] L. Yang, B. Nie, A. Jog, and E. Smirni, "SUGAR: Speeding up GPGPU application resilience estimation with input sizing," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 5, no. 1, pp. 1–29, 2021.

[49] A. R. Anwer, G. Li, K. Pattabiraman, M. Sullivan, T. Tsai, and S. K. S. Hari, "GPU-Tiident: efficient modeling of error propagation in GPU programs," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[50] M. H. Rahman, A. Shamji, S. Guo, and G. Li, "Peppa-x: finding program test inputs to bound silent data corruption vulnerability in hpc applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–13.

[51] A. Milluzzi and A. George, "Exploration of TMR fault masking with persistent threads on Tegra GPU SoCs," in *2017 IEEE Aerospace Conference*. IEEE, 2017, pp. 1–7.

[52] V. Sridharan and D. R. Kaeli, "Using hardware vulnerability factors to enhance AVF analysis," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 461–472. [Online]. Available: https://doi.org/10.1145/1815961.1816023

[53] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated gpu modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[54] V. Sridharan and D. R. Kaeli, "Quantifying software vulnerability," in *Proceedings of the 2008 Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies*. ACM, 2008, pp. 323–328.

[55] J. Calhoun, L. Olson, and M. Snir, "FlipIt: An LLVM based fault injector for HPC," in *European Conference on Parallel Processing*. Springer, 2014, pp. 547–558.

[56] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *ACM SIGPLAN Notices*, vol. 47, no. 4. ACM, 2012, pp. 123–134.

[57] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–14.

[58] M. Kaliorakis, D. Gizopoulos, R. Canal, and A. Gonzalez, "Merlin: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 241–254.

[59] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, "Modeling soft-error propagation in programs," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 27–38.

[60] D. Chen, G. Jacques-Silva, Z. Kalbarczyk, R. K. Iyer, and B. Mealey, "Error behavior comparison of multiple computing systems: A case study using linux on pentium, solaris on sparc, and aix on power," in *Dependable Computing, 2008. PRDC'08. 14th IEEE Pacific Rim International Symposium on*. IEEE, 2008, pp. 339–346.

[61] K. S. Yim, Z. Kalbarczyk, and R. K. Iyer, "Measurement-based analysis of fault and error sensitivities of dynamic memory," in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*. IEEE, 2010, pp. 431–436.

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

[63] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight silent data corruption error detector for GPGPU," in *2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2011, pp. 287–300.

[64] F. G. Previlon, C. Kalra, D. Tiwari, and D. R. Kaeli, "PCFI: Program counter guided fault injection for accelerating GPU reliability assessment," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 308–311.

[65] H. Sarbazi-Azad, *Advances in GPU Research and Practice*. Morgan Kaufmann, 2016.

[66] B. Nie, A. Jog, and E. Smirni, "Characterizing accuracy-aware resilience of GPGPU applications," in *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID 2020, Melbourne, Australia, May 11-14, 2020*. IEEE, 2020, pp. 111–120.

[67] A. Vallero, D. Gizopoulos, and S. Di Carlo, "SIFI: AMD southern islands GPU microarchitectural level fault injector," in *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, 2017, pp. 138–144.

[68] T. Santini, L. Carro, F. R. Wagner, and P. Rech, "Reliability analysis of operating systems and software stack for embedded systems," *IEEE Transactions on Nuclear Science*, vol. 63, no. 4, pp. 2225–2232, 2016.

[73] 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

[69] Á. B. de Oliveira, G. S. Rodrigues, and F. L. Kastensmidt, "Analyzing lockstep dual-core arm cortex-a9 soft error mitigation in freertos applications," in *Proceedings of the 30th Symposium on Integrated Circuits and Systems Design: Chip on the Sands*, 2017, pp. 84–89.

[70] 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.

[71] A. Martínez-Álvarez, F. Restrepo-Calle, S. Cuenca-Asensi, L. M. Reyneri, A. Lindoso, and L. Entrena, "A hardware-software approach for on-line soft error mitigation in interrupt-driven applications," *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 4, pp. 502–508, 2014.

[72] V. Fratin, D. Oliveira, C. Lunardi, F. Santos, G. Rodrigues, and P. Rech, "Code-dependent and architecture-dependent reliability behaviors," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 13–26.
2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 2019, pp. 26–38.

[74] F. F. dos Santos, L. Draghetti, L. Weigel, L. Carro, P. Navaux, and P. Rech, "Evaluation and mitigation of soft-errors in neural network-based object detection in three GPU architectures," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2017, pp. 169–176.

[75] C. Lunardi, F. Previlon, D. Kaeli, and P. Rech, "On the efficacy of ECC and the benefits of FinFET transistor layout for GPU reliability," *IEEE Transactions on Nuclear Science*, vol. 65, no. 8, pp. 1843–1850, 2018.

[76] S. K. S. Hari, P. Rech, T. Tsai, M. Stephenson, A. Zulfiqar, M. Sullivan, P. Shirvani, P. Racunas, J. Emer, and S. W. Keckler, "Estimating silent data corruption rates using a two-level model," *arXiv preprint arXiv:2005.01445*, 2020.

[77] 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.

[78] D. Agiakatsikas, G. Papadimitriou, V. Karakostas, D. Gizopoulos, M. Psarakis, C. Belanger-Champagne, and E. Blackmore, "Impact of voltage scaling on soft errors susceptibility of multicore server cpus," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 957–971. [Online]. Available: https://doi.org/10.1145/3613424.3614304

[79] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 375–382.

[80] H. Schirmeier and M. Breddemann, "Quantitative cross-layer evaluation of transient-fault injection techniques for algorithm comparison," in *2019 15th European Dependable Computing Conference (EDCC)*. IEEE, 2019, pp. 15–22.

[81] A. Tyagi, Y. Gan, S. Liu, B. Yu, P. Whatmough, and Y. Zhu, "Thales: Formulating and estimating architectural vulnerability factors for dnn accelerators," *arXiv preprint arXiv:2212.02649*, 2022.

[82] Z. He, Y. Huang, H. Xu, D. Tao, and G. Li, "Demystifying and mitigating cross-layer deficiencies of soft error protection in instruction duplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–13.