Thales: Formulating and Estimating Architectural Vulnerability Factors for DNN Accelerators

Abhishek Tyagi
University of Rochester
atyagi2@ur.rochester.edu

Yiming Gan
University of Rochester
ygan10@ur.rochester.edu

Shaoshan Liu
PerceptIn
shaoshan.liu@perceptin.io

Bo Yu
PerceptIn
bo.yu@perceptin.io

Paul Whatmough
Arm Research
paul.whatmough@arm.com

Yuhao Zhu
University of Rochester
yzhu@rochester.edu

***Abstract*—As Deep Neural Networks (DNNs) are increasingly deployed in safety-critical and privacy-sensitive applications such as autonomous driving and biometric authentication, it is critical to understand the fault-tolerance nature of DNNs. Prior work primarily focuses on metrics such as Failures In Time (FIT) rate and the Silent Data Corruption (SDC) rate, which quantify *how often* a device fails. Instead, this paper focuses on quantifying the DNN accuracy *given* that a transient error has occurred, which tells us how well a network behaves when a transient error occurs. We call this metric Resiliency Accuracy (RA).**

We show that existing RA formulation is fundamentally inaccurate, because it incorrectly assumes that software variables (model weights/activations) have equal faulty probability under hardware transient faults. We present an algorithm that captures the faulty probabilities of DNN variables under transient faults and, thus, provides correct RA estimations validated by hardware. To accelerate RA estimation, we reformulate RA calculation as a Monte Carlo integration problem, and solve it using importance sampling driven by DNN-specific heuristics.

Using our lightweight RA estimation method, we show that transient faults lead to far greater accuracy degradation than what today's DNN resiliency tools estimate. We show how our RA estimation tool can help design more resilient DNNs by integrating it with a Network Architecture Search framework.

I. Introduction

Deep neural networks (DNNs) have become integral components in many safety-critical and/or privacy-sensitive application domains. Thus, the fault tolerance of a deep learning system has become crucial. Among many sources of vulnerability, a major concern is transient errors, which are radiation-induced accidental bit flips [56], [71], [72], [77]. While transient bit flips can occur anywhere on a chip [16], [17], [34], [42], [54], they are particularly detrimental to flip-flops (FFs); other memory structures (SRAMs and DRAMs) are usually protected by error correction codes [9], [75].

Prior work primarily focuses on metrics such as Failures In Time (FIT) rate and the Silent Data Corruption (SDC) rate, which quantify *how often* a device fails. Instead, this paper focuses on quantifying the DNN accuracy *given* that a transient error has occurred, which tells us how well a network behaves when a transient error occurs. We call this metric Resiliency Accuracy (RA), the counterpart of the network's fault-free, Standard Accuracy (SA).

The RA metric can be seen as the Architectural Vulnerability Factor (AVF) [43], [53] of a DNN accelerator. As traditionally defined (for CPU and GPU architectures), the AVF of a fault site (i.e., a single bit in a single cycle) is either 0 (the value of that bit in that cycle does not affect the correctness of the program) or 1 (the value of that bit in that cycle does affect the correctness of the program); this is averaged across all bits and cycles to get an average AVF for an application. In contrast, the AVF of a fault site for a DNN accelerator is not binary, because DNN's accuracy (over a test set) is a percentage. Thus, a DNN accelerator's AVF is a spectrum between 0 and 1.

Prior work conducted real beam experiments on GPUs [21] and TPUs [67] to demonstrate the DNN accuracy loss under transient faults. The focus of our work is to show an approach that estimates RA using fault injections. Existing DNN fault injection tools such as PyTorchFI [50], Ares [66], TensorFI [13] inject faults into DNN variables (i.e., weights and activations) and average the resulting inference accuracies. This approach resembles the classic Software Vulnerability Factor (SVF) analysis [49], [76], [81], and does not accurately reflect the AVF because it fundamentally assumes that DNN variables are equally faulty. We show that the faulty probability varies by orders of magnitude across DNN variables (Sec. II), leading to the mismatch between AVF and SVF in DNNs; our results corroborate and complement the CPU-centric results in Papadimitriou and Gizopoulos [56].

Following how AVF is defined for classic architectures, we define RA as an *expected value* of the DNN's inference accuracy, which weighs the inference accuracy by how probable a software variable receives a transient fault at each cycle. Crucially, each software variable has a potentially different faulty probability. In resiliency parlance, our RA metric is AVF modulated by the impact of the bit flip on DNN accuracy.

We then describe how the new RA metric can be analytically derived from simple statistics of a DNN and the accelerator. This is achieved by leveraging the regular data reuse pattern in DNNs to map a software variable to hardware FFs over time (Sec. III). We show that our analytical modeling method correlates well with results obtained from large-scale (over 2.6

billion) RTL fault simulations (Sec. V).

Accurately calculating RA using the new formulation is time-consuming: it requires exhaustively enumerating all the bits in all the model weights/activations, each of which requires inferencing over an entire test set. We resort to sampling. Critically, the faulty probabilities in software vary significantly; uniform sampling, a common strategy used in prior work, would lead to significant estimation variance or require an excessive amount of samples to converge.

We propose a sampling strategy that estimates the true RA with several orders of fewer samples (Sec. IV). The key is to formulate RA estimation as a Monte Carlo integration problem [61]. Taking this perspective, we propose to use importance sampling [37] to reduce estimation variance and accelerate convergence. We show that a perfect estimator based on important sampling is impossible, and propose heuristics that leverage DNN-specific characteristics to approximate importance sampling while retaining the main benefit. We show that our importance sampling method significantly accelerates the convergence rate of RA estimation (Sec. VI).

We apply our RA formulation and estimation method for a set of case studies. For instance, we show that transient errors lead to much a higher DNN accuracy degradation compared to what today's RA formulation estimates; selectively hardening control FFs improves RA the most (Sec. VII). We also show how our RA formulation helps design DNNs that are more resilient to transient faults by integrating our RA estimation into a Network Architecture Search (NAS) framework (Sec. VIII).

In summary, our paper makes the following contributions:

- We propose a new formulation of Resiliency Accuracy (RA) to quantify a DNN's accuracy under transient faults.
- We propose a method to estimate RA in software. The method formulates RA estimation as a Monte Carlo integration problem and uses DNN-specific importance sampling to solve the integration problem.
- We apply our RA formulation to common DNNs and show that their accuracy loss is far greater than what is estimated by today's RA metric.
- We demonstrate how our RA formulation can help design more resilient DNNs when coupled with NAS.

II. Background and Motivation

We first define the scope of transient faults this paper focuses on (Sec. II-A). We then discuss the intuition behind needing a resiliency accuracy metric (Sec. II-B), followed by describing the common formulation of the metric in existing tools (Sec. II-C). We quantitatively analyze why the existing formulation is fundamentally flawed (Sec. II-D).

II-A. Scope and Assumptions

Consistent with recent prior work such as FIdelity [29] and Mahmoud et al. [51], we consider only single-bit errors in FFs, which are "*the most prominent abstraction for transient errors including soft errors and voltage variations.*" [29]. We assume that other major memory structures (e.g., DRAM, SRAM) are protected by ECC and/or parity bits [51]. Note that FF faults capture not only transients faults in FFs but also faults taken place in logic computation.

As with Mahmoud et al. [51], we also assume that different bit positions in an FF have the same raw FIT rate[1]. In real designs, some FF bits (e.g., exponent in float) could be hardened to resist transient errors [38]. Note, however, that we do *not* assume that all the FFs have the same fault probability. In fact, each FF (or each category of FFs) is characterized by its raw FIT rate, which will participate in our new metric, as we will discuss in Sec. III-A.

II-B. Why a New Metric?

Failures In Time (FIT) rate and the Silent Data Corruption (SDC) rate of a device (accelerator) are common metrics used in the fault tolerance literature for quantifying the impact of transient errors [12], [22], [27]. These metrics quantify *how often* a device fails. In the context of DNN accelerators, an SDC is an inference mis-prediction [13], and the FIT rate calculation considers not only mis-predictions but also system anomalies such as a crash [29].

This paper focuses on an orthogonal metric, which quantifies the accuracy of a DNN *given* that a transient error has occurred. We call this metric Resiliency Accuracy (RA), the counterpart of the network's fault-free inference accuracy, which we call Standard Accuracy (SA). Intuitively, RA tells us how well/badly a network behaves *when* a transient error occurs. Knowing RA lets us develop DNN algorithms and accelerators that can still perform reasonably well even when the execution is "corrupted" by soft errors.

II-C. Existing RA Formulation

In theory, the RA is defined at the hardware level:

$$RA = \frac{1}{N}\sum_{i=1}^{N} A(i) \tag{1}$$

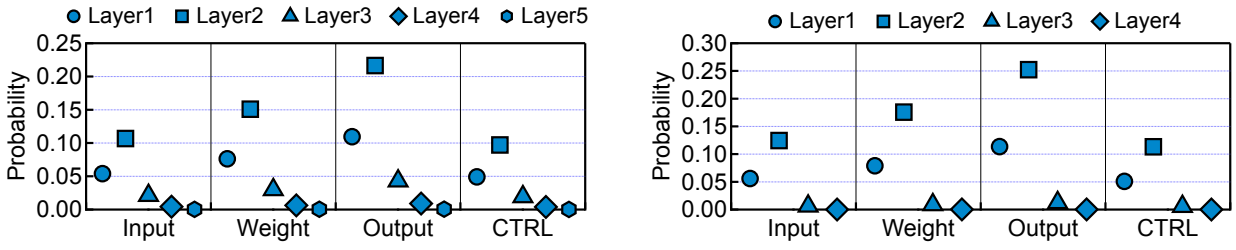
where N is the total number of FFs in the hardware, and $A(i)$ is the resulting inference accuracy under a fault at the hardware fault site i. This equation is infeasible to use in practice due to the need to perform time-consuming RTL fault injections and simulations, which are four to five orders of magnitude slower than an inference in software. Nevertheless, Equ. 1 expresses the ground truth RA of a DNN on a given hardware and serves as a reference that other more lightweight methods should strive to match.

Due to the infeasibility of performing RTL simulations, a common formulation of RA as used in today's DNN resiliency analysis tools such as PyTorchFI [50] and TensorFI [13] is to model soft errors directly in a DNN model. In this method, a fault site is a bit position (BP) of a weight or an activation. Thus, the total number of fault sites is:

$$\widehat{N_{fs}} = (\#of\ weights + \#of\ activations) \times \#of\ BPs$$

Given the software faults sites, RA is estimated by 1) first uniformly-at-random sampling of the software fault sites and,

[1] Section V-A: "*each error injection is performed on a single random bit of a random neuron for a random image.*"



(a) LeNet.

(b) MNIST-Hogwild.

Fig. 1: The probability a transient fault is received by an input activation, a weight, an output activation, or a control variable across layers in LeNet-5 and MNIST-Hogwild. Software variables do not have uniform faulty probabilities. Control variables do not correspond to variables in a DNN model — they are proxies for modeling control FFs; see Sec. III-A for details.

for each sampled fault site, perform an fault injection to obtain the corresponding inference accuracy under that fault injection, and then 2) averaging the resulting inference accuracies across fault injections[2]. This is equivalent to calculating RA by:

$$RA_{SW} = \frac{1}{\widehat{N_{fs}}} \sum_{j=1}^{\widehat{N_{fs}}} A(j) \tag{2}$$

where $A(j)$ is the inference accuracy under a fault at a software fault site j. $A(j)$ is estimated through software fault injection: flipping the bit corresponding to the software fault site j and performing an inference.

II-D. Error Sources in Existing RA Formulation

Existing RA formulation is flawed, because of incorrect assumptions made in both terms in Equ. 2. The issue with the existing RA formulation is a DNN-specific instance of the general AVF-vs-SVF mismatch as analyzed in detail by Papadimitriou and Gizopoulos [56]. Let us elaborate this mismatch in the DNN context.

Inaccurate Inference Accuracy. Estimating $A(j)$ using software fault injection does not reflect accuracy impact of actual soft errors in hardware — for two reasons.

First, software fault injection assumes that, based on the data dependencies, one bit flip in a kernel weight corrupts *all* the output activations. However, the actual number of corrupted activations depends on the reuse factor of the FF that holds the weight: if the weight FF is reused, say, 4 times before a new weight is written to the FF (e.g., due to the scheduling algorithm), the original weight affects only 4 output activations. The same argument applies to activations.

Second, software faults considered in existing tools do not reflect hardware fault sites. For instance, these tools make no distinction between an output activation and its corresponding input activation in the next layer (because numerically they are the same value), but input activations and output activations are stored in two different sets of hardware FFs that might have different faulty probabilities. Software fault injection also does not consider the effects of global *control FFs*, whose bits when corrupted drastically reduce accuracy. When transient faults take place in control FFs, the accelerator usually crashes, severely degrading the inference accuracy [29].

FIdelity [29] establishes a RTL-validated method that addresses the two issues above and, thus, correctly obtains the accurate $A(j)$ from software fault injection. Our work builds on FIdelity to obtain an accurate $A(j)$ estimation when needed, and does *not* claim it as our contribution.

Inaccurate Fault Site Probability. By simply averaging the inference accuracy across all the software fault injections, the RA formulation in Equ. 2 implicitly assumes that all software fault sites are equally probable to transient errors. This assumption, however, is incorrect, because of how software variables are stored and reused in hardware FFs.

To demonstrate this Fig. 1 shows the faulty probability of software fault sites in two DNNs, LeNet-5 [19] and MNIST-Hogwild [57], running on NVDLA. Effectively, we uniformly inject transient faults to hardware and analyze which weights/activations are effected. In the end, we aggregate statistics of the faulty probability of software fault sites. We assume a raw FIT rate of 600/MB following the study from Jagannathan et al. [33] and consistent with FIdelity [29].

The probabilities of software fault sites vary significantly, both across variable type (e.g., input activations vs. weights) and across layers. For instance, the faulty probabilities of variables in the last layer are two orders of magnitude lower than those in earlier layers. This is because weights/activations in layers that take longer time to execute stay in the hardware longer and, thus, are more vulnerable to transient errors.

In addition, Fig. 1 also shows the faulty probability of control variables, which do not correspond to any real variables in a DNN model; rather, they are proxies used to capture control FFs in hardware (see Sec. III-A for details). Those control variables generally have lower faulty probabilities than weights/activations, because there are fewer control FFs than FFs for weights/activations. As a result, they are less likely to receive transient faults than FFs of other categories.

[2]Section IV-A in PyTorchFI [50]: "*In each inference run, we inject a single-bit flip in a randomly selected neuron in the DNN to emulate a computational hardware error that may occur during inference.*". Section IV-A in TensorFI [13]: "*we perform 1000 random FI experiments per fault configuration and input.*"

III. Defining and Modeling RA

We first introduce our RA formulation (Sec. III-A), followed by an algorithm that calculates the faulty probability of any software fault site (Sec. III-B)

III-A. Defining RA as an Expected Value

Recall the intuition behind RA: it should capture the network inference accuracy *given* that a fault has occurred. Borrowing ideas from probability theory, we define RA as the *expected value* [2] of a network's inference accuracy under faults. Following the definition of the expected value, the equation to calculate RA takes the average of the network inference accuracy under each software fault *weighted* by the faulty probability of each software fault site:

$$RA = \sum_{j=1}^{N_{fs}} p(j)A(j) \tag{3}$$

where N_{fs} denotes the total number of software fault sites, $p(j)$ denotes the probability that a software fault site j experiences a transient error, and $A(j)$ denotes the inference accuracy given the transient error at fault site j and can be calculated using the FIdelity framework [29].

The intuition behind our formulation is that a network's RA is a random variable, whose outcome depends on a large number of independent events, i.e., transient faults. Each transient fault affects one software fault site j (e.g., one bit flip in a weight FF affects a weight in the model) with a different probability $p(j)$ and results in an inference accuracy $A(j)$. Note that the expected value equation inherently iterates over all software fault sites, but this is different from saying that each inference will actually have a transient fault in reality.

Our RA metric is nothing more than the *DNN-specific AVF*. For CPUs and GPUs, AVF is defined as the probability (0 or 1) that a transient fault in the hardware leads to an error in the application; the so-derived AVFs of all bits and cycles are averaged to get the overal AVF. The difference in DNNs is that the AVF of a single bit in a single cycle is the DNN inference accuracy under that fault and, thus, can be any value between 0 and 1, which is captured by our $A(j)$. In addition, since we aim to estimate DNN RA/AVF in software, the fault sites are defined at the software level, which we elaborate next.

Software Fault Sites. A software fault site is a 2-tuple $< \mathbf{Var}, \mathbf{BP} >$, where Var is a software variable that can potentially become faulty due to a transient error and BP is a bit position in a Var. In particular, there are four types of software variables we model: 1) input activations, 2) output activations, 3) weights, and 4) control variables. Note that while a layer's output activations are the next layer's input activations, they must be separately modeled because they are held in different FFs in common DNN accelerators and, thus, are independently vulnerable to transient errors. This is one key difference compared to software fault sites considered in the existing RA formulation in Equ. 2.

Another key difference is control variables, which do not correspond to any real variable in a DNN model; rather, they are a proxy to capture FFs that store neither weights nor activations. Control FFs fall into two categories: *global* control FFs that are used for data sequencing and address generation and *local* control FFs that are tightly coupled with the datapath (e.g., valid bit, MUX select bits). Faults in global control FFs generally crash the accelerator, providing no inference result for a particular input. Local control flops affect neurons depending on where they are located. In our model, we assign a software variable to each control FF in hardware.

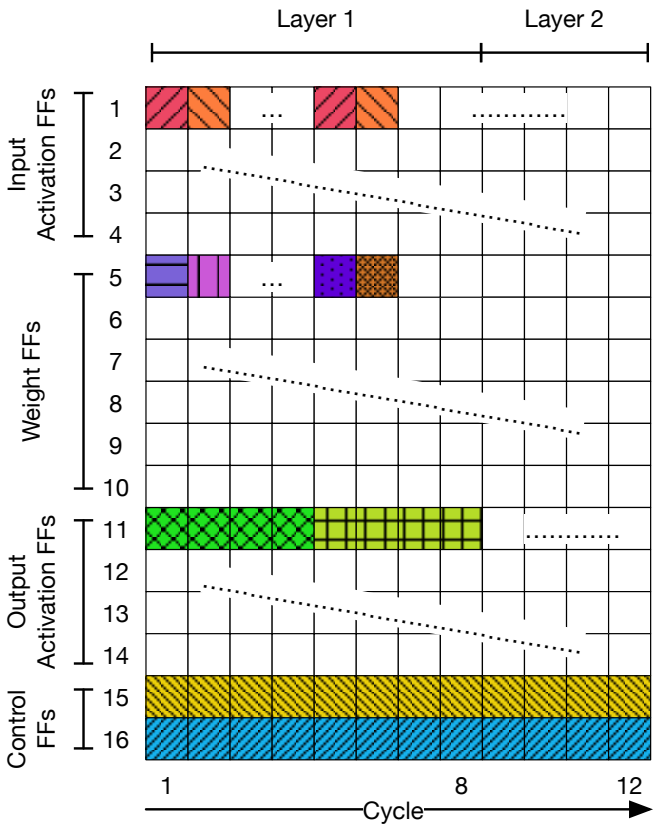


Fig. 2: An (not-to-scale) illustration of the idea of calculating the faulty probability of a software fault site. Each cell represents a hardware fault site (i.e., a particular FF at a particular cycle). Each stripe pattern (color) represents a software variable, which can occupy multiple cells because of data reuse. The goal is to calculate the number of cells each software variable is mapped to.

For simplicity purposes, we omit BP in later discussions and equate a software fault site with a software variable. The faulty probabilities so derived must be multiplied by $\frac{1}{BP}$ to obtained the actual probabilities of software fault sites, given the assumption that the probability of the bit flip occurring at a specific bit is the same across all bit positions in a FF.

III-B. Calculating RA Through Probability Transfer

We present an algorithm to estimate the transient error probability of software fault sites; it works by transferring the faulty probability from the hardware to the software.

Intuition. Our idea is illustrated in Fig. 2, where each row corresponds to a hardware FF and each column, from left to right, represents a cycle over time. Thus, each cell represents a unique hardware fault site (i.e., a particular FF at a particular cycle). A transient error can be thought of as dropping a pin on the 2D grid. The probability of a pin landing at a cell is dictated by the raw FIT rate of the corresponding FF. For instance, if the raw FIT rates for hardware FFs are uniform, the pin has an equal probability of landing at any cell.

The central question is, what are the probabilities of a randomly dropped pin landing on *each of the software fault sites*? The crux is to model *how software fault sites are mapped to the cells*. A useful thought experiment is that if a software fault site j is mapped to $T(j)$ cells out of a total amount of M cells, and each cell has the same probability of receiving the pin (e.g., if hardware FFs have the same raw FIT rate), the probability that j receives a transient error is $\frac{T(j)}{M}$. This can be easily generalized to cases where hardware FFs do not have the same raw FIT rate as we will show later.

Note that each software variable occupies multiple cells because of data reuse. For instance, in a weight stationary accelerator a weight will stay in a FF for C cycles and reused by C input activations (the exact number of C depends on the scheduling/tiling policy). As a result, that weight will occupy C consecutive cells belonging to the holding FF. The same argument applies to activations, too. A control variable stays in a control FF throughout the execution, because each control variable is uniquely associated with a control FF.

Assumptions. To model the mapping, we make the following assumptions about DNN execution generally true to common DNN accelerators:

ASSUMPTION 1 A DNN is executed layer by layer.

ASSUMPTION 2 Each software variable resides in only one type of FFs. For instance, a weight is always found in a weight FF (although it could be found in different weight FFs at different cycles) and is not in an activation FF. In other words, a FF will not hold software variables of different types. While perhaps obvious, we perform a large-scale RTL fault injection (over 2.6 billion in total) on a systolic array accelerator, and inspect the resulting Fast Signal DataBase (FSDB) waveforms, to confirm this. See Sec. V for the setup.

The implication is that, *given* that a transient fault occurs, the probability that the fault occurs to a particular FF type is equivalent to the percentage of the FFs of that type.

ASSUMPTION 3 Different software variables of the same type spend the same amount of time in a FF. For instance, a weight FF throughout the execution will hold different weights, which spend the same number of cycles in that FF. This is fundamentally true since variables of the same type have the same number of reuses in a DNN. Again, from the over 2.6 billion RTL fault injections and simulations above we inspect the FSDB waveforms to confirm this. Across 200,736 weights of a layer, the standard deviation of the number of cycles a weight spent on a weight FF is 0.025 (not exactly 0 due to the ramp-down in the end).

The implication is that *given* that a transient fault occurs in a weight (activation) FF, all the weights (activations) in the same layer are equally probable of receiving that fault.

ASSUMPTION 4 The accelerator utilization is near 100%. That is, every cell is associated with a software fault site throughout the execution (no "empty cell"). This assumption does not hold in extreme cases where the hardware resources are abundant but the network is extremely small (e.g., a 1K $\times$ 1K PE array executing a layer where the feature map dimension is 4 $\times$ 4). At the end of this section, we will relax this constraint and show how RA is estimated with an actual utilization factor.

Derivation. Given the observations above, the probability of a software fault site j can be expressed as:

$$p(j) = T(j) \times p_{T_j} \tag{4}$$

where $T(j)$ denotes the number of cells that a software fault site j (of type T_j in layer L_j) occupies, and p_{T_j} denotes the probability that a cell of type T_j receives a transient fault.

$T(j)$ is further modeled as:

$$T(j) = M \times LP(L_j) \times TP(T_j) \times VP(j) \tag{5}$$

where:

- M is the total number of cells, which is 192 in Fig. 2.
- $LP(L_j)$ is the percentage of the cells used by layer L_j. In the example of Fig. 2, if L_j is the first layer, $LP(L_j)$ would be $\frac{2}{3}$. Thus, $M \times LP(L_j)$ is the total number of cells belonging to layer L_j.
- $TP(T_j)$ denotes the percentage of the FFs of type T_j. It is equivalent to the percentage of the cells that belong to type T_j. In the example of Fig. 2, if T_j is the input activation type, $TP(T_j)$ would be $\frac{1}{4}$. Thus, $M \times LP(L_j) \times TP(T_j)$ is the total number of cells belonging to FF type T_j in layer L_j.
- $VP(j)$ is the probability that a transient fault occurs to a variable j given that a fault has occurred to some variable of type T_j. Thus, $M \times LP(L_j) \times TP(T_j) \times VP(j)$ is the total number of cells belonging to the variable j.

Taking another perspective, Equ. 5 essentially computes the joint probability of three independent events: 1) a fault occurs while layer L_j is executing; the probability of this event is $LP(L_j)$, 2) a fault occurs in an FF that is of type T_j; the probability of this event is $TP(T_j)$, and 3) a fault occurs to a particular variable j among all the variables of the type $T(j)$; the probability of this event is $VP(j)$. Let us now analytically derive each of these terms in Equ. 5.

$LP(L_j)$, given ASSUMPTION 1, is equal to the percentage of the execution time layer L_j takes, which, given ASSUMPTION 4 (the hardware utilization is nearly full), can be modeled as the ratio between the Multiply and Accumulate (MAC) operation count in that layer over the total number of MAC operations in the network:

$$LP(L_j) = \frac{MAC(L_j)}{\sum_{i=1}^{UF(j)+p(j)SA(1-UF(j))L} MAC(L_i)} \tag{6}$$

where $MAC(L_j)$ denotes the number of MAC operations performed in layer L_j, and L is the total number of layers.

$TP(T_j)$ denotes the percentage of the FFs of type T_j. Given ASSUMPTION 2, $TP(T_j)$ is given as:

$$TP(T_j) = \frac{FF(T_j)}{\sum_{i=1}^{T} FF(T_i)} \tag{7}$$

where $FF(T_j)$ denotes the number of FFs of type T_j, and T is the total number of FF types.

Based on ASSUMPTION 3 that all variables of the same type are equally probable of receiving a fault, $VP(j)$ is equal to the probability that a variable j is selected out of all the variables of the same type T_j, and can be expressed as:

$$VP(j) = \frac{1}{V(T_j)} \tag{8}$$

where $V(T_j)$ is the total number of variables of type T_j. For control variables, $V(T_j)$ is always 1 as each control FF is mapped to one control variable.

p_{T_j}, the probability of a cell of type T_j receiving a transient fault, is given by:

$$p_{T_j} = \frac{rawFIT(T_j)}{M \times \sum_{i=1}^{T}(TP(T_i) \times rawFIT(T_i))} \tag{9}$$

where $rawFIT(T_j)$ denotes the raw FIT rate of FFs of type T_j. One way to interpret Equ. 9 is that if all the FFs have the same raw FIT rate, p_{T_j} is reduced to $\frac{1}{M}$, matching our intuition that when all FFs are equally faulty each cell has the same chance of receiving the fault.

Technically, the traditional AVF metric is not concerned with the raw FIT rate, which, among other factors, could vary between different FF types (e.g., different sizes for different FF types). We include the raw FIT rate in the p_{T_j} formulation just to show how to incorporate the impact of different raw FIT rates, as we will later show in Fig. 7. To calculate the actual AVF, p_{T_j} should simply be $\frac{1}{M}$.

Putting it together, Equ. 4 requires only basic statistics of the hardware accelerator and the DNN, and is extremely lightweight to calculate. Thus, one could calculate the fault probability for all the software fault sites, even if there are billions of them. Our approach can be seen as transferring the hardware fault probability (raw FIT rate) to software fault probability, hence the name "probability transfer."

Using Actual Utilization Factor. So far we have assumed that the *Utilization Factor* $UF(j)$ of all the cells in Fig. 2 is 100%, which is obviously not always true. When a cell is not utilized, i.e., a FF does not hold an actual value pertaining to inference, a soft error occurring to that FF has no impact on the inference accuracy. While UF in theory is a hardware concept, statistically speaking the UF of a hardware FF type is the same as the UF of the corresponding software variables. This is because each software variables of the same type is assigned the same amount of cells/FFs (ASSUMPTION 3). Therefore, RA can be formulated as the average of the network inference accuracy under each software fault weighted by both the faulty probability of each software fault site and the *Utilization Factor* $UF(j)$ of a software fault site j:

$$RA = \sum_{j=1}^{N_{fs}} (p(j)UF(j)A(j) + p(j)(1 - UF(j))SA) \tag{10}$$

where (SA) is the fault-free, standard inference accuracy of the network. Intuitively, Equ. 10 says a software fault site j can affect the inference accuracy for just $UF(j)$ portion of the time; for the remaining $(1 - UF(j))$ portion, it will have no impact on the overall accuracy of the network, resulting in an $A(j)$ equal to the SA of the network. We will show in Sec. V-B that using an actual UF obtained from RTL simulation improves the RA estimation accuracy.

IV. Estimating RA Using Importance Sampling

To accelerate RA convergence, we propose to formulate RA estimation as a Monte Carlo integration problem and solve it using importance sampling (Sec. IV-A). We show that perfect importance sampling is fundamentally impossible, and propose two heuristics that leverage DNN-specific characteristics to approximate importance sampling while retaining the main benefits (Sec. IV-B).

IV-A. Compute RA via Monte Carlo Integration

Equ. 3 presents an analytical form of accurately calculating the RA of a DNN. However, that equation is impractical to calculate because of the large N number, i.e., the number of software fault sites. While $p(j)$ for each fault site j can be trivially calculated using Equ. 4, obtaining $A(j)$ requires inferencing over an entire test set. Take MobileNet as an example: it has about 3.5 billion software fault sites; assuming obtaining $A(j)$ for each fault site requires 1 second, it would take 111 years to precisely calculate the RA for MobileNet using the exact formulation in Equ. 3.

Our observation is that calculating RA can be seen as integrating a discrete function over a finite domain:

$$RA = \sum_{j=1}^{N} f(j) \tag{11}$$

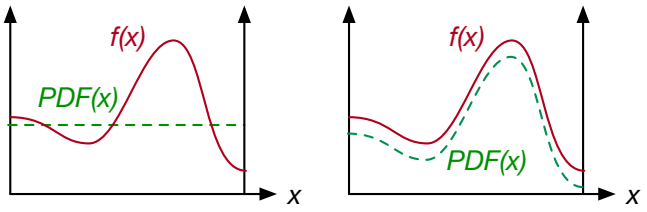
$$f(j) = p(j)A(j), \quad j \in \mathbb{Z} : j \in [1, N] \tag{12}$$

The integrand $f(\cdot)$ does not have an analytical form that can be calculated in practice. We propose to solve the integration numerically using Monte Carlo integration [61].

Formally, RA can be estimated by drawing K independent samples $X_1, ..., X_K$ using a probability density function (PDF) and calculate:

$$\widehat{RA} = \frac{1}{K} \sum_{j=1}^{K} \frac{f(X_j)}{PDF(X_j)}, \quad \sum_{j=1}^{N} PDF(X_j) = 1 \tag{13}$$

$\widehat{RA}$ is called the Monte Carlo estimator of RA. It can be shown that when K approaches infinity $\widehat{RA}$ converges to RA (or alternatively, the expected value of $\widehat{RA}$ is RA). However,



(a) Uniform sampling (constant PDF). (b) Importance sampling, where the PDF is proportional to integrand $f(\cdot)$.

Fig. 3: Comparison of two PDFs used in sampling. Note that the PDF must integrate to 1 by definition.

different sampling methods have significantly different convergence rate. A naive method is to sample the domain uniformly as shown in Fig. 3a, essentially using a PDF of $\frac{1}{N}$. While a common choice, uniform sampling takes a lot of samples to converge as we will show later[3].

Importance Sampling. To improve the convergence rate, an observation is that we have the freedom to choose the PDF used to draw the samples. Intuitively, we want to place more samples where the contribution to the integration is numerically high (i.e., "important"), hence capturing most of the integral contributions. Fig. 3b illustrates the idea.

In particular, it is established that if the $PDF(\cdot)$ is exactly proportional to the integrand $f(\cdot)$, that is:

$$PDF(j) = cf(j), \tag{14}$$

where c is a normalization value that guarantees that $PDF(j)$ integrates to 1:

$$c = \frac{1}{\sum_{j=1}^{N} f(j)} \tag{15}$$

then the variance of the Monte Carlo estimator is zero, indicating that exactly *one* sample is sufficient [41].

However, perfect important sampling is impossible, because determining c requires us to calculate the integration that we set out to estimate in the first place.

IV-B. Heuristics for Importance Sampling

We do not know $f(\cdot)$ precisely and, thus, cannot construct a PDF exactly proportional to $f(\cdot)$. However, we observe that $f(\cdot)$ is a product of two terms: $p(\cdot)$ and $A(\cdot)$. Our idea is to use *a priori* knowledge of $p(\cdot)$ and $A(\cdot)$ to construct a PDF that is *approximately* proportional to $f(\cdot)$.

In particular, $p(\cdot)$ can be completely constructed in advance given a particular DNN and the underlying hardware using Equ. 4. In contrast, $A(\cdot)$ cannot be constructed offline in practical terms since it requires enumerating hundreds of billions of fault sites, each of which requires performing inference on an entire test set.

[3]More formally, the standard deviation of uniform sampling is proportional to $\frac{1}{\sqrt{N}}$, indicating that we must quadruple the number of samples to reduce the error by half.

TABLE I: Workloads for validation.

Category	Network	Layer(s)	Dataset
-	Synthetic	7×7 CONV	Random inputs
Classification	CifarNet [31]	All the layers	CIFAR-10 [39]
Classification	ResNet-18 [28]	3x3 CONV (layer 2,4,6)	ImageNet [18]
Detection	DETR-R50 [7]	3x3 CONV (layer 7)	COCO [47]
Speech-To-Text	DeepSpeech [26]	FC (layer 10)	LibriSpeech [55]

We propose two heuristics to empirically model $A(\cdot)$. Each heuristic gives a unique approximation of $A(\cdot)$, which when multiplied with $p(\cdot)$ gives a $\hat{f}(\cdot)$, which is an approximation of $f(\cdot)$. We then construct the PDF proportional to $\hat{f}(j)$ for importance sampling.

Constancy Heuristic. The simplest heuristic is to assume that $A(j)$ is constant. That is, the inference accuracy under any software fault site j is the same. This heuristic in turn equates $f(\cdot)$ to $p(\cdot)$. Thus, the sampling PDF is constructed proportional to $p(\cdot)$.

BP Heuristic. We can improve upon the uniform assumption by observing that transient errors in certain bit positions (BPs) have a higher impact on accuracy than other positions. Taking the floating point representation as an example, a bit flip in the sign and exponent fields changes the numerical values much more significantly than bit flips in the fraction field; as a result, prior work has shown that transient errors in the sign and exponent fields degrade the inference accuracy more than the fraction field [45].

Given this empirical observation, we can model $A(\cdot)$ with respect to the BP. The exact impact of each BP in a number representation can be profiled offline.

It is worth emphasizing that profiling does not have to (and will not) capture the exact impact of BPs on $A(\cdot)$. This is because $\hat{f}(\cdot)$, as an heuristic for importance sampling, does not have to be exactly the same as $f(\cdot)$. The approximation of $\hat{f}(\cdot)$ affects only the convergence rate, *not* the accuracy of RA estimation. We will show in Sec. VI that even simple assumptions of the BP-wise impact are sufficient.

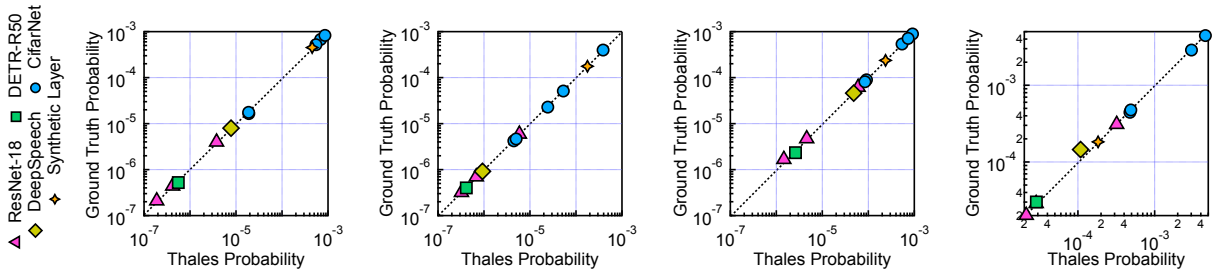
V. Validation

We first describe the RTL-based validation setup (Sec. V-A), and show that the software fault site probability and RA estimation analytically derived in Sec. III-B match well with results obtained from RTL fault injection (Sec. V-B).

V-A. Validation Setup

We use a TPU-like systolic array with a 32 × 32 MAC array using the output stationary data-flow. After synthesis, the distribution across the input activation, weight, output activation, and control FFs is 30.56%, 30.56%, 21.87%, 17.0%.

Similar to prior work on (DNN) resiliency [20], [29], [48], we use RTL fault injection to obtain ground truth data. We use Synopsys Z01X™ [4], an industry-scale fault injection and simulation tool, to pick possible hardware fault sites written as $< \mathbf{Cycle}, \mathbf{FFType}, \mathbf{BP} >$. Each run, the tool injects a bit-flip injection at a fault site and performs the RTL simulation that takes the DNN inference to the end. Therefore, this methodology correctly captures the fact that one transient fault



(a) Input Activations. (b) Weights. (c) Output Activations. (d) Control.

Fig. 4: Comparison of ground truth FF fault probabilities (y-axis) with analytically derived probabilities by THALES (x-axis).

in a software variable (e.g., a weight) could result in errors in subsequent variables that depend on the variable receiving the initial transient fault (e.g., all the output activations).

Tbl. I lists the workloads we use for validation. The first workload is a synthetic 7 × 7 CONV layer, for which we exhaust all the fault sites (i.e., every single bit at every single FF at every single cycle), resulting in a total of 2.629 Billion RTL-level fault injections and simulations (900 × 240 CPU hours in total). The remaining workloads are individual layers from popular DNNs, for which we run fault injections and simulations on 332K randomly sampled fault sites, which are obtained using Synopsys Z01X to achieve a 99% confidence level with a 1% error margin.

V-B. Validation Results

Fault Probability Validation. Our goal is to validate that the software fault site probability (i.e., $p(j)$ in the RA formulation in Equ. 3) is correctly calculated by our analytical model given in Equ. 4. We take each sampled fault site and categorize them as either Input Activations, Weights, Output Activations or Control FFs. We then compare the faulty probability of each category that our model calculates with the data obtained from RTL fault injection and simulation. We assume that the FFs have a raw FIT rate of 600/MB following the study from Jagannathan et al. [33] and consistent with FIdelity [29].

Fig. 4 shows the validation results for the four FF categories. Overall, the Pearson correlation coefficient is 0.9998 between results obtained by our analytical model and the ground truth, suggesting the accuracy of our approach. The main source of inaccuracy of is FC layers, where the analytically calculated probabilities are 6.4% away from the ground truth. This is because FC layers do not fully utilize the hardware (74.0% on average), while our model, absent detailed RTL-level data, assumes 100% hardware utilization. When using the actual hardware utilization data, the estimation accuracy is brought to within 4.0% of the ground truth.

RA Validation We also validate that our RA estimation matches well with the ground truth RA (RA_{True}) obtained from RTL simulation. To that end, we perform RTL fault injections and simulations on the entire CifarNet [31] and ResNet-18 [28] performing inference on 20 randomly selected images from the CIFAR-10 [39] dataset. We obtain the Utilization Factor (UF) of both the networks from RTL simulations. We show two RA estimatios (RA_{Est}), first by assuming UF = 1 (i.e., using Equ. 3) and then by using the actual UF values (i.e., using Equ. 10). The results are shown in Tbl. II.

TABLE II: RA estimation validation. RA_{True} is the ground truth RA, RA_{Est} is the RA estimated with Thales, and UF is the utilization factor of the hardware obtained from RTL simulations. The error percentage is for RA_{Est} estimated using the actual UF.

Network	UF	RA_{True}	RA_{Est}		% Error
			Actual UF	UF = 1	
CifarNet [31]	0.2632	0.930517	0.923981	0.890132	0.70240
ResNet-18 [28]	0.36733	0.88445	0.870438	0.828221	1.5842

We observe that the RA estimation is much closer to the ground truth RA when using the actual UF. Specifically, the RA when assuming UF = 1 is lower than that when using the actual UF. This is down to the fact that with higher utilization of the hardware, more FFs are vulnerable to soft errors. The percentage error in RA_{Est} for CifarNet [31] and ResNet-18 [28] reduces from 4.34% and 6.35% to 0.7% and 1.58% respectively when considering the actual UF.

VI. EVALUATING SAMPLING STRATEGIES

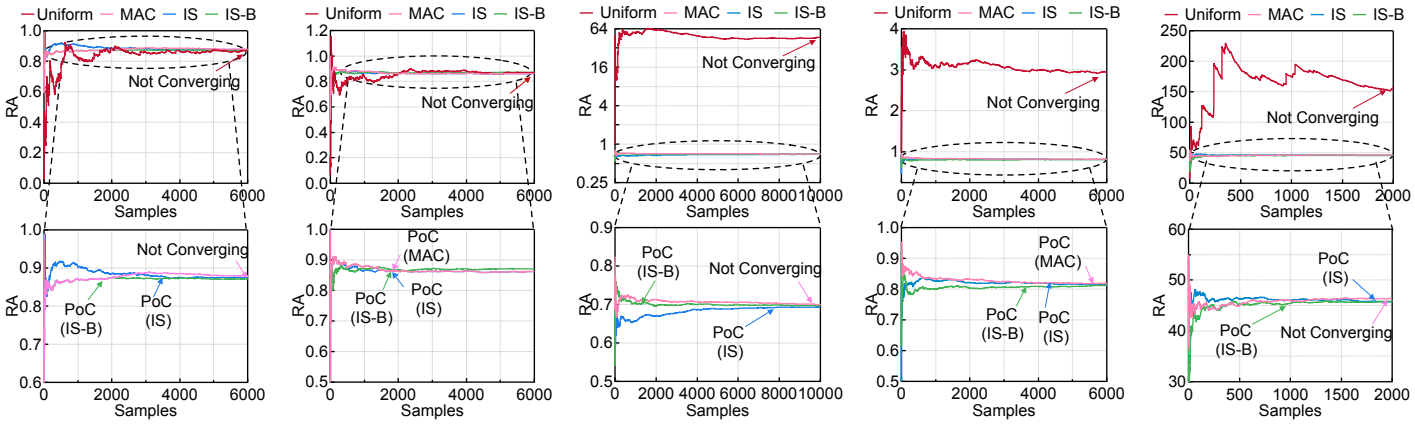
We now evaluate the importance sampling strategy proposed in this paper on different DNNs and accelerators.

VI-A. Experimental Setup

DNN Accelerators. We evaluate on two DNN accelerators: a systolic array, whose configuration is the same as that used in Sec. V, and Nvidia's NVDLA [3]. We configure the NVDLA with 2 convolution cores, each with a 4×4 MAC units using the FP16 data type. Thse configuration is exactly the same as in FIdelity [29] , which also reports the FF distribution.

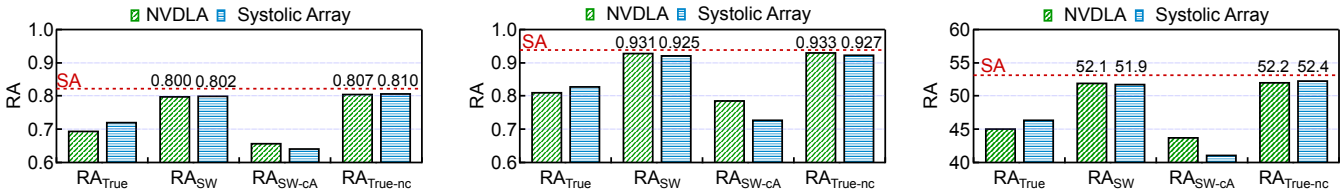
DNN Models. We evaluate five common DNNs covering both image classification and object detection. Two DNNs are small in scale: a 5-layer LeNet [19] and a 4-layer MNIST-Hogwild [57], on the MNIST dataset. We also consider three larger networks: AlexNet [40] and MobileNet [32] for image classification on the CIFAR-10 dataset [39] and YOLOv3 [68], an object detection DNN, using the COCO dataset [47].

Evaluation Plan. We aim to show that our sampling method 1) converges to the ground truth RA, and 2) converges faster than existing sampling methods.



(a) LeNet. (b) MNIST-Hogwild. (c) AlexNet. (d) MobileNet. (e) YOLOv3.

Fig. 5: Convergence comparison of different sampling strategies. PoC stands for Point of Convergence. Overall, the order of convergence speed is IS-B > IS > MAC > UNIFORM, which does not converge within given several thousands samples and produces physically unrealizable results.



(a) AlexNet. (b) MobileNet. (c) YOLOv3.

Fig. 6: Comparison of RA formulations on the three large networks; the trend on the two small MNIST networks is similar. RA_{True} is our RA formulation (Equ. 3); $RA_{Truc-nc}$ uses Equ. 3 but assumes all control FFs are fault-free; RA_{SW} is obtained using PyTorchFI and represents Equ. 2; RA_{SW-cA} uses PyTorchFI but with the correct $A(j)$ estimation using FIdelity [29].

To obtain the ground-truth RA, we exhaust all the fault sites for two small networks LeNet (352K) and MNIST-HogWild (412K), but for three larger networks we evaluate, it would take unrealistically long time. Instead, our strategy is to intentionally target a subset of the hardware fault sites and validate within that subset. Specifically, we randomly sample 500K fault sites for AlexNet, MobileNet, and YOLOv3., and assume that these are all the hardware fault sites. All subsequent evaluations are restricted to only those fault sites. The raw FIT rate is the same as described in Sec. V-B

VI-B. Comparing Sampling Methods

We also aim to compare how effective different sampling strategies are for estimating RA_{True}. To that end, we will compare the following four sampling methods:

- UNIFORM: this strategy draws a software fault site uniformly at random.
- MAC: this sampling strategy is adapted from Mahmoud et al. [51]; the probability of a weight or an activation getting sampled in fault injection is proportional to the number of MAC operations they are involved in.
- IS: this is importance sampling using the constancy heuristic in Sec. IV-B, where the PDF for sampling is proportional to the faulty probabilities of the software fault sites $p(j)$ while assuming $A(j)$ is constant.
- IS-B: this is importance sampling using the BP heuristic in Sec. IV-B. $A(j)$ that considers the different accuracy impacts of different BPs. We randomly inject faults into different BPs and observe the accuracy drop. Averaging our profiling results, we assume an inference accuracy drop of 15% for the MSB in the exponent and an 8% accuracy drop for the next four exponent bits; the accuracy drop of other BPs are assumed to be zero.

The convergence curves of different sampling methods are shown in Fig. 5. The Point of Convergence (PoC) must satisfy two criteria: the average RA in a window of 300 samples must be within 0.3% of the ground truth and the variance in the same window must be below 10^{-2}.

Immediately noticeable is that the RA under uniform sampling for the three large networks is well above one, which is physically impossible, indicating that UNIFORM would take a lot more samples to converge. The reason that UNIFORM has a hard time converging is that the probabilities of the software fault sites vary by orders of magnitude, for which uniform sampling is inefficient (Sec. IV). For instance, in YOLOv3 the faulty probability of a weight of the first convolution layer is two orders of magnitude higher than that of in the twelfth

convolution layer $1.61 \times 10^{-4}\%$ vs. $2.67 \times 10^{-6}\%$). The same difference exists within a layer, too.

Comparing other sampling strategies, we observe a general trend, IS-B converges faster than IS, which is faster than MAC, confirming the benefits of using a sampling PDF that is more proportional to the integrand during Monte Carlo integration. MAC converges more slowly than the two importance sampling-based methods, because MAC, while considers different reuse factors of weights/activations based on the MAC operations, does not take into account the actual FF distribution in the underlying hardware architecture.

VII. Comparative Studies

Comparing RA Methods. Using the three larger networks and the setup described in Sec. VI-A, Fig. 6 compares the RA estimated from a set of different formulations/methods:

- RA_{True}: RA estimated using our formulation.
- $RA_{True-nc}$: RA estimated using our formulation, but assuming all global FFs are fault free.
- RA_{SW}: RA estimated using the formulation in Equ. 2, which represented the method used by common tools today such as PyTorchFI and TensorFI.
- RA_{SW-cA}: RA estimated using the Equ. 2 formulation but using the correct $A(j)$ estimated from FIdelity [29], but still (incorrectly) assumes that all software fault sites are equally faulty.

RA_{SW} is always higher than RA_{True}, indicating an over-estimation. In fact, under RA_{SW} a DNN's RA is very close to SA. For instance, AlexNet has only a 0.7% accuracy drop and YOLOv3 has only a 1.2 mAP drop compared to their SAs, giving a perhaps rather optimistic impression that today's DNNs are resilient to transient faults.

A primary source of over-estimation in RA_{SW} is that software fault injection does not consider global control FFs, which in NVDLA contribute to 11.3% of the total FFs. To tease apart the effects of global control FFs and other FFs, we compare $RA_{True-nc}$ with RA_{True}. "RA_{True}" is over 10% lower than "$RA_{True-nc}$", matching our intuition that application crashes dramatically lower the accuracy.

$RA_{True-nc}$ is slightly, but consistently, higher than RA_{SW} across all DNNs even though both do not consider control FFs. This is because RA_{SW} assumes that a corrupted weight affects all the output activations, where in reality the affected neurons are fewer due to the limited reuse factor of a FF.

RA_{SW-cA} consistently under-estimates the RA compared to RA_{True}. The reason is that a uniform faulty probability distribution across software fault sites made by RA_{SW-cA} exaggerates the accuracy degradation effect caused by the control FFs, which occupy a small portion of the total FFs but result in much lower (zero) RAs given bit flips.

Comparing Networks. YOLOv3 has a lower SA to RA drop compared to the two image classification networks AlexNet and MobileNet. Specifically, AlexNet and MobileNet have a SA-to-RA drop of 11.4% and 12.0%, whereas the SA-to-RA drop for YOLOv3 is 8.2%. The reason is that object detection is generally more resilient: a slight change in bounding box position and size can be tolerated as long as its intersection with the ground truth bounding box over the union of them is higher than the threshold.

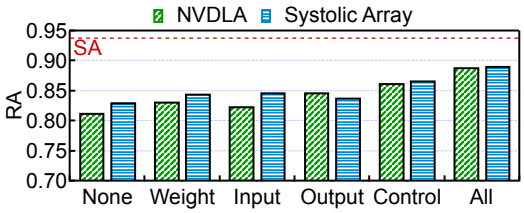


Fig. 7: We pick one FF type (x-axis) to harden at a time and evaluate the resulting MobileNet RA (y-axis).

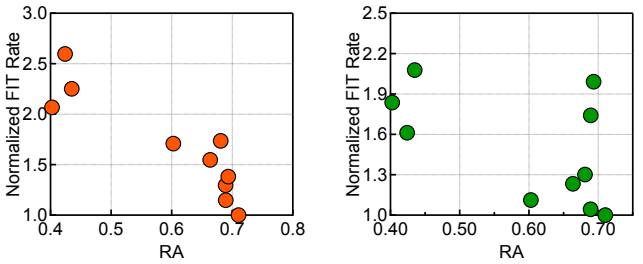
Comparing Accelerators. We make two observations when comparing the results between NVDLA and systolic array. First, RA_{True} on NVDLA is higher than that on the systolic array. This is due to the absence of the FFs before the on-chip memory in our simple systolic array design. Faults in FFs before the on-chip memory corrupt data in the SRAM and propagate faults to a significantly larger number of neurons. Second, RA_{SW-cA} on NVDLA is higher than that on the systolic array. This difference, again, arises from the absence of FFs before the on-chip memory in the systolic array. Without those FFs, the faulty probability for control FFs increases, reducing RA_{SW-cA} on the systolic array.

Sensitivity of Raw FIT Rates. Hardening FFs in hardware is a widely used technique to mitigate transient errors [58], [64]. To study the impact of FF hardening on RA, we perform an experiment where we harden one FF type at a time. We assume that FFs are hardened using the common Dual Interlocked storage Cell (DICE) technology [6], and the raw FIT rate for DICE-hardened FFs is 200/MB (cf. 600/MB for unhardened FFs) following Jagannathan et al. [33].

Fig. 7 shows the resulting RA of MobileNet (y-axis); the x-axis shows the FF type that is hardened. "None" and "All" mean no and all FF types are hardened, respectively. Unsurprisingly, RA with one hardened FF type is between that of "None"and "All".

Our results provide insight into which category of FFs should be hardened for improving RA. For instance, hardened the control FFs has the highest RA improvements, since transient faults on control FFs usually crash an accelerator altogether. Interestingly, while in NVDLA output FFs are the largest in number of all categories, hardening them does not provide the highest RA improvement. This is because the reuse factor of output FFs is the smallest (one), so transient faults on them have low impact on accuracy.

RA, FIT, and SDC Correlation. We use ten networks found by the NAS framework (Sec. VIII) to study the RA vs FIT rate correlation. Like prior work on DNN resiliency [29], [45], we use a threshold T in estimating the FIT rate. Specifically, a transient fault is considered to generate a failure if the inference accuracy under that fault is lower than the fault-free accuracy by T. Similar to Fidelity [29], we use two thresholds for this study: 20% and 40%.



(a) 20% threshold. (b) 40% threshold.

Fig. 8: RA (x-axis) vs FIT rate (y-axis) correlation under two thresholds for ten NAS-generated networks.

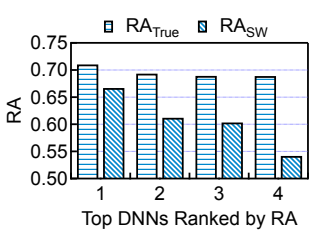


Fig. 9: Top architectures' RA comparison between using RA_{True} and RA_{SW}.

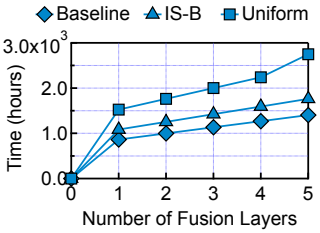


Fig. 10: NAS latency of using IS-B and UNIFORM.

Fig. 8 shows the results under the two thresholds. The FIT rate is normalized to the lowest FIT rate among the ten networks. While in general RA is inversely correlated with the FIT rate, notable outliers exist especially when the threshold is large: the Pearson correlation coefficients are -0.90 and -0.53 under the 20% and 40% threshold, respectively. In addition, the FIT rate does not provide the actual inference accuracy results, which could be critical to algorithm designers.

SDC rate is similarly defined as the FIT rate (and thus shares the same caveats above) except SDC does not consider the potential system crashes under transient errors, which is non-trivial. On CifarNet [31], ResNet-18 [28], DETR-R50 [7], and DeepSpeech [26], 6.86%, 7.13%, 5.55% and 4.79% of the transient faults yield crashes, respectively.

VIII. RA-AWARE NAS

We show how our RA estimation can help automatically design DNNs with high RAs through Network Architecture Search (NAS). We first introduce our RA-aware NAS framework (Sec. VIII-A) followed by the experimental setup (Sec. VIII-B). We show that networks generated using our NAS framework possess significantly higher RA than NAS that uses other RA estimation methods (Sec. VIII-C).

VIII-A. The NAS Framework

As a case study, we focus on multi-modal DNNs, which fuse two or more data modalities and are widely used in application domains such as autonomous machines [1], [36]. We study multi-modal fusion DNNs for two reasons. First, they are widely used in applications such as autonomous machines, where resiliency against soft errors is critical. For instance, autoware [36] and Baidu Applo [1] fuse RGB cameras and LiDAR-generated point clouds for object tracking. Second, these networks are challenging workloads for NAS, because they expose additional degree of freedom such as at what layers and using what operators to fuse different modalities.

We propose a RA-aware NAS algorithm for search fusion DNNs. Our NAS algorithm is built on top of the MFAS fusion NAS framework [59], which was originally designed to search fusion networks with the highest standard accuracy. We claim *no* algorithmic novelty over MFAS. Rather, our intention is to show that our RA estimation algorithm can be readily integrated into existing NAS frameworks, such as MFAS, to enable RA-aware NAS.

Problem Setup. We consider a basic fusion DNN containing two branches, each processing a particular input modality (e.g., RGB images vs. point clouds). The intermediate feature maps at certain layers of each branch are fused before going through a unified branch to generate the output. The searching space includes three degrees of freedom:

1) the number of layers, L, from each branch to fuse; note that each branch could have more than L layers, but only L layers from each branch are used for fusion;
2) the exact L layers from the first branch $(X_1, ..., X_L)$ and the exact L layers from the second branch $(Y_1, ..., Y_L)$ to fuse; layers are fused in order — that is, X_i and Y_i are implicitly a pair of fusion layers;
3) the activation function after each of the L fusion pairs.

Our goal is to identify a network whose fusion configuration maximizes the RA.

VIII-B. Experimental Setup

Network and Datasets. We target two-branch fusion networks for action recognition. The first branch takes RGB videos as the input and the second branch takes skeleton-based pose data as the input. Following the setup in MFAS [59], we use the inflated ResNet-50 [5] for the video branch and the deep co-occurrence model [44] for the pose branch. We use the NTU RGB+D dataset [73]. We set the maximum L to be 3, i.e., at most three fusion layers are allowed in NAS.

Evaluation Plan. We have two aims. First, we aim to show that our RA estimation algorithm allows the NAS framework to identify networks with higher RAs than today's software-level RA estimation algorithms. Second, we aim to show that our importance sampling method improves the NAS speed over random sampling.

VIII-C. Analyzing the Networks

We show that NAS guided by our RA estimation algorithm RA_{True} yields networks that have higher RAs than those guided by RA_{SW}. Fig. 9 compares the RAs of the top 4 networks generated by NAS using the two RA estimation methods. Note that the RAs of all networks, regardless of how they are generated, are all evaluated using our RA estimation algorithm to obtain the true RAs. The average improvement of RA is 9.0% across the four networks. In fact, the fourth

TABLE III: Top five networks generated by NAS when the number of fusion layers L is 2 vs. 3 (L is capped at 3). Each <X, Y> denotes a fusion pair. For instance, <4, 1> means the output of layer 4 in the first branch and the output of layer 2 in the second branch are fused together.

RA (L=2)	<X,Y>	RA (L=3)	<X,Y>
40.2%	<4,2>,<2,2>	66.4%	<4,3>,<3,2>,<3,1>
42.4%	<4,3>,<3,4>	68.1%	<2,1>,<4,4>,<4,2>
43.5%	<4,3>,<2,3>	68.9%	<4,3>,<3,2>,<4,1>
60.3%	<2,1>,<4,4>	69.3%	<4,3>,<2,3>,<4,1>
68.9%	<4,3>,<3,2>	71.0%	<4,3>,<3,4>,<4,1>

resilient network using RA_{True} to guide NAS has a higher RA compared to the best network searched using RA_{SW}.

Analyzing the NAS-ed Networks. We find that fusion DNNs that have higher RAs share two common characteristics.

First, using more fusion layers generally improve the resilience of the fusion networks. Tbl. III compares the RAs of the top five DNNs when $L = 2$ (two fusion layers) vs. $L = 3$ (three fusion layers). By adding one more fusion layer, NAS is able to improves the highest RA of a network by 2.1% and the average RA of the top five networks by 17.8%.

Second, high-RA DNNs usually fuse feature maps from later stages of both branches. Tbl. III shows the fusion configuration of the top-RA DNNs, where <X, Y> indicates that layer X from the first branch and layer Y from the second branch are fused. All these networks make extensive use of the last two layers for fusion (i.e., frequent use of <$4, 3$>, <$3, 4$>, and <$4, 4$>). That said, while earlier layers are rarely fused with each other, they do participate in certain fusion pairs, indicating the need to mix different layers in fusion.

Improving NAS Speed. Our importance sampling strategy significantly improves the NAS speed by requiring fewer samples in each NAS iteration, which must estimate the RA of the top K network that the surrogate function predicts.

Fig. 10 shows the total NAS time on two Nvidia 2080Ti GPUs across different schemes. "Baseline" denotes the NAS time when SA is the search target, i.e., no time spent on RA estimation. The other curves represent the NAS time under different sampling strategies. The x-axis shows the number of the fusion layers allowed to be searched. Naturally, when more fusion layers are allowed the total NAS time increases. Compared to the baseline, UNIFORM sampling almost double the NAS time. IS-B limits the overhead to be only 25.5%.

IX. Related Work

The main novelty of our work is two-fold. First, we formulate the new RA metric, which quantifies the accuracy of a network *given* that a fault has occurred. In contrast, FIdelity [29] estimates the FIT rate; TensorFI [13] and PyTorchFI [50] both report the SDC rate. Both the FIT rate and the SDC rate quantify *how often* hardware faults show up in software, rather than how bad the network inference would be under faults. Mahmoud et al. [51] estimates the "relative vulnerability of each fmap in the CNN" (for selectively protecting feature maps [46]), a metric different from our RA.

Second, we provide an importance sampling-based method to estimate the RA metric without exhausting all the software fault sites. Prior work mostly sample faults sites uniformly. Uniform sampling in software (PyTorchFI [50] and TensorFI [13]) under-estimates RA (Sec. VII). Uniformly sampling in hardware, e.g., Li et al. [45], in theory could yield the same RA estimation as ours. But hardware simulation is known to be extremely time consuming, which our work avoids. Mahmoud et al. [51] breaks away from uniform sampling. They instead sample based on the number of MAC operations to calculate each activation. As shown in Fig. 5, such a method, while significantly better than uniform sampling, converges more slowly than importance sampling.

We use importance sampling with PDF heuristics derived from DNN-specific characteristics. The PDF heuristics are reminiscent of the classic Monte Carlo methods used in solving the rendering equation for physically-based rendering [60].

Much of the prior work studies the DNN resiliency against faults in memory [8], [52], [66]. Memory faults could also be introduced by an attacker [11], [30], [65]. This work focuses on the transient faults in FFs as discussed in Sec. II-B. Hong et al. [30] considers faults just in the most significant bits of a floating point representation and, thus, over-estimates the impact of transient faults.

Transient faults are most accurately studied in hardware [14], [15], [80], which is impractical due to slow RTL simulations. Modeling transient errors at the model level, instead, has been the dominant approach for studying DNN resiliency [10], [12], [13], [30], [50]. We discuss the sources of inaccuracies of this approach in Sec. II-D. FIdelity [29] provides a methodology that captures the impact of a single hardware fault without RTL simulation, which this paper uses.

Of course, transient faults are not the only source of vulnerability in real-world applications such as autonomous machines, where safety issues arise from latency variation [85], adversarial attacks to DNNs [24], [63], and others; different sources of vulnerability have different implications on the safety [25]. An interesting line of future work is to examine DNN soft errors in the context of end-to-end applications.

Our RA estimation framework assumes that a DNN's data reuse is regular: all variables of the same type in a layer have the same reuse. This assumption does not hold for Graph Neural Networks [69], [74], [83] and point cloud DNNs [62], [79], [84], [86], where data reuses are not uniform because of the irregular connections in graphs and/or neighbor searches [23], [35], [70], [78], [82], [87].

X. Conclusion

We propose a statistically-based formulation to quantify a DNN's RA. The crux of the formulation is to correctly model the faulty probabilities of software fault sites. Our RA formulation can be seen as the DNN-specific AVF. We show that importance sampling coupled with proper sampling functions is key to efficiently estimating the RA in software. We show that transient faults present far greater degradation to DNN's inference accuracy than what existing metrics estimate.

REFERENCES

[1] "Baidu Apollo team (2017), Apollo: Open Source Autonomous Driving, howpublished = https://github.com/apolloauto/apollo note = Accessed: 2019-02-11."

[2] "Expected Value, howpublished = https://en.wikipedia.org/wiki/expected_value."

[3] "Nvdla open source project, howpublished = http://nvdla.org/primer.html, note = Accessed: 2018."

[4] "Z01X Functional Safety Assurance, howpublished = https://www.synopsys.com/verification/simulation/z01x-functional-safety.html."

[5] F. Baradel, C. Wolf, J. Mille, and G. W. Taylor, "Glimpse clouds: Human activity recognition from unstructured feature points," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 469–478.

[6] T. Calin, M. Nicolaidis, and R. Velazco, "Upset hardened memory design for submicron cmos technology," *IEEE Transactions on nuclear science*, vol. 43, no. 6, pp. 2874–2878, 1996.

[7] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-end object detection with transformers," in *European conference on computer vision*. Springer, 2020, pp. 213–229.

[8] N. Chandramoorthy, K. Swaminathan, M. Cochet, A. Paidimarri, S. Eldridge, R. V. Joshi, M. M. Ziegler, A. Buyuktosunoglu, and P. Bose, "Resilient low voltage accelerators for high energy efficiency," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 147–158.

[9] C.-L. Chen and M. Hsiao, "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," *IBM Journal of Research and development*, vol. 28, no. 2, pp. 124–134, 1984.

[10] 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 *2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*. IEEE, 2008, pp. 339–346.

[11] H. Chen, C. Fu, J. Zhao, and F. Koushanfar, "Proflip: Targeted trojan attack with progressive bit flips," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 7718–7727.

[12] Z. Chen, G. Li, K. Pattabiraman, and N. DeBardeleben, "Binfi: An efficient fault injector for safety-critical machine learning systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–23.

[13] Z. Chen, N. Narayanan, B. Fang, G. Li, K. Pattabiraman, and N. DeBardeleben, "Tensorfi: A flexible fault injection framework for tensorflow applications," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2020, pp. 426–435.

[14] E. Cheng, S. Mirkhani, L. G. Szafaryn, C.-Y. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose *et al.*, "Clear: C ross-l ayer e xploration for a rchitecting r esilience-combining hardware and software techniques to tolerate soft errors in processor cores," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[15] 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*, 2013, pp. 1–10.

[16] D. A. G. G. de Oliveira, L. L. Pilla, T. Santini, and P. Rech, "Evaluation and mitigation of radiation-induced soft errors in graphics processing units," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 791–804, 2015.

[17] V. Degalahal, N. Vijaykrishnan, and M. J. Irwin, "Analyzing soft errors in leakage optimized sram design," in *16th International Conference on VLSI Design, 2003. Proceedings.* IEEE, 2003, pp. 227–233.

[18] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[19] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[20] F. F. dos Santos, J. E. R. Condia, L. Carro, M. S. Reorda, and P. Rech, "Revealing gpus vulnerabilities by combining register-transfer and software-level fault injection," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 292–304.

[21] F. F. dos Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, "Analyzing and increasing the reliability of convolutional neural networks on gpus," *IEEE Transactions on Reliability*, vol. 68, no. 2, pp. 663–677, 2018.

[22] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 385–396, 2010.

[23] Y. Feng, G. Hammonds, Y. Gan, and Y. Zhu, "Crescent: taming memory irregularities for accelerating deep point cloud analytics," *arXiv preprint arXiv:2204.10707*, 2022.

[24] Y. Gan, Y. Qiu, J. Leng, M. Guo, and Y. Zhu, "Ptolemy: Architecture support for robust deep learning," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 241–255.

[25] Y. Gan, P. Whatmough, J. Leng, B. Yu, S. Liu, and Y. Zhu, "Braum: Analyzing and protecting autonomous machine software stack," in *ISSRE*, 2022.

[26] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates *et al.*, "Deep speech: Scaling up end-to-end speech recognition," *arXiv preprint arXiv:1412.5567*, 2014.

[27] 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)*. IEEE, 2012, pp. 1–12.

[28] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[29] Y. He, P. Balaprakash, and Y. Li, "Fidelity: Efficient resilience analysis framework for deep learning accelerators," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 270–281.

[30] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitraș, "Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 497–514.

[31] J. Hosang, M. Omran, R. Benenson, and B. Schiele, "Taking a deeper look at pedestrians," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 4073–4082.

[32] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[33] S. Jagannathan, T. Loveless, B. Bhuva, N. Gaspard, N. Mahatme, T. Assis, S.-J. Wen, R. Wong, and L. Massengill, "Frequency dependence of alpha-particle induced soft error rates of flip-flops in 40-nm cmos technology," *IEEE Transactions on Nuclear Science*, vol. 59, no. 6, pp. 2796–2802, 2012.

[34] S. M. Jahinuzzaman, D. J. Rennie, and M. Sachdev, "A soft error tolerant 10t sram bit-cell with differential read capability," *IEEE Transactions on Nuclear Science*, vol. 56, no. 6, pp. 3768–3773, 2009.

[35] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

[36] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2018, pp. 287–296.

[37] T. Kloek and H. K. Van Dijk, "Bayesian estimates of equation system parameters: an application of integration by monte carlo," *Econometrica: Journal of the Econometric Society*, pp. 1–19, 1978.

[38] K. Kobayashi, K. Kubota, M. Masuda, Y. Manzawa, J. Furuta, S. Kanda, and H. Onodera, "A low-power and area-efficient radiation-hard redundant flip-flop, dice acff, in a 65 nm thin-box fd-soi," *IEEE Transactions on Nuclear Science*, vol. 61, no. 4, pp. 1881–1888, 2014.

[39] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[40] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[41] D. P. Kroese, T. Taimre, and Z. I. Botev, *Handbook of monte carlo methods*. John Wiley & Sons, 2013.

[42] L. Lantz, "Soft errors induced by alpha particles," *IEEE Transactions on Reliability*, vol. 45, no. 2, pp. 174–179, 1996.

[43] 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*. IEEE, 2009, pp. 502–506.

[44] C. Li, Q. Zhong, D. Xie, and S. Pu, "Co-occurrence feature learning from skeleton data for action recognition and detection with hierarchical aggregation," *arXiv preprint arXiv:1804.06055*, 2018.

[45] 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.

[46] F. Libano, B. Wilson, J. Anderson, M. J. Wirthlin, C. Cazzaniga, C. Frost, and P. Rech, "Selective hardening for neural networks in fpgas," *IEEE Transactions on Nuclear Science*, vol. 66, no. 1, pp. 216–222, 2018.

[47] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*. Springer, 2014, pp. 740–755.

[48] 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.

[49] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," in *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*. IEEE, 2000, pp. 417–426.

[50] A. Mahmoud, N. Aggarwal, A. Nobbe, J. R. S. Vicarte, S. V. Adve, C. W. Fletcher, I. Frosio, and S. K. S. Hari, "Pytorchfi: A runtime perturbation tool for dnns," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2020, pp. 25–31.

[51] A. Mahmoud, S. K. S. Hari, C. W. Fletcher, S. V. Adve, C. Sakr, N. Shanbhag, P. Molchanov, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing selective protection for cnn resilience," in *32nd IEEE International Symposium on Software Reliability Engineering, ISSRE 2021*. IEEE Computer Society, 2021, pp. 127–138.

[52] S. Mittal, "A survey on modeling and improving reliability of dnn algorithms and accelerators," *Journal of Systems Architecture*, vol. 104, p. 101689, 2020.

[53] 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.

[54] R. Naseer, Y. Boulghassoul, J. Draper, S. DasGupta, and A. Witulski, "Critical charge characterization for soft error rate modeling in 90nm sram," in *2007 IEEE International Symposium on Circuits and Systems*. IEEE, 2007, pp. 1879–1882.

[55] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, "Librispeech: an asr corpus based on public domain audio books," in *2015 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2015, pp. 5206–5210.

[56] 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)*. IEEE, 2021, pp. 902–915.

[57] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[58] C. Peng, J. Huang, C. Liu, Q. Zhao, S. Xiao, X. Wu, Z. Lin, J. Chen, and X. Zeng, "Radiation-hardened 14t sram bitcell with speed and power optimized for space application," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 2, pp. 407–415, 2018.

[59] J.-M. Pérez-Rúa, V. Vielzeuf, S. Pateux, M. Baccouche, and F. Jurie, "Mfas: Multimodal fusion architecture search," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 6966–6975.

[60] M. Pharr, W. Jakob, and G. Humphreys, *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.

[61] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[62] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 652–660.

[63] Y. Qiu, J. Leng, C. Guo, Q. Chen, C. Li, M. Guo, and Y. Zhu, "Adversarial defense through network profiling based path extraction," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4777–4786.

[64] R. Rajaei, M. Tabandeh, and M. Fazeli, "Single event multiple upset (semu) tolerant latch designs in presence of process and temperature variations," *Journal of Circuits, Systems and Computers*, vol. 24, no. 01, p. 1550007, 2015.

[65] A. S. Rakin, Z. He, and D. Fan, "Tbt: Targeted neural network attack with bit trojan," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 13 198–13 207.

[66] B. Reagen, U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G.-Y. Wei, "Ares: A framework for quantifying the resilience of deep neural networks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[67] 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)*. IEEE, 2022, pp. 376–381.

[68] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[69] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.

[70] T. Seidl and H.-P. Kriegel, "Optimal multi-step k-nearest neighbor search," in *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, 1998, pp. 154–165.

[71] N. Seifert, V. Ambrose, B. Gill, Q. Shi, R. Allmon, C. Recchia, S. Mukherjee, N. Nassif, J. Krause, J. Pickholtz *et al.*, "On the radiation-induced soft error performance of hardened sequential elements in advanced bulk cmos technologies," in *2010 IEEE International Reliability Physics Symposium*. IEEE, 2010, pp. 188–197.

[72] N. Seifert, P. Slankard, M. Kirsch, B. Narasimham, V. Zia, C. Brookreson, A. Vo, S. Mitra, B. Gill, and J. Maiz, "Radiation-induced soft error rates of advanced cmos bulk devices," in *2006 IEEE International Reliability Physics Symposium Proceedings*. IEEE, 2006, pp. 217–225.

[73] A. Shahroudy, J. Liu, T.-T. Ng, and G. Wang, "Ntu rgb+ d: A large scale dataset for 3d human activity analysis," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 1010–1019.

[74] O. Shchur, M. Mumme, A. Bojchevski, and S. Günnemann, "Pitfalls of graph neural network evaluation," *arXiv preprint arXiv:1811.05868*, 2018.

[75] C. W. Slayman, "Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 397–404, 2005.

[76] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 2009, pp. 117–128.

[77] G. Srinivasan, P. Murley, and H. Tang, "Accurate, predictive modeling of soft error rate due to cosmic rays and chip alpha radiation," in *Proceedings of 1994 IEEE International Reliability Physics Symposium*. IEEE, 1994, pp. 12–16.

[78] Y. Tao, D. Papadias, and Q. Shen, "Continuous nearest neighbor search," in *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 2002, pp. 287–298.

[79] M. Tiator, C. Geiger, and P. Grimm, "Point cloud segmentation with deep reinforcement learning," in *ECAI 2020*. IOS Press, 2020, pp. 2768–2775.

[80] N. J. Wang, J. Quek, T. M. Rafacz *et al.*, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *International Conference on Dependable Systems and Networks, 2004*. IEEE Computer Society, 2004, pp. 61–61.

[81] 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.

[82] T. Xu, B. Tian, and Y. Zhu, "Tigris: Architecture and algorithms for 3d perception in point clouds," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 629–642.

[83] C. Zhang, D. Song, C. Huang, A. Swami, and N. V. Chawla, "Heterogeneous graph neural network," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 793–803.

[84] J.-F. Zhang and Z. Zhang, "Point-x: A spatial-locality-aware architecture for energy-efficient graph-based point-cloud deep learning," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1078–1090.

[85] H. Zhao, Y. Zhang, P. Meng, H. Shi, L. E. Li, T. Lou, and J. Zhao, "Safety score: A quantitative approach to guiding safety-aware autonomous vehicle computing system design," in *2020 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2020, pp. 1479–1485.

[86] Y. Zhou and O. Tuzel, "Voxelnet: End-to-end learning for point cloud based 3d object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4490–4499.

[87] Y. Zhu, "Rtnn: accelerating neighbor search using hardware ray tracing," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 76–89.