FIdelity: Efficient Resilience Analysis Framework for Deep Learning Accelerators

Yi He
University of Chicago
Chicago, IL, USA
yiizy@uchicago.edu

Prasanna Balaprakash
Argonne National Laboratory
Argonne, IL, USA
pbalapra@anl.gov

Yanjing Li
University of Chicago
Chicago, IL, USA
yanjingl@uchicago.edu

***Abstract*—We present a resilience analysis framework, called FIdelity, to accurately and quickly analyze the behavior of hardware errors in deep learning accelerators. Our framework enables resilience analysis starting from the very beginning of the design process to ensure that the reliability requirements are met, so that these accelerators can be safely deployed for a wide range of applications, including safety-critical applications such as self-driving cars.**

Existing resilience analysis techniques suffer from the following limitations: 1. general-purpose hardware techniques can achieve accurate results, but they require access to RTL to perform time-consuming RTL simulations, which is not feasible for early design exploration; 2. general-purpose software techniques can produce results quickly, but they are highly inaccurate; 3. techniques targeting deep learning accelerators only focus on memory errors.

Our FIdelity framework overcomes these limitations. FIdelity only requires a minimal amount of high-level design information that can be obtained from architectural descriptions/block diagrams, or estimated and varied for sensitivity analysis. By leveraging unique architectural properties of deep learning accelerators, we are able to systematically model a major class of hardware errors – transient errors in logic components – in software with high fidelity. Therefore, FIdelity is both quick and accurate, and does not require access to RTL.

We thoroughly validate our FIdelity framework using Nvidia's open-source accelerator called NVDLA, which shows that the results are highly accurate – out of 60K fault injection experiments, the software fault models derived using FIdelity closely match the behaviors observed from RTL simulations. Using the validated FIdelity framework, we perform a large-scale resilience study on NVDLA, which consists of 46M fault injection experiments running various representative deep neural network applications. We report the key findings and architectural insights, which can be used to guide the design of future accelerators.

I. Introduction

Deep learning (DL) accelerators have been deployed in a wide range of application domains, from edge computing, self-driving cars, to cloud servers [8], [16]. Hardware error resilience is a top priority for these accelerators. The importance of resilience for safety-critical applications such as self-driving cars has already been pointed out by Nvidia [10], Tesla [25], and many others. Furthermore, in general, resilience analysis provides better understanding of application/design requirements, and enables efficient architectural exploration to achieve optimal tradeoffs between power, performance, area, and reliability. It also provides a means to quantitatively compare resilience properties of different designs/applications (e.g., for benchmarking purposes). Resilience analysis can even be used to assess the impact of fault attacks (e.g., using hardware trojans, injecting optical/electromagnetic disturbances, exploiting variations, and so on, for malicious purposes), and to guide the design of secure architectures. Because of its importance, resilience analysis should be performed starting from the very beginning of the design process.

There exist several resilience analysis studies targeting DL accelerators [1], [11], [19], [27], but they only focus on memory errors. These studies are not sufficient, because transient errors in logic components, referred to as *logic transient errors* for short, are a major reliability concern. Our results show that, for a DL accelerator in which resilience protection of sequential elements is not provided, the FIT (failure in time) rate of these sequential elements (> 9.5) is significantly higher than the stringent automotive safety requirement (< 0.2), even just as a result of random transient errors that occur infrequently, Therefore, in this paper, we focus on a new resilience analysis framework that targets logic transient errors in DL accelerators.

A well-known resilience analysis approach is to perform large-scale fault injection experiments. However, existing fault injection techniques suffer from several key limitations. In general, RTL-level fault injection techniques (or mixed-mode techniques that combine RTL and software simulations) can achieve accurate results. However, RTL is not likely to be available (or easily modified to test out different implementations) during the early phases of the design process. Even if RTL is available, simulation time would be prohibitively long. On the other hand, software-level fault injection techniques are quick, but their accuracy cannot be guaranteed [4]. Various optimizations for fault injection techniques have been developed for CPUs [9], [13], [20], but they cannot be directly applied to DL accelerators.

To overcome the above challenges, we present FIdelity, which models hardware logic transient errors in software with high fidelity. The accurate mapping is possible because of the following insights: as a special-purpose design, the majority of the hardware operations in DL accelerators closely match software operations. Moreover, because of the well-defined dataflow and scheduling algorithms, all operations that are affected by a given hardware error, and how they are affected by the error, can be systematically derived. **The novel contribution of this work is that, using just a minimal amount of hardware information, our framework is able to generate accurate software fault models for both datapath and control components, without the need**

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00033

to access RTL. As such, the behavior of a logic transient error can be modeled in software to enable quick and accurate software fault injection.

To validate our framework, we apply it to Nvidia's open-source DL accelerator, NVDLA [16], and obtain its software fault models. We then perform 60K RTL fault injection experiments using various representative DNN workloads. By manually analyzing all RTL fault injection cases that lead to non-masked outcomes, we confirm that, for the datapath, the software fault models derived using FIdelity capture the *exact* fault behaviors obtained from RTL simulations. For the control portion, FIdelity's models closely match with RTL results.

Using the validated FIdelity framework, we perform a large-scale resilience study on NVDLA, which consists of 46M fault injection experiments. Our workloads include various representative deep neural networks: CNNs used for image classification tasks (Inception, ResNet, Mobilenet), the Yolo CNN used for object detection tasks, and the Transformer network for language translation tasks. The key findings are:

1. Our results quantitatively demonstrate the crucial need for resilience analysis and protection solutions for DL accelerators. Although DL workloads exhibit certain tolerance to errors, such tolerance alone cannot guarantee that a DL accelerator will meet the resilience requirement of a target application.

2. Our results reveal how hardware design choices, data precision, and correctness metrics affect the overall resilience of the design, as well as fault properties that can be leveraged to develop new resilience techniques.

In summary, the major contributions of this paper are:

1. We create the FIdelity resilience analysis framework for DL accelerators.

2. We thoroughly validate this framework.

3. We use FIdelity to perform large-scale (46M) fault injection experiments, which reveals new resilience knowledge and architectural insights.

This paper is organized as follows. We provide background and motivation in Sec. II. We present the FIdelity framework in Sec. III, and validation methodology and results in Sec. IV. Large-scale fault injection experiments and results are discussed in Sec. V, followed by related work in Sec. VI and conclusions in Sec. VII.

II. Background and Motivation

A. Overview of Deep Learning Accelerators

In this paper, we focus on digital electronic inference accelerators for deep neural networks (DNNs), which are widely deployed in many important application domains such as vision, medical applications, and so on. These accelerators generally share a common dataflow architecture that perform the following main operations (Fig. 1):

(1) **Input/weight fetching and processing**, which fetches inputs/weights from memory and stores them in one or more levels of on-chip buffers. Optional data processing (e.g., padding, compression, etc.) can also be performed.

(2) **input/weight sequencing**, which distributes various input-weight pairs to the compute units. The sequencing order is determined by the specific scheduling and dataflow (reuse) algorithm (e.g., weight reuse, row stationary, etc. [23]).

(3) **Multiply-accumulate (MAC) operations**, which are the core computation primitive for all DNN workloads. These operations are performed by a large number of MAC units.

(4) **MAC output processing and write back**, which includes: a. performing optional linear operations (e.g., bias addition); b. applying a non-linear activation function; c. pooling; and, d. storing the final results to memory.

Different designs may differ in the number of pipeline stages, memory organization, the number and organization of MAC units, and the scheduling/reuse algorithm. However, the high-level dataflow architecture is similar.

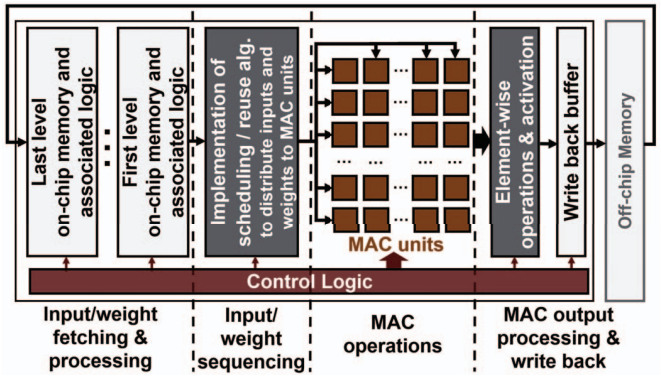


Fig. 1: General Digital DL Accelerator Architecture.

B. Transient Errors: A Major Concern

In this work, we focus on logic transient errors including soft errors and dynamic variations. Soft errors are radiation-induced transient errors caused by neutrons from cosmic rays and alpha particles from packaging materials. While soft errors are not considered a big concern in combinational logic, and soft error rates in flip-flops (FFs) stay roughly constant or even decrease over technology generations, the system-level soft error rate increases as more devices are integrated on a single chip [5], [14], [21], [22]. Moreover, soft error rates can increase when supply voltage is reduced to improve energy efficiency [12], [17]. Another major source of transient errors is dynamic variations, such as voltage droop caused by complex circuit/workload interactions [2].

Transient errors pose critical reliability risks to DL accelerators. However, to the best of our knowledge, there is *no* logic transient error analysis framework that targets DL accelerators, while the frameworks for general-purpose systems all exhibit limitations (as discussed in Sec. I). This motivates us to create the FIdelity framework.

III. The FIdelity Framework

FIdelity is a new logic transient error analysis framework targeting digital DNN inference accelerators. FIdelity provides high-fidelity software models for all single-cycle, single FF bit-flip errors (or multiple single-cycle bit-flips in a single register), which are the most prominent abstraction for transient

errors including soft errors [4] and voltage variations [3]. Thus, it achieves quick and accurate results without relying on the availability of RTL. FIdelity is broadly applicable to a wide variety of DL accelerators because it leverages their common architectural properties. An overview of the framework is shown in Fig. 3.

A. Insights and Novelty

Existing software fault injection techniques model a logic transient error as a single-cycle bit-flip in a single architectural (i.e., software-visible) state, which is highly inaccurate because in reality logic transient errors can result in various architectural effects, including single or multiple bit-flips in one or more architectural states, system time-outs, and so on. For complex general-purpose designs such as CPUs, it is very challenging to model the effects of these errors without detailed RTL models. However, for DL accelerators, we identify four key properties pertaining to resilience analysis, which suggest that the architectural effects of a FF bit-flip can be accurately and systematically derived.

Accelerator Property (1): For a FF bit-flip to cause errors in the final output of a DL application, it must first cause errors in the output neurons of the current DNN layer. Moreover, the bit-flip cannot directly affect the computation in other layers. Thus, to capture the effects of a FF bit-flip, it is *equivalent* to first obtaining the list of faulty output neurons in the current layer, as well as their faulty values, and then determining how these faulty neurons affect the final application output through software simulation.

Accelerator Property (2): As a special-purpose design, the datapath FFs in a DL accelerator closely match the variables in a DNN software framework. For example, in NVDLA, datapath FFs only store the following information: a DNN's inputs, weights, bias values, partial sums, and output values, all of which are visible in software.

Accelerator Property (3): Due to the regular structure and precisely-defined dataflow architecture, in every cycle, the value stored in a datapath FF only affects a deterministic set of output neurons in the current DNN layer. Note that, multiple neurons can be affected by a single-cycle FF value because an important design principle of DL accelerators is to *reuse* the input/weight/partial sum values effectively to optimize energy efficiency [23].

Accelerator Property (4): Control FFs in DL accelerators are classified into two categories: local and global. A local control FF directly interacts with a deterministic set of datapath FFs, so its effects on the output neurons can be derived based on the corresponding datapath FFs. On the other hand, global FFs (e.g., FFs that store the number of kernels or data precision in the current DNN layer) control the computations of a large number of, or even all, output neurons.

Therefore, given a *fault site*, which specifies both the FF where a fault is injected and the cycle during which the fault is injected, *software fault models* that accurately capture hardware logic transient errors can be obtained by answering two questions:

1. How to obtain the set of *faulty output neurons*, i.e., output neurons that are affected by this fault?
2. How to change the values of faulty output neurons to reflect the effects of the fault?

To answer the first question, we create a systematic approach called *Reuse Factor Analysis*, presented in Sec. III-B. The answer to the second question follows the information provided by Reuse Factor Analysis, as discussed in Sec. III-C.

The novelty of our approach is that, using just a few pieces of information – that can be obtained from design plans, block diagrams, architectural descriptions, and estimated values (that can be varied to perform sensitive analysis) – Reuse Factor Analysis can generate accurate software fault models, without the need to access RTL, or the presence of RTL at all.

B. Reuse Factor Analysis

Given a target FF in a DL accelerator design, our Reuse Factor Analysis technique provides information on: 1. the maximum number of faulty neurons that can be generated if this FF experiences a single cycle bit-flip, which is defined as the *reuse factor (RF)* of the FF; 2. the relative location(s) of all possible faulty neuron(s); as well as 3. the order in which these faulty neurons are calculated.

1) ***Reuse Factor Analysis for Datapath FFs****:* For the general accelerator architecture depicted in Fig. 1, the RF of a datapath FF depends on various design parameters such as the scheduling/reuse algorithm, memory organization, and connections from the FF to various compute units. To take these factors into account, we separate datapath FFs in the following partitions, where each partition consists of one or more pipeline stages: 1. before each level of the on-chip memories; 2. between the first-level (L1) on-chip memory and MAC units; 3. inside MAC units; and 4. after MAC units. We also categorize these FFs into different types of variables (inputs, weights, partial sums, and outputs[1]), and refer to them as input FFs, weight FFs, and so on. The pipeline stage and variable type together determine a datapath FF's *category*.

Based on our analysis, we derive the following unique properties in different datapath FF categories:

Datapath RF Property (1): For datapath FFs before each level of the on-chip memories, a single-cycle bit-flip manifests as one incorrect value stored in the corresponding memory. Therefore, their RF values (as well as the relative locations and computation order of faulty neurons) can be determined by the scheduling/reuse algorithm since these on-chip memories are software-managed caches. For example, in NVDLA, there is one level of on-chip memory, and it stores both inputs and weights. The scheduling/reuse algorithm specifies that values stored on-chip should be reused for all MAC operations that involve these values. Therefore, any error occurring on the datapath preceding the on-chip memory can affect all output neurons that use the corresponding input/weight value in the current DNN layer.

[1]Output FFs store output values after accumulation is done, but can precede element-wise operations such as ReLU or bias addition.

TABLE I: Summary of Reuse Factor Analysis for Datapath FFs in DL Accelerators.

Faulty FF positions	Possible variable types	Properties	RF, and computation order / relative locations of faulty neurons
Before each level of on-chip memory	Input, weight, bias	A transient fault manifests as one incorrect value in memory	Specified by scheduling/reuse algorithm
Between L1 on-chip memory & MAC units, and inside MAC units	Input, weight, bias	Datapath RF properties (3), (4)	Obtained by using Algorithm 1
Inside and after MAC units	Partial sum, output	RF = 1	Specified by scheduling/reuse algorithm
After MAC units	Bias	Affect neurons that use the bias	Obtained by using Algorithm 1

Datapath RF Property (2): For output FFs inside and after the MAC units, since each FF corresponds to exactly one output neuron, the RF of all of these FFs equals to 1. Moreover, since the mapping from MAC units to output neurons are defined by the scheduling/reuse algorithm, the locations of the faulty neurons can be obtained as well.

Datapath RF Property (3): All datapath FFs in the same category (i.e., they have the same variable type and belong to the same pipeline stage) have the same RF. For example, in NVDLA, special NaN and zero values are signified using extra FFs in addition to the normal floating point values, and all of these FFs that belong to the same pipelines stage are reused in the same way to produce the same set of output neurons. Therefore, their RF values are the same. This means that we only need to perform Reuse Factor Analysis for each datapath FF category, which is a tractable problem.

Datapath RF Property (4): There are three independent datapath flows: weight FFs or input FFs or bias FFs→partial sum FFs→output FFs. Moreover, for each independent datapath flow, the RF of a FF in pipeline stage i must be greater than or equal to the RF of a FF in stage k, $\forall k > i$, because datapath FFs in later stages are driven by those in earlier stages.

As summarized in Table I, given the above properties, the Reuse Factor Analysis algorithm (Algorithm 1) focuses on input, weight, and bias FFs that are positioned after the L1 on-chip memory, while the RF values and corresponding faulty neuron information of other datapath FFs can be directly obtained from a DL accelerator's scheduling/reuse algorithm.

Algorithm 1 uses inputs 2-4 to account for how many compute units (line 3, capturing the spatial reuse aspect) and how many cycles of computation in each compute unit (line 4, capturing the temporal reuse aspect) can be affected by a single-cycle bit-flip in a target FF[2]. Our algorithm also considers the possibility that a faulty value may stay in the target FF for multiple cycles (line 2). As such, the relationship between the target FF and the compute units is established. Next, the relationship between each compute unit and each output neuron (specified in input 5) is taken into account to link the effects of the target FF to output neurons (lines 5-6).

The inputs of Algorithm 1 are both minimal and sufficient, and they can all be obtained from high-level block diagram or microarchitectural description of the design, or from the scheduling/reuse algorithm. Specifically, in the hardware level, only the relationship between the target FF and the compute units is required. This is because, based on Datapath RF Property (4), a FF cannot drive another FF with a higher RF value. Thus, the compute units that have a connection to the target FF already include all the compute units that the target FF affects, which means that a detailed hardware model (e.g., one that specifies the connectivity between different datapath FFs) is not needed.

The algorithm returns the RF value of the target FF, the corresponding set of unique faulty output neurons, and the

[2]Without loss of generality, we assume that a compute unit is responsible for the computation of one output neuron per cycle, and can produce a new output per cycle. We also assume that partial sums are not reused by multiple output neurons. Algorithm 1 can be adapted easily without these assumptions.

Algorithm 1: Reuse Factor Analysis

Definition: *compute units* are multipliers for input FFs and weight FFs, and accumulators/adders for partial sum FFs and bias FFs.

Inputs:

1. Variable type and pipeline stage of a target FF;
2. FF_value_cycles: the maximum number of cycles for which the target FF holds the same output value;
3. M_l, $\forall l = 0, ..., FF\_value\_cycles - 1$: the set of compute units that use the target FF's value for its computation at the target FF's lth loop (i.e., the lth cycle after the target FF last updates its output value);
4. $in\_effect\_cycles(m)$, $\forall m \in M_l$, and $\forall l = 0, ..., FF\_value\_cycles - 1$: the number of cycles during which a single-cycle value in the target FF is in effect (i.e., is used) by compute unit m;
5. $neurons(m)_{y,l}$, $\forall m \in M_l$, $\forall y = 0, ..., in\_effect\_cycles(m) - 1$, and $\forall l = 0, ..., FF\_value\_cycles - 1$: the set of relative (batch, height, width, channel) indices of the output neurons that are computed in the yth cycle by compute unit m since m starts to use target FF's value at the lth loop. The first neuron in $neurons(M_0[0])_{0,0}$ serves as the reference neuron.

```
1  FaultyNeurons = [];
2  for l ← 0 to FF_value_cycles − 1 do
3      for m ∈ M_l do
4          for cycle ← 0 to in_effect_cycles(m) − 1 do
5              for neuron ∈ neurons(m)_{cycle,l} do
6                  insert((neuron, l), FaultyNeurons);
7              end
8          end
9      end
10 end
11 RF = length(FaultyNeurons);
12 return RF, FaultyNeurons;
```

order that they are generated. The order is indicated by a time stamp (l in line 6 of Algorithm 1) attached to each faulty neuron. The faulty neurons are represented in relative indices, since the actual set of faulty output neurons (in absolute indices) depends on which cycle the fault is injected. To model a random fault injection cycle, we randomly choose one set of faulty neurons from all possible sets. Moreover, if the target FF holds its output for more than 1 cycle, we further randomly choose an integer p between 0 and $FF\_value\_cycles - 1$, and only consider the subset of $FaultyNeurons$ whose time stamp l is greater than or equal to p.

*2) **Datapath FF Reuse Factor Analysis Examples**:* Figure 2 demonstrates how Algorithm 1 is applied to various datapath FFs in DL accelerators. In all examples, the workload is a convolution layer and the computations are done for a single batch.

In Fig. 2(a), we demonstrate the datapath of a NVDLA-like accelerator. There are k^2 parallel MAC units. The same inputs are sent to all MAC units, but the weights are different. In each MAC unit, the weights are reused for multiple operations, but a new input (shared by all MAC units) is fetched for each operation. The MAC units perform computations in the row-major order, and they compute the output neurons with the same (height, width) position in k^2 consecutive channels at the same time. We show four examples in Fig. 2(a): targets a1-a3 are weight FFs, and target a4 is an input FF. The output of target a1 is only sent to one multiplier (m_{00}). However, downstream in the weight FFs→partial sum FFs→output FFs datapath flow, the max FF_value_cycles is t, i.e., the output of a1 is eventually sent to another FF (target a2 in this example) that keeps the same output for t cycles before it reaches multiplier m_{00}. Moreover, a2's $in\_effect\_cycles(m_{00})$ value is 1. Thus, a1's $in\_effect\_cycles(m_{00})$ is t, and a fault in target a1 affects t consecutive neurons in one output channel. Target a2 affects the same set of faulty neurons as a1. The difference is that a2's $FF\_value\_cycles = t$, while $in\_effect\_cycles(m_{00}) = 1$. Thus, a fault in a2 affects a random number of neurons between 1 to t. In the case of target a3, the only difference from target a2's case is that the RF is 1 instead of t because the faulty value only lasts for 1 cycle. For target a4, its value is sent to k^2 multipliers which produce k^2 output neurons each cycle, so its RF is k^2. All of a4's faulty neurons belong to the same 2D matrix position and span k^2 consecutive output channels.

Figure 2(b) shows the datapath of an Eyeriss-like accelerator, where target b1 is a weight FF, b2 is an input FF, and b3 is a bias FF. In this design, MAC units are arranged as a $k \times k$ systolic array. The MAC units belonging to the same column perform the MAC operations for one row of the output neurons in consecutive cycles, and consecutive MAC columns compute consecutive rows of the output neurons. Each MAC unit computes a row of partial sum values using one row of inputs and one row of weights, and the corresponding row of output neurons are obtained by accumulating the partial sum values of all MAC units in each column. In each cycle, a weight value is passed from one MAC unit to its neighbor in the next column so that it can be reused in the computations of different output rows. Therefore, the RF of b1 is k, and the set of faulty output neurons occupy k consecutive rows in the same column. Meanwhile, an input value is reused by the MAC units diagonally, and it is also reused inside each MAC unit to compute output neurons in consecutive channels and consecutive columns. Here we assume that, at the fault site of target b2, the FF stores an input that is only needed for computing the last output column, so it is only reused across t output channels. Therefore, the RF of b2 is $k \times t$. The set of faulty output neurons for b2 are located in t consecutive channels. Within each channel, they also occupy k consecutive rows in the last column. In the case of target b3, since it is connected to only one compute unit (BiasAdd) and is not reused temporally, its RF is 1.

*3) **Reuse Factor Analysis for Control FFs**:* Control FFs in a DL accelerator can be broadly classified into two categories, *global* and *local*.

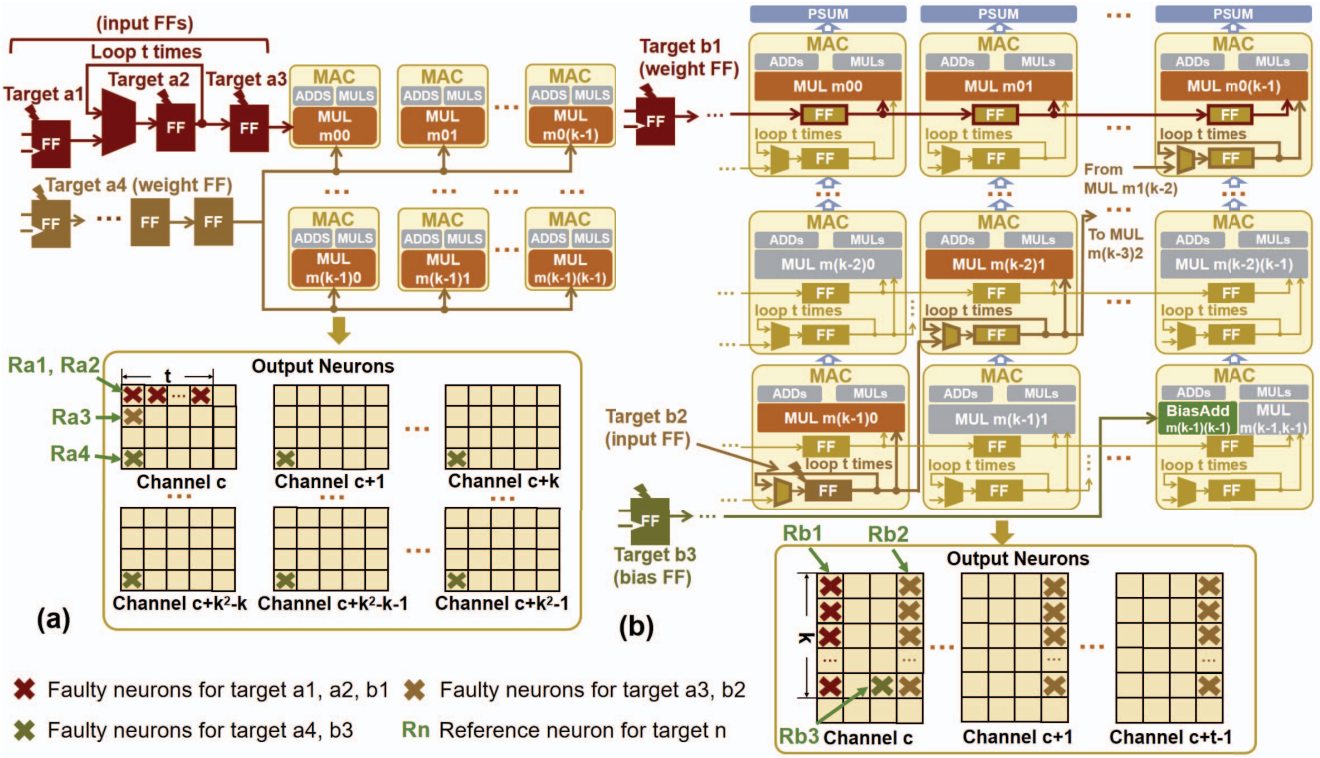
Global control FFs include those that store configuration information for the execution of an entire DNN layer (e.g., the size of inputs and weights, base addresses, and data types), and FFs that are responsible for sequencing data to/from on-chip memories (e.g., counters for advancing the address of a memory to read input/weight values). These FFs affect a large number of, if not all, output neurons. Moreover, one observation based on our own experiments is that, if the number of faulty neurons is large, it is very likely that an application output error or a system anomaly (e.g., time-out) would occur. For example, when $\sim 10\%$ of all output neurons in a layer are faulty, the probability that the final application output is correct is only $\sim 15\%$. Therefore, in FIdelity, we model a fault injected to an active global control FF as one that always results in application error or system anomaly.

Local control FFs, on the other hand, are closely coupled with certain datapath FFs. For example, the control FF that indicates whether or not the output of a multiplier is valid and ready to be passed to the next pipeline stage is only coupled with the output FFs of the corresponding multiplier. Therefore, the RF value and the set of faulty output neurons of the valid signal are the same as those derived for the output FFs in the same multiplier. Also, it is possible for a local control FF to affect multiple datapath FFs – for example, if the valid signal controls the outputs of multiple parallel multipliers. In this case, we take the sum of the RF values and the union of $FaultyNeurons$ from all the datapath FFs it is coupled with. Another example of a local control FF is the select signal of a datapath multiplexer, which only affects the output of the multiplexer.

C. Deriving Faulty Output Neuron Values

After performing Reuse Factor Analysis on an accelerator design, our next task is to determine how to change the values of the faulty output neurons, and we consider datapath and local control FFs separately.

For datapath FFs, recall Accelerator Property (2) in Sec. III-A, i.e., each datapath FF already corresponds to a



Input 1:	Target a1	Target a2	Target a3	Target a4	Target b1	Target b2	Target b3
Input 2:	1	t	1	1	1	t	1
Input 3:	$M_0=[m_{00}]$	$M_l=[m_{00}]$, $l \in [0, t)$	$M_0=[m_{00}]$	$M_0=[m_{00}, m_{01}, \dots, m_{(k-1)(k-1)}]$	$M_0=[m_{00}, m_{01}, \dots, m_{0(k-1)}]$	$M_l=[m_{(k-1)0}, m_{(k-2)1}, \dots, m_{0(k-1)}]$, $l \in [0, t)$	$M_0=[m_{(k-1)(k-1)}]$
Input 4:	t	1	1	$1, \forall m \in M_0$	$1, \forall m \in M_0$	$1, \forall m \in M_l, l \in [0, t)$	1
Input 5: Ref. position: [h, w, c]	neurons $(m_{00})_{y,0}$= {[h, w+y, c] | y ∈ [0,t)}	neurons $(m_{00})_{0,l}$ = {[h, w+l, c] | l ∈ [0, t)}	neurons $(m_{00})_{0,0}$ ={[h, w, c]}	neurons$(m_{ab})_{0,0}$ ={[h, w, c+ak+b] | a ∈ [0, k), b ∈ [0, k)}	neurons$(m_{0a})_{0,0}$ ={[h+a, w, c] | a ∈ [0,k)}	neurons$(m_{ab})_{0,l}$ ={[h+b, w, c+l] | l ∈ [0, t), a+b=k-1}	neurons $(m_{(k-1)(k-1)})_{0,0}$ = {[h, w, c]}
Output: RF	t	t	1	k^2	k	k × t	1
Output: FaultyNeurons	[([h, w, c],0), ([h, w+1, c],0), ..., ([h, w+t-1, c],0)]	[([h, w, c],0), ([h, w+1,c],1), ..., ([h, w+t-1, c],t-1)]	[([h, w, c],0)]	[([h, w, c],0), ([h, w, c+1],0), ..., ([h, w, c+k^2-1],0)]	[([h, w, c],0), ([h+1, w, c],0), ..., ([h+k-1, w, c],0)]	[([h, w, c],0), ([h+1, w, c],0), ..., ([h+k-1, w, c],0), ..., ([h+k-1, w, c+t-1],t-1)]	[([h, w, c],0)]

Fig. 2: Datapath FF Reuse Factor Analysis for (a) A NVDLA-like accelerator; (b) An Eyeriss-like accelerator.

software-visible state. In other words, for each datapath FF bit-flip, there exists an *equivalent* bit-flip in software, which can be used to calculate the values of the corresponding faulty neurons. For example, if the faulty FF corresponds to a bit of a weight value, then the whole set of faulty neurons are computed by using the faulty weight value.

For local control FFs, the effects of a fault on the corresponding datapath value(s) are not deterministic. Recall the valid signal example presented in the previous session, which is a local control FF that is used to indicate whether the output of a multiplier is valid or not. If a fault flips its value from 'valid' to 'not valid', effectively it means that the result generated by the multiplier at this cycle is dropped, so its value will be replaced by the next output generated by the same multiplier. However, if the value is flipped from 'not valid' to 'valid', then a non-deterministic intermediate value will be incorrectly interpreted as a valid output. As another example, if a multiplexer's select signal encounters a fault, then any input of the multiplexer (including inputs that are not currently driven) may be passed to the output, resulting in a non-deterministic output value. Therefore, for each neuron in the $FaultyNeurons$ set of a local control FF, we replace its value with a random value.

D. FIdelity's Fault Injection Flow

The accurate software fault models, derived based on our Reuse Factor Analysis algorithm, is the key component in the FIdelity framework. Using the software fault models, FIdelity's entire fault injection process is shown in Fig. 3.

The inputs of FIdelity, summarized in Fig. 3, include: 1. a DNN workload (e.g., its layer type, kernel size, etc.); 2. a raw FF FIT rate, which is the probability that a transient error occurs in one FF, and its value depends on the technology node

of the design; 3. the scheduling/reuse algorithm and hardware configuration parameters of the accelerator design, which are available from architectural descriptions; and 4. high-level microarchitecture information. The required microarchitecture information can be estimated from design plans/sketches, block diagrams, or previous design generations, and the estimated values can be varied for sensitivity analysis to obtain resilience bounds. As the design process evolves and more detailed hardware information becomes available, FIdelity's inputs are more accurate, which in turn improves its accuracy.

The output of FIdelity is the accelerator FIT (failure in time) rate, a standard resilience metric, which denotes the number of system failures in 1 billion device hours, where system failure in the context of DL accelerators is either DNN application output error or system anomaly (e.g., time-out).

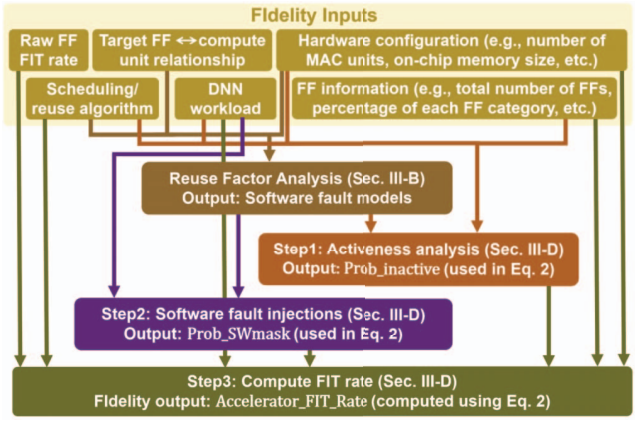


Fig. 3: Overview of the FIdelity Framework.

The fault injection flow of FIdelity consists of three steps. In **step 1**, we perform FF activeness analysis to account for error masking scenarios, since a fault injected to an inactive FF will always be masked.

The probability that a FF is inactive can be estimated by simply assuming a reasonable range. Given a few pieces of high-level microarchitecture information, a more detailed analysis can be performed by considering three classes of inactive FFs during the execution of a DNN layer:

Class 1. *Component not used*: a FF belongs to a hardware component that remains idle for the entire execution of the workload. For example, if the weights are not compressed, then all FFs in the decompression unit are idle.

Class 2. *Signal not used*: a FF belongs to an active hardware component, but the FF itself stays inactive the entire time. For example, FFs responsible for floating point calculations remain inactive if the current workload uses an integer representation.

Class 3. *Temporally not used*: a hardware component – and therefore, all FFs in this component – is inactive for a portion of the time. For example, the MAC units are inactive when they are waiting for data to be fetched from memory.

These classes are mutually exclusive and complete. The first two classes are determined by the DNN layer's characteristics (e.g., whether it uses integer or floating point representation, or whether its weight values are compressed). For the "temporally not used" class, the percentage of time for which a hardware component is inactive can be estimated using high-level architectural information. For example, in NVDLA, a performance tool, which uses information solely obtained from the scheduling/reuse algorithm and hardware configuration parameters such as the number of MAC units, is available [15]. Given a workload (a DNN layer), the tool breaks down the time required to fetch data and to perform MAC/linear/non-linear/etc. operations. This breakdown indicates how long a FF positioned before CBUF (NVDIA's on-chip memory) is inactive, or how long a FF inside a MAC unit is inactive. Note that, it is possible for a FF to be inactive when the corresponding hardware component is active. However, based on our study, we found that this case constitutes a very small fraction of all cases, so it is not included in our analysis.

Given a DNN layer r, we define Perc_inactive(cat, cl, r) for FFs that are mapped to software fault model cat and inactive class cl, which equals 1 if $cl \in Class\ 1, 2$, and equals to the percentage of inactive time of the corresponding component if $cl \in Class\ 3$. Let FF_Perc(cat, cl) represent the percentage of FFs in cl out of all FFs in cat. Then, the average probability that an FF belonging to cat is inactive during the execution of r, denoted as Prob_inactive(cat, r), is calculated as shown in Eq. 1.

$$\text{Prob\_inactive}(cat, r) = \sum_{cl} \text{FF\_Perc}(cat, cl) \times \text{Perc\_inactive}(cat, cl, r) \quad (1)$$

In **step 2**, given the accurate software fault models based on our Reuse Factor Analysis, we perform large-scale software fault injection experiments for a statistically significant number of samples using each software fault model, and record the outcome of each experiment run.

For DNN applications, the outcome of logic transient errors can be classified into two categories: 1. *masked*, which means that the final output generated in the presence of a fault is sufficiently similar to the golden output, such that the effects of the fault can be considered negligible; and 2. *system failure*, which captures all non-masked cases including application output error and system anomaly. The outputs of step2 are the probabilities of masked cases for all software fault models in each layer of a given DNN application (*Prob_SWmask*$(cat, r)\ \forall r$). Note that, *Prob_SWmask*(global control FFs, $r) = 0\ \forall r$, since this is how FIdelity models faults in active global control FFs.

In **step 3**, we calculate $Accelerator\_FIT\_rate$ using Eq. 2. Let FIT_raw be the raw FF FIT rate, N_{ff} be the number of FFs in the accelerator, $FF\_Perc(cat)$ be the percentage of FFs whose faulty behavior is modeled by software fault model cat, and $exec\_time(r)$ be the execution time of layer r in a given DNN application, which can be estimated or obtained based on high-level architectural information. Accelerator_FIT_Rate is the product of the probability that a fault occurs in a FF (FIT_raw), and the probability of an application error or a system anomaly, given that a fault has occurred in a single FF:

$$\begin{aligned}\text{Accelerator\_FIT\_rate} = FIT\_raw \times N_{ff} \times \sum_{r} [\ exec\_time(r) \\ \times \sum_{cat} \text{FF\_Perc}(cat) \times (1 - \text{Prob\_inactive}(cat, r)) \\ \times (1 - \text{Prob\_SWmask}(cat, r))\ ] / \sum_{r} exec\_time(r)\end{aligned} \quad (2)$$

TABLE II: NVDLA Software Fault Models for Convolution (Conv), Fully Connected (FC), & Matrix Multiplication (MatMul) layers.

Datapath positions / Control type	Variables	%FF	RF	Faulty Neuron Information	Software Fault Model
Before CBUF	Input	2.5%	Total number of neurons using the target FF value	**Conv:** All neurons that use the input value are faulty. The locations depend on layer parameters such as stride, dilution, and the size of the kernels. **FC:** All neurons are faulty. **MatMul:** All neurons in the output row that this input value maps to are faulty.	One random bit-flip at one randomly chosen input, affecting all neurons that use the input value.
	Weight	4.8%		**Conv:** All neurons in the output channel that this weight value maps to are faulty. **FC:** One neuron in each batch uses the weight value and thus is faulty. **MatMul:** All neurons in the output column that this weight value maps to are faulty.	One random bit-flip at one randomly chosen weight, affecting all neurons that use the weight value.
Between CBUF & MAC units, and inside MAC units	Input	16.2%	16	**Conv:** 16 neurons in the same 2D matrix position and spanning 16 consecutive output channels use the same faulty value. See target a4 in Fig. 2(a) as an example. **FC:** 16 consecutive output neurons use the same faulty value. **MatMul:** 16 consecutive neurons in the output row that this input value maps to are faulty.	One random bit-flip at one randomly chosen input, affecting the corresponding 16 faulty neurons.
	Weight	21.6%	16	**Conv:** All or a subset of 16 neurons belonging to the same output channel, and are consecutive in the same row, use the same faulty value. See target a1/a2 in Fig. 2(a) as an example. **FC:** One out of 16 output neurons are faulty, for a total number of $<= 16$ faulty neurons. **MatMul:** All or a subset of the 16 consecutive neurons in the output column that this weight value maps to are faulty.	One random bit-flip at one randomly chosen weight, affecting the corresponding $<= 16$ neurons.
Inside and after MAC units	Output, partial sum	37.9%	1	The one neuron that uses the target FF value is faulty.	One random bit-flip at one randomly chosen output neuron or partial sum.
Local control	N/A	5.7%	1	The faulty neuron is the same as the one that is derived for the datapath FF that the target FF controls.	Random faulty value at one randomly chosen output neuron.
Global control	N/A	11.3%	ALL	A large number of output neurons are faulty.	System failure.

E. Broader applicability

Although we focus on logic transient errors, FIdelity can be used to model memory errors as well, based on Datapath RF property (1) in Sec.III-B1. Reuse factor analysis for errors occurring in one memory word is the same as that for the datapath FFs that serve as input data to the memory (Table I, row2). For multiple memory errors, the set of faulty neurons are the union of the faulty output neurons of each error. After the memory software fault models are established, the fault injection flow can be carried out exactly as shown in Fig. 3.

TABLE III: Workloads (DNN Layers) Used for FIdelity Validation. Precision: FP16.

Networks	Layer	Dataset
Inception	A 3×3 Conv layer in inception module.	Imagenet
ResNet50	A 3×3 Conv layer in residual block.	
Transformer	A FC layer in feed-forward network.	IWSLT2014
	A MatMul layer in attention.	
RNN	A FC layer in LSTM.	UCI HAR
Yolo	A 3×3 Conv layer in residual block.	COCO

IV. FIdelity Framework Validation

We use NVDLA as a case study to demonstrate that FIdelity's software fault models are accurate. Our methodology is to first perform RTL injection to obtain golden references on the number of faulty neurons, their relative positions, as well as the order in which they are generated, and then compare the software fault models generated using Reuse Factor Analysis with the information obtained from RTL simulations.

A. Accurate Software Fault Models for NVDLA

We apply the Reuse Factor Analysis to NVDLA. The design of NVDLA is similar to our example in Fig. 2(a), which is configured with $k = 4$ and $t = 16$ in our case study. Its software fault models are shown in Table II for three types of representative DNN layers: convolution, fully-connected, and matrix multiplication.

B. Validation Methodology Details

RTL fault injection experiments are performed using Synopsys VCS for various representative workloads shown in Table III. After a fault site is selected for a given workload, we perform RTL simulation until all output neurons of the current layer are generated, or until the simulation reaches a system time-out value. By comparing the output neuron results against

the fault-free results, we obtain the set of faulty neurons and their faulty values. We ran experiments with 10K fault sites for each workload to achieve 95% confidence interval. Moreover, in order to validate FIdelity's results for global control FFs, for all experiments where faults are injected into global control FFs, we further perform mixed-mode simulations to check the correctness of the final DNN application outputs (correctness metrics are discussed in Sec. V and specified in Table IV).

Out of the 60K total RTL fault injection experiments, 9956 generate errors in the output neurons, and 72 lead to system time-out (the time-out cases are all due to faults in global control FFs). For all of the 10028 cases where the injected faults are not masked, we use the same fault sites to derive the corresponding software fault models from Table II. For datapath FFs and local control FFs, we perform software fault injection using FIdelity's software fault models in TensorFlow (which we have modified to support all software fault models). Each software fault injection experiment generates a set of faulty neurons and their faulty values, which are manually compared with those obtained form RTL simulations.

C. Results and Discussions

For each of the datapath FF cases (8262 total), our software fault model generates *exactly* the same set of faulty neurons and faulty values as the RTL result. Our model for global control FFs, i.e., faulty global control FFs always result in system failures, is also accurate. The RTL+TensorFlow mix-mode simulation results suggest that only ∼9.5% of faults injected to global control FFs are masked.

For the local control FF cases (138 total), RTL simulation results confirm that indeed only one output neuron is faulty (RF=1) in each case. Moreover, our analysis derives the same faulty neuron as the RTL simulation result in each case. In terms of the faulty values, although our results differ from RTL simulation results, we expect that, with a sample set that is sufficiently statistically-significant, the results derived using the FIdelity framework will be similar to RTL simulation results. This is because, as discussed in Sec. III-B3, the behavior of a fault in a local control FF is non-deterministic and its effects can be approximated using a random faulty value.

In summary, the accuracy of the software fault models generated by the Reuse Factor Analyses algorithm in our FIdelity framework is thoroughly validated.

V. Large-Scale Resilience Study

We apply the validated FIdelity framework to NVDLA, and perform large-scale software fault injection experiments in TensorFlow (using software fault models presented in Table II) to obtain resilience results for four typical CNNs (Inception, ResNet, MobileNet, Yolo) and the Transformer network. The details of the experiments are shown in Table IV. Note that, the FP16 networks are taken from public resources, and we trained the INT16 and INT8 models with TensorFlow's support for quantization.

TABLE IV: FIdelity's Fault Injection Experiment Setup.

Platform: Tensorflow (modified to support FIdelity's fault injection flow)			
Total number of Experiments: 46M			
DNN Workload	Inception, ResNet50, MobileNet	Transformer	Yolo
Dataset	Imagenet, Cifar10	IWSLT14	COCO
Correctness Metric	Top 1 label match	$< 10\%/20\%$ BLEU score difference	$< 10\%/20\%$ precision difference
Data Precision	FP16, INT16, INT8	FP16	FP16

The correctness metric of a DNN application, which determines whether the final output is correct or not, is an important consideration. For Inception, ResNet, MobileNet, which are CNNs used for image classification tasks, we compare the top1 label of each fault injection experiment with the top1 label obtained from the fault-free execution, and the application output is considered correct if they match. For the Transformer and Yolo networks, each application output is given a quality score, and an output can be considered correct even if its score does not match the fault-free score exactly. Here we apply two metrics similar to previous work [11]: an application output of the Yolo or Transformer network is considered correct if the difference of its score is within 10% or 20% of the score obtained from the fault-free execution.

Our resilience analysis results are shown in Figs. 4 and 5, breaking down the Accelerator_FIT_rate contributions from datapath, local, and global FFs. The raw FF FIT rate used to derive our results is 600/MB for soft errors [7]. Other raw FF FIT rates (e.g., for voltage variations, or soft errors in a different technology node) can be used, and the general conclusions of our results remain the same.

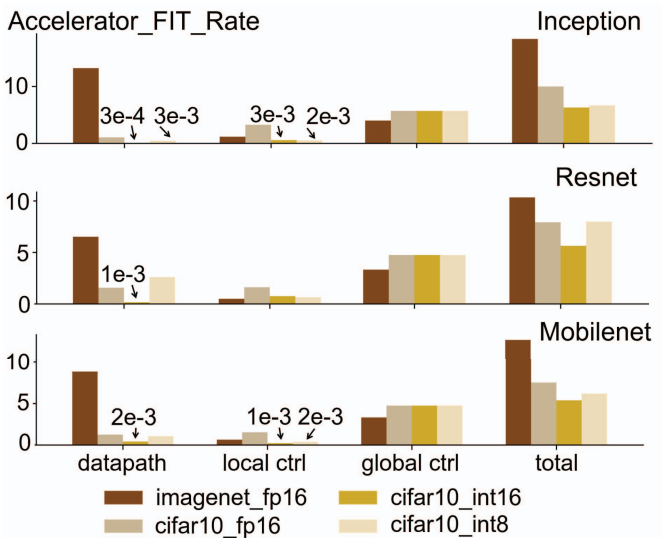
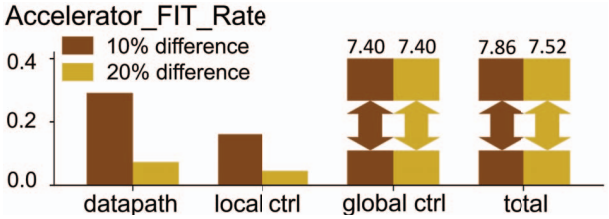
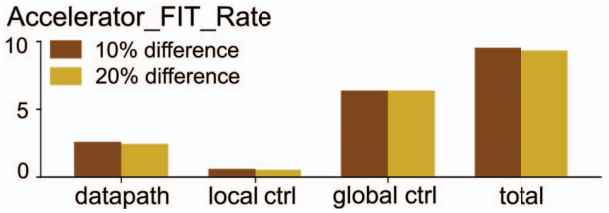


Fig. 4: Accelerator_FIT_Rate Values for Inception, Resnet, and Mobilenet.

We present five key results that are revealed by our study below.



(a) Transformer



(b) Yolo

Fig. 5: Accelerator_FIT_Rate Values for Transformer & Yolo.

Key result (1): There is a need for resilience protection in the logic portion of DL accelerators. One objective of this study is to find out if NVDLA's resilience level can meet the ASIL-D level of the ISO26262 standard (i.e., the highest level of automotive safety standard) [6], if the FFs in NVDLA are left unprotected from transient errors.

According to the safety requirement, the overall FIT rate for an entire self-driving car chipset must be $<$10. We follow the standard approach [11] that assigns a fraction of the FIT rate requirement to each individual hardware component based on the area of the component, and estimate that the FIT rate requirement for all FFs in NVDLA must be $<$0.2 since the FFs in NVDLA occupies $\sim 2\%$ of the total chipset area [28].

However, our results in Figs. 4 and 5 show that, without resilience protection, NVDLA's Accelerator_FIT_rate values are significantly higher than 0.2. For example, for the Yolo network, which is widely used for object detection tasks for self-driving cars, the FIT rate is 9.5 when the 10% precision difference is used as the correctness metric.

In general, various DNN applications are subject to various resilience requirements. Safety-critical applications impose high resilience standards as shown in the self-driving car example above. Moreover, even for non-safety-critical applications, the rate of hardware errors can be quite high under certain operating conditions and environments, which will result in significant degradation in inference accuracy. Thus, resilience analysis and resilience protection techniques are essential to ensure that the resilience target of any given application is met under all conditions.

Key result (2): There is a need for resilience analysis and protection for datapath and local control FFs in DL accelerators. Our results show that global control FFs contribute to the largest portion of the Accelerator_FIT_rate values. If all global control FFs are protected, is there still a need to perform resilience analysis and provide resilient protection for the other FFs?

The answer is yes. In Fig. 6, we show the Accelerator_FIT_rate values for three CNN applications assuming that the raw FIT rate of all global control FFs is 0. We can see that the FIT rates are still larger than 0.2, the largest FIT rate allowed by the automotive safety standard. Therefore, frameworks such as FIdelity is crucial to analyze the resilience properties of datapath and local control FFs in DL accelerators.

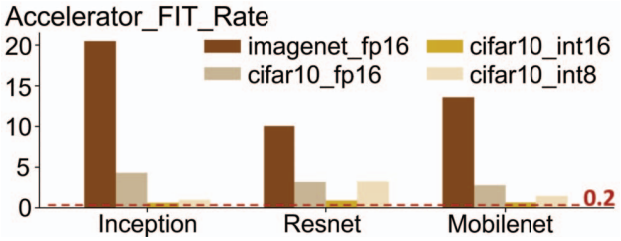


Fig. 6: Accelerator_FIT_Rate Values Assuming that Global Control FFs are Protected.

Key result (3): The correctness metric can impose a large impact on Accelerator_FIT_Rate. For example, in Fig. 5, the FIT rates for datapath and local control FFs are very different when different correctness metrics (10% vs. 20% BLEU score difference) are used. Therefore, application characteristics and requirements must be taken into consideration when resilience analysis is performed.

Key result (4): Data precision also influences Accelerator_FIT_rate. From Fig. 4, we can see that the FP16 networks result in higher Accelerator_FIT_rate values compared to their INT16 and INT8 counterparts in most cases. One possible reason is that the actual dynamic range of FP16 is much larger than INT16 and INT8, which allows bigger perturbations in the presence of errors, and thus results in a higher probability of application output errors according to Key result (5). We can also see that the FIT rates of the INT8 networks are in general higher than those of the INT16 networks. One hypothesis is that, due to the precision loss, if INT8 and INT16 networks are quantized with similar min and max values, then the same absolute perturbation that a fault brings to a neuron will result in a bigger effect in INT8 networks than INT16 networks, resulting in a higher number of Top 1 label mismatches.

Key result (5): Large perturbations in the output neurons are more likely to cause application output errors than smaller perturbations. We separate a subset of the fault injection experiments, the ones that result in one faulty output neuron from the FP16 Inception, Resnet, and Mobilenet networks, into two cases: 1. the difference between the faulty and fault-free values is $\leq$100; and 2. the difference is $>$100. In the former case, there is only a $< 4\%$ probability that a fault results in an incorrect application output; while in the latter, this probability is $> 45\%$. These results suggest that large perturbations in the values of faulty output neurons are more likely to cause application output errors.

Architectural Insights: The key results revealed by FIdelity provide architectural insights to guide resilience designs, and can inspire new resilience techniques. As an example, if errors in certain FF categories contribute more to Accelerator_FIT_rate than others, the design may be optimized by min-

imizing the number of FFs in these categories. Alternatively, selectively protecting only the FFs in these categories may be sufficient to achieve a given resilience target while minimizing system-level costs. As these resilience-critical FF categories are workload dependent, the selective protection scheme can be implemented in an adaptive manner. As another example, hardware-software co-design techniques can be explored, by bounding the values of output neurons based on Key result (5), or by experimenting with different data precision based on Key result (4), to achieve higher resilience.

VI. Related Work

Since resilience is a top priority in DL accelerators, various resilience studies targeting these accelerators exist. However, they mainly focus on memory errors, e.g., soft errors that occur in SRAMs [11], or errors caused by reduced memory voltage [1], [19], [27]. In previous work, transient errors in datapath FFs are modeled as single bit-flips in a single architectural state [11], which is highly inaccurate. Furthermore, previous work either assumes that control FFs have little resilience impact [11], or that a faulty control FF always causes visible system anomaly [24], which is also highly inaccurate. For example, in NVDLA, 34% of the control FFs (5.7% overall) can only affect at most one output neuron, and their resilience impact is usually small, while the rest of the control FFs have global effects (see Table II). To the best of our knowledge, this is the first detailed resilience study on logic transient errors in DL accelerators.

Large-scale statistical fault injection is a well-known approach for analyzing logic transient errors [3], [4], [18], [26]. However, techniques that produce accurate results require access to a RTL model to perform detailed fault injection experiments, which takes a prohibitively long time. We quantitatively compare RTL simulation time, mixed-mode simulation time, and the time required to perform FIdelity's software fault injection experiments, for all workloads in Table III. For NVDLA, FIdelity achieves >10000X and 40X-2200X speedup vs. RTL and mixed-mode simulations, respectively, while achieving similar accuracy as validated in Sec. IV[3].

On the other hand, software fault injection techniques are quick (similar to FIdelity), but their results are highly inaccurate. Again using the workloads in Table III, we compare the accurate Accelerator_FIT_Rate values obtained from FIdelity with those obtained from a naive software fault injection technique where single bit-flips are injected to the architectural states. We found that the naive technique underestimates NVDLA's Accelerator_FIT_Rate by up to 25X across different workloads. Such inaccurate results are misleading and can impose significant safety/reliability risks.

Various techniques have also been proposed to optimize the fault injection process. However, many of these techniques require access to RTL [9], [13]. Moreover, these techniques are specifically designed for CPUs, but the architectural properties of DL accelerators are fundamentally different from CPUs. For example, both MeRLiN [9] and Relyzer [20] aim to reduce the number of fault sites by identifying CPU instructions that generate equivalent/similar faults, so they are not applicable for DL accelerators. Raven [13] obtains the error probability of each hardware block through RTL simulations, which takes less time than simulating an entire design, and then it combines these probabilities to obtain the final FIT rate. Raven also cannot be applied to DL accelerators because an error that occurs in a hardware block may not lead to a DNN application output error or system anomaly. Thus, FIT rate estimations obtained using the Raven approach are inaccurate (pessimistic).

AVF (Architectural Vulnerability Factor) can be used to analyze the effects of transient errors in CPUs without performing fault injection experiments. However, it relies heavily on the architecture and microarchitecture of microprocessors. Thus, it cannot be applied to DL accelerators.

[3]It is infeasible to directly compare Accelerator_FIT_Rate calculated using FIdelity vs. RTL or mixed-mode simulations because it will take $> 100M$ CPU hours to perform RTL simulation to reach statistically significant results.

VII. Concluding Remarks

FIdelity is a quick and accurate framework for analyzing the resilience of deep learning accelerators, which is essential because resilience is a top priority of these accelerators. FIdelity performs high-speed software fault injection to obtain resilience results, and achieves the level of accuracy similar to RTL or mixed-mode fault injection techniques. The most crucial component of this framework is the detailed and comprehensive software fault models, which capture the effects of hardware faults with high fidelity. These software fault models can be systematically derived given high-level architectural/hardware information, without the need to access RTL, using a new technique called Reuse Factor Analysis. Reuse Factor Analysis leverages unique characteristics that are common among DL accelerators. Therefore, the FIdelity framework is widely applicable to many deep learning accelerator designs.

Enabled by the FIdelity framework, for the first time we are able to perform accurate large-scale resilience analysis, from which we gain new resilience knowledge and architectural insights, such as how the behaviors of logic transient errors are influenced by hardware design choices, workloads, data precision, and resilience metrics. Our study also quantitatively demonstrates that it is important to investigate resilience properties and develop efficient resilience techniques to mitigate various sources of hardware errors in DL accelerators. Resilience analysis frameworks, such as FIdelity, will play a major role in this effort and inspire new resilience studies.

Acknowledgement

We thank all anonymous reviewers, Professors Hank Hoffman, Michael Maire, and Rick Stevens of the University of Chicago, and Dr. Daniele Scarpazza of Citadel for their advice and assistance. Computing resources for our experiments were provided by the University of Chicago Research Computing Center and the Argonne Leadership Computing Facility.

REFERENCES

[1] N. Chandramoorthy *et al.*, "Resilient low voltage accelerators for high energy efficiency," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 147–158.

[2] E. Cheng, "Cross-layer resilience to tolerate hardware errors in digital systems," Ph.D. dissertation, Stanford University, 2018.

[3] E. Cheng, S. Mirkhani, L. G. Szafaryn, C.-Y. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose, and S. Mitra, "Clear: Cross-layer exploration for architecting resilience - combining hardware and software techniques to tolerate soft errors in processor cores," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2897937.2897996

[4] H. Cho *et al.*, "Quantitative evaluation of soft error injection techniques for robust system design," in *Proceedings of the 50th Annual Design Automation Conference*, 2013, pp. 1–10.

[5] B. Gill *et al.*, "Comparison of alpha-particle and neutron-induced combinational and sequential logic error rates at the 32nm technology node," in *IEEE International Reliability Physics Symposium Proceedings*, 2009, pp. 199–205.

[6] International Organization for Standardization, "Iso 26262-1:2018 road vehicles – functional safety," https://www.iso.org/standard/68383.html, 2018.

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

[8] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," *CoRR*, vol. abs/1704.04760, 2017. [Online]. Available: http://arxiv.org/abs/1704.04760

[9] M. Kaliorakis, D. Gizopoulos, R. Canal, and A. Gonzalez, "Merlin: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 241–254.

[10] S. Keckler, "High performance computing in a world of embedded intelligence," https://www.esweek.org/sites/default/files/ES_Week_101419_Keckler.pdf, 2019.

[11] G. Li *et al.*, "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*, ser. SC '17, 2017, pp. 8:1–8:12.

[12] N. N. Mahatme *et al.*, "Impact of supply voltage and frequency on the soft error rate of logic circuits," *IEEE Trans. Nucl. Sci.*, vol. 60, no. 6, pp. 4200–4206, 2013.

[13] S. Mirkhani, S. Mitra, C. Cher, and J. Abraham, "Efficient soft error vulnerability estimation of complex designs," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2015, pp. 103–108.

[14] P. Nsengiyumva *et al.*, "A comparison of the seu response of planar and finfet d flip-flops at advanced technology nodes," *IEEE Trans. Nucl. Sci.*, vol. 63, no. 1, pp. 266–272, 2016.

[15] NVDLA, "Nvdla opensourced performance," https://github.com/nvdla/hw/tree/nvdlav1/perf, 2018.

[16] NVIDIA Corporation, "Nvdla open source project," http://nvdla.org/primer.html, 2018.

[17] R. Pawlowski *et al.*, "Characterization of radiation-induced sram and logic soft errors from 0 . 33v to 1 . 0v in 65nm cmos," in *Custom Integrated Circuits Conference (CICC), 2014 IEEE Proceedings of the*, 2014, pp. 33–36.

[18] P. Ramachandran, P. Kudva, J. Kellington, J. Schumann, and P. Sanda, "Statistical fault injection," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008, pp. 122–127.

[19] B. Reagen *et al.*, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 267–278.

[20] S. K. Sastry Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Application resiliency analyzer for transient faults," *IEEE Micro*, vol. 33, no. 3, pp. 58–66, May 2013.

[21] N. Seifert *et al.*, "Soft error susceptibilities of 22nm tri-gate devices," *IEEE Trans. Nucl. Sci.*, vol. 59, no. 6, pp. 2666–2673, 2012.

[22] ——, "Soft error rate improvements in 14-nm technology featuring second-generation 3d tri-gate transistors," *IEEE Trans. Nucl. Sci.*, vol. 62, no. 6, pp. 2570–2577, 2015.

[23] V. Sze *et al.*, "Efficient processing of deep neural networks: A tutorial and survey," *CoRR*, vol. abs/1703.09039, 2017. [Online]. Available: http://arxiv.org/abs/1703.09039

[24] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, June 2012, pp. 356–367.

[25] Tesla, "How does tesla's new self-driving ai chip match up to competitors?" https://www.electronicspoint.com/opinion/how-does-teslas-new-self-driving-ai-chip-match-up-to-competitors/, 2019.

[26] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *International Conference on Dependable Systems and Networks, 2004*, 2004, pp. 61–70.

[27] P. N. Whatmough *et al.*, "14.3 a 28nm soc with a 1.2ghz 568nj/prediction sparse deep-neural-network engine with $>$0.1 timing error rate tolerance for iot applications," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, Feb 2017, pp. 242–243.

[28] WikiChip, "Full self-driving chip - tesla," https://en.wikichip.org/wiki/tesla_(car_company)/fsd_chip, 2019.