API-based Hardware Fault Simulation for DNN-Accelerators

Patrik Omland, Yang Peng, Michael Paulitsch (*Senior Member, IEEE*), Jorge Parra, Gustavo Espinosa (*Member, IEEE*) Abishai Daniel, Gereon Hinz and Alois Knoll (*Senior Member, IEEE*)

***Abstract*—Computer hardware experiences random faults in some cases causing program failure. The rate of faults can accumulate to notable effect in computer clusters and warrants consideration for high dependability applications.**

Deep Neural Networks (DNNs) are relatively robust to hardware faults. When designing DNN-accelerators, performance may be gained by optimizing the hardware protection overhead, subject to dependability requirements.

We propose a novel method for hardware fault simulation to investigate the end-to-end, hardware fault to DNN output failure probability for common DNN-accelerators. In a proof of concept, speed ups of the order of 10^5 compared to full hardware simulations have been achieved.

***Index Terms*—deep neural networks, hardware faults, hardware reliability, fault model, fault tolerance, fault injection, program vulnerability**

I. Introduction

CONTINUED transistor scaling results in lower operating voltages that enable increased levels of integration within a given silicon area footprint. However, it also entails an increase in the likelihood of unintended bit flips and data corruption at the device level.

The rate of these faults per computational resource requires special consideration when

1) combining many computational resources (e.g. super computers, server farms).
2) executing applications with high dependability requirements, such as in automotive (requiring failure rates below 10^{-8} failures/hour for safety critical functions).

To reduce the likelihood of data corruption, hardware designers identify high-risk components and add protection circuitry, such as parity checks and error correction codes (ECC). However, protection circuitry requires die area and increases power consumption which could otherwise be used to increase performance. The more comprehensive the protection, the higher the error detection or correction capabilities, but the more area it occupies.

A balance must be found in the trade-off between an integrated circuit's dependability and performance. Experiments indicate that Deep Neural Networks (DNNs) are more resilient to hardware faults than other programs.[1] In this context, special "DNN-accelerators" have been designed for efficient DNN execution [3]: These may require lower than usual levels of hardware protection, while satisfying the same dependability targets for DNN applications.

So what is the probability of output failure due to hardware faults for DNNs running on these DNN-accelerators? In this work we present a novel method for estimating this probability. Our approach works by expanding the primitives of application program interfaces (APIs) used by DNNs with hardware-specific fault simulations: First, the original primitive is run, then the output is modified in the way it would be corrupted due to faults in the target hardware. The actual hardware is not required. By executing a DNN with this modified API simulating hardware faults, statistics may be generated on output failures. Unlike existing approaches our approach uniquely combines

- Accuracy: The actual workload is run on an accurate hardware fault simulation
- Speed: The simulation time is not constrained by the lack of, nor the speed of hardware to be simulated
- Scale: By sharing the modified API implementation, accurate dependability estimates for specific workloads may be generated without hardware / algorithmic knowledge.

In the following, we first define what we mean by "DNN-accelerator" (II). We then argue that established methods to estimate the probability of DNN output failure due to hardware faults have their limitations (III) and present the proposed method (IV, V).

II. DNN-Accelerators

Computing platforms tailored specifically to the needs of DNNs have become more common over the past years. Prominent examples are Google Tensor Processing Units, Nvidia Tensor Cores, Intel Xeon Tile Matrix Multiply units and Intel Xe HPC GPUs [3].

By far, most computer operations carried out by DNNs are spent on matrix multiplication: In DNN-terminology, the fully-connected and convolution layers are calculated by algorithms using matrix multiplication.[2] For DNNs such as ResNet-50, these multiplications involve large matrices with dimensions, n, in the thousands. Matrix multiplication is, approximately, an $\mathcal{O}(n^3)$-operation. All other commonly used DNN-operations

P. Omland, Y. Peng, M. Paulitsch, J. Parra, G. Espinosa and A. Daniel are with the Intel Corporation.

P. Omland, G. Hinz and Alois Knoll are with the Department of Informatics, Technical University Munich, Garching, Munich, Germany.

Direct questions and comments about this article to Patrik Omland, Dependability Research Lab, Intel Deutschland GmbH, Lilienthalstraße 15, 85579 Neubiberg; patrik.omland@intel.com.

Manuscript received 2021-05-14; revised 2022-06-03.

[1]Compare bit error rate thresholds found in [1] with requirements in [2].
[2][4] Explains how to convert convolution to matrix multiplication.

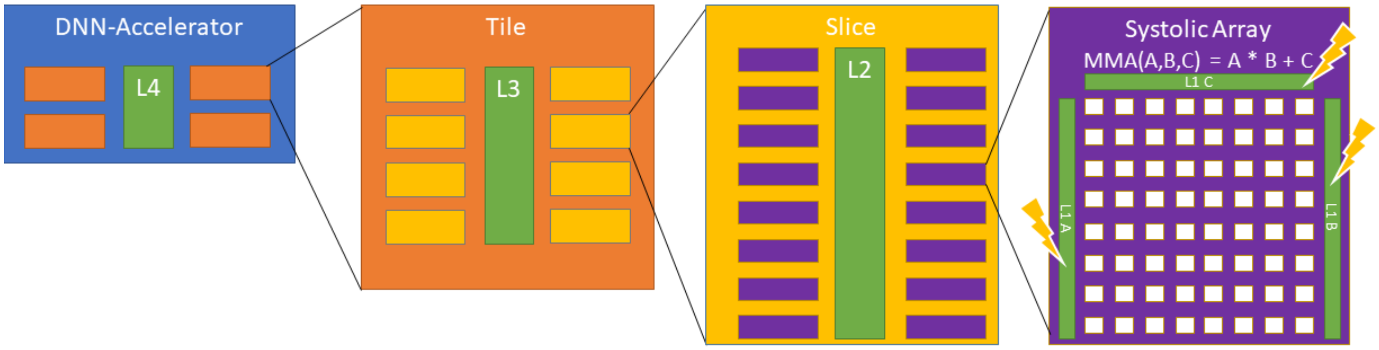

Fig. 1. Typical architecture of a DNN Accelerator. Memory hierarchy depth (L4-L1) and number of units on each level (four, eight, 16) chosen arbitrarily.

are $\mathcal{O}(n)$-operations. Consequently, accelerators geared towards DNNs specifically aim to accelerate large matrix multiplications. Most of them

1) adopt an architecture which consists of Systolic Arrays operating in parallel, and
2) feature a memory hierarchy designed to maximize the reuse of data cached close to the Systolic Arrays,

where each Systolic Array computes small matrix-multiply-accumulate (MMA) operations, $D = A \cdot B + C$.

We will refer to this class of accelerators as "DNN-accelerators". The typical architecture of a DNN-accelerator is shown in fig. 1. The white blocks inside the Systolic Array represent Multiply-Accumulate-Fused (MAF) units, performing the actual calculations.

When designing DNN-accelerators, the relative robustness of DNNs w.r.t. hardware faults is taken advantage of by optimizing the level of hardware protection for performance gains. In this context, Systolic Arrays and their caches present particularly good opportunities for such gains.

Protection circuitry for large caches (L4-L2 in fig. 1) requires relatively little die area. In comparison, the Systolic Array caches (L1 in fig. 1) are very small and there may be thousands of them - here, protection carries a high performance cost. Analogously, while an ALU on the "slice level" in fig. 1 may be implemented with hardware protection, doing the same for each of the dozens of MAF-units comprising a single Systolic Array places a large burden on performance.

III. Related work

Many methods of estimating the likelihood of program failure due to hardware faults exist. Below, we present the most prominent ones.

A. Statistical Fault Injection

In statistical fault injection, faults are injected at program runtime. These faults may be injected at different system abstraction levels (gate, microarchitecture, etc.). In general, lower level fault injection provides more accurate results but may not be scalable in practice due to long execution times, while higher level fault injection may run much faster, but at the price of lower accuracy [5].

Hierarchical simulations have been applied to address this trade-off by simulating different parts of the system at different abstraction levels, so that required details are modeled only for the parts of interest [5]. The proposed method in this work follows a similar concept as hierarchical simulations.

B. Vulnerability Factors

In the vulnerability factor approach, simulating lower system abstraction levels individually for each program is avoided by estimating the fraction of faults affecting a given level from the next lower level. Frequently used factors are the Hardware Vulnerability Factor (HVF, [6]), the Program Vulnerability Factor (PVF, [7]) and the Timing Vulnerability Factor (TVF, [8]). The overall failure rate for a program, P, is then estimated by (1), where F denotes the fraction of time in a particular use condition, uc, itself dependent on the clock frequency, f_{clk}.

$$\text{Failure Rate}(P) \approx \sum_{uc} F_{uc,P}\left(f_{\text{clk}}\right) \cdot \sum_{c \in \text{circuits}} \text{Fault-Rate}_c \cdot \text{TVF}_{uc,c} \cdot HVF_{uc,c} \cdot PVF_{uc,c,P} \quad (1)$$

However, not much is gained if $PVF_{uc,c,P}$ has to be estimated individually for each DNN, each use condition and each circuit.[3] As will be shown in V, hierarchical fault injection simulations not only deliver more accuracy, but may be implemented in a general, scalable fashion.

C. Evaluating vulnerability of DNN-based applications

To understand the vulnerability of DNN-based applications, many existing work (e.g. [9][1]) adopt application level fault injection by, say, injecting faults directly into the DNN model (e.g. weights). However, this approach does not reflect the actual impact of the underlying platform on which the DNN is executed. As will be shown in V, microarchitectural details of DNN accelerator designs have profound impact on how hardware faults propagate to the level of the DNN model.

[3]For many CPU applications, the approximation $PVF_{uc,c,P} \approx 1$ may be used, making this approach useful for rough estimates. However, in this work, we are particularly interested in the $PVF_{uc,c,P} << 1$ property of DNNs.

IV. Problem Statement

The task at hand is a risk assessment for DNNs when facing hardware faults on DNN-accelerators. Generally, given a program, p, the *risk* of a hardware fault, f, causing failure with $severity \in \{0 = none,\ 1,\ \dots\}$, may be defined as

$$risk = \Pr\big(f,\ severity \mid p\big) \cdot severity \tag{2}$$

commonly known as the risk matrix approach, where shorthand $\Pr$ denotes probability.

The probability on the right-hand side of (2) may be separated into two parts

$$\Pr\big(f,\ severity \mid p\big) = \underbrace{\Pr\big(f \mid p\big)}_{\text{exposure}} \cdot \underbrace{\Pr\big(severity \mid f,\ p\big)}_{\text{conditional failure}} \tag{3}$$

The "exposure probability" measures the likelihood of the fault, f, occurring while a given program, p, is exposed to it. For instance, if a program makes no use of floats, and the hardware fault considered is a fault in an FPU, the program's exposure probability to that fault equals zero.

The "conditional failure probability" measures the likelihood of the program, p, failing with $severity$, conditional on it being exposed to a fault, f. For instance, if the program's output is a single-precision floating point value and the fault only ever flips the least significant bit of that value, the relative output error equals 2^{-23}: For most programs, this error won't be considered program failure, so the associated conditional failure probability would equal zero.

As calculating the *risk* using (2) becomes trivial once the failure probability, (3), has been estimated, moving forward, we only consider the latter problem.

V. Novel API-based fault simulation

Numerical programs, in particular DNNs, rely on standards-based application program interfaces (APIs) to implement mathematical operations such as matrix multiplication. The actual operation is usually implemented by the hardware manufacturer, requiring intimate knowledge of the accelerator's memory hierarchy, instruction pipelining, etc. In the proposed approach, hardware fault simulations are implemented into these APIs for the very same reason. Also, fault simulations thus implemented become available immediately to every program linking the given API.

The proposed API-based fault simulation for a given API comprises the following steps: For each API-operation executed on the accelerator

1) *MoC*: Develop a model of computation (MoC), modeling how the operation is executed on the actual hardware
2) *Fault MoC-Scope*: For the hardware fault under consideration, find the execution steps affected in the MoC by one such fault.
3) *API Fault Simulation*: Develop a fault simulation for these execution steps making as much use of the API-operation's (efficiently computed) output as possible and include the simulation with the API-operation.

Without loss of generality, we provide a sample application with a simplified model of computation, following the steps outlined above, to illustrate the proposed method.

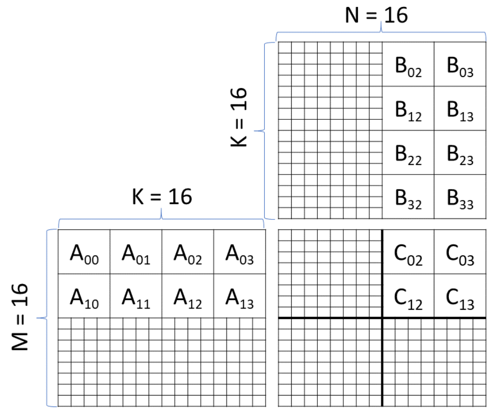

Fig. 2. Matrix multiplication on DNN-Accelerators

A. Simplified Model of Computation

We implement the fictitious general matrix multiply API-function $\mathrm{GEMM16}(A, B) = A^{16\times16} \cdot B^{16\times16}$, on a DNN-accelerator featuring four Systolic Arrays. Each Systolic Array itself may execute a matrix multiply accumulate (MMA) instruction, $A^{4\times4} \cdot B^{4\times4} + C^{4\times4}$. The generalization to arbitrary dimensions and number of Systolic Arrays is straight forward.

The multiplication is depicted in fig. 2. The 16 submatrices, $C_{mn}^{4\times4}$, may be calculated by

$$C_{mn}^{4\times4} = \sum_{k=0}^{k<4} A_{mk}^{4\times4} B_{kn}^{4\times4} \tag{4}$$

The GEMM16-algorithm using MMA-instructions is given by alg. 1. It divides C into quadrants, each assigned one Systolic Array (cp. fig.2).

Algorithm 1 GEMM16: Returns $A^{16\times16} \cdot B^{16\times16}$ using four Systolic Arrays (SAs) capable of $A^{4\times4} \cdot B^{4\times4} + C^{4\times4}$-MMA

```
Input: Matrices $A^{16\times16}$, $B^{16\times16}$
Output: $C = A \cdot B$
1: $C = 0$
2: for $SA = 0$ to $3$ do
3:     $m_0 = \lfloor SA/2 \rfloor * 2$
4:     $n_0 = \lfloor SA \mod 2 \rfloor * 2$
5:     for $k = 0$ to $3$ do
6:         for $m = m_0$ to $m_0 + 1$ do
7:             for $n = n_0$ to $n_0 + 1$ do
8:                 $C_{mn} = \mathrm{MMA}_{SA}\left(A_{mk}, B_{kn}, C_{mn}\right)$
9: return $C$
```

Unrolling the m, n loops for the upper right quadrant we get

```
1: for $k = 0$ to $3$ do
2:     $C_{02} = \mathrm{MMA}_1\left(A_{0k}, B_{k2}, C_{02}\right)$
3:     $C_{03} = \mathrm{MMA}_1\left(A_{0k}, B_{k3}, C_{03}\right)$
4:     $C_{12} = \mathrm{MMA}_1\left(A_{1k}, B_{k2}, C_{12}\right)$
5:     $C_{13} = \mathrm{MMA}_1\left(A_{1k}, B_{k3}, C_{13}\right)$
```

Notice, that each k-iteration requires only four different A, B inputs, namely A_{0k}, A_{1k}, B_{k2}, B_{k3}. Now, consider the memory hierarchy in fig. 1: For L1A (L1B) large enough to cache one (two) 4×4-submatrices, data requests to L2 for these inputs is halved.[4] Moving forward we assume just that.

B. Simulating Transient L1 Cache faults

Suppose one of the L1 caches of the upper right quadrant's Systolic Array experiences a transient bit-flip - what is the fault's MoC-scope? From the unrolled loop above, we see that any such fault is confined to one k-iteration (data is not reused across k-iterations) and affects at most two C_{mn} (e.g. if B_{12} is corrupted in line 2, it affects C_{02} and then C_{12} in line 4).

Next, we develop the API fault simulation. In the unrolled loop above, suppose the fault occurs at iteration $k = 2$: line 4 and has the effect $B_{23} \overset{\not{}}{\mapsto} \tilde{B}_{23}$. The corresponding effect on the output, $C \overset{\not{}}{\mapsto} \tilde{C}$, reads

$$\tilde{C}_{12} = C_{12} - \mathrm{MMA}_1(A_{12}, B_{23}, 0) + \mathrm{MMA}_1(A_{12}, \tilde{B}_{23}, 0)$$

As C_{12} is returned by the regular API-operation, we do not need to calculate it ourselves but can utilize the high-performance API-implementation as input to the simulation.

More generally, the effect of a cache fault occurring in Systolic Array SA, at iteration $(k, m, n) \in [0, 3] \times [0, 1] \times [0, 1]$ and cache index $X \in \{A, B_0, B_1, C\}$, may be modeled by alg. 2.[5]

Algorithm 2 GEMM16_FI_C: Simulate L1A/B/C Cache fault during GEMM16 execution

```
Input: $A$, $B$, $C = \mathrm{GEMM16}(A, B)$, $SA$, $(k, m, n)$, $X$
Output: $\tilde{C}$, fault-simulated output of GEMM16
 1: $m = m + \lfloor SA/2 \rfloor * 2$
 2: $n = n + \lfloor SA \mod 2 \rfloor * 2$
 3: if $(X == A)$ then
 4:     if $(n == 0)$ then
 5:         $C_{m0} = \mathrm{MMA}(-A_{mk}, B_{k0}, C_{m0})$
 6:         $C_{m0} = \mathrm{MMA}(\tilde{A}_{mk}, B_{k0}, C_{m0})$
 7:     $C_{m1} = \mathrm{MMA}(-A_{mk}, B_{k1}, C_{m1})$
 8:     $C_{m1} = \mathrm{MMA}(\tilde{A}_{mk}, B_{k1}, C_{m1})$
 9: else if $(X == B_0)$ then
10:     ...
11: ...
12: return $C$
```

Note that in alg.2 timing matters: If a fault in L1A happens at $n = 0$, then two of C's 4×4-submatrices are affected, otherwise only one. Similarly, if L1B suffers a fault corrupting B_0 at $n = m = 1$, C will not be affected.

The API hardware fault simulation is listed in alg. 3. To inject one random fault into a program making multiple use of GEMM16, we count the overall MMA-instruction calls, MMA_total, of that program, and pick a positive random number, $MMA\_FI \leq MMA\_total$, representing one of these calls.

[4] For real-world DNN-accelerators with $M \times K \times N$-MMA: If L1A caches a single $M \times K$-submatrix and L1B caches L_B $K \times N$-submatrices, each Systolic Array may be assigned $L_B \times L_B$ $M \times N$-submatrices in the output to reduce L2-requests for A, B by a factor of $1/L_B$ using alg. 1.

[5] By not using the actual C_{mn}-input to MMA for the given (k, m, n), alg.2 does not account for "$(a + b) + c \neq a + (b + c)$". To account for that, the k-loop needs to be executed as in alg. 4.

Algorithm 3 GEMM16_FSIM: Simulate fault in GEMM16 execution

```
Input: $A$, $B$. Global: $MMA\_Calls$, $MMA\_FI$
Output: $\tilde{C}$, fault-simulated output of GEMM16
 1: $C = \mathrm{GEMM16}(A, B)$
 2: if $MMA\_FI \in [MMA\_Calls, MMA\_Calls + 64)$ then
 3:     Choose random $(SA, k, m, n, X)$
 4:     $C = \mathrm{GEMM16\_FI\_C}(A, B, C, SA, k, m, n, X)$
 5: $MMA\_Calls = MMA\_Calls + 64$
 6: return $C$
```

By far most of the work in alg.3 is performed through the API-call to GEMM16: This will be executed with maximal performance on any hardware with a GEMM16-implementation. In comparison, the up to two MMA-calls from GEMM16_FI are insignificant - in particular for real world large GEMM operations with thousands of MMA calls.

Coming back to the original problem of estimating (3): We may approximate $\Pr(severity \mid f, p)$ by the relative failure frequency of program runs with hardware fault simulation. To estimate $\Pr(f \mid p)$, the likelihood of encountering a transient fault, random in time and space, does not depend on the level of parallelization: Whether four Systolic Arrays are used, or a single one four-times as long, does not matter. Accordingly, given the fault rate, R_f, of one L1 cache and the duration, τ_{MMA}, of one MMA-execution, we may estimate

$$\Pr(f \mid p) \approx 1 - \exp\left(-MMA\_total \cdot \tau_{\mathrm{MMA}} \cdot R_f\right) \qquad (5)$$

where the exponential failure distribution was used to model the probability of fault given fault rate and duration.

C. Simulating Transient faults inside Systolic Arrays

The same method applied for simulating transient faults in the Systolic Array's caches (V-B), may be used for the simulation of arbitrary faults inside the Systolic Arrays $\mathrm{MMA} \overset{\not{}}{\mapsto} \widetilde{\mathrm{MMA}}$. For the Systolic Array's digital arithmetic, however, simulating the correct C_{mn}-input to the MMA-instruction matters[6] and thus needs to be calculated by simulating the k-loop - see alg. 4. For real world applications with large k-loops, the additional simulation overhead is notable.

Algorithm 4 GEMM16_FI_L: Simulate fault inside Systolic Array Logic during GEMM16 execution.

```
Input: $A$, $B$, $C = \mathrm{GEMM16}(A, B)$, $SA$, $(k_{FI}, m, n)$
Output: $\tilde{C}$, fault-simulated output of GEMM16
 1: $m = m + \lfloor SA/2 \rfloor * 2$
 2: $n = n + \lfloor SA \mod 2 \rfloor * 2$
 3: $C_{mn} = 0$
 4: for $k = 0$ to $3$ do
 5:     if $k \neq k_{FI}$ then
 6:         $C_{mn} = \mathrm{MMA}_{SA}(A_{mk}, B_{kn}, C_{mn})$
 7:     else
 8:         $C_{mn} = \widetilde{\mathrm{MMA}}_{SA}(A_{mk}, B_{kn}, C_{mn})$
 9: return $C$
```

[6] The Systolic Array may, for instance, perform optimizations if $C_{mn} = 0$.

TABLE I
TRANSIENT BUFFER FAULT SIMULATION FOR DNN-ACCELERATORS RUNNING RESNET-50 INFERENCE.

$M \times K \times N$	L_B	#MMA	ΔTop [%]	Overh.[%]
$32 \times 32 \times 32$	4	117,216	-1.12	11.0
	2		-1.05	3.3
$16 \times 16 \times 16$	4	994,368	-1.07	1.2
	2		-0.92	0.5
$8 \times 8 \times 8$	4	7,923,456	-0.96	0.1
	2		-0.76	0.0

D. Simulating Permanent faults

A permanent fault in a Systolic Array or its caches affects every k, m and n and thus alg. 4 needs to be modified accordingly. The challenge in simulating permanent faults lies in modeling the likelihood of encountering the faulty Systolic Array.

Suppose we execute a program with ten $\mathrm{GEMM16}$ invocations on a DNN-accelerator with 16 Systolic Arrays. As $\mathrm{GEMM16}$ requires four Systolic Arrays, each time $\mathrm{GEMM16}$ is invoked, there are $16!/(16-4)! = 43680$ ways of assigning four output quadrants to 16 Systolic Arrays. For the whole program we get $43680^{10} \approx 10^{46}$ possibilities.

One approach to handle this problem is to model worst- and best-case scenarios. For instance, considering fig.1, in a worst-case the program's $\mathrm{GEMM16}$s might always be mapped to the same slice and randomly distributed among its 16 Systolic Arrays, one of which has a permanent fault. In a best-case each Systolic Array might be chosen at random from the 512 Systolic Arrays comprising fig.1's DNN-accelerator.

VI. RESNET-50 PROOF OF CONCEPT

We applied the method presented in V to ResNet-50 [10] inference on several DNN-accelerator configurations by modifying the oneDNN API. Two of the oneDNN operations used by ResNet-50 utilize Systolic Arrays: Matrix multiplication and convolution.[2] We analyzed the algorithms implemented by oneDNN for DNN-accelerators and developed fault models according to the method described above. The buffers (FP16 data format) were corrupted by a single random transient bit-flip each inference, analogously to alg. 2. 24.8k ImageNet [11] inferences were executed for each configuration. The results are shown in table I.

The simulation was run on an Intel i9-7960X CPU. A single inference without fault injection took 104ms. The overhead in table I is given w.r.t. to this duration. "$M \times K \times N$" specifies the MMA dimensions and "L_B" the number of $K \times N$ matrices cached in L1B.[4] "ΔTop" lists the change in percentage of inputs for which the highest rated output label is correct with vs. without fault simulation. "#MMA" lists the number of MMA calls for a single inference.

As expected from section V, the conditional failure probability (3), which may be identified with "ΔTop", decreased with decreasing MMA dimensions: The corrupted buffer element affects fewer output elements. The effect on the exposure probability (3) is more complicated: While smaller buffers result in a smaller frequency of buffer corruption, accounting for the time the application is exposed to these buffers is not straight forward: While the number of required MMA calls obviously increases with decreasing MMA dimensions, estimating the duration of each such call for different dimensions requires knowledge of the Systolic Array's implementation. Consequently, one should not draw conclusions on the risk (2) associated with different Systolic Array configurations from table I without accounting for these factors.

In conclusion, we successfully applied our novel methodology to a large workload, performing hundreds of thousand hardware fault simulations within hours on a regular CPU, where more traditional approaches would have taken days for a single simulation.

VII. FUTURE WORK

The best-/worst-case approach for modeling permanent faults (V-D) does not yield the single probability for program failure we are after (3). Rather, it delivers upper/lower bounds on that probability. Moving forward we are developing models of computation incorporating scheduling algorithms for DNN-accelerators to accurately estimate this probability.

In section V-C, we suggest running a hardware simulation, $\widetilde{\mathrm{MMA}}$, for the complete MMA-instruction. When modeling permanent faults inside the Systolic Arrays, this simulation overhead becomes significant. In future work, we will develop methods reducing the simulation overhead to simulating single MAF-units (II) only.

Finally, while our research has focused on utilizing DNN-accelerators for the class of DNN programs, other classes of matrix multiplication heavy programs would profit from using DNN-accelerators (e.g. finite element methods). In upcoming work we will investigate the effect of hardware protection design choices on the dependability of these kinds of programs.

ACKNOWLEDGMENT

We would like to thank Yue Qi, Fangwen Fu and Mourad Gouicem for technical insights on DNN-accelerators and algorithms utilizing their architecture.

REFERENCES

[1] B. Reagen et al., "Ares: A framework for quantifying the resilience of deep neural networks," in *55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018.
[2] H. T. Nguyen and et al., "Chip-level soft error estimation method," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, 2005.
[3] A. Rodriguez, *Deep Learning Systems: Algorithms, Compilers, and Processors for Large-Scale Production*. Morgan & Claypool Publishers, 2020.
[4] S. Chetlur et al., "cuDNN: Efficient Primitives for Deep Learning," *arXiv:1410.0759v3*, Oct. 2014.
[5] Z. Kalbarczyk et al., "Hierarchical Simulation Approach to Accurate Fault Modeling for System Dependability Evaluation," *IEEE Transactions on Software Engineering*, 1999.
[6] V. Sridharan and D. R. Kaeli, "Using hardware vulnerability factors to enhance AVF analysis," *Proc. Int. Symp.Comput. Arch. (ISCA)*, 2010.
[7] ——, "Eliminating microarchitectural dependency from Architectural Vulnerability," *International Symposium on High Performance Computer Architecture (HPCA-15)*, 2009.
[8] N. Seifert and N. Tam, "Timing vulnerability factors of sequentials," *IEEE Transactions on Device and Materials Reliability*, 2004.

[9] G. Li et al., “Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications,” *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (ACM)*, 2017.
[10] K. He et al., “Deep Residual Learning for Image Recognition,” arXiv:1512.03385v1.
[11] J. Deng et al., “ImageNet: A Large-Scale Hierarchical Image Database,” in *CVPR09*, 2009.

Patrik Omland is a Research Scientist at Intel Corporation and is pursuing a PhD with the Department of Informatics, Technical University Munich. His research interests include the effect of hardware faults on program execution and digital arithmetic / numerical algorithms. He holds a Masters degree in Mathematical Physics from the Ludwig-Maximilians-University Munich.

Yang Peng is a research scientist and system architect at Intel Corporation. His research interests include system architecture for dependable AI/ML-based systems. He holds a Ph.D. in Electrical Engineering from the Technical University of Munich.

Michael Paulitsch (Senior Member, IEEE) is a Principal Engineer at Intel Corporation, where he leads the Dependability Research Labs. His research interests include novel architectures for dependable systems and machine learning. He has (co)authored 50+ technical papers and holds a Ph.D from the Technical University Vienna and a Ph.D from the Vienna University of Economics and Business.

Jorge Parra is a computer architect currently working on Intel’s Xe GPU products. He holds a Ph.D. and a M.Sc. from the University of New Mexico in Electrical Engineering and a B.Sc. in Electronics Engineering from Pontificia Universidad Javeriana. His interests are in Computer Architecture, Machine Learning Hardware Architectures, and Artificial Intelligence.

Gustavo Espinosa (Member, IEEE) is Senior Principal Engineer at Intel Corporation, where he leads reliability and security architecture development for discrete GPU products. He previously led the architecture development of a number of Intel’s processors. He holds a Masters degree in Computer Engineering from Boston University and a Bachelors degree in Electrical Engineering from Cornell University.

Abishai Daniel is a staff RAS quality and reliability engineer at Intel Corporation with a focus on statistical predictive model development and application of novel machine learning techniques to reliability modeling. He has served as program committee member/session chair for various IEEE conferences, published more than 15 papers and holds 2 patents. Abishai has an MSEE and a PhD from the University of Michigan and an AB from Wabash College.

Gereon Hinz is CEO of STTech, providing solutions to current and upcoming technological challenges in the autonomous systems domain. For the Technical University Munich, he organizes a yearly Autonomous Driving lecture series. He holds a Masters degree in cybernetics from the University of Stuttgart.

Alois Knoll (Senior Member, IEEE) is Professor of Computer Science with the Department of Informatics of the Technical University Munich. His research interests include robotics, artificial intelligence and real-time systems. He has (co)authored 600+ technical papers and guest-edited international journals. He initiated the First IEEE/RAS Conference on Humanoid Robots and was General and Program Chair of various IEEE conferences. He is Specialty Chief Editor of the *Frontiers in Neurorobotics*.