# API-Based Hardware Fault Simulation for DNN Accelerators

**Patrik Omland, Yang Peng, and Michael Paulitsch**
Dependability Research Laboratory
Intel Deutschland GmbH
85579 Neubiberg, Germany

**Jorge Parra, Gustavo Espinosa, and Abishai Daniel**
Intel Corporation, Santa Clara, CA 95054 USA

**Gereon Hinz**
STTech
82031 Grünwald, Germany

**Alois Knoll**
Department of Informatics
Technical University Munich
85748 Munich, Germany

*Editor's notes:*
This article presents an application program interface (API)-based hardware fault simulation method to investigate the effect of hardware faults on the failure probability of deep neural network (DNN) accelerators.
*—Fei Su, Intel Corporation*

**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

- combining many computational resources (e.g., supercomputers and server farms) and
- 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 (ECCs). 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 tradeoff 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.

*Digital Object Identifier 10.1109/MDAT.2022.3180977*
*Date of publication: 8 June 2022 ; date of current version: 10 March 2023.*

[1]Compare bit error rate thresholds found in [1] with requirements in [2].

- *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.

## 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 graphics processing units (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 $O(n^3)$ operation. All other commonly used DNN operations are $O(n)$ operations. Consequently, accelerators geared toward DNNs specifically aim to accelerate large matrix multiplications. Most of them

- adopt an architecture that consists of systolic arrays (SAs) operating in parallel and
- feature a memory hierarchy designed to maximize the reuse of data cached close to the SAs,

where each SA 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 Figure 1. The white blocks inside the SA represent multiply-accumulate-fused (MAF) units, performing the actual calculations.

When designing DNN accelerators, the relative robustness of DNNs with reference to hardware faults is taken advantage of by optimizing the level of hardware protection for performance gains. In this context, SAs and their caches present particularly good opportunities for such gains.

Protection circuitry for large caches (L4–L2 in Figure 1) requires relatively little die area. In comparison, the SA caches (L1 in Figure 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 Figure 1 may be implemented with hardware protection, doing the same for each of the dozens of MAF units comprising a single SA places a large burden on performance.

## Related work

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

### 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, and so on). 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].

[2]Chetlur *et al.* [4] explain how to convert convolution to matrix multiplication.

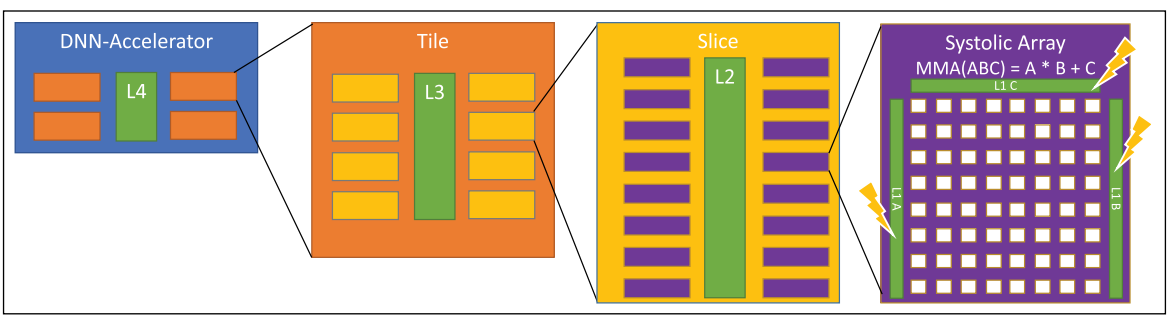


**Figure 1. Typical architecture of a DNN accelerator. Memory hierarchy depth (L4–L1) and the number of units on each level (4, 8, 16) chosen arbitrarily.**

Hierarchical simulations have been applied to address this tradeoff 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.

### 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, $f_{\text{clk}}$

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

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

### Evaluating vulnerability of DNN-based applications

To understand the vulnerability of DNN-based applications, many existing works (e.g., [1] and [9]) 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 the upcoming section, microarchitectural details of DNN accelerator designs have a profound impact on how hardware faults propagate to the level of the DNN model.

## 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 $\in \{0 = \text{none}, 1, \ldots\}$, may be defined as

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

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

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

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 floating-point unit (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 will not 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.

## Novel API-based fault simulation

Numerical programs, in particular, DNNs, rely on standards-based 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, and so on. 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.

[3]For many central processing unit (CPU) applications, the approximation $\text{PVF}_{uc,c,p} \approx 1$ may be used, making this approach useful for rough estimates. However, in this work, we are particularly interested in the $\text{PVF}_{uc,c,p} \ll 1$ property of DNNs.

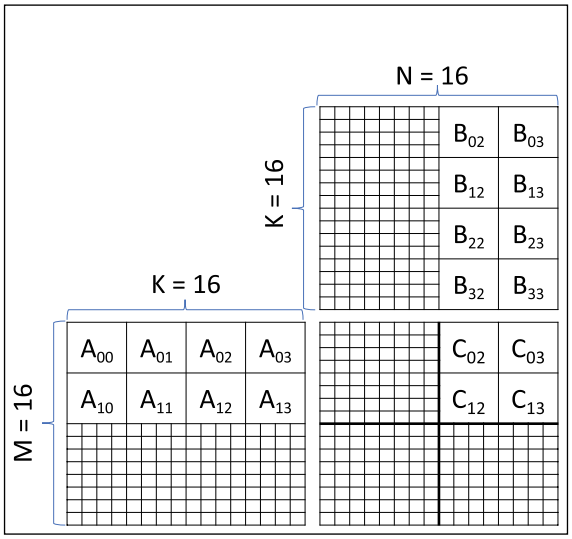


**Figure 2. Matrix multiplication on DNN accelerators.**

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

1) *Model of computation (MoC):* Develop an 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 including the simulation with the API operation.

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

## Simplified model of computation

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

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

$$C_{mn}^{4\times4} = \sum_{k=0}^{k<4} A_{mk}^{4\times4} B_{kn}^{4\times4}. \quad (4)$$

**Algorithm 1.** GEMM16: returns $A^{16\times16} \cdot B^{16\times16}$ using four 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: $C_{mn} = \mathrm{MMA}_{SA}(A_{mk}, B_{kn}, C_{mn})$
9: **return** $C$

The GEMM16 algorithm using MMA instructions is given by Algorithm 1. It divides $C$ into quadrants, each assigned one SA (see Figure 2).

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

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

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

## Simulating transient L1 cache faults

Suppose one of the L1 caches of the upper right quadrant's SA 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 $B_{23} \overset{\lightning}{\mapsto} \tilde{B}_{23}$. The corresponding effect on the output, $C \overset{\lightning}{\mapsto} \tilde{C}$, 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

[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 SA 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 Algorithm 1.

**Algorithm 2.** GEMM16_FI_C: Simulate L1A/B/C cache fault during GEMM16 execution.

**Input:** $A$, $B$, $C = \text{GEMM16}(A, B)$, $SA$, $(k, m, n)$, $X$
**Output:** $\tilde{C}$, fault-simulated output of GEMM16

```
m = m + ⌊SA/2⌋ * 2
n = n + ⌊SA mod 2⌋ * 2
if (X == A) then
    if (n == 0) then
        C_m0 = MMA(−A_mk, B_k0, C_m0)
        C_m0 = MMA(Ã_mk, B_k0, C_m0)
    C_m1 = MMA(−A_mk, B_k1, C_m1)
    C_m1 = MMA(Ã_mk, B_k1, C_m1)
else if (X == B_0) then
    ...
...
return C
```

the high-performance API implementation as input to the simulation.

More generally, the effect of a cache fault occurring in 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 Algorithm 2.[5]

Note that in Algorithm 2, timing matters: If a fault in L1A happens at $n = 0$, then two of $C$'s $4 \times 4$-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 Algorithm 3. To inject one random fault into a program making multiple uses 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.

By far, most of the work in Algorithm 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 | $f, p$) by the relative failure frequency of program runs with hardware fault simulation. To estimate Pr($f$ | $p$), the likelihood of encountering a transient fault, random in time and space, does not depend on the level of parallelization: Whether four SAs are used, or a single one four-times as long, does not matter. Accordingly,

**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

```
C = GEMM16(A, B)
if MMA_FI ∈ [MMA_Calls, MMA_Calls + 64) then
    Choose random (SA, k, m, n, X)
    C = GEMM16_FI_C(A, B, C, SA, k, m, n, X)
MMA_Calls = MMA_Calls + 64
return C
```

**Algorithm 4.** GEMM16_FI_L: simulate fault inside SA logic during GEMM16 execution.

**Input:** $A$, $B$, $C = \text{GEMM16}(A, B)$, $SA$, $(k_{FI}, m, n)$
**Output:** $\tilde{C}$, fault-simulated output of GEMM16

```
m = m + ⌊SA/2⌋ * 2
n = n + ⌊SA mod 2⌋ * 2
C_mn = 0
for k = 0 to 3 do
    if k ≠ k_FI then
        C_mn = MMA_SA(A_mk, B_kn, C_mn)
    else
        C_mn = MMA~_SA(A_mk, B_kn, C_mn)
return C
```

given the fault rate, $R_f$, of one L1 cache and the duration, $\tau_{\text{MMA}}$, of one MMA execution, we may estimate

$$\Pr(f \mid p) \approx 1 - \exp(-\text{MMA\_total} \cdot \tau_{\text{MMA}} \cdot R_f) \quad (5)$$

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

### Simulating transient faults inside SAs

The same method applied for simulating transient faults in the SA's caches (previous section) may be used for the simulation of arbitrary faults inside the SAs $\text{MMA} \overset{f}{\mapsto} \widetilde{\text{MMA}}$. For the SA'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 Algorithm 4). For real-world applications with large $k$-loops, the additional simulation overhead is notable.

### Simulating permanent faults

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

[5]By not using the actual $C_{mn}$-input to MMA for the given $(k, m, n)$, Algorithm 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 Algorithm 4.

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

Suppose we execute a program with ten GEMM16 invocations on a DNN accelerator with 16 SAs. As GEMM16 requires four SAs, each time GEMM16 is invoked, so there are 16!/(16–4)! = 43,680 ways of assigning four output quadrants to 16 SAs. For the whole program, we get $43{,}680^{10} \approx 10^{46}$ possibilities.

One approach to handle this problem is to model worst- and best-case scenarios. For instance, considering Figure 1, in the worst case, the program's GEMM16s might always be mapped to the same slice and randomly distributed among its 16 SAs, one of which has a permanent fault. In the best case, each SA might be chosen at random from the 512 SAs comprising Figure l's DNN accelerator.

## ResNet-50 proof of concept

We applied the method presented in the previous section 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 SAs: 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 bitflip for each inference, analogous to Algorithm 2. 24.8k ImageNet [11] inferences were executed for each configuration. The results are shown in Table 1.

The simulation was run on an Intel i9-7960X CPU. A single inference without fault injection took 104 ms. The overhead in Table 1 is given with reference 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 versus without fault simulation. "#MMA" lists the number of MMA calls for a single inference.

**Table 1. 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 |

As expected from the previous section, 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 straightforward. 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 SA's implementation. Consequently, one should not draw conclusions on the risk (2) associated with different SA configurations from Table 1 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.

## Future work

The best- and worst-case approaches for modeling permanent faults (previous section) do 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 the previous section, we suggest running a hardware simulation, $\widetilde{\text{MMA}}$, for the complete MMA instruction. When modeling permanent faults inside the SAs, this simulation overhead becomes significant. In future work, we will develop methods reducing the simulation overhead to simulating single MAF units (previous section) 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. ■

## Acknowledgments

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 *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, Jun. 2018, pp. 1–6.

[2] H. T. Nguyen et al., "Chip-level soft error estimation method," *IEEE Trans. Device Mater. Rel.*, vol. 5, no. 3, pp. 365–381, Sep. 2005.

[3] A. Rodriguez, *Deep Learning Systems: Algorithms, Compilers, and Processors for Large-Scale Production*. San Rafael, CA, USA: Morgan & Claypool, 2020.

[4] S. Chetlur et al., "CuDNN: Efficient primitives for deep learning," Oct. 2014, *arXiv:1410.0759*.

[5] Z. Kalbarczyk et al., "Hierarchical simulation approach to accurate fault modeling for system dependability evaluation," *IEEE Trans. Softw. Eng.*, vol. 25, no. 5, pp. 619–632, Oct. 1999.

[6] V. Sridharan and D. R. Kaeli, "Using hardware vulnerability factors to enhance AVF analysis," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2010, pp. 461–472.

[7] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *Proc. Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2009, pp. 117–128.

[8] N. Seifert and N. Tam, "Timing vulnerability factors of sequentials," *IEEE Trans. Device Mater. Rel.*, vol. 4, no. 3, pp. 516–522, Sep. 2004.

[9] G. Li et al., "Understanding error propagation in deep learning neural network (DNN) accelerators and applications," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (ACM)*, 2017, pp. 1–12.

[10] K. He et al., "Deep residual learning for image recognition," 2015, *arXiv:1512.03385*.

[11] J. Deng et al., "ImageNet: A large-scale hierarchical image database," in *Proc. CVPR*, 2009, pp. 248–255.



**Patrik Omland** is a research scientist with Intel Deutschland GmbH, 85579 Neubiberg, Germany. He is pursuing a PhD with the Department of Informatics, Technical University Munich, Munich, Germany. His research interests include the effect of hardware faults on program execution and digital arithmetic/numerical algorithms. Omland has a master's in mathematical physics from the Ludwig-Maximilians-University Munich, Munich.

**Yang Peng** is a research scientist and system architect with Intel Deutschland GmbH, 85579 Neubiberg, Germany. His research interest includes system architecture for dependable artificial intelligence/machine learning (AI/ML)-based systems. Peng has a PhD in electrical engineering from the Technical University of Munich, Munich, Germany.

**Michael Paulitsch** is a principal engineer with Intel Deutschland GmbH, 85579 Neubiberg, Germany, where he leads the Dependability Research Laboratories. His research interests include novel architectures for dependable systems and machine learning. Paulitsch has a PhD from Technical University Vienna, Vienna, Austria, and a PhD from the Vienna University of Economics and Business, Vienna. He is a Senior Member of IEEE.

**Jorge Parra** is a computer architect with Intel Corporation, Santa Clara, CA 95054 USA, working on Intel's Xe GPU products. His research interests include computer architecture, machine learning hardware architectures, and artificial intelligence. Parra has a PhD and an MSc in electrical engineering from the University of New Mexico, Albuquerque, NM, USA.

**Gustavo Espinosa** is a senior principal engineer with Intel Corporation, Santa Clara, CA 95054 USA, where he leads reliability and security architecture development for discrete GPU products. Espinosa has a master's in computer engineering from Boston University, Boston, MA, USA. He is a member of IEEE.

**Abishai Daniel** is a staff reliability, availability and serviceability (RAS) quality and reliability engineer with Intel Corporation, Santa Clara, CA 95054 USA, with a focus on statistical predictive model development and application of novel machine learning techniques to reliability modeling. Abishai has an MSEE and a PhD from the University of Michigan, Ann Arbor, MI, USA.

**Gereon Hinz** is the CEO of STTech, 82031 Grünwald, Germany, providing solutions to current and upcoming technological challenges in the autonomous systems domain. Hinz has a master's in cybernetics from the University of Stuttgart, Stuttgart, Germany.

**Alois Knoll** is a professor of computer science with the Department of Informatics, Technical University Munich, 85748 Munich, Germany. His research interests include robotics, artificial intelligence, and realtime systems. He is a Senior Member of IEEE.

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