Low-Cost Online Convolution Checksum Checker

Dionysios Filippas, Nikolaos Margomenos, Nikolaos Mitianoudis, *Senior Member, IEEE*, Chrysostomos Nicopoulos, *Member, IEEE*, and Giorgos Dimitrakopoulos

***Abstract*— Managing random hardware faults requires the faults to be detected online, thus simplifying recovery. Algorithm-based fault tolerance has been proposed as a low-cost mechanism to check online the result of computations against random hardware failures. In this case, the checksum of the actual result is checked against a predicted checksum computed in parallel by a hardware checker. In this work, we target the design of such checkers for convolution engines that are currently the most critical building block in image processing and computer vision applications. The proposed convolution checksum checker, named ConvGuard, utilizes a newly introduced invariance condition of convolution to predict *implicitly* the output checksum using only the pixels at the border of the input image. In this way, ConvGuard reduces the power required for accumulating the input pixels without requiring large buffers to hold intermediate checksum results. The design of ConvGuard is generic and can be configured for different output sizes and strides. The experimental results show that ConvGuard utilizes only a small percentage of the area/power of an efficient convolution engine while being significantly smaller and more power efficient than a state-of-the-art checksum checker for various practical cases.**

***Index Terms*— Algorithm-based fault tolerance, convolution, error detection, reliability.**

I. Introduction

CONVOLUTION is an essential operation in image processing and it is widely applied in image signal processors [1], [2], camera processing pipelines [3], and computational photography [4]. The importance of convolution has increased considerably with the emergence of deep learning and, more specifically, convolutional neural networks (CNNs). A CNN is a special type of neural network architecture that relies on convolution layers and has shown remarkable performance in many application fields, such as computer vision [5], natural language processing [6], and robotics [7].

This widespread adoption of CNNs has triggered the need to accelerate them directly in hardware, using a variety of customized architectures that attempt to balance the need for high throughput with energy efficiency [8]. Specialized vector and tensor processors are designed to accelerate convolutions by first mapping them to equivalent matrix algebra operations, covering both dense and sparse data representations [9]–[11]. Dataflow and systolic architectures follow a similar approach, but orchestrate computation differently, with the goal being to break the register file bottleneck [12], [13]. Specialized convolution engines compute convolutions using sliding-window architectures and unrolled hardware units, thus taking better advantage of local buffering and memory reuse [14].

Besides performance and energy-efficiency requirements, the increasing prevalence of CNNs in safety-critical systems also increases the need for building resilient CNNs as an essential piece in guaranteeing the correctness of inference applications [15], [16]. This combined need for high-performance computation and functional safety is prevalent in various application domains, such as automotive systems. Guaranteeing correct computation in the presence of random hardware faults is necessary for safety and possible standards compliance [17]. For instance, ISO 26262 functional safety compliance requires that systems must function correctly, with potentially unsafe faults detected and controlled to prevent a hazard [18]. Thus, compliant systems must have very high fault detection capabilities.

Managing random hardware faults, such as soft [19] and hard errors [20], requires special hardware modules for fault detection [21] that allows faults to be detected online and rapidly, possibly within a few cycles of their occurrence, thus simplifying recovery. The importance of online error detection is further increased, if one considers the additional reliability constraints imposed by modern implementation technologies, including process variations, device wear-out, and aging [20]. The problem is accentuated in ultralow-power applications that execute CNN models at the edge in low-voltage setups to enable always-on intelligence on mobile and Internet-of-Things (IoT) devices [22], [23].

Algorithm-based fault tolerance (ABFT) techniques [24], [25] offer a low-cost mechanism to detect abnormal behavior in matrix-based computations [26] by comparing the true output checksum with a predicted one. Checksum computation and checking can be done either in software [27], [28] or in hardware [29]. In this work, we focus on *convolution-specific ABFT hardware checkers*.

In the case of a hardware online checker, as shown in Fig. 1, the checker is attached to the input and the output of the convolution engine and computes the true and the predicted checksums that characterize the result of convolution. When the two checksums differ, an error flag is asserted.

Manuscript received July 1, 2021; revised September 19, 2021; accepted October 6, 2021. The work of Dionysios Filippas, Nikolaos Margomenos, and Giorgos Dimitrakopoulos was supported by Siemens EDA Research Grant to Democritus University of Thrace, Greece, on "High-Level Synthesis Research for System on Chip." *(Corresponding author: Dionysios Filippas.)*

Dionysios Filippas, Nikolaos Margomenos, Nikolaos Mitianoudis, and Giorgos Dimitrakopoulos are with the Department of Electrical and Computer Engineering, Democritus University of Thrace, 67100 Xanthi, Greece (e-mail: dfilippa@ee.duth.gr; nikomarg2@ee.duth.gr; nmitiano@ee.duth.gr; dimitrak@ee.duth.gr).

Chrysostomos Nicopoulos is with the Department of Electrical and Computer Engineering, University of Cyprus, 1678 Nicosia, Cyprus (e-mail: nicopoulos@ucy.ac.cy).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TVLSI.2021.3119511.

Digital Object Identifier 10.1109/TVLSI.2021.3119511

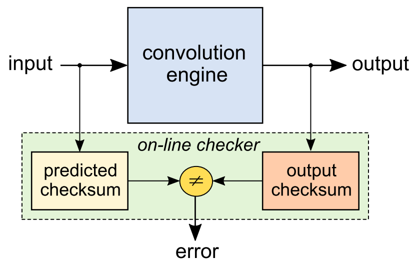


Fig. 1. Online checksum checker operates in parallel to the convolution engine and compares the true and the predicted checksums of convolution.

The checker does not interfere with, or interrupts, the operation of the convolution engine but simply provides online fault detection at the checksum level. Checksum checking cannot distinguish the correctness of every output pixel produced by the convolution engine. Instead, it checks whether the sum of a group of output pixels matches the expected sum. In a similar vein, checking the result of convolution can be done using residue checking architectures [21], [30] that fall behind checksum-based hardware checkers, as shown in [29].

The prediction of the convolution checksum should be done in a cost-efficient manner. In state-of-the-art approaches, such as [28], [29], and [31], the predicted output checksum is computed explicitly using the same input pixels used for the actual convolution. Significant amount of computation is saved [29] by reusing efficiently the already computed checksums at the cost of additional buffering to store those reusable results.

On the contrary, the proposed checksum checker, called ConvGuard, follows a different approach, enabled by a new fundamental property of convolution checksums introduced in this work. Instead of accumulating the input pixels used for the actual convolution, ConvGuard predicts the output checksum of convolution implicitly by accumulating only the peripheral pixels at the border of the input image that are dropped, or not computed, at the output. In this way, ConvGuard significantly reduces the power required for accumulating the input pixels, without requiring large buffers to store intermediate checksum results. Overall, the key contributions of this work can be summarized as follows.

1) ConvGuard introduces a novel invariant condition for 2-D convolutions and utilizes it to predict implicitly the checksum of the convolution output. This alternative checksum computation can be computed rapidly with a low-cost hardware module that can easily track the performance (clock frequency and throughput) of the monitored convolution engine.
2) The proposed checksum convolution checker can be configured to various convolution structures, including output size and stride. Especially in the case of nonunity-stride convolutions, only useful input pixels are accumulated and no redundant computation is involved.
3) The experimental results, using detailed hardware analysis of synthesized designs, highlight that ConvGuard utilizes only a small percentage of the area/power of an efficient convolution engine while being significantly smaller than a state-of-the-art checksum checker [29]. The results scale well for increased image and filter sizes. Also, the minimum buffering requirements of ConvGuard reduce its susceptibility to bit-flip errors.

$$y_B = \begin{bmatrix} y_{00} & y_{01} & y_{02} & y_{03} \\ y_{10} & \mathbf{y_{11}} & \mathbf{y_{12}} & \mathbf{y_{13}} \\ y_{20} & \mathbf{y_{21}} & \mathbf{y_{22}} & \mathbf{y_{23}} \\ y_{30} & \mathbf{y_{31}} & \mathbf{y_{32}} & \mathbf{y_{33}} \end{bmatrix} \quad y_D = \begin{bmatrix} y_{00} & y_{01} & y_{02} & y_{03} \\ y_{10} & \mathbf{y_{11}} & \mathbf{y_{12}} & y_{13} \\ y_{20} & \mathbf{y_{21}} & \mathbf{y_{22}} & y_{23} \\ y_{30} & y_{31} & y_{32} & y_{33} \end{bmatrix}$$

Fig. 2. Examples of output images as a result of the convolution of a 3×3 input image and a 2×2 filter. The useful output pixels are highlighted in blue. The rest are the extra outputs that should have been dropped or not calculated.

The rest of this article is organized as follows. Section II introduces the invariance condition of convolution checksum and the implicit checksum prediction. Section III presents the hardware checker that implements implicit prediction. Section IV presents the checker for nonunity-stride convolutions. Experimental results are given in Section V, while conclusions are drawn in Section VI.

II. Prediction of Convolution Checksum

The convolution of an $H \times W$ image x with a filter h of size $M \times M$ is calculated as follows [32]:

$$y_{mn} = \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} h_{ij} x_{m-i,n-j} \qquad (1)$$
$$m \in [0, P-1], \quad n \in [0, Q-1].$$

Formally, the size of the output y is $P \times Q$, with $P = M + H - 1$ and $Q = M + W - 1$, and is larger than the input image. However, in practice, the output pixels on the border of the image may not be computed. In such cases, the output image is either of equal size to the input image or, most often, smaller. Fig. 2 shows two possible convolution outputs (the pixels in blue) for a 3×3 input image and a 2×2 filter

$$x = \begin{bmatrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \\ x_{20} & x_{21} & x_{22} \end{bmatrix}, \qquad h = \begin{bmatrix} h_{00} & h_{01} \\ h_{10} & h_{11} \end{bmatrix}.$$

In the case of y_D, convolution is performed only on the pixels of the input image without requiring any border padding [14]. Hence, the output image is smaller than the input image.

Convolution can be equivalently expressed as a matrix–vector multiplication [8]

$$\mathbf{y} = A\mathbf{h}_{\text{vec}}. \qquad (2)$$

Vector $\mathbf{h}_{\text{vec}}$ contains all the $\hat{M} = M \times M$ coefficients of the filter arranged one after the other in a row-wise fashion in one column. Output vector $\mathbf{y}$ contains all elements of convolution ($P \times Q$ in total) again in a row-wise fashion. For the multiplication to be valid, matrix A contains one row for each application of the filter to the input image, i.e., for each possible position of the sliding window, including the outer border. Therefore, since the $\hat{M}$ filter coefficients will be multiplied with an equal number of input pixels, matrix A consists of $\hat{M}$ columns.

For the convolution of a 3×3 input with a 2×2 filter, A consists of all elements of the input from where the filter h would slide over, assuming a zero-padded border

$$A = \begin{bmatrix} 0 & 0 & 0 & x_{00} \\ 0 & 0 & x_{00} & x_{01} \\ 0 & 0 & x_{01} & x_{02} \\ 0 & 0 & x_{02} & 0 \\ 0 & x_{00} & 0 & x_{10} \\ x_{00} & x_{01} & x_{10} & x_{11} \\ x_{01} & x_{02} & x_{11} & x_{12} \\ x_{02} & 0 & x_{12} & 0 \\ 0 & x_{10} & 0 & x_{20} \\ x_{10} & x_{11} & x_{20} & x_{21} \\ x_{11} & x_{12} & x_{21} & x_{22} \\ x_{12} & 0 & x_{22} & 0 \\ 0 & x_{20} & 0 & 0 \\ x_{20} & x_{21} & 0 & 0 \\ x_{21} & x_{22} & 0 & 0 \\ x_{22} & 0 & 0 & 0 \end{bmatrix}. \tag{3}$$

A. Invariant Condition for Convolution Checksum

Since $\mathbf{y} = A\mathbf{h}_{\text{vec}}$, and $A = [a_{ij}]$, every element y_i of $\mathbf{y}$ is computed as

$$y_i = \sum_{j=0}^{\hat{M}-1} a_{ij} h_j^{\text{vec}}. \tag{4}$$

Summing all y_i's yields

$$\sum_{i=0}^{PQ-1} y_i = \sum_{i=0}^{PQ-1} \sum_{j=0}^{\hat{M}-1} a_{ij} h_j^{\text{vec}}. \tag{5}$$

By rearranging the order of the two sums in (5), we get

$$\sum_{i=0}^{PQ-1} y_i = \sum_{j=0}^{\hat{M}-1} \sum_{i=0}^{PQ-1} a_{ij} h_j^{\text{vec}} = \sum_{j=0}^{\hat{M}-1} \left(\sum_{i=0}^{PQ-1} a_{ij} \right) h_j^{\text{vec}}. \tag{6}$$

The sum inside the parentheses of (6) corresponds to the sum of the elements of the jth column of A. It can easily be observed in (3) and proven in the general case that the sum of the input pixels of each column of A is the same for all columns and equal to the sum of all pixels of the input. Therefore, we can replace $\sum_{i=0}^{PQ-1} a_{ij}$ with $\sum_{k=0}^{H-1} \sum_{l=0}^{W-1} x_{kl}$. Based on this observation, we can write

$$\begin{aligned} \sum_{i=0}^{PQ-1} y_i &= \sum_{j=0}^{\hat{M}-1} \left(\sum_{k=0}^{H-1} \sum_{l=0}^{W-1} x_{kl} \right) h_j^{\text{vec}} \\ &= \left(\sum_{k=0}^{H-1} \sum_{l=0}^{W-1} x_{kl} \right) \left(\sum_{j=0}^{\hat{M}-1} h_j^{\text{vec}} \right). \end{aligned} \tag{7}$$

In other words, in (7), we have shown that the sum of all output pixels of the convolution is equal to the product of the sum of all input data elements x_{kl} with the sum of all the filter's coefficients.

Let S_y denote the set of indices that support image y and S_h and S_x denote the indices that support h and x, respectively. The invariance condition (7) becomes

$$\sum_{i \in S_y} y_i = \left(\sum_{k \in S_x} x_k \right) \left(\sum_{j \in S_h} h_j \right). \tag{8}$$

The set S_y can be divided into two sets S_y^{xtr} and S_y^{crp} that denote the pixel indices that are cropped from the original image and the pixel indices that remain in the cropped image, respectively. The set of S_y^{xtr} is not fixed and it represents all pixels that are left off, depending on the choice of the useful output and the structure of the convolution. It is straightforward to see that $S_y = S_y^{\text{crp}} + S_y^{\text{xtr}}$. Thus, (8) becomes

$$\sum_{i \in S_y^{\text{crp}}} y_i + \sum_{i \in S_y^{\text{xtr}}} y_i = \left(\sum_{k \in S_x} x_k \right) \left(\sum_{j \in S_h} h_j \right). \tag{9}$$

Let us see an arithmetic example of this newly introduced invariance condition for the convolution of a 3×3 input image x with a 2×2 filter h

$$x = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 1 & 2 \\ 1 & 1 & 2 \end{bmatrix}, \qquad h = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}.$$

According to (1), the complete output consists of 4×4 pixels as follows:

$$y = \begin{bmatrix} 1 & 3 & 4 & 4 \\ 4 & \mathbf{10} & \mathbf{14} & 12 \\ 4 & \mathbf{10} & \mathbf{14} & 12 \\ 3 & 7 & 10 & 8 \end{bmatrix}.$$

Depending on which output pixels are considered useful, invariant condition (9) would take a different form. For the case of unity-stride convolutions and assuming that convolution is performed only on the pixels of the input image without padding (such as case y_D in Fig. 2), the useful pixels that will actually be computed by the convolution engine are the ones highlighted in bold. Inevitably, the remaining pixels at the periphery of the output image are the extra pixels that will not be computed by the convolution engine. The sum of the highlighted outputs is equal to $\sum y_{\text{crp}} = 48$, while the sum of the unused outputs $\sum y_{\text{xtr}} = 72$. In all cases, according to (9), the sum of the two disjoint sets of pixels $(48 + 72 = 120)$ is equal to the product of sums $(\sum x_k)(\sum h_j) = 12 \times 10$.

B. Explicit and Implicit Prediction of the Output Checksum

An online checksum checker, similar to the one shown in Fig. 1, would accumulate all useful output pixels coming out of the convolution engine and compare the derived checksum with the predicted one. Predicting the output checksum either explicitly, or implicitly, means to recompute $\sum_{i \in S_y^{\text{crp}}} y_i$ directly from the input without interfering at any point with the convolution engine.

The useful output pixels y_{crp} and the extra ones y_{xtr} can both be computed according to (2) as follows:

$$y_{\text{crp}} = A^{\text{crp}} \mathbf{h}_{\text{vec}}, \qquad y_{\text{xtr}} = A^{\text{xtr}} \mathbf{h}_{\text{vec}}. \tag{10}$$

Matrices A^{crp} and A^{xtr} contain only the rows of A that correspond to each disjoint set of outputs. For our running example (case y_D in Fig. 2),

$$A^{\text{crp}} = \begin{bmatrix} x_{00} & x_{01} & x_{10} & x_{11} \\ x_{01} & x_{02} & x_{11} & x_{12} \\ x_{10} & x_{11} & x_{20} & x_{21} \\ x_{11} & x_{12} & x_{21} & x_{22} \end{bmatrix}, \quad A^{\text{xtr}} = \begin{bmatrix} 0 & 0 & 0 & x_{00} \\ 0 & 0 & x_{00} & x_{01} \\ 0 & 0 & x_{01} & x_{02} \\ 0 & 0 & x_{02} & 0 \\ 0 & x_{00} & 0 & x_{10} \\ x_{02} & 0 & x_{12} & 0 \\ 0 & x_{10} & 0 & x_{20} \\ x_{12} & 0 & x_{22} & 0 \\ 0 & x_{20} & 0 & 0 \\ x_{20} & x_{21} & 0 & 0 \\ x_{21} & x_{22} & 0 & 0 \\ x_{22} & 0 & 0 & 0 \end{bmatrix}.$$

From (10), we can compute the elements of y_{crp} and y_{xtr} as follows:

$$y^i_{\text{crp}} = \sum_{j=0}^{\hat{M}-1} a^{\text{crp}}_{ij} h_j, \qquad y^i_{\text{xtr}} = \sum_{j=0}^{\hat{M}-1} a^{\text{xtr}}_{ij} h_j. \tag{11}$$

To compute the checksum of the useful output pixels, we need to sum all elements y^i_{crp}. Let us assume that the number of useful output pixels is equal to K

$$\begin{aligned} \sum_{i=0}^{K-1} y^i_{\text{crp}} &= \sum_{i=0}^{K-1}\sum_{j=1}^{\hat{M}-1} a^{\text{crp}}_{ij} h_j = \sum_{j=0}^{\hat{M}-1}\sum_{i=0}^{K-1} a^{\text{crp}}_{ij} h_j \\ &= \sum_{j=0}^{\hat{M}-1}\left(\sum_{i=0}^{K-1} a^{\text{crp}}_{ij}\right) h_j. \end{aligned} \tag{12}$$

The derived equation (12) tells us how to explicitly predict the output checksum using only the pixels that participate in the convolution (i.e., the ones in the center of the input image that are grouped in A^{crp}). To do so, we need to compute the sum of each column of A^{crp}, i.e., $\sum_{i=0}^{K-1} a^{\text{crop}}_{ij}$ for column j, and multiply the result with the corresponding filter coefficient. Then, we should reduce the derived partial products to one final predicted checksum.

Alternatively, with ConvGuard, we can compute *implicitly* the same sum of output pixels using the new invariance condition (9), which can be rewritten as

$$\sum_{i=0}^{K-1} y^i_{\text{crp}} = \left(\sum_{k\in S_x} x_k\right)\left(\sum_{j\in S_h} h_j\right) - \sum_{i=0}^{\hat{K}-1} y^i_{\text{xtr}}. \tag{13}$$

Since the number of useful pixels is assumed to be equal to K, the number of extra pixels (zero and nonzero) is equal to $\hat{K} = PQ - K$. The sum of extra output pixels can be expressed similar to (12) as follows:

$$\sum_{i=1}^{\hat{K}-1} y^i_{\text{xtr}} = \sum_{j=0}^{\hat{M}-1}\left(\sum_{i=0}^{\hat{K}-1} a^{\text{xtr}}_{ij}\right) h_j. \tag{14}$$

Also, the product of sums $(\sum x_k)(\sum h_j)$ can be restructured as

$$\left(\sum_{k\in S_x} x_k\right)\left(\sum_{j\in S_h} h_j\right) = \sum_{j=0}^{\hat{M}-1}\left(\sum_{k\in S_x} x_k\right) h_j. \tag{15}$$

By replacing (14) and (15) into (13), we get

$$\begin{aligned} \sum_{i=0}^{K-1} y^i_{\text{crp}} &= \sum_{j=0}^{\hat{M}-1}\left(\sum_{k\in S_x} x_k\right) h_j - \sum_{j=0}^{\hat{M}-1}\left(\sum_{i=0}^{\hat{K}-1} a^{\text{xtr}}_{ij}\right) h_j \\ &= \sum_{j=0}^{\hat{M}-1}\left(\sum_{k\in S_x} x_k - \sum_{i=1}^{\hat{K}-1} a^{\text{xtr}}_{ij}\right) h_j. \end{aligned} \tag{16}$$

Equation (16) corresponds to the implicit prediction of the output checksum. Instead of directly using the central pixels of the input image, we accumulate the columns of A^{xtr} that consist only of peripheral pixels, i.e., $\sum_{i=1}^{\hat{K}-1} a^{\text{xtr}}_{ij}$ for each column j. Each one of those accumulated sums (one for each filter coefficient) is subtracted from a common sum that corresponds to the sum of all input pixels, irrespective of their position. Then, the derived differences are multiplied with their corresponding filter's coefficients and reduced to a final sum.

In the case of multiple filters, the same approach holds for each separate filter. In addition, the approach can be applied on the case of 3-D convolution. Since 3-D convolutions are commonly decomposed to depth-wise convolutions or pseudo-3-D convolutions [33], it is straightforward to apply the above approach to each separable filter and check the result of the corresponding convolution.

For realistic image sizes and unity-stride convolutions, the number of extra pixels is much smaller than the useful ones. Therefore, choosing to accumulate the peripheral input pixels is expected to lead the overall fewer additions. This choice is unique to ConvGuard and a direct consequence of the invariance condition (9) introduced in this work.

III. Online Checker Architecture

The architecture of ConvGuard is shown in Fig. 3. The checker module operates in parallel to the convolution engine, without interfering with its operation. ConvGuard monitors the input x and the output y of the engine and predicts implicitly the output checksum by computing online equation (16).

A. Checker Organization

In each cycle, ConvGuard performs two tasks. On the output side, it accumulates the valid output samples produced by the convolution engine. Recall that, without loss of generality, we assume that the convolution engine computes only the useful output pixels and does not produce any invalid output. If it does, it just needs to mark the pixels as invalid so that ConvGuard can skip them. On the input side, to check the correctness of convolution, ConvGuard computes one sum for each column of A^{xtr} and a common sum of all input pixels. To do this, it employs one accumulator for each column of A^{xtr} ($\hat{M}$ in total) and one accumulator for the common sum.

Initially, all accumulators are reset to zero. Then, as each input pixel arrives (one per cycle, more pixels can arrive per cycle after marginal design changes), it is added to the appropriate accumulator, while all of the incoming pixels are added to the common-sum accumulator. Depending on the arriving input pixels, multiple accumulators may be enabled in the same cycle. For instance, in our running example, when

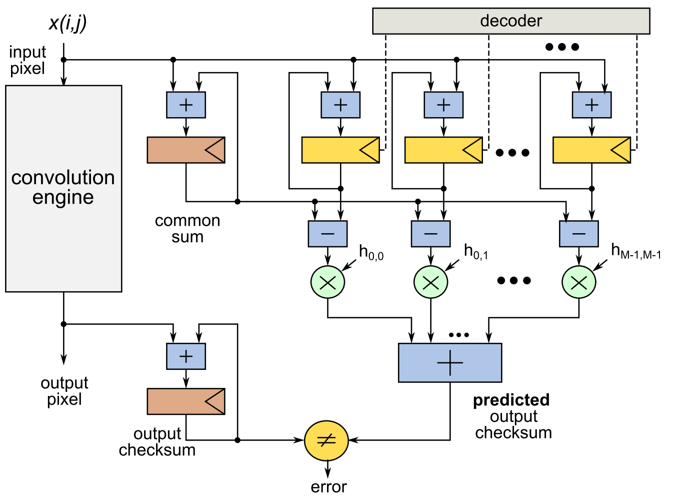


Fig. 3. Organization of ConvGuard. It runs in parallel to the convolution engine and it receives the same input and the engine's output pixels. Convguard accumulates the sum of the output pixels and compares it to its predicted checksum value. The predicted output checksum utilizes a set of accumulators—one for each filter coefficient—and a common-sum accumulator that computes the sum of all input pixels.

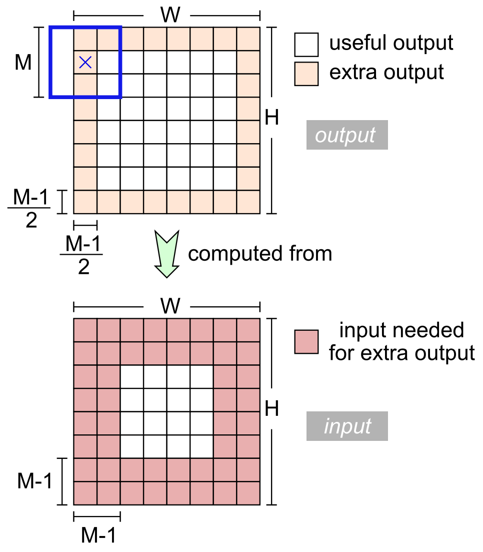


Fig. 4. Peripheral pixels that should be added for the computation of the sum of extra output pixels. Each highlighted pixel may contribute to many accumulators in the same cycle, as dictated by the decoder function.

input pixel x_{00} arrives, it contributes to the running sum of accumulators that correspond to the filter's coefficients h_{01}, h_{10}, and h_{11}. On the contrary, when input pixel x_{20} arrives, only the accumulator of h_{11} is activated. It should be noted that central pixels—like x_{11}—that do not appear in A^{xtr} are skipped and not added to any accumulator besides the common one.

The decision to which accumulator each input pixel contributes is done in the decoder, which is also shown in Fig. 3. The decoder decides two things: 1) which peripheral input pixels contribute to the computation of the extra output pixels and 2) to which accumulator they should be added.

As shown in Fig. 4, the extra output pixels consist of the $(M - 1)/2$ rows and columns on the border of the output. To derive those outputs, a larger border of $M - 1$ rows and columns is actually used from the input image. The decoder would allow only those input pixels to be added to the appropriate accumulators. On the contrary, all input pixels are added to the common-sum accumulator. Stated formally, an input pixel x_{ij} is added to the accumulator that corresponds to the filter's coefficient h_{mn} when at least one of the following inequalities is satisfied:

$$m > i > H - M + m \quad n > j > W - M + n. \tag{17}$$

When all input pixels have passed through the convolution engine, the checker's accumulators have accumulated all needed sums: one common sum and one sum for each column of A_{xtr}. At this point, the sum that corresponds to each coefficient is subtracted from the common sum, in order to correctly compute the term in the parentheses of (16). Then, each resulting term is multiplied with the corresponding filter's coefficient and the products are added to produce the final value, which corresponds to the predicted output checksum of the convolution.

For fixed-point implementations, which is the focus of this work, all registers and arithmetic units are sized appropriately so as to avoid any overflow conditions that would ruin the output checksum prediction. For checking a floating-point-based convolution engine, we cannot guarantee that the predicted output checksum would match the true output checksum, even under error-free operation. In these cases, the equality comparison should be transformed to a bounds check. If the predicted and the true output checksums differ by a certain small error bound, the convolution would still be considered fault free [21], [34].

Finally, it should be stressed that the prediction of the output checksum is computed gradually without requiring any buffering of intermediate results. This lack of buffering is critical in reducing the cost of the checksum checker. It is expected that an online checker should consume only a small percentage of the area of the convolution engine and leave only an incremental energy footprint, compared to the energy consumed in computing the actual convolution.

B. When Does Implicit Prediction of the Output Checksum Make Sense?

Predicting the output checksum implicitly using (16) is useful only when it can be computed with fewer additions, compared to an explicit prediction of the checksum. To understand when the two approaches break even, we need to count the number of additions needed in each case. Equivalently, we need to count the number of nonzero elements of matrices A^{crp} and A^{xtr}.

In the case of explicit checksum prediction, A^{crp} consists of only nonzero elements. It has as many rows as the number of useful output pixels. According to Fig. 4, the number of useful output pixels is $K = (H - M + 1)(W - M + 1)$ for odd values of M. The number of columns of matrix A^{crp} is always equal to the number of the filter's coefficients $\hat{M}$ (equal to M^2). Therefore, by multiplying the two, the number of additions required for the explicit computation of the checksum is

$$\#\text{explicit adds} = \hat{M}\,K = M^2(H - M + 1)(W - M + 1). \tag{18}$$

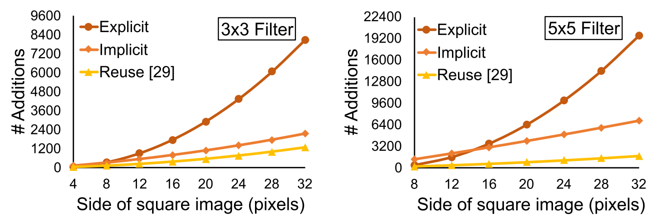


Fig. 5. Number of additions needed for predicting the output checksum explicitly, implicitly, and with maximum reuse of already computed results [29] for two filter cases and various square input image sizes.

On the contrary, the implicit checksum prediction has to do with all the remaining output pixels. Recall from (3) that each column of A contains all input pixels and some zero elements. Therefore, the nonzero elements of every column of A^{xtr} that should be added are equal to the number of all input pixels $H\,W$ minus the elements of the same column of A^{crp}, i.e., $H\,W - (H - M + 1)(W - M + 1)$. Since there are $\hat{M}$ columns in A^{xtr}, the number of additions required is equal to $\hat{M}(H\,W - (H - M + 1)(W - M + 1))$. By replacing (18) in the derived formula, we conclude that, to compute the sum of the extra output pixels, we need

$$\hat{M}\,H\,W - \#\text{explicit adds}$$

additions. The checker computes also a common sum that involves the sum of all input pixels. Therefore, ConvGuard requires $H\,W$ more additions. Overall,

$$\#\text{implicit adds} = (1 + \hat{M})H\,W - \#\text{explicit adds}. \quad (19)$$

Using (19) and (18), we can compare the efficiency of these two approaches for arbitrary image and filter sizes. The implicit approach proposed by ConvGuard requires fewer additions than the explicit approach when

$$\#\text{explicit adds} > \left(\frac{\hat{M} + 1}{2}\right)H\,W. \quad (20)$$

Fig. 5 shows the number of additions required in each case (“Explicit” and “Implicit”) for a 3×3 and a 5×5 filter for various square input dimensions. For really small input images, it is more efficient—in terms of number of additions—to predict the output checksum explicitly. When the input image is larger, implicit prediction is more efficient than explicit prediction. For instance, for a 3×3 input filter, implicit prediction is more efficient for any input image larger than 8×8 pixels. The minimum input image size required to make implicit prediction more cost-efficient for various filter sizes and for stride $S = 1$ is presented in the first column of Table I. The presented sizes are encountered in many existing applications. For instance, VGG-16 [35] has an input image of size 224×224, which is convolved with a filter of size 3×3. Furthermore, the second-to-fifth convolutional layers of AlexNet [36] perform convolutions on images with sizes of 27×27 and 13×13 using 5×5 or 3×3 filters.

Fig. 5 also shows the number of additions required by the state-of-the-art “Reuse” checksum checker, as presented in [29]. This approach relies on explicit prediction of the output checksum and reduces the total number of additions by reusing many of the already computed sums. However, the reduced number of additions comes at the cost of extra buffering to store the required intermediate results. As clearly shown in the experimental results in Section V-B, this extra buffering significantly increases the total area and power of this checker, relative to ConvGuard. Moreover, the extra buffers make the checker more susceptible to random bit-flips that would lead to false detection alarms, as analyzed in Section V-C.

TABLE I
MINIMUM SIZE OF THE SIDE OF A SQUARE IMAGE THAT FAVORS IMPLICIT OVER EXPLICIT PREDICTION OF THE OUTPUT CHECKSUM

Filter	Stride - S			
	1	2	4	6
3×3	8	14	-	-
5×5	15	17	43	-
7×7	23	26	34	64
11×11	35	38	46	54

IV. CHECKING NONUNITY-STRIDE CONVOLUTIONS

In the nonunity-stride convolutions found in many practical applications, the useful output pixels are even fewer. Furthermore, in these cases, extra pixels are present not only at the periphery of the image but also in the center as well. In such cases, predicting the output checksum implicitly would always require more additions than the explicit prediction. To enable the applicability of ConvGuard to nonunity-strided convolutions, we utilize a recently proposed transformation [37], [38] that allows the computation of any convolution with stride $S > 1$ using multiple channels of unity-stride convolutions. By applying the implicit checksum prediction on each independent unity-stride channel, we can still design a low-cost checksum checker.

A. Checking Independently Per Channel

In a unity-stride convolution, the filter is applied to every pixel of the input. On the contrary, in the case of a nonunity-stride convolution, the filter moves on the input with a step of S. In this case, each input pixel will not be multiplied with every filter coefficient, but with a subset of them. Fig. 6 groups the input pixels based on which filter's coefficient “touches” them. The blue input pixels will be multiplied only with the blue filter coefficients, while the green ones will be multiplied only with the green filter coefficient. Based on this observation, the work in [37] and [38] proposed to compute any nonunity-stride convolution by summing the result of S^2 smaller and independent unity-stride convolutions. The unity-stride convolutions are applied on selected subimage and subfilter pairs, as also shown in Fig. 6.

Being able to transform a nonunity-stride convolution into multiple unity-stride ones allows us to apply ConvGuard efficiently to arbitrary strides. More precisely, ConvGuard predicts the output checksum implicitly using (16) separately per channel. Since—according to [38]—the result of the each subconvolution is added to form the final convolution result, then the final prediction of the output checksum is the sum of all intermediate implicit predictions.

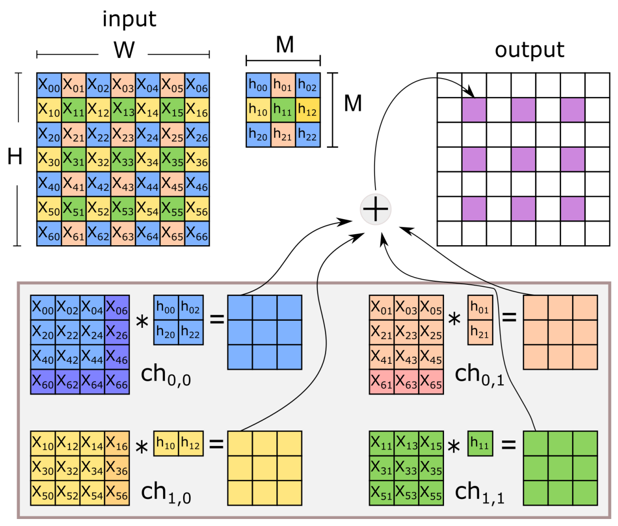


Fig. 6. Transformation of a strided convolution with $S = 2$ to a four-channel unity-stride convolutions, depicted with symbol $*$.

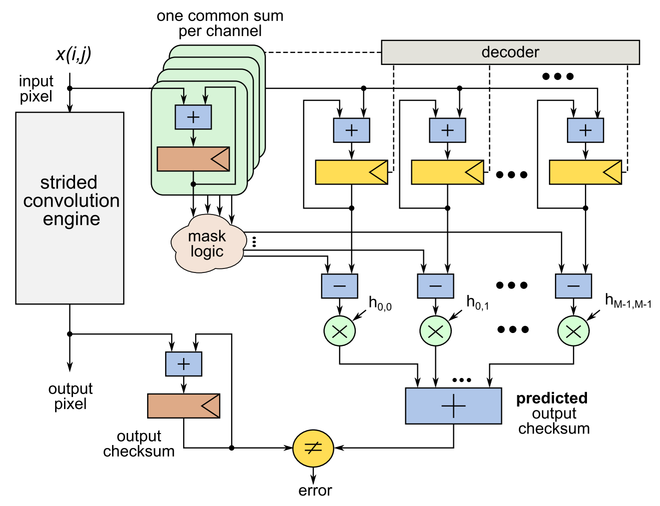


Fig. 7. Generalized ConvGuard architecture that supports arbitrary strided convolutions.

B. Generalized Checker

The organization of generalized ConvGuard is shown in Fig. 7. To compute (16) for each channel, we need more accumulators. Since the number of the filter's coefficients does not change, the number of accumulators that sum the input pixels per coefficient remains the same as in the baseline case ($S = 1$). However, we need more than one common-sum accumulators. Since we compute a common sum for the subimage of each channel, we need S^2 common-sum accumulators in total (one per channel). Thus, overall, for supporting convolutions with stride S, we need $S^2 + M^2$ accumulators.

For convolutions of arbitrary stride, decoding is a two-step procedure. The first step decides to which channel each pixel belongs, and the second step decides whether it is a peripheral pixel of the channel's subimage or not. The first check determines to which channel's common-sum accumulator the pixel should be added, and the second check (also using the result of the first check) decides to which filter coefficients the incoming pixel refers.

For the first check, the common-sum accumulator of channel (k, l) is increased when, for the input pixel (i, j), the following hold: $k = i \mod S$ and $l = j \mod S$.

For the second check, we actually need to check whether at least one of the inequalities in (17) holds after mapping the indices of the input pixels (i, j) and the filter's coefficients (m, n) to the "smaller" coordinates of each channel. The considered sizes for the subimages and subfilter should be scaled too. To achieve this, we merely need to integer-divide each variable of the inequalities with the selected stride S.

Once the common sums per channels have been accumulated and the coefficient accumulators get their final values, the appropriate common sums are subtracted from the appropriate accumulators, as shown in Fig. 7. The mask logic of Fig. 7 decides the assignment by identifying the common sums and the filter coefficients that belong to the same channel.

The number of additions required for the implicit prediction of the output checksum depends on the size of the input image, as well as the size of the filter. In addition, in the case of nonunity-stride convolutions, the efficiency of ConvGuard also depends on the sizes of all subimages and subfilters that emerge after the transformation to multichannel unity-stride convolutions. Thus, the number of additions depends on the selected stride as well.

Table I shows the minimum number of pixels that an input image should have to make the implicit prediction of the output checksum more efficient than its explicit counterpart. For instance, for stride $S = 2$ and a filter or size 5×5, the input image should be at least 17×17 pixels, while for a larger 11×11 filter, the minimum image size increases to 38×38 pixels. When the stride is larger than the filter, the multichannel decomposition of the original strided convolution is degenerated. In this case, each channel may contain only one filter coefficient or none. Hence, in such cases, the differentiation between implicit and explicit prediction no longer makes sense.

V. Experimental Evaluation

In the experimental results, we aim to highlight three aspects of ConvGuard. In the first set of experiments, our plan is to measure the hardware overhead of ConvGuard, relative to a customized convolution engine. Then, ConvGuard is compared, in terms of hardware complexity, with a state-of-the-art checker that minimizes the number of required additions to explicitly predict the output checksum. Finally, in the third set of experiments, we explore the fault detection properties of both checkers.

A. Hardware Overhead Added to Check an Optimized Convolution Engine

Convolution engines can employ various architectures. Choosing a high-throughput, but area-efficient, sliding-window-based architecture—similar to the one used in [14] and [39]—would reveal the worst case overhead expected from ConvGuard. In such sliding-window-based convolution engines, the incoming pixels are streamed in the engine and stored in an active window buffer of the same size as the filter and in row buffers that keep the $M - 1$ recently fetched

TABLE II

AREA AND POWER COMPLEXITY OF AN APPLICATION-SPECIFIC CONVOLUTION ENGINE AND CONVGUARD OPERATING AT 1 GHz

Filter	Image	Area (μm^2)		Power (mW)@ 1GHz	
		Engine	Checker	Engine	Checker
3×3	14×14	26908	9860	3.99	1.31
	28×28	32947	11278	4.90	1.53
	56×56	36678	12164	5.26	1.84
5×5	28×28	74104	18651	10.36	2.09
	56×56	100900	20184	11.16	2.35
	112×112	115380	22451	15.56	2.78
11×11	56×56	508873	63536	37.88	4.65
	112×112	544887	74463	54.95	5.37
	224×224	616364	68110	65.39	5.46

lines of the input image [14]. Row buffers can be built either using registers or SRAM blocks. The filtering function is an unrolled and possibly pipelined arithmetic datapath. The baseline input–output throughput of 1 pixel/cycle of these architectures can easily be increased to facilitate parallelism by accepting and producing more pixels/cycle [40].

The sliding-window-based convolution engine and the ConvGuard checker that operates in parallel have been designed in C++ and synthesized to Verilog RTL using Catapult HLS and driven by a commercial-grade 45-nm standard-cell library. Final timing/area results are derived from the Oasys logic synthesis tool. Line buffer memories are mapped to SRAM macro blocks to further minimize the area of the convolution engine. Keeping line buffers in registers would have increased the area of the convolution engine significantly and would unrealistically minimize the overhead of the checker. Power was estimated after synthesis using the PowerPro power analysis and optimization tool. Switching activity information was gathered after simulating the convolution engine and the checker using actual images and filters.

Both the convolution engine and ConvGuard have been synthesized for various image and filter sizes assuming 16-bit wide input pixels. In all cases, we assumed a target clock frequency of 1 GHz. Table II shows the area and power of each constituent part of a protected convolution engine. In addition, Fig. 8 shows the area and power percentage of ConvGuard, relative to the total area and power of the protected convolution engine, for each one of the examined cases.

ConvGuard provides checking capability to the convolution engine by incurring only a small additional area and power overhead. The overhead added is below 10% for 11×11 filters and increases for smaller filters and smaller image widths. The cost of ConvGuard is mostly determined by the size of the filter and is only slightly affected by the size of the image. Image size determines only logarithmically the bit-width of the checker's accumulators. Furthermore, when increasing the bit-width of the input pixels, the cost of the convolution engine that buffers actual pixels increases faster than the cost of ConvGuard, which only stores their sum. For instance, for 32-bit inputs (instead of 16-bit), the highest overhead shown in Fig. 8(a) for the case of a small 3×3 filter and a small

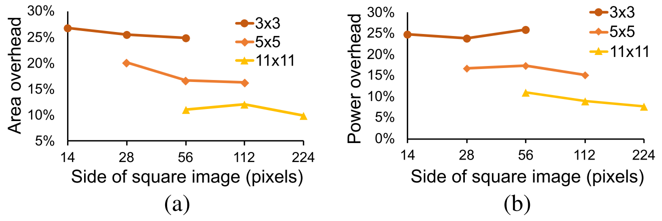


Fig. 8. (a) Area and (b) power cost of the ConvGuard checker as a percentage of the total area and power of the protected convolution engine.

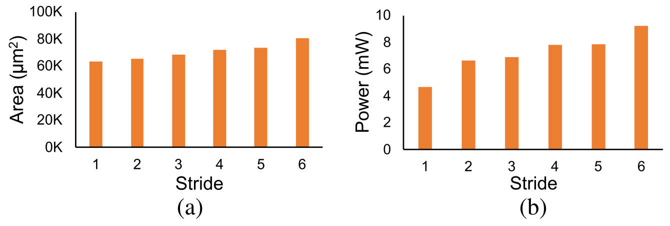


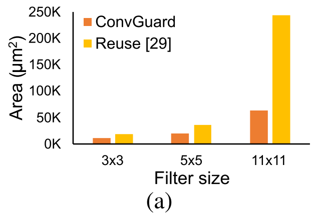
Fig. 9. (a) Area and (b) power scaling of ConvGuard with increasing stride for an 11×11 filter and a 56×56 example image.

14×14 input image drops from 25%—for 16-bit inputs—to 19% for 32-bit inputs (not shown in the figure).

Fig. 9 shows the area and power scaling of the ConvGuard architecture for increasing stride. The synthesized designs assume an 11×11 filter, where using nonunity strides makes more sense. From the reported results, we can see that increasing the stride only marginally increases the total area of the checker. Roughly, for every step of increasing stride, the area and power increase by 6% and 11%, respectively. This result can easily be explained since the majority of the area of ConvGuard is occupied by the area of the accumulators per filter coefficient and their associated datapath logic and less by the area of the common-sum accumulators used in each channel (see Fig. 7). Moreover, part of the area/power increase observed when increasing the stride is the result of the complexity of the mask logic shown in Fig. 7. The mask logic introduces additional multiplexing to forward the result of the multiple common-sum accumulators (one per channel) to the appropriate subtraction units.

B. Hardware Complexity Comparison With a State-of-the-Art Checker Using ASIC and FPGA Implementations

Having quantified the overhead of adding ConvGuard to a customized convolution engine, we now aim to highlight its efficiency relative to a recent state-of-the-art checker architecture [29]. In [29], the prediction of the output checksum is done explicitly and the already computed sums of pixels are kept and reused when forming larger sums. On one hand, this approach significantly reduces the number of additions, as shown in Fig. 5, but, on the other hand, it increases the number of buffers added to store the intermediate results. The HLS-ready C implementation of this "Reuse" architecture is publicly available in Git and used in this work after easily modifying the Vivado-specific synthesis constraints to Catapult-HLS-specific constraints. Although the design



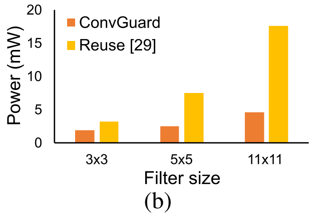


Fig. 10. (a) Area and (b) power overhead cost of the ConvGuard checker, compared to the reuse architecture [29] for an input image size of 56 × 56.

mentioned in [29] was targeting an FPGA implementation for testing overclocking possibilities, it was easily ported to an ASIC implementation with marginal modifications that kept the original organization of the checker. To enable a comparison against ConvGuard of the hardware cost, the C model was successfully synthesized to 1 GHz using a 45-nm technology library.

The area/power results obtained after synthesizing both designs for various filter sizes, and assuming an input image of 56 × 56 pixels, are shown in Fig. 10. The trend for other image sizes is the same. The cost of both checkers is mostly affected by the size of the filter, while the size of the input image only determines the bit-width of the accumulators.

In all cases, it is evident that ConvGuard requires significantly less area and power. This attribute of ConvGuard is attributed to the complete lack of buffering resources that fits well to its low-cost profile. Besides its accumulators, ConvGuard does not store any incoming pixels nor any previous intermediate checksum result. On the contrary, the “reuse” architecture requires a set of accumulators that handle final additions (equal in number to ConvGuard) and an additional set of accumulators for storing intermediate results. This extra sequential storage is inherent to the organization of the “reuse” architecture.

Similar conclusions are drawn when considering FPGA implementations of ConvGuard and the “reuse” architecture. We implemented two versions of the protected convolution engine—i.e., the engine and the checker—on a Xilinx Artix-7 chip (XC7A100T) targeting a clock frequency of 100 MHz. The first version includes ConvGuard as the checker, while the second uses the “reuse” architecture [29]. To study the impact of the filter size and the input image size on the final design, we report the implementation results for input image sizes of 56 × 56 and 112 × 112 and for filter sizes of 5 × 5 and 11 × 11.

The results obtained after mapping the Catapult-derived Verilog RTL to the FPGA using Xilinx Vivado 2021.1 are shown in Table III. The results include the resource utilization of an unprotected engine and the two versions of protected engines. The convolution engine utilizes as many DSP blocks as the square of the filter size to enable a fully unrolled implementation of the datapath. On the contrary, the number of BRAM blocks that implement the line buffers of the convolution engine [14] is determined linearly, both by the size of the input image and the size of the filter. The addition of the checker in parallel to the convolution engine increases the resource utilization for the two protected

TABLE III

FPGA Resource Utilization of an Unprotected Convolution Engine and Two Protected Engines Using the ConvGuard and Reuse [29] Checkers

Image		56 × 56		112 × 112	
Filter		5 × 5	11 × 11	5 × 5	11 × 11
Unprotected Engine	SLICE	511	2587	524	2707
	BRAM	2	5	2	5
	DSP	25	121	25	121
ConvGuard Protected Engine	SLICE	1268	5406	1274	5513
	BRAM	2	5	2	5
	DSP	26	122	26	122
Reuse [29] Protected Engine	SLICE	2201	13861	2317	14418
	BRAM	2.5	5.5	2.5	5.5
	DSP	26	123	26	123

convolution engines, as shown in Table III. Both checkers need additional DSP blocks and LUT slices to accommodate their arithmetic datapaths. “Reuse” also needs an extra half BRAM to implement its buffers. In all examined cases, it is evident that the ConvGuard checker leads to implementations with lower cost than “reuse” that scales favorably with increasing image and filter sizes.

C. Fault Detection Properties and Comparison With a State-of-the-Art Checker

In the last set of experiments, the goal is to quantify the fault detection properties of ConvGuard and compare them to the state-of-the-art checker analyzed in Section V-B. In this way, we highlight the additional benefit offered by the reduced buffering requirements, compared to reducing the number of additions. The smaller the number of buffers a checker needs, the smaller the probability to experience a fault inside the checker itself. Checker faults may lead to false alarms and/or missed fault detections.

Our experiments are based on injecting bit-flips in random clock cycles during the time interval needed to complete a convolution. Faults are injected to random storage elements in both the convolution engine and the checker. The number of faults injected in each run is a user parameter. The probability to experience a bit-flip is proportional to the area of the corresponding storage elements. For instance, the SRAM-based row buffers of the convolution engine are expected to experience a bit-flip more often than the accumulators of the checker. The input pixels and the filter coefficients used in each run are the ones used for power estimation. At the end of each convolution, we record the outcome of the fault injection campaign. The observed behavior may fall into one of four categories.

1) *Detected:* A fault occurred in the convolution engine and the checker detected it.
2) *Silent:* A fault occurred in the convolution engine and the checker did not detect it. In this case, we must be certain that the checker did not experience any faults. The effect of the fault was masked at the checksum level.
3) *False positive (FP):* The checker flagged a fault detection, but no fault occurred in the convolution engine.
4) *False negative (FN):* A fault occurred in the convolution engine and the checker did not detect it. In this case,

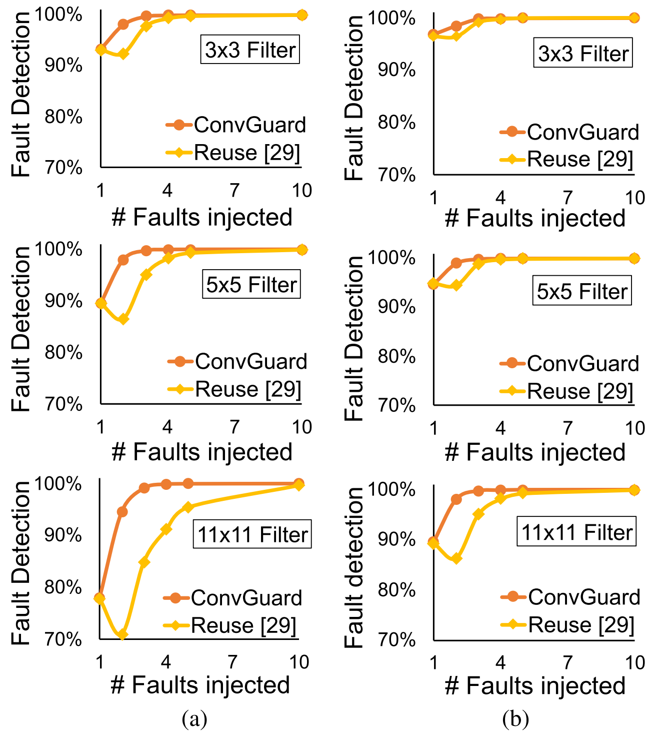


Fig. 11. Fault detection efficiency of the ConvGuard and "reuse" [29] architectures after injecting a varying number of faults to the same 10k convolutions of (a) 56 × 56 and (b) 112 × 112 input images using 3 × 3, 5 × 5, and 11 × 11 filters. In the case of one injected fault, it is assumed that this is injected only into the convolution engine in a random clock cycle.

we must be certain that the checker experienced a fault that caused its malfunction.

Fig. 11 shows the percentage of faults detected by ConvGuard and the "reuse" architecture [29] after executing the same 10k convolutions of 56 × 56 and 112 × 112 input images and using 3 × 3, 5 × 5, and 11 × 11 filters. In each case, an increasing number of faults were injected per convolution. In the case of injecting only a single fault, we assume that the fault is always injected in the convolution engine and the checker remains error-free. This is the reason why the fault detection performances of both checkers match (their performance depends solely on the fault detection properties of checksum-based checking). In the following cases, the faults are injected randomly to both the convolution engine and the checker. The increased number of buffers in the "reuse" architecture, relative to ConvGuard, reduces its fault detection efficiency. This difference is mostly the result of FP outcomes. Even if the convolution engine is error-free, the checker signals an error. When the number of faults injected is increased, both approaches converge to a high fault detection rate. The multiple faults occurring almost certainly cause a difference between the true and the predicted checksum in both cases.

Increasing the filter size increases the number of accumulators needed in both checkers, thus decreasing their fault detection capabilities. However, the effect is more pronounced in the "reuse" architecture, due to the buffers needed by construction to store the intermediate results. On the other hand, increasing the input image size only marginally affects the detection capabilities of both checkers since it

TABLE IV

OBSERVED BEHAVIOR AFTER INJECTING EITHER TWO OR FOUR RANDOM FAULTS IN CONVOLUTIONS OF INCREASING INPUT SIZE AND A 3 × 3 FILTER

Faults Injected	Image	Fault Categories			
		Detected	Silent	FP	FN
2	14 × 14	91.36%	3.98%	4.62%	0.01%
	28 × 28	95.95%	2.03%	2.01%	0.01%
	56 × 56	97.96%	1.45%	0.59%	0.00%
	112 × 112	98.58%	1.17%	0.25%	0.00%
4	14 × 14	99.54%	0.23%	0.23%	0.00%
	28 × 28	99.85%	0.12%	0.02%	0.00%
	56 × 56	99.93%	0.07%	0.00%	0.00%
	112 × 112	99.94%	0.06%	0.00%	0.00%

only affects logarithmically the bit-width of the checksum accumulators.

To quantify the fault detection efficiency of ConvGuard when increasing the image size, we injected two and four random faults in 10k convolutions using a 3 × 3 filter. The same number of convolutions was repeated for different input image sizes. The results are presented in Table IV. With small images, the probability of injecting a fault into the checker is higher, which leads to a measurable amount of false alarms (FP and FN cases). Instead, when the input image increases, the area of the line buffers increases, compared to the rest of the sequential storage. Thus, the line buffers inevitably experience the majority of the faults. Since the checker is less likely to experience a fault, it can detect the errors of the convolution engine more often. As the number of injected errors increases, the possibility of having a false alarm drops to almost zero. Overall, due to its low cost and high fault detection efficiency, ConvGuard can act complementary to other protection mechanisms, such as parity checking added to memory blocks [21].

VI. CONCLUSION

ABFT is a generic approach for detecting random hardware failures by identifying when there is a difference between the actual and the expected outcome at the checksum level. Such approaches can be used even for post-silicon design validation. In this work, we focus on convolution-specific ABFT implemented directly in hardware.

Our proposal identifies a generic invariance checking condition for convolution and uses it to design simpler online checksum checkers. To avoid any performance degradation, the prediction is computed in the same time frame that the convolution engine produces the true output. The proposed ConvGuard architecture does not recompute any output pixel; it only quickly predicts their sum. The simple mathematical formulation that guides the design of ConvGuard allows it to adapt to any convolution structure, including arbitrary stride parameters. Its algorithmic nature simplifies the design process and allows its easy adoption in both ASIC and FPGA chips.

In addition to reducing the number of additions by predicting the output checksum implicitly, ConvGuard operates using minimum buffering. Consequently, it saves considerable amount of area relative to a current state-of-the-art checker architecture [29], and it is less susceptible to false negative or false positive alarms for precisely the same reason.

References

[1] H. An *et al.*, "An ultra-low-power image signal processor for hierarchical image recognition with deep neural networks," *IEEE J. Solid-State Circuits*, vol. 56, no. 4, pp. 1071–1081, Apr. 2021.

[2] P. Hansen *et al.*, "ISP4ML: The role of image signal processing in efficient deep learning vision systems," in *Proc. 25th Int. Conf. Pattern Recognit. (ICPR)*, Jan. 2021, pp. 2438–2445.

[3] J. Hegarty *et al.*, "Darkroom: Compiling high-level image processing code into hardware pipelines," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 1–11, Jul. 2014.

[4] A. Adams *et al.*, "The Frankencamera: An experimental platform for computational photography," *Commun. ACM*, vol. 55, no. 11, pp. 90–98, Nov. 2012.

[5] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 779–788.

[6] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," *IEEE Comput. Intell. Mag.*, vol. 13, no. 3, pp. 55–75, Aug. 2018.

[7] K. Tateno, F. Tombari, I. Laina, and N. Navab, "CNN-SLAM: Real-time dense monocular SLAM with learned depth prediction," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 6243–6252.

[8] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

[9] N. P. Jouppi *et al.*, "Ten lessons from three generations shaped Google's TPUv4i," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2021, pp. 1–14.

[10] K. Patsidis, C. Nicopoulos, G. C. Sirakoulis, and G. Dimitrakopoulos, "RISC-V^2: A scalable RISC-V vector processor," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Oct. 2020, pp. 1–5.

[11] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-GHz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm FD-SOI," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 2, pp. 530–543, Feb. 2020.

[12] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 367–379.

[13] X. Wei *et al.*, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Proc. 54th Annu. Design Automat. Conf.*, Jun. 2017, pp. 1–6.

[14] L. Ioannou, A. Al-Dujaili, and S. A. Fahmy, "High throughput spatial convolution filters on FPGAs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 6, pp. 1392–1402, Jun. 2020.

[15] 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.*, Nov. 2017, pp. 1–12.

[16] B. Reagen *et al.*, "Ares: A framework for quantifying the resilience of deep neural networks," in *Proc. 55th ACM/ESDA/IEEE Design Automat. Conf. (DAC)*, Jun. 2018, pp. 1–6.

[17] R. Salay, R. Queiroz, and K. Czarnecki, "An analysis of ISO 26262: Using machine learning safely in automotive software," Sep. 2017 *arXiv:1709.02435*. [Online]. Available: https://arxiv.org/abs/1709.02435

[18] A. Hopkins, "Silicon evolution for the automotive revolution," ARM, Cambridge, U.K., White Paper, 2019. [Online]. Available: https://arxiv.org/abs/1808.01556

[19] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Trans. Device Mater. Rel.*, vol. 5, no. 3, pp. 305–316, Sep. 2005.

[20] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, Nov./Dec. 2005.

[21] I. Koren and C. Krishna, *Fault-Tolerant Systems*. San Mateo, CA, USA: Morgan Kaufmann, 2020.

[22] P. Whatmough, S. Lee, D. Brooks, and G.-Y. Wei, "DNN engine: A 28-nm timing-error tolerant sparse deep neural network processor for IoT applications," *IEEE J. Solid-State Circuits*, vol. 53, no. 9, pp. 2722–2731, Jun. 2018.

[23] S. K. Lee, P. N. Whatmough, D. Brooks, and G.-Y. Wei, "A 16-nm always-on DNN processor with adaptive clocking and multi-cycle banked SRAMs," *IEEE J. Solid-State Circuits*, vol. 54, no. 7, pp. 1982–1992, Jul. 2019.

[24] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. C-33, no. 6, pp. 518–528, Jun. 1984.

[25] S.-J. Wang and N. K. Jha, "Algorithm-based fault tolerance for FFT networks," *IEEE Trans. Comput.*, vol. 43, no. 7, pp. 849–854, Jul. 1994.

[26] P. Wu *et al.*, "Towards practical algorithm based fault tolerance in dense linear algebra," in *Proc. 25th ACM Int. Symp. High-Perform. Parallel Distrib. Comput.*, May 2016, pp. 31–42.

[27] J. Chen, X. Liang, and Z. Chen, "Online algorithm-based fault tolerance for Cholesky decomposition on heterogeneous systems with GPUs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2016, pp. 993–1002.

[28] K. Zhao *et al.*, "FT-CNN: Algorithm-based fault tolerance for convolutional neural networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 7, pp. 1677–1689, Jul. 2021.

[29] T. Marty, T. Yuki, and S. Derrien, "Safe overclocking for CNN accelerators through algorithm-level error detection," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 12, pp. 4777–4790, Dec. 2020.

[30] S. J. Piestrak and P. Patronik, "Design of fault-secure transposed FIR filters protected using residue codes," in *Proc. 17th Euromicro Conf. Digit. Syst. Design*, Aug. 2014, pp. 575–582.

[31] S. K. S. Hari, M. Sullivan, T. Tsai, and S. W. Keckler, "Making convolutions resilient via algorithm-based error detection techniques," *IEEE Trans. Dependable Secure Comput.*, early access, Mar. 2, 2021, doi: 10.1109/TDSC.2021.3063083.

[32] R. Gonzalez and R. Woods, *Digital Image Processing*, 3rd ed. London, U.K.: Pearson, 2007.

[33] R. Ye, F. Liu, and L. Zhang, "3D depthwise convolution: Reducing model parameters in 3D vision tasks," in *Proc. Can. AI, Adv. Artif. Intell.* Cham, Switzerland: Springer, 2019, pp. 186–199.

[34] I. Bayraktaroglu and A. Orailoglu, "Concurrent test for digital linear systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits*, vol. 20, no. 9, pp. 1132–1142, Sep. 2001.

[35] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Int. Conf. Learn. Represent.*, 2015, pp. 1–14.

[36] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2012, pp. 1097–1105.

[37] C. Kong and S. Lucey, "Take it in your stride: Do we need striding in CNNs?" 2017, *arXiv:1712.02502*. [Online]. Available: http://arxiv.org/abs/1712.02502

[38] J. Pan and D. Chen, "Accelerate non-unit stride convolutions with Winograd algorithms," in *Proc. Asia South Pacific Design Automat. Conf. (ASPDAC)*, 2021, pp. 358–364.

[39] A. Shawahna, S. M. Sait, and A. El-Maleh, "FPGA-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2019.

[40] G. Stitt, A. Gupta, M. N. Emas, D. Wilson, and A. Baylis, "Scalable window generation for the Intel Broadwell+Arria 10 and high-bandwidth FPGA systems," in *Proc. Int. Symp. Field-Program. Gate Arrays*, Feb. 2018, pp. 173–182.

Dionysios Filippas received the Diploma degree in electrical and computer engineering and the M.Sc. degree in computer engineering from Democritus University of Thrace, Xanthi, Greece, in 2019 and 2021, respectively, where he is currently working toward the Ph.D. degree.

His research interests include energy-efficient machine-learning accelerators, high-level synthesis, and fault-tolerant systems.

Nikolaos Margomenos received the Diploma degree in electrical and computer engineering from Democritus University of Thrace, Xanthi, Greece, in 2021.

His research interests include the design of energy-efficient programmable data-parallel accelerators and high-level synthesis design flows.

Nikolaos Mitianoudis (Senior Member, IEEE) received the Diploma degree in electronic and computer engineering from the Aristotle University of Thessaloniki, Thessaloniki, Greece in 1998, the M.Sc. degree in communications and signal processing from Imperial College London, London, U.K., in 2000, and the Ph.D. degree in audio source separation using independent component analysis from Queen Mary, University of London, London, U.K., in 2004.

Between 2003 and 2009, he was a Research Associate at Imperial College London working on the Data Information Fusion-Defense Technology Center project "Applied Multidimensional Fusion," sponsored by General Dynamics U.K. and QinetiQ. From 2009 until 2010, he was an Academic Assistant at the International Hellenic University, Thessaloniki. Since 2010, he has been with the Electrical and Computer Engineering Department, Democritus University of Thrace, Xanthi, Greece, where he currently serves as an Associate Professor of Audio and Image Processing. He also serves as an Associate Editor for IEEE TRANSACTIONS ON IMAGE PROCESSING (2018–2024) and at *MDPI Journal of Imaging*. His research interests include machine learning, deep learning, computer vision, music information retrieval, and blind source separation/extraction.

Chrysostomos Nicopoulos (Member, IEEE) received the B.S. and Ph.D. degrees in electrical engineering with a specialization in computer engineering from Pennsylvania State University, State College, PA, USA, in 2003 and 2007, respectively.

He is currently an Associate Professor with the Department of Electrical and Computer Engineering, University of Cyprus, Nicosia, Cyprus. His current research interests include networks on chip, computer architecture, multicore/many-core microprocessor, and computer system design.

Giorgos Dimitrakopoulos received the B.S., M.Sc., and Ph.D. degrees in computer engineering from the University of Patras, Patras, Greece, in 2001, 2003, and 2007, respectively.

He is currently an Associate Professor with the Department of Electrical and Computer Engineering, Democritus University of Thrace, Xanthi, Greece. He is interested in the design of digital integrated circuits, energy-efficient data-parallel accelerators, functional safety architectures, and the use of high-level synthesis for agile ASIC and FPGA design flows.

Dr. Dimitrakopoulos received two Best Paper Awards at the Design Automation and Test in Europe (DATE) Conference in 2015 and 2019, respectively. Also, he received the HIPEAC Technology Transfer Award in 2015.