Safe Overclocking for CNN Accelerators through Algorithm-Level Error Detection

Thibaut Marty, Tomofumi Yuki, Steven Derrien

Safe Overclocking for CNN Accelerators through Algorithm-Level Error Detection

Thibaut Marty, Tomofumi Yuki, Steven Derrien

***Abstract*—In this paper, we propose a technique for improving the efficiency of CNN hardware accelerators based on timing speculation (overclocking) and fault tolerance. We augment the accelerator with a lightweight error detection mechanism to protect against timing errors in convolution layers, enabling aggressive timing speculation. The error detection mechanism we have developed works at the algorithm-level, utilizing algebraic properties of the computation, allowing the full implementation to be realized using High-Level Synthesis tools. Our prototype on ZC706 demonstrated up to 60% higher throughput with negligible area overhead for various wordlength implementations.**

I. Introduction

TIMING speculation, also known as overclocking, is a well known approach to increase the computational throughput of programmable processors and hardware accelerators. When used aggressively, timing speculation can lead to incorrect results due to timing anomalies that typically occur within long combinational paths. As reported in the literature [1]–[3], timing errors can cause large numerical errors in the computation. Although many applications are robust to low amplitude errors (e.g., errors due to quantization), occasional large errors can have devastating effect even for such applications.

The frequency of such errors depends on a number of factors, including the intensity of overclocking, operating temperature, voltage drops, variability within and across boards, input data, and so on. This makes it extremely difficult to determine a "safe" overclocking speed analytically or empirically as acknowledged in prior work. This has led to the use of online monitoring to avoid having excessive slacks based on static timing analysis [4], [5].

In this work, we propose to combine timing speculation with algorithm-level error detection to make overclocking a viable option for Convolutional Neural Networks (CNNs). CNN is a variant of multi-layered neural networks that constructs features from local information through convolutions. CNN models used in state-of-the-art applications are computation intensive and often need to be accelerated on GPUs or FPGAs to achieve high performance and/or to reach better energy efficiency [6], [7]. The core computations of CNNs have abundant parallelism, both task-level and fine-grained, that can be efficiently mapped to these accelerators. Furthermore, CNNs are known to be tolerant to noise. The reasons above make CNN an interesting class of computations to target.

Online error detection is necessary to prevent errors from affecting the final output, and it must be lightweight so that the gains by overclocking are not nullified. Although many error detection techniques exist, they are implemented at the circuit-level and are tied to the exact design at hand [4], [5], [8]. This limits the ease of design space exploration for a given algorithm, especially in the context of High-Level Synthesis (HLS) where these techniques cannot be directly expressed.

We therefore propose a higher level error detection scheme similar to Algorithm-Based Fault Tolerance (ABFT) [9], which is used in High Performance Computing as a lightweight protection from both soft and hard errors [10], [11].

Specifically, our contributions are the following:

- A lightweight error detection for convolution layers based on algorithmic properties. The developed method gives two-degree savings in complexity with respect to the full convolution layer.
- An open-source prototype implementation[1] of a timing speculative CNN accelerator based on the proposed error detection algorithm. This FPGA prototype is fully implemented in C++ with HLS tools and targets Zynq based systems.
- An analysis on how our approach fits into the CNN accelerator design space, and how timing speculation improves throughput in spite of the misspeculation cost.
- An experimental validation of the approach showing significant throughput and energy efficiency improvements with negligible area overhead.

The remainder of this paper is organized as follows. We provide background on timing speculation techniques and CNNs in Section II. We describe our proposed approach with lightweight error detection in Sections III and IV. We demonstrate the approach with a prototype implementation in Section V, and then discuss related work in Section VI. We conclude and give directions for future work in Section VII.

II. Background

In this section, we introduce convolutional neural networks and discuss existing implementation strategies for CNN accelerators. We then explain timing speculation and show the need for online error detection by empirical observation of its impact on application-level accuracy.

A. Convolutional Neural Networks

In this paper, we are interested in the forward pass of CNNs, i.e., when a trained network is applied to new inputs. Typically, a forward pass of CNNs processes three-dimensional matrices in a pipelined manner through multiple *layers*. The input is usually an image (with color depth), and the final output is a

[1] https://github.com/ThibautMarty/conv-hls-overclocking/

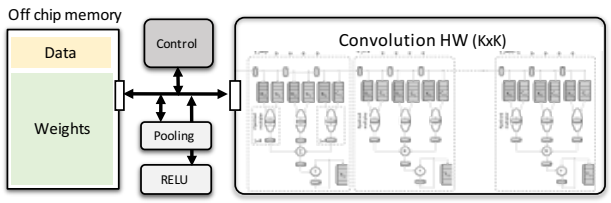


Fig. 1. Single Computation Engine

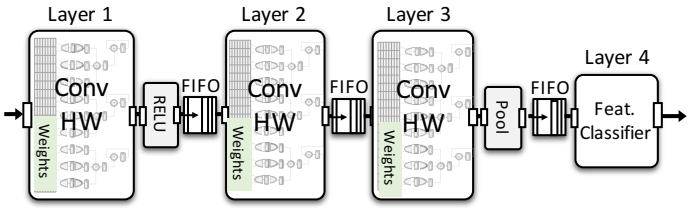


Fig. 2. Streaming Accelerator

vector of length M indicating the likelihood of each category, which can be viewed as a $1 \times 1 \times M$ matrix.

We are interested in the convolution layer of CNNs, which is known to be the main bottleneck. Given a $P \times Q \times N$ input matrix x, it computes a $R \times C \times M$ output y. Each of the M two-dimensional outputs are computed as a three-dimensional convolution over x with a kernel (weights) of size $K \times K \times N$. The convolution may be strided by some factor S. The R and C dimensions of the output matrix is usually smaller than the input, depending on the stride and padding at boundaries. Different layers take different values of the parameters described; the output $R \times C \times M$, called feature maps, can be viewed as an "image" where the depth is the extracted features.

Given a $P \times Q \times N$ input matrix x and $K \times K \times N \times M$ matrix holding the weights w, a convolution layer outputs a $R \times C \times M$ matrix y through the following equation:

$$y_{r,c,m} = \sum_{n=0}^{N-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} x_{n,Sr+i,Sc+j} \cdot w_{m,n,i,j} \qquad (1)$$

Fully-Connected layers are usually employed as the last stages of the pipeline where the size of the input matrix has become significantly smaller than the original image. They perform a collection of dot products with weights and produce a one-dimensional vector as outputs. They are identical to the hidden layers in regular artificial neural networks and can be computed as matrix-matrix products.

Additionally, activation functions (e.g., rectifier) or sub-sampling (e.g., max-pooling) may be considered as layers in CNN. Since these layers are inexpensive and can be merged with the preceding layer, we do not discuss them in this paper.

State-of-the-art CNN model are known to be computationally demanding (a single run of a VGG16 model requires more than 30 G operations), with more than 90% of the computational workload spent on the convolutions layers. Our approach therefore aims at accelerating the convolution layer.

B. Accelerating CNNs with FPGA

There has been numerous work on accelerating CNNs on FPGAs and other platforms. In this section, we briefly describe the most common system-level architectures, which can be classified into three families [12].

- **Single Computation Engine (SCE)** architecture, depicted in Figure 1, offloads convolution kernels to the accelerator, and processes layers in a sequential manner. SCEs generally support all types of convolution layers, and can also execute RELU or max-pooling layers. This accelerator style typically employs a form of decoupled access/execute model to hide memory access latency. Whenever a convolution is to be executed by the SCE, both its inputs and weights are fetched from external memory. This often results in memory bandwidth bottlenecks. To alleviate this issue, it is possible to take advantage of batching, where a given layer configuration is reused to process several data-sets. However, batching improves overall throughput at the cost of latency.
- **Streaming Accelerators (SA)**, as shown in Figure 2, use multiple convolution accelerator instances, typically one for each layer, that are pipelined to provide high throughput. In this approach, each convolution accelerator must store all its weights on the chip, which can be challenging given the relative size of CNN models. Another difficulty is to balance processing resources among layers to maximize the overall throughput.
- **Fused Accelerators (FA)** use a combination of previous approaches, where the hardware accelerator executes a sequence of convolution layers. Such approach offers a good trade-off between on-chip memory cost, off-chip memory bandwidth requirements, and inference latency.

Our approach augments the convolution hardware depicted in Figures 1 and 2 with error detection to enable safe overclocking. The implementation of the convolution hardware is discussed in Section IV. Our approach fits all the architectures described above; our only requirement is that the inputs/outputs of the convolution accelerator is streamed through FIFOs.

C. Timing Speculation through Overclocking in FPGAs

The minimum clock period at which a given FPGA design is expected to work is obtained from a Static Timing Analysis (STA). This analysis assumes the worst case scenario, and hence the design may be operated on higher operating frequencies without observing incorrect behaviors. However, this technique, widely known as *overclocking* or *timing speculation*, also has many pitfalls:

- Variability among chips and operating conditions makes it difficult to determine how much overclocking can be tolerated. The fact that errors do not manifest often (or the inability to observe errors in a given setup) does not mean that errors do not happen.
- The impact of timing errors on the circuit output is difficult to evaluate a priori. Unlike errors arising from truncation/quantization, the impact is not limited to Least

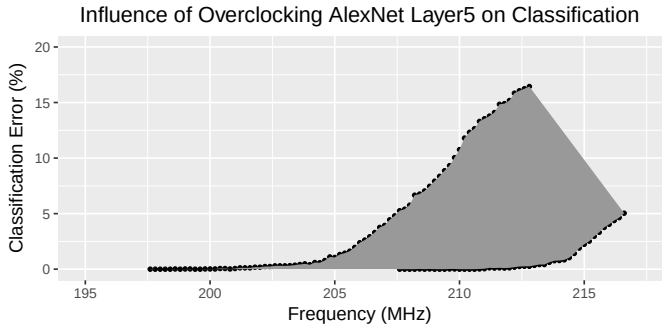


Fig. 3. Impact of overclocking on classification; classification error is when the output class differs from that of error-free execution, regardless of the ground-truth class. The data points are the highest/lowest error rate observed for the given frequency, and the shaded regions show the range of possible error rates influenced by various sources of variability.

Significant Bits (LSBs). Thus, it may result in large numerical errors, compromising the design functionality.

D. Impact of Overclocking on Classification Accuracy

We performed a series of experiments to asses the impact of overclocking on final outputs. In this section, we report some of the results obtained from classifying 5000 images (the first $10\,\%$ of ILSVRC 2012 validation dataset) using AlexNet. We used our accelerator prototype, 16 bits fixed-point design, to execute the last convolutional layer. The last convolutional layer is selected for these experiments because the last layers are known to be more sensitive to errors than the first layers. We obtained the input data for the accelerator by quantizing the previous layer's error-free floating-point output in order to isolate effect on accuracy. Using low precision fixed-point requires special attention to training and quantization to avoid accuracy loss, which is beyond the scope of this work.

Figure 3 depicts the impact of overclocking on classification accuracy. We have performed this experimentation over a set of Zybo platforms to capture inter-board variability, using different data-sets. Our results show that there are significant variation when the accuracy degradation starts to happen (note that we obtained similar results for ZC706 system).

It is acknowledged by prior work that a "safe" frequency is impossible to determine statically due to multiple sources of variability [5]. For example, when considering only a subset of variability (inter-board variability), there is as wide as $10\,\mathrm{MHz}$ gap in the frequency where accuracy starts to drop. In other words, it is difficult to ensure that the overclocked accelerators do not significantly alter the application outputs.

E. Online Error Detection for Convolution

The combination of variability and high impact on outputs makes overclocking impractical without some form of online error detection/correction. This has led to circuit-level techniques for dynamically checking for incorrect behaviors, including the Razor technique [4], [5], [13], [14]. In particular, Nunez-Yanez [8] has used this technique in the context of timing-speculative Binary Neural Network accelerators. These circuit-level approaches do not support designs with DSP blocks, and often need semi-manual design steps to be implemented on FPGAs (see Section VI for further discussion).

III. Algorithm-Level Error Detection

In this section, we describe our lightweight error detection for convolutions. Our approach is at the algorithm-level, in contrast to circuit (register) level techniques.

A. Overview of the Approach

Our approach builds on a *lightweight* mechanism for detecting errors based on algorithmic properties. This approach shares some ideas with earlier work on Algorithm-Based Fault Tolerance (ABFT) for matrix operations [9].

This error detection mechanism uses two checksums, one computed from the outputs, and another computed directly from the inputs, which we refer to as *output-checksum* and *input-checksum*, respectively. We flag the computation as erroneous whenever the checksums do not match.

The key intuition is that the computation of the input-checksum can be simplified by taking advantage of algebraic properties, leading to a low-complexity error detector.

B. ABFT for 2D Convolutions

The 2D case is missing the depth dimension in the processed matrices, but the main ideas carry over to the 3D case. Given the 2D output y, the output-checksum is:

$$\sigma = \sum_{r=0}^{R} \sum_{c=0}^{C} y_{r,c}$$

Substituting the definition of y (Eq. 1 without m, n and $S = 1$) gives the direct computation from the inputs:

$$\rho = \sum_{r=0}^{R} \sum_{c=0}^{C} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} (x_{r+i,c+j} \cdot w_{i,j})$$

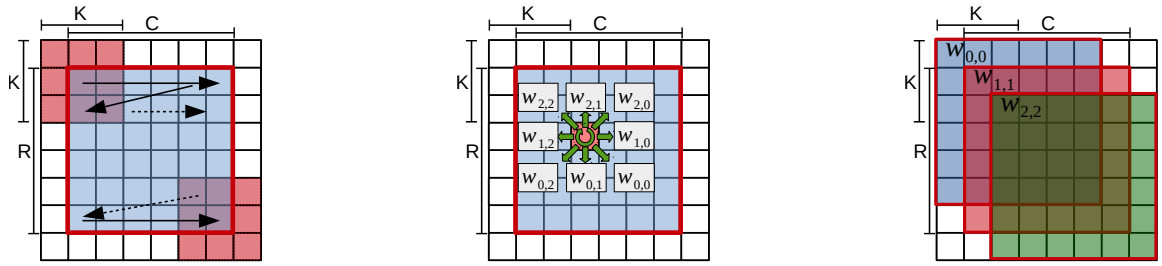
The goal is to simplify the computation of ρ so that the checksum comparison can be performed with reduced cost.

The additional two-dimensional summation provides two sources for simplification. The combination of the two simplifications described below reduces the cost of computing the input-checksum for 2D case from $O(RCK^2)$ to K^2 multiplications and $O(RC)$ additions.

1) Factorization: Multiplications can be factored out to eliminate RC multiplications. This may be viewed as a reordering of the summations followed by factorization:

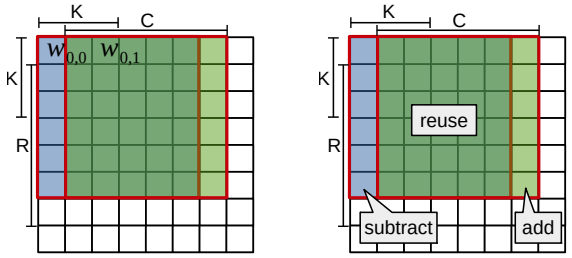
$$\rho = \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} w_{i,j} \left(\sum_{r=0}^{R} \sum_{c=0}^{C} x_{r+i,c+j} \right)$$

A graphical illustration is given in Figure 4. This reduces the number of multiplications from K^2RC to K^2.



(a) 2D convolution without zero-padding (b) Contribution of an input by weights (c) Input groups by weights

Fig. 4. Illustration of the factorization for 2D case. As depicted in Figure 4a, 2D convolution can be viewed as a dot product between the weights and the neighboring inputs with a sliding window. An alternative view shown in Figure 4b is that an input value is used to compute K^2 output values, contributing to each of them through multiplication by different weights. We can thus group the input data into (overlapping) subsets based on the weights, which is what is shown for three weight values in Figure 4c. Since sum of convolution outputs is completely linear, the multiplication can be factorized to save work.



(a) Two neighboring groups (b) Reuse pattern

Fig. 5. The reuse between two input groups corresponding to weights $w_{0,0}$ and $w_{0,1}$. The sum of all elements in group $w_{0,1}$ can be computed by addition/subtraction of columns from that of $w_{0,0}$, instead of repeating $R \times C$ additions.

2) Reuse in Summations: The groups of summations after factorization have significant overlaps, which can be used to reduce the number of additions. This simplification concerns the computation of the inner two summations, or *input groups*:

$$X_{i,j} = \sum_{r=0}^{R} \sum_{c=0}^{C} x_{r+i,c+j}$$

Note that each input group, $X_{i,j}$, is a summation over slightly different regions of x due to the offsets by i and j. These values of X takes K^2RC additions when computed naïvely. However, once a value of X for a specific instance of i, j is computed, the remaining instances can be computed by only $O(C)$ or $O(R)$ additions as explained in Figure 5.

There are multiple ways to rewrite the definition of X to take advantage of this reuse. One example is as follows:

$$X_{i,j} = \begin{cases} \sum_{r=0}^{R} \sum_{c=0}^{C} x_{r,c} & : \begin{matrix} i = 0 \\ j = 0 \end{matrix} \\ X_{i,j-1} + \sum_{r=0}^{R} x_{r+i,C-1+j} - \sum_{r=0}^{R} x_{r+i,j-1} & : \begin{matrix} i = 0 \\ j > 0 \end{matrix} \\ X_{i-1,j} + \sum_{c=0}^{C} x_{R-1+i,c+j} - \sum_{c=0}^{C} x_{i-1,c+j} & : i > 0 \end{cases}$$

In the above, the $R \times C$ summation for $X_{0,0}$ is performed first. Then the remaining instances of $X_{i,j}$ are computed by addition and subtraction of one row/column. The $R \times C$ summation is not repeated for each $X_{i,j}$ ($K \times K$ instances) reducing the complexity by $O(K^2)$ in exchange for $2R$ or $2C$ additions.

C. Lightweight Checksum for 3D Convolution Layers

For the 3D case, the output y has the third dimension corresponding to the different kernels applied to the input. The checksum is over all three dimensions of the output:

$$\sigma = \sum_{m=0}^{M-1} \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} y_{m,r,c} \tag{2}$$

Substituting Eq. 1 (again, assuming unit stride) gives the direct checksum computation from the inputs:

$$\rho = \sum_{m=0}^{M-1} \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} \left(\sum_{n=0}^{N-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} x_{n,r+i,c+j} \cdot w_{m,n,i,j} \right)$$

Reordering of the summations permits the factorization of the multiplication by weights:

$$\rho = \sum_{n=0}^{N-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} \left(\left(\sum_{r=0}^{R-1} \sum_{c=0}^{C-1} x_{n,r+i,c+j} \right) \cdot \sum_{m=0}^{M-1} w_{m,n,i,j} \right)$$

Note that in the 3D case, the different convolution kernels applied to the same input (the m dimension) can be first added together, since m is invariant to the expression involving x.

Taking advantage of the reuse in summations yields:

$$\rho = \sum_{n=0}^{N-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} \left(X_{n,i,j} \cdot \sum_{m=0}^{M-1} w_{m,n,i,j} \right) \tag{3}$$

where X takes the same form as the 2D case, except for the additional n dimension, which does not have any reuse.

D. Managing Strided Convolutions

Some CNNs use strided convolution in their first layers, especially for large convolution kernels. The striding factor is usually proportional to the kernel size. For example, the typical stride is 2 for 5×5 kernels, whereas a striding factor of 4 is used for 11×11 kernels. In the following, we show how our checksums support strides, at the price of a slight increase in algorithmic complexity.

Strided convolution adds a parameter S as shown in Eq. 1. This changes the convolution behavior: the dot product is now applied on a sliding window over the input with a step of S in both directions. This further reduces the size of the outputs and the number of multiply-accumulate operators by a factor of S^2.

Non-unit strides introduce sparsity to how each input is multiplied by the weights. All weights are not used for each input value, but only for one out of S, on the two dimensions. This results in S^2 partitions of inputs and weights, where each partition may be viewed as a separate non-strided convolution using subsets of inputs and weights sampling every S elements in both dimensions. The factorization of multiplications when computing the input-checksum is unaffected by strides, since the total number of input groups remains the same; each partition contributes to disjoint subsets of the input groups. Then, the input-checksum can be computed with Eq. 3 with a strided definition of X:

$$X_{n,i,j} = \sum_{r=0}^{R} \sum_{c=0}^{C} x_{n,Sr+i,Sc+j}$$

The reuse in summations now happens within each partition when the inputs are used for multiple weights, that is, when $S < K$. Exploiting the reuse in each partition gives:

$$X_{n,i,j} = \begin{cases} \sum_{r=0}^{R} \sum_{c=0}^{C} x_{n,Sr,Sc} & : i < S \\ & : j < S \\ X_{n,i,j-S} + \sum_{r=0}^{R} x_{n,Sr+i,S(C-1)+j} & : i < S \\ \quad - \sum_{r=0}^{R} x_{n,Sr+i,j-S} & : j \geq S \\ X_{n,i-S,j} + \sum_{c=0}^{C} x_{n,S(R-1)+i,Sc+j} & : i \geq S \\ \quad - \sum_{c=0}^{C} x_{n,i-S,Sc+j} & \end{cases}$$

There are two differences from the non-strided case: it requires separate initial computations for each partition (S^2 total) and the difference computations are strided. Note that $S = 1$ is simply a special case of the above.

The computation of output-checksum remains the same as the non-strided case.

We note that layers with non-unit strides typically have larger values of R, C, and K increasing the reuse. For AlexNet, the first layer uses $S = 4, R = C = 55$, and $K = 11$, which corresponds to a factor of 189 savings on additions and a factor of 8 savings on multiplications. These savings are similar to those for the last layers with $S = 1, R = C = 13$, and $K = 11$, which corresponds to savings on additions and multiplications with a factor of 169 and 9, respectively. Thus, we expect similar levels of overhead for strided and non-strided layers. Furthermore, the calculation of X for each partition is independent, and may be parallelized if latency becomes an issue, at the cost of extra hardware resources.

TABLE I
COMPLEXITY OF CHECKSUMS VS. CONVOLUTION.

	additions	multiplications
convolution	$O(MNRCK^2)$	$O(MNRCK^2)$
output-checksum	$O(MRC)$	0
input-checksum	$O\left(NK^2(M+C) + RCS^2\right)$	$O(NK^2)$

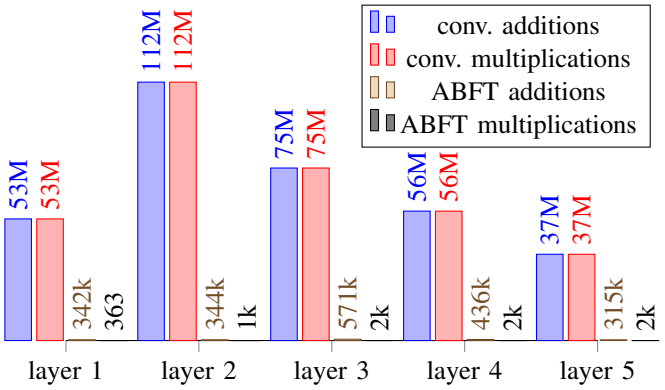


Fig. 6. Number of additions and multiplications for convolution and ABFT checksums for AlexNet [7] convolutional layers. Layer 1 has $S = 4$, other layers have $S = 1$.

E. Max Pooling and RELU Layers

Other layers such as pooling or rectified linear unit are not compute-intensive. These layers may simply be executed without overclocking to ensure that no errors are introduced.

F. Checksum Overhead Analysis

Table I shows the complexity (in terms of number of arithmetic operations: additions and multiplications) of both the checksums and the convolution layer. Note that $R \times C$ is the size of output images. Input images dimensions are $(K + S(R-1)) \times (K + S(C-1))$. The complexity reduction translates into differences of several orders of magnitude in number of arithmetic operations as exemplified in Figure 6. The checkums require much fewer arithmetic operations compared to the convolution kernel for AlexNet convolution layers.

G. Correctness of Error Detection

In some cases, our checksum-based mechanism may lead to a false positive (detecting an error whereas no error occurred) or a false negative (not detecting an actual error), like any error detection mechanism.

A timing error can impact any computation in the circuit and the erroneous results will be stored in registers. The erroneous values will in turn impact the convolution output and/or the checksums. Therefore, there are several scenarios depending on the number of timing errors and their impact.

The scenarios are summarized in Table II. The two basic cases are highlighted: when no error occur or when an error in convolution layer occurs. The algebraic properties guarantee that a single error in the convolution output is detected. In other cases, the result may be a false positive or a false negative. A false positive only affects performance: the tile is

TABLE II
ERROR DETECTION OUTCOME FOR ALL SCENARIOS: NO ERROR, SINGLE ERROR OR MULTIPLE ERRORS IN THE CHECKSUMS AND/OR THE CONVOLUTION OUTPUT. EACH OUTCOME IS LABELED AS ERROR-FREE EXECUTION, FALSE POSITIVE (FP), OR FALSE NEGATIVE (FN).

		Errors in convolution layer		
		none	single	multiple
Errors in checksums	none	**error-free**	**100% detection**	possible FN
	single	100% FP	possible FN	
	multiple	possible FP		

TABLE III
DATA AND CHECKSUM WORDLENGTHS USED IN EXPERIMENTAL RESULTS (SEE SECTION V). THIS LAYER'S DIMENSIONS ARE $N = 192$, $M = 128$, $R = C = 13$, AND $S = 1$, BUT THE CHECKSUMS ARE COMPUTED ON TILES. THE DESIGNS USE TILES OF SIZE 32 FOR THE N DIMENSION AND 64 FOR THE M DIMENSION.

Input data	Weight data	K	Checksums
16 bits	16 bits	3	55 bits
16 bits	8 bits	3	47 bits
16 bits	4 bits	3	43 bits
16 bits	2 bits	3	41 bits
16 bits	1 bits	3	40 bits
8 bits	8 bits	3	39 bits
4 bits	4 bits	3	31 bits
1 bits	1 bits	3	25 bits
16 bits	8 bits	5	48 bits
16 bits	8 bits	11	50 bits

computed again even though it was correct. A false negative is problematic, as the errors are not detected. We have empirically confirmed that the false negatives are rare in Section V-E. In the following, we give an intuition for the low rate of false negatives using a simplistic model of error.

Although they are unlikely, false negatives are still possible when multiple errors occur. The errors during the convolution layer affect the output-checksum. Several errors can cancel each other if their effects on the checksum are opposite. Intuitively, the probability of such cancellation to happen is affected by the precision (wordlength) used for the checksum calculation. Since the checksum is computed with full precision, its wordlength b depends on the input data types and the number of accumulations. For a full convolution checksum, we have $b = D + W + \lceil log_2(NK^2) \rceil + \lceil log_2(RCM) \rceil$, where D is the input data wordlength, W is the weight data wordlength and N, K, R, C, and M are the convolution parameters as defined in Section II-A.

Table III shows wordlengths of checksums for designs used in our evaluation. The wider the checksum's wordlength, the lower the chances to get a false negative. Assuming a simple model where an error ultimately leads to a single bit flip in the checksum with uniform probability, the chances that more than one error cancel each other is $\frac{1}{2^b}$.

Errors can also occur in the checksums computation circuit. If errors occur only in checksums computation, a false positive is highly probable (except if errors cancel each other). If errors occur in both the convolution and the output-checksum computation, a false negative is possible, as explained above. However, the convolution kernel circuit is expected to be much less sensitive than the checksums circuit to timing errors. This is because the hardware implementation of checksums are much simpler compared to the convolution implementation due to lower levels of parallelism. Therefore, we expect the convolution computation to be impacted by timing errors at a lower frequency than the checksums computation.

IV. IMPLEMENTATION

In this section, we describe how we designed the fault-tolerant convolution accelerator for FPGA and give an overview on how the errors and frequency can be managed.

A. Baseline Accelerator Design

Our accelerator kernel follows an implementation strategy proposed by Zhang et al. [6], which is described in Figure 7. Note that we do not claim that this is the optimal convolution accelerator implementation. Finding the best design instance often require extensive exploration of a large design space involving loop transformations and data layout optimization [15], and this problem is completely orthogonal to our approach. One important notion in their work that we also use is the *tiles*. The CNN computations are largely data parallel, and can be decomposed into smaller chunks to reduce on-chip memory cost. The granularity of these chunks is usually selected such that the accelerator fits on the target board. We use the word *tile* to refer to the unit of computation performed by the accelerator.

B. Accelerator with Error Detection

Our modified architecture is depicted in Figure 8. Two additional modules that compute the checksums are added between the convolution kernel and the FIFOs. The important property is that the checksums are computed in a streaming fashion such that the latency of the accelerator is unaffected. The entire accelerator sits within its own clock domain managed by software through dynamic reconfiguration of the Mixed-Mode Clock Manager (MMCM) module.

To preserve the overall performance of the accelerator, both input- and output-checksum components are designed to be able to sustain the same I/O throughput as the convolution accelerator. In our current implementation, we support up to one input data per cycle, but higher throughput is possible by exposing more parallelism in the hardware component, which can be easily achieved in HLS through an unroll directive.

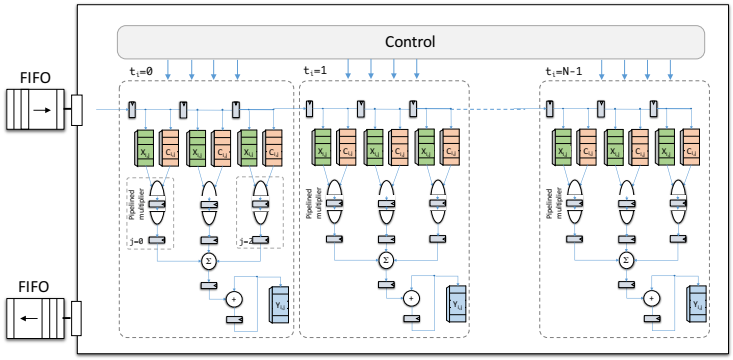
In our proposed design, the entire accelerator operate in the overclocked domain. We made this choice to simplify the design of the checksum components; more resources would have been needed to compensate for the clock speed gap.

C. Implementation of Checksum Calculations

The output-checksum is implemented as a simple accumulator over convolution outputs, with a small area overhead compared to the rest of the datapath. The hardware component responsible for computing the input-checksum (Eq. 3) is more

```
for(to = 0; to < M ; to++) {
  for(row = 0; row < R ; row++) {
    for(col = 0; col < C ; col++) {
      #pragma pipeline
      for(i = 0; i < K ; i++) {
        #pragma unroll
        for(ti = 0; ti < N ; ti++) {
          #pragma unroll
          for(j = 0; j < K ; j++) {
            y[to][row][col] +=
                w[to][ti][i][j] *
                x[ti][S*row+i][S*col+j];
          }
        }
      }
    }
}}}
```

Convolution HLS input code



Convolution accelerator datapath

Fig. 7. The convolution kernel based on the design by Zhang et al. [6]. The convolution kernel has three sources of parallelism. The main convolution has ample parallelism, which is used as the fine-grained parallelism (unrolling in HLS). This datapath is also aggressively pipelined to process different inputs. Furthermore, this datapath itself is replicated for convolutions by different weights to the same input. The factor of unrolling and/or replication controls the throughput of the accelerator.

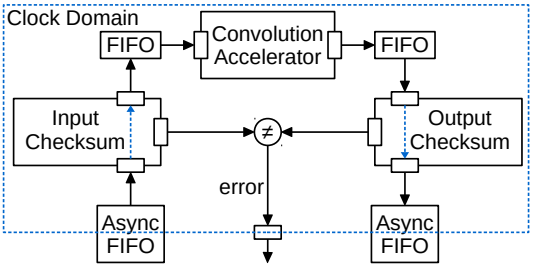


Fig. 8. Accelerator with Error Detection.

costly, as it involves a multiplier, storage for partial sums, and non-trivial control logic.

Both of these components are significantly less complex compared to the main kernel. The main performance constraint is to ensure that the data processing rate matches that of the convolution kernel. The checksum calculations need sufficient parallelism to keep up with the rate of input consumption as well as the rate of output production. The parallelism in these calculations are realized as aggressive pipelining ($II = 1$) to minimize area cost.

The output-checksum, which is an accumulation, has a trivial parallelism to process the outputs. The computation is independent except for the aggregation at the end.

The input-checksum computes the input groups and the summation of weights over M as the data is streamed through. Then, the rest of the computation is overlapped with the convolution kernel: the computation of the input groups (X), multiplication by the summed weights, and the final accumulation. Note that the inputs memory transfer (and summations) for next tile is overlapped with the kernel, the rest of the input-checksum calculation, and the outputs memory transfer for the previous tile. This ensures that the input-checksum calculation does not impact the overall latency of the accelerator.

D. Integration with Complete CNN Accelerators

As mentioned in Section II-B, accelerator of the convolution kernel is only a component of CNN accelerators. In this section, we discuss how our convolution accelerator can be integrated with different styles of CNN accelerators.

In the case of single computation engine, using our approach is straightforward. The convolution accelerator operates within its specific clock domains, and asynchronous FIFOs are used to stream in and out data to and from the accelerator. As described in Section V, this architecture has been choosen for our prototype.

Taking advantage of our technique in streaming accelerator or fused accelerator architectures is not more difficult. Convolution layer can be mapped to different pieces of hardware, Thus, they may be overclocked at different frequencies, depending on the number of clock managers available on the devices (Virtex-7 have up to 24 of such components). In the case where the number of clock manager is not high enough, they may need to be clustered into a smaller subset of clock domains. Asynchronous FIFOs must be used for data transfer between clock domains.

Note that RELU and max-pool layers cannot benefit from ABFT, and thus they must operate either at a safe clock speed, or use alternative timing error detection techniques. Since these layers are not computational bottleneck, this does not impact the performance of the accelerator.

E. Failure Recovery Cost

The misspeculation overhead depends on the system-level architecture discussed in Section II-B. This section proposes an analytical model of the misspeculation cost.

For a single compute engine, an erroneous tile is simply fed back to the accelerator at a later time. The macro-pipeline does not have to be stalled because the tiles are parallel; they can be executed in any order.

For a streaming accelerator, the overhead depends on the details of the architecture. If there are buffers between layers, e.g., for crossing FPGA boundaries, then full pipeline stall can be avoided. In the worst case, the full pipeline must be restarted, discarding all intermediate results.

As a simplistic model, consider the following:

- each tile takes 1 unit of time with baseline frequency;

- failed tiles are executed at the baseline frequency;
- N: total number of tiles;
- O: mean overclocked frequency, normalized to the baseline;
- E: error rate, probability of a tile to fail;
- S: number of stages in SA.

Note that the time it takes to switch the frequency is negligible — less than $1\,\%$ of a tile computation in our prototype.

Then the total time T can be modeled as: $T = \frac{N}{O} + S \cdot N \cdot E$ where the SCE is a special case when $S = 1$.

The gain by overclocking is $N - \frac{N}{O}$, which should be lower than the overhead $S \cdot N \cdot E$ for overclocking to be profitable. Clearly, the error rate plays an important role, and we may derive the break-even point (overhead = gain) as: $E = \frac{O-1}{S}$

In other words, even for a small gain in frequency, completely negating its gain takes a high error rate. For instance, with $25\,\%$ overclocking on average with 5-stage SA architecture, it requires $5\,\%$ error rate to negate the gain. With a strict dynamic scaling policy that only attempts to increase frequency in long intervals once an error is detected, the error rate is extremely low ($< 0.1\,\%$), rendering the recovery cost negligible. In addition, we expect that the error-free execution is not necessary in most use cases.

In the case where the application can tolerate a few errors (e.g., missclassifications), in a non critical context for instance, a small number of erroneous tiles may be tolerated. In this specific case, the detected errors do not need to be corrected. Hence, there is no additional latency overhead with our approach. However, how to precisely find the tolerable amount of errors is use-case specific and is orthogonal to this work.

F. Frequency Scaling

This section discusses how to achieve high throughput given an accelerator with adjustable frequency and online error detection capability.

As discussed in Section IV-E, the error rate is the main factor that determines throughput. Figure 9 illustrates this behavior. It shows the normalized throughput (related to the non overclocked throughput) taking into account the misspeculation cost for a frequency range based on macro-pipeline depth and measured error rate. We have measured the error rate and the average processing time for 100,000 tiles computed from uniform random data per frequency with a step of $0.25\,\mathrm{MHz}$ around the frequency where errors start to occur. Then, we have calculated the effective throughput using the model described in Section IV-E. We can see that the resulting throughput is linear with the frequency, until errors occur. The macro-pipeline depth influences the throughput penalty but the main factor is the error rate.

As the environment (temperature, input data, etc) changes, the frequency at which the maximum throughput is achieved may change over time and can not be known in advance, as exemplified by the red arrows in the figure. Note that these results reflect only one instance of the experiment under specific operating conditions and the results may differ depending on the environment.

Therefore, there should be a dynamic mechanism that is able to select the suitable frequency, i.e., the highest one without errors, while keeping the error rate as low as possible. As the target may change over time, the dynamic mechanism needs to be used during the whole application execution, not only at the beginning.

We propose a simple algorithm to dynamically scale the frequency at runtime. The algorithm has two parameters:

- G: granularity of frequency adjusted at a time;
- I: interval; number of successful tiles before attempting to increasing frequency.

The algorithm initially increases the frequency by G after each tile until an error is observed. Then the algorithm repeatedly takes one of the following actions:

- decrease by G if an error is observed;
- increase by G after I successful tiles.

Parameters I and G expose a tradeoff between reactivity and error rate and they must be chosen by the user.

This algorithm is very simple, and we acknowledge that more sophisticated control algorithms could have been used. However, our results (see Section V-F) show that this approach performs well in our context.

V. Experimental results

We present our empirical study in this section. The study consist of two parts: (i) the area overhead, and (ii) frequency scaling on AlexNet image classification.

A. Experimental Platform

The whole accelerator was designed using SDSoC (2018.2), demonstrating the suitability of our approach to modern FPGA tools that operate at higher levels of abstractions. We use the ZC706 board (XC7Z045) from the Xilinx Zynq-7000 family as our target platform.

The hardware designs used in this section target the fifth convolutional layer in the AlexNet CNN architecture [7] with the following standard configuration: $N = 192$, $M = 128$, $R = C = 13$, $K = 3$, and $S = 1$. Preliminary experiences showed that this layer was the one that most impact classification rate when impacted by timing errors. Note that the third and fourth layers of AlexNet also have similar configurations with larger N and/or M.

B. Area Results

We report the results for various kernel size (K), and wordlengths. The wordlengths are denoted as $D \times W$, where D is the input data wordlength and W is the weight data wordlength. Other design parameters (tile sizes and unroll factors) were selected to achieve highest throughput on the target board with the largest wordlengths, and the same parameters were used for lower wordlength designs (except for $K = 11$ where an unroll factor has been divided by 2).

All designs were synthesized with several target frequencies in order to get the maximum achievable STA frequency. All reported numbers are after P&R.

The amount of hardware resources used by different hardware configurations are shown in Figure 10, from which we can make the following observations:

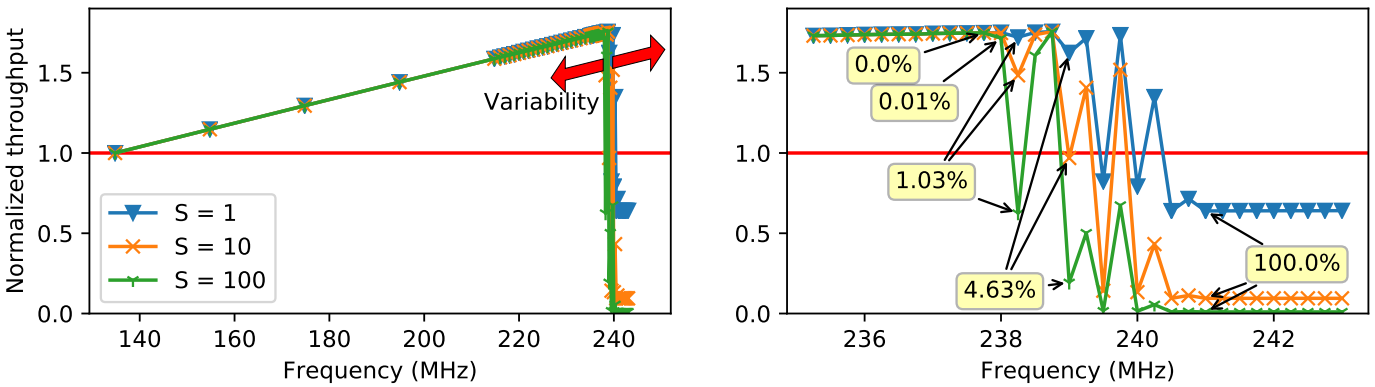


Fig. 9. Normalized throughput as a function of the frequency for three scenarios computed from the model introduced in Section IV-E using empirically collected error rate and processing time. The left plot shows the full frequency range. The red arrows illustrate variability: the frequency at which the maximum throughput is achieved cannot be known in advance and can change. The right plot shows the same data over a narrower frequency range where errors start to manifest. The labels correspond to error rates, ranging from 0 % to 100 %. At a given frequency, the error rate is the same for the three scenarios, the only difference being the misspeculation cost.

- DSP and BRAM cost, which often are the limiting resources for CNN implementation, is mostly unaffected by the ABFT mechanism.
- There is a small overhead in terms of slice count.

Overall, our results show that the overhead is negligible.

For the sake of completeness, we also compared our approach to error detection techniques based on Residue Numbering Systems (RNS) protection as proposed in previous work [16]. We considered a RNS protection using `mod` 3, `mod` 7, and `mod` 15 residue code protecting 16 bits data. Protection with `mod` 3 code is the RNS error detection strategy with the lowest possible area overhead, and also with the lowest error coverage.

Figure 10 shows the area overhead of RNS protection compared to the unprotected baseline. These results show that our ABFT approach has a significantly lower area footprint (1.3k slices vs 2.7k/8k/15.2k slices). This is despite the fact that ABFT provides much better error coverage than RNS. For example `mod` 3 RNS provides only 66% error coverage in the case of multiple bit-flips, even when these bit-flips are caused by a single timing error. In addition, RNS codes are much more fine-grained, and additional mechanisms are needed to protect the full accelerator in the same manner as our approach. For instance, errors in control-logic may cause a loop to be skipped, which would be undetected by typical uses of RNS codes if the check happens at the end of the loop.

C. Performance Results

The overclocked frequencies enabled by our approach are shown in Figure 11 for accelerators with various wordlengths and $K = 3$. For all data wordlengths except when $W = 1$ bit, results show a mean performance improvement of 68% with little difference across configurations. For $W = 1$ bit cases, the performance improvement is more modest (21.3% and 26.6%) and, for the 1×1 case, well below the average 50% improvements reported by Nunez-Yanez [8] with the Elongate framework. We believe that this is because our accelerator is designed to work for multiple wordlengths, and is not specialized for binary CNNs. In particular, we observed that for designs with 1 bit weights, errors manifest much earlier.

Figure 14a shows the performance of accelerators with and without overclocking; the mean improvement was 54%. The ABFT-enabled frequency would be the one eventually reached by a monitoring system in the same environment. The potential failure recovery penalty is not taken in account as it has been shown to be negligible in Section IV-E.

D. Observed Errors

We also took a detailed look at how timing errors affect the output of convolution layers and observed that LSBs have much higher chances to be corrupt in all frequency ranges, as shown in Figure 12. It may seem counter-intuitive, but it makes sense as the routing phase tries to balance paths. On FPGAs, routing has become an important portion of the delays, and it impacts the timing behavior of the circuit as well as the combinational paths. Furthermore, the dot product operations use DSPs that are blackboxes in which the timing behaviour can not be easily analyzed.

At frequency ranges where timing errors start to manifest, almost all errors are at the LSBs. Many of these errors are benign, since some of the LSBs are usually truncated to match the data representation used in the next CNN layer. This gives an opportunity to detect timing errors before they impact the final output with no penalty as the tile does not need to be recomputed. The convolution is computed in an exact manner, with the intermediate results stored with enough bits for the multiply-accumulate chain. The output-checksum is computed on untruncated output; otherwise the invariant would not hold due to rounding errors. Therefore, a timing error may be eventually masked by the truncation, resulting in correct, truncated outputs although the checksums do not match. When comparing the two checksums, it is possible to know if an error affected the truncated output by looking at which bits are incorrect. Assuming that only one error happened, and truncation of T LSBs, then the error did not affect the truncated output if the error in the checksum is located in the T LSBs. For such errors, recomputing the tile

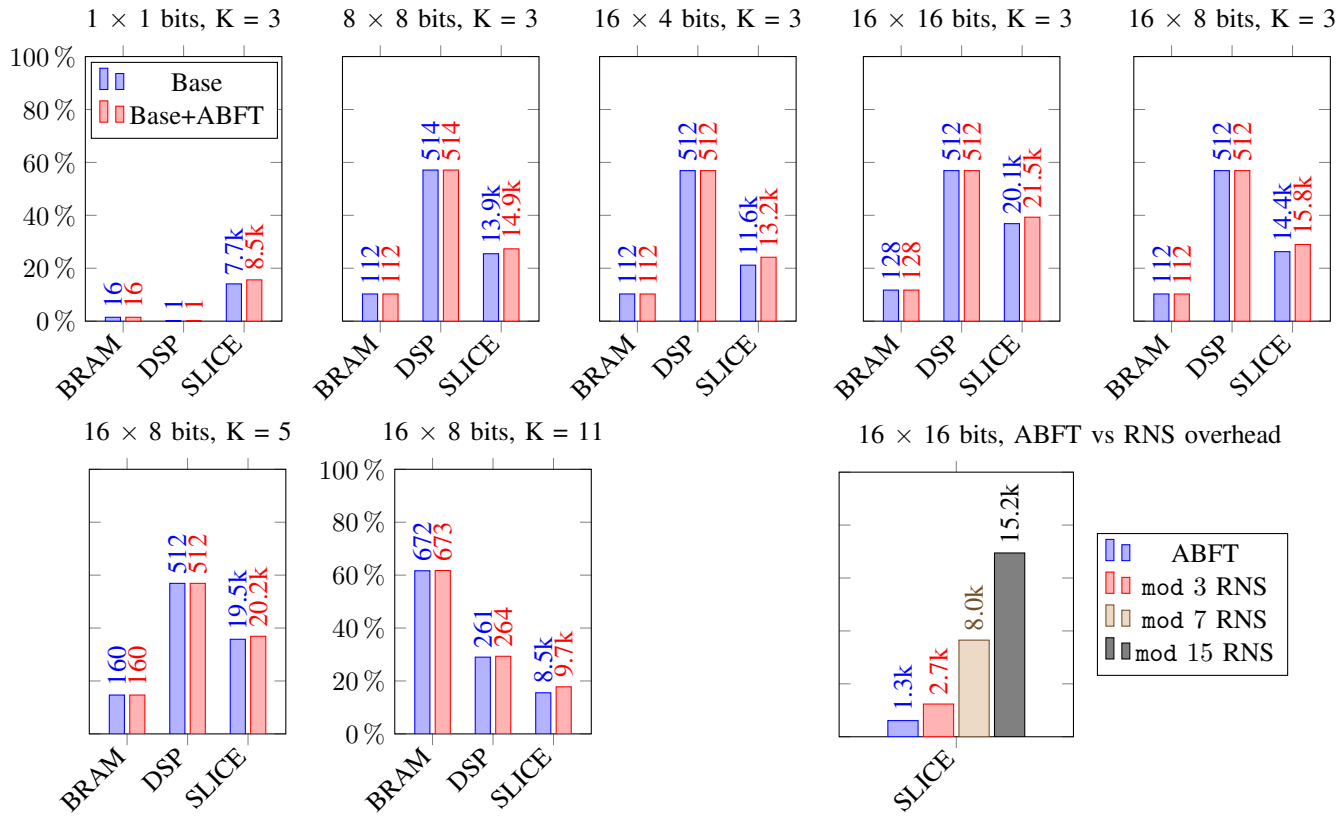


Fig. 10. Area results for ZC706 after P&R with and without error detection for different configurations.

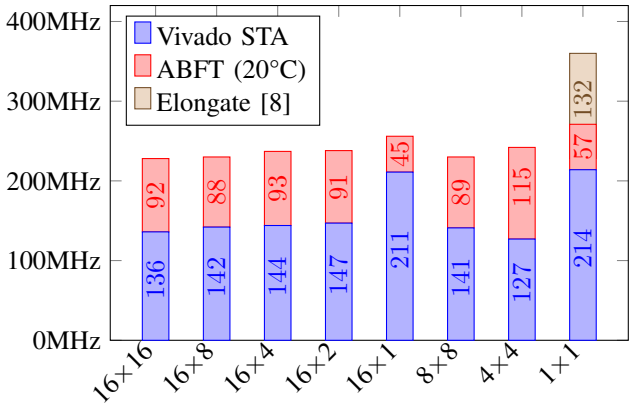


Fig. 11. ZC706 clock speed by Static Timing Analysis (STA), our proposed approach (ABFT) for a 0.1% tile error rate at room temperature, and the Elongate technique proposed by Nunez-Yanez [8].

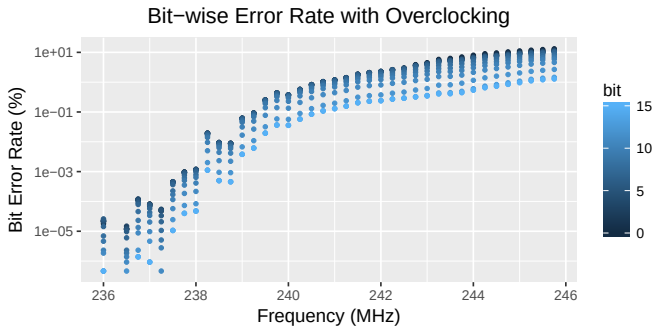


Fig. 12. Observed bit-wise error rate in the outputs with 16×16 bits design, with uniform random inputs.

is unnecessary, but this information can still be used to tune the frequency and avoid subsequent errors.

E. False Negatives

As discussed in Section III-G, the risk of having false negatives exists. The rate of false negatives depends on several factors such as the overclocking factor, the fabric, the wordlengths of data and operators, the design, etc.

We ran 8×8 and 4×4 bits accelerators over a range of frequencies to empirically show that false negative are highly improbable. We define the false negative rate as the number of missed erroneous tiles to the number of erroneous tiles. We computed more than 700,000 tiles for 21 different frequencies selected to observe 0% to 10% failed tiles. We deliberately overclocked the circuit until large numbers of errors were observed. Since the false negative rate is expected to be extremely small, we would not be able to observe false negatives under normal configuration (with less than 0.1% error rate) in a reasonable amount of time.

We did not observe any false negative for the 8×8 bits configuration, but observed some for the 4×4 bits configuration. This agrees with our analysis in Section III-G that shows designs with shorter checksum lengths are more likely to have

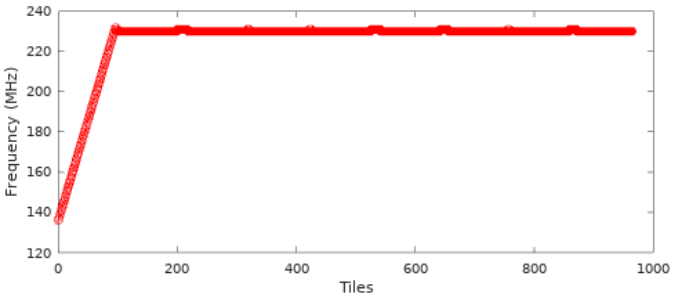


Fig. 13. The fifth layer of AlexNet with frequency scaling for classification of 1000 images (= tiles for this configuration) using the 16×16 bits design. The algorithm in Section IV-F was configured to $G = 1$ MHz and $I = 100$.

false negatives. The false negative rate was at most 0.4% of the incorrectly computed tiles. This translates to 0.06% of the total number of tiles computed at the same frequency, which is being aggressively overclocked to exhibit many errors. The false negative rate is expected to be much lower in a normal setting where the error rate is much lower. Therefore, we claim that false negatives are not an issue in our approach.

F. Dynamic Frequency Scaling

Figure 13 shows the result of applying frequency scaling on the fifth layer of AlexNet with the 16×16 bits design. The frequency scales up to around 230 MHz and then stabilizes with occasional bumps. The mean frequency over the 1000 images was 225.5 MHz. Compared to the baseline frequency, this is about 66 % increase in throughput. Note that this result is from a single board, single setup. The gains are expected to vary under different operating conditions.

G. Impact on Energy Efficiency

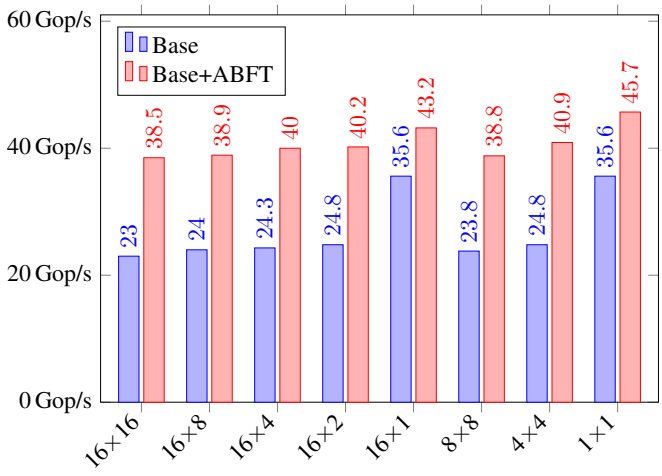
Since energy efficiency is one of the reason for resorting to FPGAs, we measured the impact of our technique on the overall energy efficiency of the system. The power dissipated by an FPGA accelerator can be decomposed into a static power contribution P_{stat} and a dynamic power contribution P_{dyn}, the latter being proportional to the circuit clock frequency.

Because FPGAs are built out of SRAM technology, they tend to have relatively high static to dynamic power ratio. In addition, for a fair analysis, the value P_{stat} must also include static power contributions from all the components of the system: external memory, peripherals, regulators, etc.

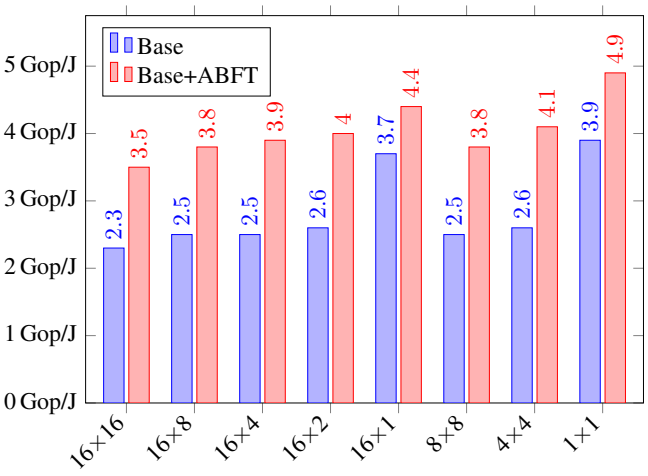
Our results are provided in Figure 14b that shows the energy efficiency in Giga-operation per Joule for different wordlengths. We measured the execution time of the accelerator computing 64 images, and the power consumption of the entire ZC706 board (except power adapter) during that time. As expected, our overclocking scheme impacts only a fraction of the total power consumption of the system and improves the overall energy efficiency of the system by 46% on average. This is perfectly in line with prior observations [8], [17].

VI. Discussion and Related Work

CNN is a rapidly changing field, and many approaches to hardware accelerations have been proposed in the literature.



(a) Throughput



(b) Energy Efficiency

Fig. 14. Performance comparison for different wordlengths in throughput (Giga-operations per second; Gop/s) and energy efficiency (Gop/J) at two frequencies: Base (f_{STA}) and ABFT-enabled frequency (f_{oc}), which is the maximum frequency at which we did not observe any error during the experiments.

In the following, we review several CNN strategies/design choices, and discuss how their impact of the feasibility or efficiency of our propose ABFT scheme.

A. Data Representations

Early implementations of CNNs on FPGAs used floating point arithmetic to perform convolution operations [6]. Floating point arithmetic is not associative and leads to rounding errors during computations. Thus, the values of input- and output-checksum obtained through ABFT may not match even in the absence of timing errors.

The invariant may be relaxed to tolerate a certain error margin to alleviate this issue, but at the price of significant increases in the rates of both false positives and false negatives. Since checksum computations are linear, it is possible to derive analytical models of the error distribution, and to determine the most adequate threshold. Moreover, such relaxations do not affect the capability to detect large errors, e.g., corrupted exponents.

In this work, we focused on fixed-point implementations, since it has been shown that CNN inference can be performed with fixed-point arithmetic (including ternary and binary neural networks, which are special cases of integer arithmetic) with little or no impact on the classification accuracy.

B. Algorithmic Variations

Independently of the choice of data representation, many research work has proposed algorithm-level optimizations to improve the performance of CNNs.

For example, **FFT based convolutions** have been used to improve the performance of CNN accelerators [18], [19]. The implementation of convolution in the frequency domain helps reducing the overall computational complexity from $O(N^2)$ to $O(NlogN)$. However, all existing implementations are based on floating point arithmetics, for which our proposed ABFT scheme cannot guarantee the same error coverage due to rounding errors as explained above.

Winograd based convolutions have shown to improve the performance of CNNs on FPGAs [20] by reducing the average number of multiplications required per convolution output. However, this reduction comes at the price of increased number of additions, and more complex reuse patterns. In contrast to FFT based convolutions, Winograd based CNN does not require intermediate quantization stages. As a consequence, our approach can be applied at the layer level almost as is.

Weight pruning is an effective technique to reduce storage/bandwidth requirements when implementing CNNs. Earlier results [21], [22] have shown that up to 90% of the weights could be pruned out without impacting classification accuracy. Efficiently implementing pruned CNN requires convolution accelerators that are different from the one used in this work as they operate on sparse data. Nevertheless, our error detection scheme can be reused as is, with a slight increase in *relative* overhead being the only difference. Since the number of operations in a 90% pruned network is one-tenth of that of an unpruned network, the relative cost of ABFT is ten times higher. This is at most 3% for the configuration shown in Figure 6, compared to more than 50% speed improvements.

C. Comparison with Existing ABFT Techniques

Our approach is similar to other ABFT techniques presented in previous work [9], [23]. The idea behind Algorithm Based Fault tolerance (ABFT) is to use high-level algebraic properties (or algebraic invariants) relating the inputs and outputs of an algorithm. When the results do not satisfy these invariants, then at least one error has occurred during the execution. In some cases, the results of the incorrect invariant may even be used to locate (and correct) errors.

ABFT was initially proposed for error detection and correction of matrix operations [9], but was later adapted to other algebraic kernels, such as matrix decomposition (QR, LU) [24], iterative stencils [25], and Fast Fourier Transforms [24]. The original ABFT for matrix product is illustrated in Figure 15.

The convolution layer and the fully-connected layer can be viewed as a collection of matrix products, making the classical ABFT directly applicable. The lightweight checksum

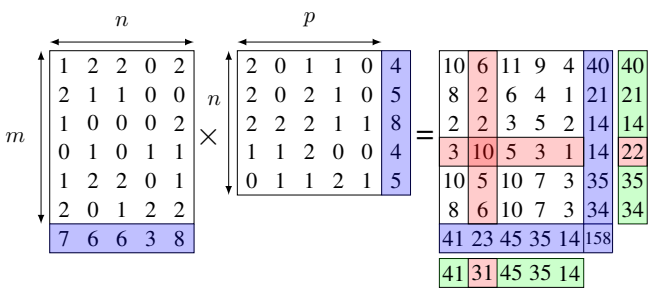


Fig. 15. ABFT for matrix product [9]. The input matrices are augmented with additional row and column (shaded ones) that are column-wise and row-wise checksums, respectively. The product of the augmented matrices have additional row and column that are themselves checksums of the inner sub-matrix (the original output without augmentation), which is due to the algebraic property. In this example, the value at 4th row, 2nd column is incorrect—it should be 2. This can be detected and located by the mismatch in the row-wise and column-wise checksums that are computed during the matrix product, and those separately computed from the output matrix.

calculation proposed in this paper is **not a direct application of the ABFT for matrix operations**. We use algorithmic invariants in collections of convolutions to further reduce the cost of checksums. This is evident in the fact that we reduce the algorithmic complexity by twofold, exploiting reuse in two dimensions, whereas the original ABFT brings one-degree savings. Some ABFT extensions employ a variant of the original method by identifying pieces of computations that can be viewed as matrix operations [25], [26]. However, ABFT is a more general concept that can be applied to other computations, e.g., FFT [23]. The original ABFT for matrix operations has been used in the context of FPGAs as well [27], [28]. However, these earlier work did not consider ABFT as a mean to support overclocking.

Some of the ABFT techniques employ two or more checksums to enable error correction. It is possible to use a similar method for the convolution layer by giving up one-degree of complexity savings. However, this is not attractive because:

- Error correction based on checksums would not work if there are two or more errors (or at the cost of less complexity reduction), which is frequently the case with overclocking.
- Recovery by recomputing erroneous outputs does not add significant cost as discussed in Section IV-E.
- Adding the error recovery logic in hardware can be costly in terms of area.

D. Other Techniques for Timing Error Detection

Other error detection techniques, such as those based on Residue Numbering Systems (RNS) could be used. RNS protection provides low overhead error detection [16] for convolution kernels. However, they provide limited protection against typical timing errors that impact several outputs at a time. We have compared the area overhead of RNS with our approach in Section V-B.

Some error detection techniques have been designed specifically to detect timing errors. Both Razor [4] and online slack measurement [5], [14] use shadow registers to detect timing

violation, or to measure timing slacks. These approaches adjust the frequency/voltage using the measured errors, or timing slack, which is similar to our work. The main difference between our approach and those based on shadow registers is that we provide error detection at the algorithm-level in contrast to register/circuit level. This makes our approach more flexible, as it can be reused without additional design effort over a large design space (tile size, unrolling, data wordlength, etc.) of CNN accelerators. In contrast, circuit-level protections must be redesigned for every new design. Although some proof-of-concept toolchains have been proposed [8], [29] to automate such approach, they are not publicly available, and it is difficult to assess how robust they are in practice.

Existing circuit-level techniques trade coverage with area overhead. One of the main difficulty is to properly select the smallest subset of the registers to protect/measure to keep the overhead low. This process is complicated as the addition of the shadow registers may impact the timing, and is also limited by the static timing analysis to determine the (near-)critical paths. Although low overhead ($\leq 4\%$) has been claimed for small kernels prototyped in FPGAs [5], the overhead of a variant of Razor added to an ASIC implementation of the ARM Cortex-M3 [13] is reported to be much higher at 20% or even more depending on the expected coverage.

It is also not clear how device level variability impacts error coverage. Because of the large variability in logic block delays within a single chip (see the work of Gojman et al. [30]), the 10% longest paths reported by STA are seldom the actual longest paths for the device at hand. Therefore, it is difficult to evaluate the actual error coverage of such approach.

More importantly, as acknowledged by Nunez-Yanez [8], key FPGA components such as DSP blocks include internal (e.g., pipelining) registers that cannot be protected due to the lack of adequate routing resources. Given that CNN workloads mainly consist of multiply-accumulate patterns, such error detectors have limited applicability for FPGA designs. Specifically, it is limited to binary neural network accelerators that do not use the accumulators in DSP blocks.

In spite of these challenges, Nunez-Yanez [8] has used this approach to explore energy/performance trade-offs using DVFS on FPGAs in the context of binary neural networks. The results reported in their work are in line with our observations (50% performance margins are common for Zynq) and suggest that more recent generations of FPGAs, such as the UltraScale, can tolerate even larger overclocking margins, possibly due to the increasing variability in smaller process nodes. The main difference lies in the results that we obtain for the overclocking margin for 1 bit CNNs. However, this is completely orthogonal to the error detection technique and is more related to convolution implementation.

Our approach does not share these limitations in circuit-level techniques, because it operates at the algorithm level. It provides excellent error coverage for lower hardware overhead: less than 1% compared to a few reference values of around 5% reported in prior work [5], [14].

The main limitation of our work is that we require the computation to have algorithmic properties to enable lightweight error detection. Although convolution is not the only computation where ABFT is available, our technique is not applicable to arbitrary computations unlike those based on shadow registers. An interesting direction of future work is to automatically identify the necessary properties to improve the applicability of our technique.

E. Applicability to Other Architectures

Although the proposed approach is specifically tailored to FPGA accelerators, we expect our technique to be useful on other types of architectures (CPUs, GPUs).

For example, Leng et al. [31] evaluated how the impact of timing errors, due to voltage reduction (DVFS), would impact the correctness of the program output. Their results show that the impact of timing errors is highly dependant of the program behavior. They also show that, under aggressive voltage reduction, convolutions and FFT kernels induce many silent data errors. Theses results suggest that our approach would be directly applicable to GPUs accelerated CNN, provided that fixed point arithmetic is used.

We also expect the technique to be applicable to in-order CPU micro-architectures with simple control logic, such as those used for CNN workloads in embedded systems [32].

VII. Conclusions and Future Work

In this paper, we propose timing speculation coupled with lightweight error detection as an approach to further improve the performance of hardware accelerators for CNNs. We have demonstrated the efficacy of our approach with a prototype implementation and an extensive empirical study. In addition, our approach is well suited for implementation in high-level design tools such as Vivado HLS/SDSoC, which is becoming more and more attractive for productivity reasons.

We believe that similar techniques can be applied to many other application domains (bioinformatics, iterative solvers, etc.) by taking advantage of existing ABFT techniques, or by devising new algorithms tailored for this task.

Acknowledgements

This work was partially funded by the Brittany Region.

References

[1] S. Chaudhuri, J. S. J. Wong, and P. Y. K. Cheung, "Timing speculation in FPGAs: Probabilistic inference of data dependent failure rates," in *Proceedings of the 2011 International Conference on Field-Programmable Technology*, ser. FPT '11, 2011, pp. 1–8.

[2] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17, 2017, pp. 8:1–8:12.

[3] K. Shi, D. Boland, and G. A. Constantinides, "Accuracy-Performance Tradeoffs on an FPGA through Overclocking," in *Proceedings of the IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '13, 2013, pp. 29–36.

[4] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation," in *Proceedings of the 36th IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36, 2003, pp. 7–.

[5] J. M. Levine, E. Stott, and P. Y. Cheung, "Dynamic Voltage & Frequency Scaling with Online Slack Measurement," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '14, 2014, pp. 65–74.

[6] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15, 2015, pp. 161–170.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[8] J. L. Nunez-Yanez, "Energy proportional neural network inference with adaptive voltage and frequency scaling," *IEEE Transactions on Computers*, pp. 1–1, 2018.

[9] K.-H. Huang and J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, 1984.

[10] Z. Chen, "Online-ABFT: An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13, 2013, pp. 167–176.

[11] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based Fault Tolerance for Dense Matrix Factorizations," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12, 2012, pp. 225–234.

[12] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 56:1–56:39, 2018.

[13] M. Fojtik, D. Fick, Y. Kim, N. Pinckney, D. M. Harris, D. Blaauw, and D. Sylvester, "Bubble Razor: Eliminating Timing Margins in an ARM Cortex-M3 Processor in 45 nm CMOS Using Architecturally Independent Error Detection and Correction," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 1, pp. 66–81, 2013.

[14] J. L. Nunez-Yanez, M. Hosseinabady, and A. Beldachi, "Energy optimization in commercial FPGAs with voltage, frequency and logic scaling," *IEEE Transactions on Computers*, vol. 65, no. 5, pp. 1484–1493, May 2016.

[15] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Patel, and M. Herbordt, "A framework for acceleration of CNN training on deeply-pipelined FPGA clusters with work and weight load balancing," in *Proceedings of the 28th International Conference on Field Programmable Logic and Applications*, ser. FPL '18. IEEE, 2018, pp. 394–3944.

[16] S. J. Piestrak and P. Patronik, "Design of Fault-Secure Transposed FIR Filters Protected Using Residue Codes," in *Proceedings of the 17th Euromicro Conference on Digital System Design*, 2014, pp. 575–582.

[17] T. Yuki and S. Rajopadhye, "Folklore confirmed: Compiling for speed = compiling for energy," in *Proceedings of the 26th International Workshop on Languages and Compilers for Parallel Computing*, ser. LCPC '13, Sep. 2013.

[18] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan *et al.*, "CirCNN: Accelerating and Compressing Deep Neural Networks Using Block-circulant Weight Matrices," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 395–408.

[19] C. Zhang and V. Prasanna, "Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 35–44.

[20] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *Proceedings of the IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '17. IEEE, 2017, pp. 101–108.

[21] S. Li, W. Wen, Y. Wang, S. Han, Y. Chen, and H. Li, "An FPGA design framework for CNN sparsification and acceleration," in *Proceedings of the IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '17, April 2017, pp. 28–28.

[22] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, pp. 1135–1143.

[23] S.-J. Wang and N. K. Jha, "Algorithm-based fault tolerance for FFT networks," *IEEE Transactions on Computers*, vol. 43, no. 7, pp. 849–854, 1994.

[24] A. L. N. Reddy and P. Banerjee, "Algorithm-based fault detection for signal processing applications," *IEEE Transactions on Computers*, vol. 39, no. 10, pp. 1304–1308, 1990.

[25] A. Roy-Chowdhury, N. Bellas, and P. Banerjee, "Algorithm-based error-detection schemes for iterative solution of partial differential equations," *IEEE Transactions on Computers*, vol. 45, no. 4, pp. 394–407, 1996.

[26] G. Bosilca, A. Bouteiller, T. Herault, Y. Robert, and J. Dongarra, "Composing resilience techniques: ABFT, periodic and incremental checkpointing," *International Journal of Networking and Computing*, vol. 5, no. 1, pp. 2–25, 2015.

[27] A. Jacobs, G. Cieslewski, and A. D. George, "Overhead and reliability analysis of algorithm-based fault tolerance in FPGA systems," in *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications*, ser. FPL '12, 2012, pp. 300–306.

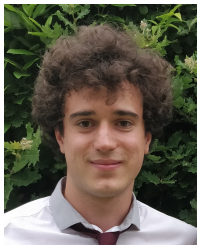
[28] J. J. Davis and P. Y. K. Cheung, "Achieving low-overhead fault tolerance for parallel accelerators with dynamic partial reconfiguration," in *Proceedings of the 24th International Conference on Field Programmable Logic and Applications*, ser. FPL '14, 2014, pp. 1–6.

[29] J. M. Levine, E. Stott, G. A. Constantinides, and P. Y. K. Cheung, "SMI: Slack Measurement Insertion for online timing monitoring in FPGAs," in *Proceedings of the 23rd International Conference on Field Programmable Logic and Applications*, ser. FPL '13, 2013, pp. 1–4.

[30] B. Gojman, S. Nalmela, N. Mehta, N. Howarth, and A. Dehon, "GROK-LAB: Generating real on-chip knowledge for intra-cluster delays using timing extraction," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 4, pp. 32:1–32:23, 2014.

[31] J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, and V. J. Reddi, "Safe limits on voltage reduction efficiency in GPUs: A direct measurement approach - IEEE Conference Publication," in *Microarchitecture (MICRO), 2015 48th Annual IEEE/ACM International Symposium on*. IEEE, 2015, pp. 294–307.

[32] (2019) HERO: Open Heterogeneous Research Platform. [Online]. Available: https://pulp-platform.org/hero.html



Thibaut Marty obtained his BSc degree and MSc degree from University of Rennes, France, in 2015 and 2017 respectively. He is currently working toward the PhD degree in computer sciences at the University of Rennes, under supervision of Steven Derrien and Tomofumi Yuki. His research interests include High-Level Synthesis and fault tolerance on FPGAs.

Steven Derrien obtained his PhD from University of Rennes 1 in 2003, and is now professor at University of Rennes 1. He is also a member of the Cairn research group at IRISA. His research interests include High-Level Synthesis, loop parallelization, and reconfigurable systems design.

Tomofumi Yuki received his Ph.D. in computer science from Colorado State University in 2012. He is currently a researcher at Inria, France. His main research interests are in parallel/high performance computing, compilers/programming models for parallel/HPC, and High-Level Synthesis.