FT-CNN: Algorithm-Based Fault Tolerance for Convolutional Neural Networks

Kai Zhao, Sheng Di, *Senior Member, IEEE*, Sihuan Li, *Student Member, IEEE*, Xin Liang, *Member, IEEE*, Yujia Zhai, Jieyang Chen, *Member, IEEE*, Kaiming Ouyang, Franck Cappello, *Fellow, IEEE*, and Zizhong Chen, *Senior Member, IEEE*

Abstract—Convolutional neural networks (CNNs) are becoming more and more important for solving challenging and critical problems in many fields. CNN inference applications have been deployed in safety-critical systems, which may suffer from soft errors caused by high-energy particles, high temperature, or abnormal voltage. Of critical importance is ensuring the stability of the CNN inference process against soft errors. Traditional fault tolerance methods are not suitable for CNN inference because error-correcting code is unable to protect computational components, instruction duplication techniques incur high overhead, and existing algorithm-based fault tolerance (ABFT) techniques cannot protect all convolution implementations. In this article, we focus on how to protect the CNN inference process against soft errors as efficiently as possible, with the following three contributions. (1) We propose several systematic ABFT schemes based on checksum techniques and analyze their fault protection ability and runtime thoroughly. Unlike traditional ABFT based on matrix-matrix multiplication, our schemes support any convolution implementations. (2) We design a novel workflow integrating all the proposed schemes to obtain a high detection/correction ability with limited total runtime overhead. (3) We perform our evaluation using ImageNet with well-known CNN models including AlexNet, VGG-19, ResNet-18, and YOLOv2. Experimental results demonstrate that our implementation can handle soft errors with very limited runtime overhead (4%∼8% in both error-free and error-injected situations).

Index Terms—Algorithm-based fault tolerance, deep learning, silent data corruption, reliability, high-performance computing

1 Introduction

Deep learning using convolutional neural networks (CNNs) is becoming the key state-of-the-art technique in science and technology fields such as image classification [1], [2], [3], object detection [4], natural language processing [5], medical image analysis [6], and drug design [7]. More and more scientific research (such as cosmological simulation and materials analysis) also is addressing the great potential of leveraging CNN techniques to analyze extremely large amounts of data in a supercomputer environment, achieving unprecedented discoveries in their domains [8].

The reliability of the CNN inference is becoming a critical concern [9] because CNN inference applications are being widely utilized in different scenarios, including high-performance scientific simulations and safety-critical systems [10], [11] such as aerospace and autonomous vehicles. CNN inference applications usually run for a long time or continuously to process many inference tasks. For example, the inference engine in autonomous vehicles is running continuously to predict road conditions. As a result, even a single inference task for one input finishes in seconds, the reliability of CNN inference is still critically important given the long execution time of the inference applications.

In the domain of CNN inference, machine learning applications could be very error prone because of two reasons. On the one hand, recent literature indicates that soft errors are inevitable in modern systems, from edge computing devices to supercomputers [12], [13], because of multiple factors [14] such as high-energy cosmic radiation [15], aging, and wear of devices [16]. On the other hand, CNN inference applications often call for power-efficient and cost-efficient machine learning accelerators, which [17], [18] may adopt overclocking with voltage underscaling, incurring more soft errors than common hardware incurs.

Soft errors may cause serious consequences to CNN inference systems. Recent studies [19], [20], [21] indicate that resilient convolutional neural networks are essential for guaranteeing the correctness of inference applications. Researchers [19] demonstrate that a single bit flip happened during CNN image classification could result in as much as 40 and 70 percent SDC rate in datapath and memory, respectively. Such high SDC rates would downgrade the CNN prediction accuracy dramatically. Furthermore, the neutron beam test [20] shows that when running YOLO [4] classification, the Failure In Time (FIT) caused by SDCs could be as much as 38 for Nvidia K40 and 96 for Nvidia Tegra X1, which fail to meet the ISO 26262 standard for functional safety of road vehicles [22].

- *Kai Zhao, Sihuan Li, Yujia Zhai, Kaiming Ouyang, and Zizhong Chen are with the Department of Computer Science and Engineering, University of California, Riverside, Riverside, CA 92521 USA. E-mail: {kzhao016, sli049, yzhai015, kouya001, chen}@ucr.edu.*
- *Sheng Di and Franck Cappello are with the Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL 60439 USA. E-mail: sdi1@anl.gov, cappello@mcs.anl.gov.*
- *Xin Liang and Jieyang Chen are with the Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831 USA. E-mail: liangx@ornl.gov, chen@ucr.edu.*

Manuscript received 30 June 2020; revised 21 Aug. 2020; accepted 6 Sept. 2020. Date of publication 31 Dec. 2020; date of current version 11 Feb. 2021. (Corresponding author: Kai Zhao.) Recommended for acceptance by P. Balaji, J. Zhai, and M. Si. Digital Object Identifier no. 10.1109/TPDS.2020.3043449

Existing resilient solutions are insufficient for protecting CNN inference applications against these soft errors. Error-correcting code (ECC), for example, suffers from memory area cost and relatively high latency and power consumption. According to [23], ECC with chip-kill applied to all data, compared with no ECC protection, has an average of 40 percent overhead in memory energy, 20 percent overhead in system energy and 20 percent overhead in performance for computation-bounded applications. Moreover, ECC cannot handle multiple bit flips or computational errors. Techniques based on instruction duplication (ID) [24] incur high overhead and require both application-specific and hardware-specific optimization; and optimizing and deploying ID techniques on all CNN accelerators is difficult.

Considering all the drawbacks and limitations of ECC and ID, algorithm-based fault tolerance (ABFT) [25] is an attractive solution to realize resilient CNN. It has much lower overhead than other techniques have; and it is architecture independent, meaning that it supports any hardware accelerator. The idea of ABFT is to detect and/or correct soft errors based on the known invariants that the algorithm has. Over the past thirty years, ABFT schemes have been successful in detecting errors for matrix operations [26], [27], [28], [29], [30], [31], iterative methods [32], [33], [34], [35], data transformation kernels [36] and sorting algorithm [37]. However, the existing ABFT schemes for matrix operations focus mainly on large and square matrices. Moreover, they incur more than 50 percent overhead when applied for CNN soft error protection (shown by our experiments in Section 6.3).

In this paper, we propose a strategy comprising a series of ABFT schemes for protecting the CNN inference stage against soft errors. We focus on the convolutional layers in CNN because they consume the major portion of the computation time [38], [39], [40], [41].

The main contributions of this paper are summarized as follows.

- We design several ABFT schemes that can be applied to any convolution implementation on any hardware. They can detect and correct errors at runtime. We provide an in-depth analysis of the ABFT schemes in terms of fault protection ability and runtime.
- We design a multischeme workflow for soft error protection with layerwise optimization to obtain a high detection/correction ability with limited runtime overhead. Additionally, our solution can protect the bias operation, grouped convolution, and back propagation.
- We implement an efficient soft error detection library for CNN, called *FT-Caffe*, and evaluate FT-Caffe on ImageNet [42] using four popular CNN models: Alexnet [1], VGG-19 [2], ResNet-18 [3], and YOLOv2 [4]. Experimental results on the Bebop supercomputer [43] using up to 128 nodes demonstrate that FT-Caffe can keep the correctness of the inferences with 4%$\sim$8% overhead in both error-free and erroneous cases.

In the rest of the paper, we first introduce background about convolutional layers and existing ABFT techniques applicable to matrix-matrix multiplication (MM)-based convolution implementation. In Section 3, we propose four novel ABFT schemes that can be applied to any convolution implementations. In Section 4, we analyze the fault protection ability and runtime of the four schemes and propose an efficient multischeme workflow integrating all the four schemes. In Section 5, we discuss how to support bias, grouped convolution, and back propagation. In Section 6, we evaluate our solutions for both error-free case and erroneous case. In Section 7, we discuss related work on fault tolerance in convolutional neural networks. We present our concluding remarks in Section 8.

TABLE 1
Notations and Symbols Used in This Paper

Notation	Description
$\mathbf{D}$	Feature map, dimension is $4D$
$\mathbf{W}$	Kernels, also called filters, dimension is $4D$
$\mathbf{O}$	Output, dimension is $4D$
$\mathbf{B}$	Bias, dimension is $1D$
$\mathbf{C}$	Checksums
$\mathbf{S}$	Block summations of $\mathbf{O}$, corresponding to checksums
$\otimes$	Convolution operation
N	First dimension of $\mathbf{D}$ and $\mathbf{O}$
M	First dimension of $\mathbf{W}$ and second dimension of $\mathbf{O}$
Ch	Second dimension of $\mathbf{D}$ and $\mathbf{W}$, also called channels
H	Third and fourth dimension of $\mathbf{D}$
R	Third and fourth dimension of $\mathbf{W}$
E	Third and fourth dimension of $\mathbf{O}$
U	Stride size

2 Background

This section introduces some high-level ideas of convolutional layers and the existing ABFT techniques to MM-based convolution algorithms. The notations and symbols used in this paper are summarized in Table 1.

2.1 Definition of Convolutional Layer

The convolutional layer can be represented as the following convolution operation.

$$\mathbf{O}[n][m][x][y] = \mathbf{B}[m] + \sum_{k=0}^{Ch-1}\sum_{i=0}^{R-1}\sum_{j=0}^{R-1}\mathbf{D}[n][k][Ux+i][Uy+j] \times \mathbf{W}[m][k][i][j]$$
$$0 \le n < N, 0 \le m < M, 0 \le x, y < E, E = \frac{H-R+U}{U}. \quad (1)$$

The convolution operation involves two significant inputs: the feature map (fmap) $\mathbf{D}$, $\mathbf{D} \in \mathbb{R}^{N \times Ch \times H \times H}$, and the convolutional kernels $\mathbf{W}$, $\mathbf{W} \in \mathbb{R}^{M \times Ch \times R \times R}$. Note that all the matrices and vectors in this paper are highlighted in bold in order to differentiate from the scalar numbers, according to the naming convention. The bias, denoted as $\mathbf{B}$, is applied to the output after convolution, and the final result is denoted as $\mathbf{O}$, $\mathbf{O} \in \mathbb{R}^{N \times M \times E \times E}$. Since the bias operation is independent of the convolution computation, in the rest of this section we describe only the protection for convolution computation. In Section 5.1, we will discuss the protection for bias.

2.2 Implementation of Convolutional Layer

Convolution can be implemented efficiently in several ways [44]. The first option is *MM-based convolution* [45], which

reshapes the kernel and feature map to two temporary matrices and then applies matrix-matrix multiplication on them. Another way to implement convolution is called *direct convolution*, which performs the convolution operation directly. It is widely used in AI accelerators including Eyeriss [39], DianNao [46] and NVIDIA Deep Learning Accelerator [47]. *Fast Fourier transform–based convolution* [48] leverages FFT to compute the convolution. It is particularly suitable for the relatively large feature map and kernel. However, it is inferior to the *Winograd convolution* [44] when the sizes of the feature map and kernel are relatively small.

Modern CNN frameworks and accelerators generally automatically choose the best implementations of convolution based on hardware resources and model structure, because various implementations have different constraints on memory, architecture, and CNN model.

2.3 ABFT for Matrix-Matrix Multiplication

Traditional ABFT designed for matrix-matrix multiplication can be applied to the MM calculation of the MM-based convolution [20], but it has at least three limitations. (1) It supports only MM-based convolution implementation, which is not always the best-fit implementation selected by the CNN framework and accelerator. (2) It incurs high overhead (more than 50 percent, as shown in Section 6.3), due to the small and irregular shape of the matrices used by MM-based convolution. (3) Moreover, it cannot cover the reorganization operations of feature before the MM calculation. Therefore, new ABFT schemes are needed in order to protect the convolutional layer more effectively.

3 Novel ABFT Schemes for Convolution

In this section, we present four novel ABFT schemes, each supporting any convolution implementation and being able to protect the whole convolution process. In Section 4, we propose a multischeme workflow using all the schemes in different stages to maximize the soft error protection ability with minimized performance overhead.

3.1 Preliminary Analysis–Convolution

For clear description, we interpret convolution at the block level. Specifically, in Equation (1), $\mathbf{D}$, $\mathbf{W}$, and $\mathbf{O}$ are all $4D$ matrices. They can be represented as being composed of multiple blocks as shown in Fig. 1. For any n and m ($0 \le n < N, 0 \le m < M$), $\mathbf{D}_n$, $\mathbf{W}_m$, and $\mathbf{O}_{nm}$ are blocks. The dimension of $\mathbf{D}_n$, $\mathbf{W}_m$, and $\mathbf{O}_{nm}$ are $Ch \times H \times H$, $Ch \times R \times R$ and $E \times E$, respectively. The notation $\otimes$ is used to represent the convolution computation between blocks $\mathbf{D}_n$ and $\mathbf{W}_m$. The convolution operation defined by Equation (1) can be simplified at the block level as follows.

$$\begin{aligned} &\mathbf{O}_{nm} = \mathbf{D}_n \otimes \mathbf{W}_m \\ &0 \le n < N, 0 \le m < M. \end{aligned} \tag{2}$$

Equation (2) can be interpreted by using the blue part in Fig. 1. Since each of the 3D substructures of $\mathbf{D}$ and $\mathbf{W}$ is treated as a block, $\mathbf{D}$ and $\mathbf{W}$ can be thought of as two $1D$ vectors of blocks. At the block level, the convolution operation is similar to matrix-matrix multiplication. The element (i, j) of $\mathbf{O}$ is calculated by using the ith element of $\mathbf{D}$ and the jth element of $\mathbf{W}$. As illustrated in Fig. 1, the elements involved in the convolutional layers can be split into two levels, which are covered by our protection solution, respectively.

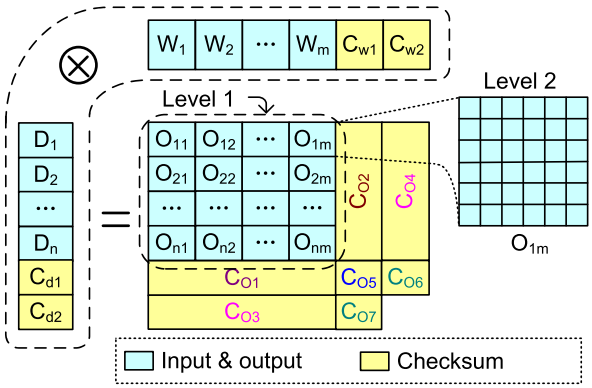


Fig. 1. Interpretation of convolution at the block level.

We can derive that the convolution operation (denoted by $\otimes$) has a distributive property as follows.

$$\begin{aligned} &\mathbf{D}_1 \otimes \mathbf{W}_1 + \mathbf{D}_2 \otimes \mathbf{W}_1 = \\ &\sum_{k=0}^{Ch-1} \sum_{i=0}^{R-1} \sum_{j=0}^{R-1} \mathbf{D}_1[k][Ux+i][Uy+j] \times \mathbf{W}_1[k][i][j] \\ &+ \sum_{k=0}^{Ch-1} \sum_{i=0}^{R-1} \sum_{j=0}^{R-1} \mathbf{D}_2[k][Ux+i][Uy+j] \times \mathbf{W}_1[k][i][j] \\ &= \sum_{k=0}^{Ch-1} \sum_{i=0}^{R-1} \sum_{j=0}^{R-1} (\mathbf{D}_1 + \mathbf{D}_2)[k][Ux+i][Uy+j] \times \mathbf{W}_1[k][i][j] \\ &= (\mathbf{D}_1 + \mathbf{D}_2) \otimes \mathbf{W}_1. \end{aligned}$$

Similarly, we can get the following equation.

$$\mathbf{D}_1 \otimes \mathbf{W}_1 + \mathbf{D}_1 \otimes \mathbf{W}_2 = \mathbf{D}_1 \otimes (\mathbf{W}_1 + \mathbf{W}_2). \tag{3}$$

The distributive property, Formula (3), is the key to proving the equivalence between the sum of the output and the output checksum. This property will be used later to prove the correctness of our design.

3.2 Preliminary Analysis–CNN Checksums

In general, we compute checksums for $\mathbf{D}$ and $\mathbf{W}$ and then use them to derive the checksums for $\mathbf{O}$. Soft errors can be detected and corrected by comparing $\mathbf{O}$ with its checksums.

We introduce all the checksums (as shown in Table 2 and as yellow blocks in Fig. 1) that are necessary for our ABFT schemes.

We define the checksums of $\mathbf{D}$ and $\mathbf{W}$ as follows.

$$\begin{aligned} \mathbf{C}_{d1} &= \sum_{n=0}^{N-1} \mathbf{D}_n \\ \mathbf{C}_{d2} &= \sum_{n=0}^{N-1} n\mathbf{D}_n \\ \mathbf{C}_{w1} &= \sum_{m=0}^{M-1} \mathbf{W}_m \\ \mathbf{C}_{w2} &= \sum_{m=0}^{M-1} m\mathbf{W}_m. \end{aligned} \tag{4}$$

TABLE 2
Checksums Used by Schemes

Scheme	Checksums of D and W	Checksums of O
Full Checksum (FC)	$\mathbf{C}_{d1}$, $\mathbf{C}_{w1}$	$\mathbf{C}_{o1}$, $\mathbf{C}_{o2}$
Row Checksum (RC)	$\mathbf{C}_{d1}$, $\mathbf{C}_{d2}$	$\mathbf{C}_{o1}$, $\mathbf{C}_{o3}$
Column Checksum (ClC)	$\mathbf{C}_{w1}$, $\mathbf{C}_{w2}$	$\mathbf{C}_{o2}$, $\mathbf{C}_{o4}$
Checksum-of-Checksum (CoC)	$\mathbf{C}_{d1}$, $\mathbf{C}_{w1}$, $\mathbf{C}_{d2}$, $\mathbf{C}_{w2}$	$\mathbf{C}_{o5}$, $\mathbf{C}_{o6}$, $\mathbf{C}_{o7}$
CoC Detection Only (CoC-D)	$\mathbf{C}_{d1}$, $\mathbf{C}_{w1}$, $\mathbf{C}_{d2}$, $\mathbf{C}_{w2}$	$\mathbf{C}_{o5}$

The four checksums (denoted as input checksums) can be treated as four blocks of $\mathbf{D}$ and $\mathbf{W}$. The checksums of $\mathbf{O}$ (denoted as output checksums) are defined as the convolution result of input checksums and/or inputs.

$$\begin{aligned}
\mathbf{C}_{o1} &= \mathbf{C}_{d1} \otimes \mathbf{W} \\
\mathbf{C}_{o2} &= \mathbf{D} \otimes \mathbf{C}_{w1} \\
\mathbf{C}_{o3} &= \mathbf{C}_{d2} \otimes \mathbf{W} \\
\mathbf{C}_{o4} &= \mathbf{D} \otimes \mathbf{C}_{w2} \\
\mathbf{C}_{o5} &= \mathbf{C}_{d1} \otimes \mathbf{C}_{w1} \\
\mathbf{C}_{o6} &= \mathbf{C}_{d1} \otimes \mathbf{C}_{w2} \\
\mathbf{C}_{o7} &= \mathbf{C}_{d2} \otimes \mathbf{C}_{w1}.
\end{aligned} \tag{5}$$

The output $\mathbf{O}$ is represented in the form of blocks (i.e., Level 1 in Fig. 1). Elements inside the same block are independent with respect to checksums (Level 2 in Fig. 1). That is, we perform the checksum comparison independently for each element across blocks. Therefore, multiple soft errors in the same block can be detected and corrected independently.

In what follows, we describe the four schemes we proposed, each involving one or more input and output checksums. The required checksums used by each scheme are summarized in Table 2.

3.3 Full Checksum Scheme (FC)

The first scheme we designed is called *full checksum scheme*, or FC, because it is based on checksums from both $\mathbf{D}$ and $\mathbf{W}$, as shown in Fig. 1 and Table 2.

$\mathbf{C}_{d1}$ and $\mathbf{C}_{w1}$ are calculated before the convolution operation, so any memory error striking $\mathbf{D}$ or $\mathbf{W}$ during the convolution would not affect $\mathbf{C}_{d1}$ or $\mathbf{C}_{w1}$. As for the output checksums, we can get the following equations by applying the distributive property of $\otimes$.

$$\mathbf{C}_{o1}[m] = \left(\sum_{n=0}^{N-1} \mathbf{D}_n\right) \otimes \mathbf{W}_m = \sum_{n=0}^{N-1} (\mathbf{D}_n \otimes \mathbf{W}_m) = \sum_{n=0}^{N-1} \mathbf{O}_{nm}$$

$$\mathbf{C}_{o2}[n] = \mathbf{D}_n \otimes \left(\sum_{m=0}^{M-1} \mathbf{W}_m\right) = \sum_{m=0}^{M-1} (\mathbf{D}_n \otimes \mathbf{W}_m) = \sum_{m=0}^{M-1} \mathbf{O}_{nm}.$$

These equations show the equality between the sum of output and the output checksums. Let $\mathbf{S}_{o1}$ and $\mathbf{S}_{o2}$ be the summation of the output, where $\mathbf{S}_{o1}[m] = \sum_{n=0}^{N-1} \mathbf{O}_{nm}$, $\mathbf{S}_{o2}[n] = \sum_{m=0}^{M-1} \mathbf{O}_{nm}$. We can compare $\mathbf{C}_{o1}$, $\mathbf{C}_{o2}$ with $\mathbf{S}_{o1}$, $\mathbf{S}_{o2}$ to detect, locate, and correct soft errors if they exist.

3.4 Row Checksum Scheme (RC)

Compared with the full checksum scheme, the second ABFT scheme we designed involves only the row checksums of output $\mathbf{O}$, so we call it *row checksum scheme*.

The row checksums used in this scheme are $\mathbf{C}_{o1}$ and $\mathbf{C}_{o3}$. $\mathbf{C}_{o3}$ is computed from convolution operation between $\mathbf{C}_{d2}$ and $\mathbf{W}$, and the related output summation is defined by $\mathbf{S}_{o3}[m] = \sum_{n=0}^{N-1} n \times \mathbf{O}_{nm}$.

For the detection of soft errors, we need to compare $\mathbf{C}_{o1}$ with $\mathbf{S}_{o1}$. If they are not equal to each other at location j, the error can be located by $i = \frac{\mathbf{C}_{o3}[j] - \mathbf{S}_{o3}[j]}{\mathbf{C}_{o1}[j] - \mathbf{S}_{o1}[j]}$ and j, and it can be corrected by adding $\mathbf{C}_{o1}[j] - \mathbf{S}_{o1}[j]$ to the block (i, j).

3.5 Column Checksum Scheme (ClC)

The third scheme we proposed is called *column checksum scheme* because it involves only the column checksums of output $\mathbf{O}$. The column checksums used in this scheme are $\mathbf{C}_{o2}$ and $\mathbf{C}_{o4}$. $\mathbf{C}_{o4}$ is defined by performing convolution operation between $\mathbf{D}$ and $\mathbf{C}_{w2}$, and the related output summation is defined as $\mathbf{S}_{o4}[n] = \sum_{m=0}^{M-1} m \times \mathbf{O}_{nm}$. To detect soft errors, we compare $\mathbf{C}_{o2}$ with $\mathbf{S}_{o2}$ first. If they are not equal to each other at location i, the error can be located by i and j ($=\frac{\mathbf{C}_{o4}[i] - \mathbf{S}_{o4}[i]}{\mathbf{C}_{o2}[i] - \mathbf{S}_{o2}[i]}$), and it can be recovered by adding $\mathbf{C}_{o2}[i] - \mathbf{S}_{o2}[i]$ to the block (i, j).

3.6 Checksum-of-Checksum Scheme (CoC/CoC-D)

Unlike the three schemes that all need $\mathbf{D}$ and/or $\mathbf{W}$ to calculate output checksums, the last scheme we proposed involves neither $\mathbf{D}$ nor $\mathbf{W}$ but only their checksums, so it is named *checksum-of-checksum scheme* (or CoC scheme for short). Specifically, C_{o5}, C_{o6}, and C_{o7} are the output checksums we will use in this scheme. Similar to $\mathbf{C}_{o1}$, using the distributive property can get three equations between the output checksums and output as follows.

$$\mathbf{C}_{o5} = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} \mathbf{O}_{nm} = \mathbf{S}_{o5}$$

$$\mathbf{C}_{o6} = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} m \times \mathbf{O}_{nm} = \mathbf{S}_{o6}$$

$$\mathbf{C}_{o7} = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} n \times \mathbf{O}_{nm} = \mathbf{S}_{o7}.$$

S_{o5}, S_{o6}, and S_{o7} are defined as output summations corresponding to C_{o5}, C_{o6}, and C_{o7}. Let $\mathbf{O}(i, j)$ be the corrupted output block, $\mathbf{O}'$ be the correct output, and let $\delta = \mathbf{O}'_{ij} - \mathbf{O}_{ij}$ be the difference. Using the output checksums, we can get the following.

$$\mathbf{C}_{o5} - \mathbf{S}_{o5} = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} \mathbf{O}'_{nm} - \mathbf{O}_{nm} = \delta$$

$$\mathbf{C}_{o6} - \mathbf{S}_{o6} = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} m \times (\mathbf{O}'_{nm} - \mathbf{O}_{nm}) = j \times \delta$$

$$\mathbf{C}_{o7} - \mathbf{S}_{o7} = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} n \times (\mathbf{O}'_{nm} - \mathbf{O}_{nm}) = i \times \delta.$$

The location i, j can be obtained by $i = (C_{o7} - S_{o7})/\delta$ and $j = (C_{o6} - S_{o6})/\delta$. Then the soft error can be fixed by adding δ to $\mathbf{O}_{ij}$.

If only soft error detection is required, we do not need to compute $\mathbf{C}_{o6}$ and $\mathbf{C}_{o7}$, thus reducing the number of computations. Input checksums regarding $\mathbf{C}_{d1}$, $\mathbf{C}_{d2}$ and $\mathbf{C}_{w1}$, $\mathbf{C}_{w2}$, however, are still required for soft error detection. We denote such a detection scheme by *CoC-D*.

4 Multischeme Workflow

In this section, we first discuss the fault protection abilities and runtime of the four schemes we proposed in Section 3. Then, we propose a multischeme workflow, powered by calibrated arrangement of the four schemes and layerwise optimization.

4.1 Analysis of Protection Ability for Convolution Checksum Schemes

In this section, we analyze the fault protection ability of all the schemes.

4.1.1 Fault Model

The fault model for soft errors that we discuss in this paper includes transient faults in computational units and data corruption faults (both transient and persistent) in memory (including cache). In the following text, we use *fault* to represent a malfunction event, and we denote its corresponding symptom as *soft error*.

Soft error protection includes error detection and error correction. Error detection means that the scheme can detect soft errors without knowing the exact location. Error correction means that the scheme can locate the soft error locations and recover the incorrect result.

Without loss of generality, in the following analysis we consider at most one fault per convolution. One convolutional neural network contains several or even tens of convolutional layers, and the total forward execution time of a CNN model is usually within seconds. Thus, we can reasonably assume that at most one fault may strike to one convolutional layer, considering the short executing time of a single layer. Multiple faults per convolution can also be detected by our schemes and recovered by recomputing the corrupted convolutional layer.

4.1.2 Analysis of Soft Error in D and W

One fault occurring during the convolution execution can result in multiple soft errors in $\mathbf{W}$ and $\mathbf{D}$. The soft errors in $\mathbf{W}$ can be detected by comparing the checksum of $\mathbf{W}$ with $\mathbf{C}_{w1}$ and corrected by reloading weights from the CNN model. The soft errors in $\mathbf{D}$ do not need correction because $\mathbf{D}$ will be discarded after convolution computation; the resulting errors in the output can be detected and corrected by the checksums of the output, as demonstrated below.

4.1.3 Analysis of Soft Error in O

One fault during the convolution execution can result in corruption of one block row or column of $\mathbf{O}$. By definition, the row i of $\mathbf{O}$ is computed by the ith block of $\mathbf{D}$ with W. Thus, one fault in $\mathbf{D}$ would result in at most one corrupted row. The column j of $\mathbf{O}$ is computed by $\mathbf{D}$ with the jth block of $\mathbf{W}$. Thus, one fault in $\mathbf{W}$ would result in at most one corrupted column. Moreover, the intermediate result will be reused only by the same row or column, such that one fault in the computational units would corrupt only values in the same row or column. Accordingly, in the following sections we discuss the soft error protection ability in the context of at most one corrupted row or column of $\mathbf{O}$.

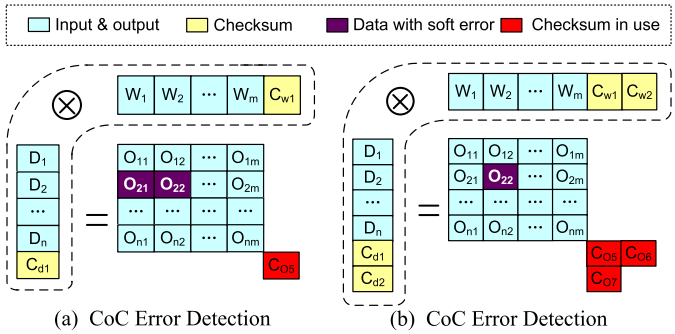


Fig. 2. Soft error protection ability of CoC scheme (soft error happens in inputs and outputs).

4.1.4 Soft Error Protection Ability of CoC Scheme

Fig. 2 demonstrates the protection ability of the CoC scheme when soft errors strike the input or output data. As shown in Fig. 2a, multiple soft errors can be detected by using only $\mathbf{C}_{o5}$. A single soft error in $\mathbf{O}$ can be corrected by CoC using all checksums including $\mathbf{C}_{o5}$, $\mathbf{C}_{o6}$, and $\mathbf{C}_{o7}$, as shown in Fig. 2b. However, CoC cannot correct soft errors across multiple blocks in $\mathbf{O}$.

Fig. 3 illustrates the protection ability of the CoC scheme when soft errors happen inside the checksums. Such soft errors can cause inconsistency among the output checksums of CoC, which can be used for error detection. For example, in Fig. 3a, $\mathbf{C}_{d1}$ is corrupted, leading to corrupted $\mathbf{C}_{o5}$ and $\mathbf{C}_{o6}$ with correct $\mathbf{C}_{o7}$. We can detect this abnormal pattern when comparing checksums with the summation of $\mathbf{O}$ to detect the input checksum corruption. The input $\mathbf{D}$, $\mathbf{W}$, and output $\mathbf{O}$ are clean and without soft errors since fault frequency is at most once per convolution. Thus, we can safely discard all the checksums and finish this convolution computation.

4.1.5 Soft Error Protection Ability of Row Checksum Scheme and Column Checksum Scheme

Since the row checksum scheme and column checksum scheme are symmetric with each other, we discuss them together in this section. As shown in Fig. 4a, the row checksum scheme can detect and correct soft errors if they are in the same row. If the soft errors are in the same column, as shown in Fig. 4b, the row checksum scheme can only detect soft errors; it has no correction ability. The column checksum scheme, on the contrary, can detect and correct errors located in the same column but fail to correct those appearing in the same row.

4.1.6 Soft Error Protection Ability of Full Checksum Scheme

The full checksum scheme has the highest ability to correct soft errors. The scheme uses both the row checksum $\mathbf{C}_{o1}$

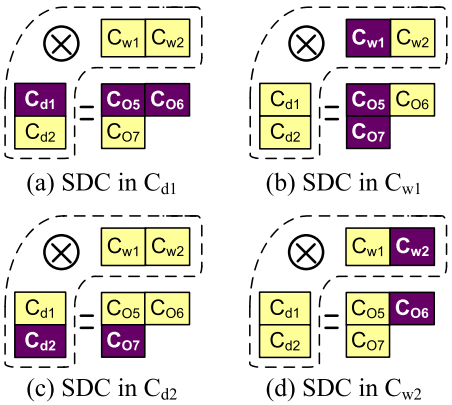


Fig. 3. Soft error protection ability of CoC scheme (soft error happens in checksums).

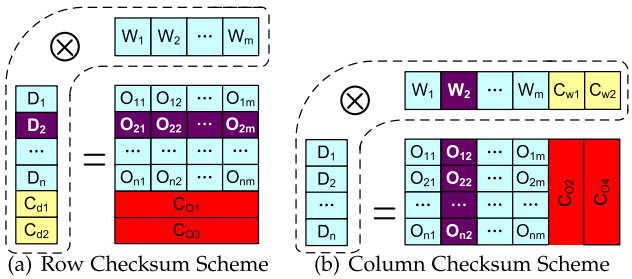


Fig. 4. Soft error protection ability of row/column checksum schemes.

and column checksum $\mathbf{C}_{o2}$ so that it can correct soft errors in both directions, as shown in Figs. 5a and 5b. If soft errors exist in $\mathbf{C}_{o1}$ (Fig. 5d), however, $\mathbf{C}_{o1}$ can no longer be used to locate or correct soft errors. To support error correction in this situation, we use checksum $\mathbf{C}_{o5}$ and $\mathbf{C}_{o6}$ from the CoC scheme to locate the corrupted column, and we then use $\mathbf{C}_{o2}$ to correct the soft errors. If soft errors exist in $\mathbf{C}_{o2}$ (Fig. 5c), $\mathbf{C}_{o5}$ and $\mathbf{C}_{o7}$ are used to locate the corrupted row, and $\mathbf{C}_{o1}$ is used to correct the soft errors.

4.1.7 Conclusion

In this section, we define our fault model and analyze the soft error protection ability of four schemes. We conclude that the CoC scheme has the lowest error correction ability and that the full checksum scheme has the best error correction ability. The abilities of the row checksum scheme and column checksum scheme are higher than that of the CoC scheme but lower than that of the full checksum scheme. CoC-D (discussed in Section 3.6) can detect multiple soft errors but without correction ability. The analysis here serves as the fundamental basis of our low-overhead high-protection design, which will be presented in Section 4.3.

4.2 Runtime Analysis

In this section, we analyze the time complexity theoretically and present runtimes of all schemes based on experiments.

Table 3 shows the time complexity of some basic checksum operations, where α is the coefficient of CPU-intensive operations and β represents the coefficient for memory-intensive operations.

Table 4 shows the theoretical time complexity of all the schemes. The full checksum scheme has the best soft error correction ability; however, its runtime is relatively long. Although the CoC scheme has lower ability than the other three schemes in correcting soft errors, it has the shortest runtime. Note that the kernel checksum $\mathbf{C}_{w1}$ and $\mathbf{C}_{w2}$ can be precalculated before the application; there is no cost in generating kernel checksum in the row, column, and CoC schemes.

To verify the correctness of the derived time complexity of the four schemes, we execute them on a supercomputer using four CNN models. We show the normalized worst-case runtime of the four schemes in the *separate* column of Fig. 6. Other columns of this Figure represent the worst-case runtime of multischeme workflows and will be discussed in the next section. Experiments confirm our conclusion that CoC and CoC-D have the shortest runtime and that the runtime of the full checksum scheme is relatively long. We also see that the column checksum scheme has a much longer runtime than the row checksum scheme does. The reason is twofold. On the one hand, $\mathbf{W}$ blocks have smaller sizes than $\mathbf{D}$ blocks have, leading to longer time to compute $\mathbf{D} \otimes \mathbf{C}_{w2}$

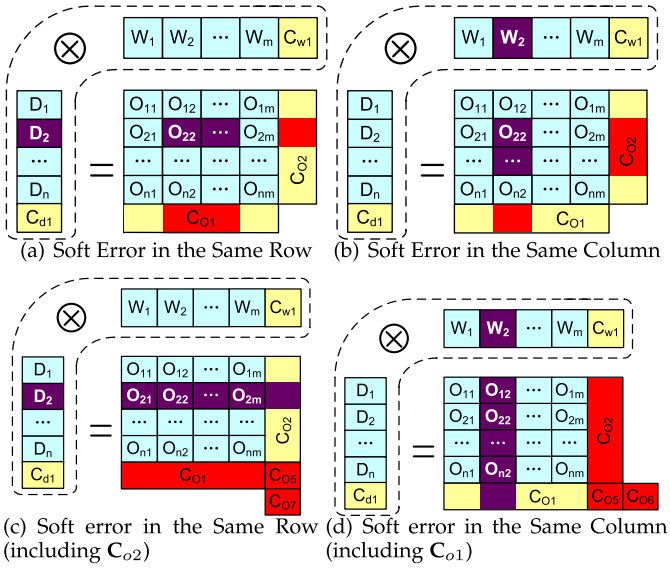


Fig. 5. Soft error protection ability of full checksum scheme.

TABLE 3
Runtimes of Basic Operations

Operation	Derived Runtime
block level convolution $\mathbf{D}_n \otimes \mathbf{W}_m$	αChR^2E^2
Total convolution operations	$\alpha NMChR^2E^2$
Compute the checksum of $\mathbf{D}$	$\beta NChH^2$
Compute the checksum of $\mathbf{O}$	βNME^2

TABLE 4
ABFT Schemes Runtime

Scheme	Derived Runtime	Soft Error Correction Ability
FC	$\alpha(N+M)ChR^2E^2 + \beta(NChH^2 + 2NME^2)$	High
RC	$2\alpha MChR^2E^2 + 2\beta(NChH^2 + NME^2)$	Middle
ClC	$2\alpha NChR^2E^2 + 2\beta(NME^2)$	Middle
CoC	$3\alpha ChR^2E^2 + \beta(2NChH^2 + 3NME^2)$	Low

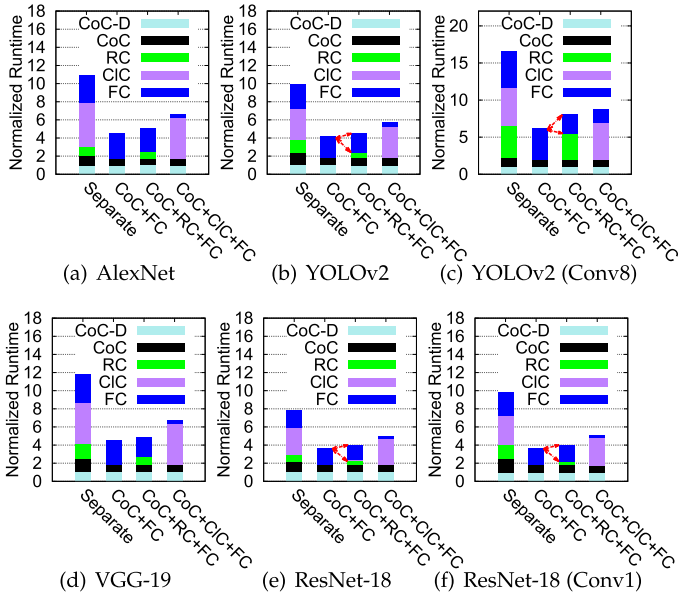


Fig. 6. Worst-case normalized runtime, baseline is CoC-D.

by the column checksum scheme than to compute $\mathbf{C}_{d2} \otimes \mathbf{W}$ by the row checksum scheme. On the other hand, computing row checksums ($\mathbf{C}_{o1}$ and $\mathbf{C}_{o3}$) is more efficient than computing column checksums ($\mathbf{C}_{o2}$ and $\mathbf{C}_{o4}$), because the row checksum calculation can be reduced to efficient column-summation operations.

4.3 Multischeme Workflow for Soft Error Protection

The four schemes we proposed have pros and cons in terms of their soft error correction ability and runtime overhead. To achieve the highest protection ability and lowest overhead, we propose a multischeme workflow by integrating the four schemes, as shown in Fig. 7. The workflow is made up of two modules: error detection and error correction. In our designed workflow, we use CoC-D to detect errors because it has the lowest overhead. For the error correction, we put CoC in the beginning because it is the most lightweight method. By comparison, FC has highest correction ability but also highest time overhead, so we put it at the end of the workflow.

The error detection modules will be executed for every execution whether there is a soft error or not. Thus, any unnecessary computations should be avoided in order to reduce the overall overhead. For instance, both CoC-D and FC are able to detect all the soft errors, but we adopt only CoC-D in the workflow for error detection because FC has a much higher overhead. RC and ClC cannot detect soft errors correctly if the checksum is corrupted.

The error correction module will not be executed until some soft errors are detected. The schemes in this module will be invoked to fix soft errors according to the workflow. If it fails to correct the errors due to inconsistency of checksum blocks or illegal error locations, the next-level scheme will be invoked.

Since the checksums can be reused among different CNN schemes in the workflow, the runtime of the workflow is actually lower than the sum of all schemes' runtimes. For example, both CoC-D and CoC use C_{o5}; if CoC-D detects soft errors and CoC is invoked to correct soft errors, CoC

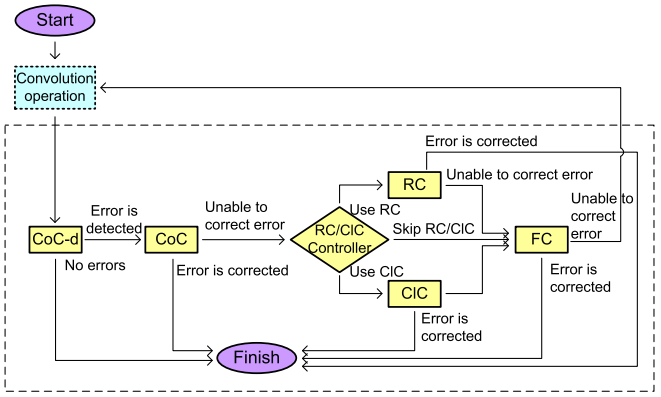


Fig. 7. Multischeme workflow designed to detect/correct soft errors.

can save the time of computing C_{o5} and its corresponding summation S_{o5}, since they have been computed by CoC-D. This analysis can also be confirmed by our experiments. As shown in Fig. 6, the relative runtime of CoC in the second column is reduced compared with that of CoC in the first column. The relative runtime of RC in the third column is reduced compared with that of RC in the first column.

The decision to put RC and ClC in the workflow between CoC and FC is controlled by each layer. The reason to control RC/ClC in each layer is that their relative runtimes differ across layers. Since RC and ClC are symmetric based, in the following we present our analysis based mainly on RC, without loss of generality.

We denote the runtime of the workflow CoC+FC as t_0, the runtime of workflow CoC+RC as t_1, and the runtime of workflow CoC+RC+FC as t_2. Enabling RC can fix some soft errors before FC, thus changing the runtime from t_0 to t_1. When RC fails to correct soft errors, however, FC still needs to be invoked; and the runtime will increase from t_0 to t_2. Denoting the probability of row soft errors by p_r and the probability of column soft errors by p_c, we can derive the average time saved by RC as $t_y = p_r(t_0 - t_1)$ and the average time increase by RC as $t_n = p_c(t_2 - t_0)$. In order to minimize the total runtime, RC should be enabled when $t_y > t_n$.

We give an example to further illustrate when RC should be enabled. Fig. 6b shows the average runtime among all the convolutional layers in YOLOv2. In this figure, the runtime of CoC+RC is much lower than that of CoC+FC, and the runtime of CoC+RC+FC is slightly higher than that of CoC+FC. Therefore, enabling RC can save significant runtime when the soft errors are able to be corrected by RC. On the other hand, a bit runtime penalty is incurred if RC fails to correct the soft errors. However, for the conv8 layer in YOLOv2 (shown in Fig. 6c), CoC+RC's runtime is close to that of CoC+FC. Thus, enabling RC in this layer would barely reduce the overall runtime even though the soft errors can be corrected by RC. Moreover, CoC+RC+FC's runtime is much higher than CoC+RC's. As a result, the total runtime will increase significantly if the soft errors cannot be corrected by RC. Hence, for this layer, it is better to use CoC+FC for error correction with RC disabled.

In practice, the runtime t_0, t_1 and t_2 can be computed by offline profiling. The probability values p_c and p_r can be estimated based on the size of $\mathbf{D}$ and size of $\mathbf{W}$. For instance, the soft error often strikes each element in the input under the independent and identical distribution. In this situation, it is

TABLE 5
Bias Adjustments for Output Checksums Comparison

Checksum	Adjust Summation
$\mathbf{C}_{o1}$	$\mathbf{S}_{o1}[m][i][j] - N \times Bias[m]$
$\mathbf{C}_{o3}$	$\mathbf{S}_{o3}[m][i][j] - (\sum_{i=1}^{N} i) \times Bias[m]$
$\mathbf{C}_{o2}$	$\mathbf{S}_{o2}[n][i][j] - \sum^{m} Bias[m]$
$\mathbf{C}_{o4}$	$\mathbf{S}_{o4}[n][i][j] - \sum^{m} m \times Bias[m]$
$\mathbf{C}_{o5}$	$\mathbf{S}_{o5}[i][j] - N \times \sum^{m} Bias[m]$
$\mathbf{C}_{o6}$	$\mathbf{S}_{o6}[i][j] - N \times \sum^{m} m \times Bias[m]$
$\mathbf{C}_{o7}$	$\mathbf{S}_{o7}[i][j] - (\sum_{i=1}^{N} i) \times \sum^{m} Bias[m]$

easy to drive that the probability of soft errors occurring in $\mathbf{D}$ is proportional to that of $\mathbf{W}$ (i.e., $\frac{p_r}{p_c} = \frac{number\ of\ elements\ in\ \mathbf{D}}{number\ of\ elements\ in\ \mathbf{W}}$.

5 Resolving Bias, Grouped Convolution, and Back Propagation

In this section, we extend our solution to support bias, grouped convolution, and the back propagation of convolutional layers.

5.1 Bias

Bias is a $1D$ vector that needs be added to the output of the convolutional layers. FT-Caffe provides protection for the bias operation.

Many CNN frameworks add bias on the fly with the convolution calculation. As a result, the output $\mathbf{O}$ already contains bias, whereas the output checksums do not contain bias since they are calculated by inputs and input checksums without bias. In order to compare the output checksums and the output $\mathbf{O}$, bias has to be subtracted from output summation before comparison. Subtracting bias from output $\mathbf{O}$ directly before verification and then adding bias to $\mathbf{O}$ after verification is not feasible, however, because of the overhead of modifying every element in $\mathbf{O}$. Table 5 shows the output checksums and adjusted output summation for comparison in order to detect errors. The bias part of the formulations can be precomputed.

5.2 ABFT for Grouped Convolution

Grouped convolution is a special kind of convolution. Our schemes need to be modified to support this convolution. Define the number of groups as G. Each fmap basic block has $\frac{Ch}{G}$ instead of Ch channels. All the M kernel basic blocks are divided into G groups, each having $\frac{M}{G}$ $3D$ basic blocks. The kernel block in the gth group does convolution only with the gth channel group of every fmap block. Fig. 8 shows this process for N=2, M=4, and G=2.

The checksums for fmap $\mathbf{C}_{d1}$ and $\mathbf{C}_{d2}$ stay the same. The checksum for kernel are redefined as

$$\mathbf{C}_{w1} = \left[\sum_{m=0}^{\frac{M}{G}-1} \mathbf{W}_m, \sum_{m=\frac{M}{G}}^{2\frac{M}{G}-1} \mathbf{W}_m, \ldots, \sum_{m=(G-1)\frac{M}{G}}^{M-1} \mathbf{W}_m\right]$$

$$\mathbf{C}_{w2} = \left[\sum_{m=0}^{\frac{M}{G}-1} m \times \mathbf{W}_m, \sum_{m=\frac{M}{G}}^{2\frac{M}{G}-1} m \times \mathbf{W}_m, \ldots, \sum_{m=(G-1)\frac{M}{G}}^{M-1} m \times \mathbf{W}_m\right],$$

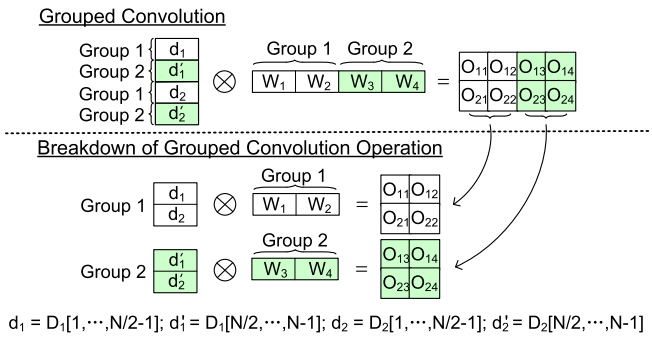


Fig. 8. Demonstration of grouped convolution, groups = 2.

where $\mathbf{C}_{w1}$ and $\mathbf{C}_{w2}$ are the combination of G checksums from each kernel group. Each checksum has $\frac{Ch}{G}$ channels, so $\mathbf{Ch}_{w1}$ and $\mathbf{Ch}_{w1}$ each have $G\frac{Ch}{G} = Ch$ channels, which are the same with every fmap block.

The definition of output checksums $\mathbf{C}_{o1}$, $\mathbf{C}_{o2}, \ldots, C_{o7}$ stays the same. Let $X[l..r]$ represent the channels from l to r in matrix X. We can prove the following property for any $\mathbf{D}_n$ and $\mathbf{W}_m$ according to Equation (1).

$$\mathbf{D}_n \otimes \mathbf{W}_m = \mathbf{D}_n[1..k-1] \otimes \mathbf{W}_m[1..k-1] + \mathbf{D}_n[k..Ch] \otimes \mathbf{W}_m[k..Ch], 0 \leq k < Ch.$$

Using this equation, we can prove the relation between $\mathbf{C}_{o2}$ and $\mathbf{O}$ as follows.

$$\mathbf{C}_{o2}[n] = \mathbf{D}_n \otimes \left[\sum_{m=0}^{\frac{M}{G}-1} \mathbf{W}_m, \sum_{m=\frac{M}{G}}^{2\frac{M}{G}-1} \mathbf{W}_m, \ldots, \sum_{m=(G-1)\frac{M}{G}}^{M-1} \mathbf{w}_m\right]$$

$$= \mathbf{D}_n[0..\frac{C}{G} - 1] \otimes \sum_{m=0}^{\frac{M}{G}-1} \mathbf{w}_m + \mathbf{D}_n\left[\frac{C}{G}..\frac{2C}{G} - 1\right] \otimes \sum_{m=\frac{M}{G}}^{\frac{2M}{G}-1} \mathbf{w}_m$$

$$+ \ldots + \mathbf{D}_n\left[\frac{(G-1)C}{G}..C-1\right] \otimes \sum_{m=\frac{(G-1)M}{G}}^{M-1} \mathbf{w}_m$$

$$= \sum_{m=0}^{M-1} \mathbf{D}_n \otimes \mathbf{w}_m = \sum_{m=0}^{M-1} \mathbf{O}_{nm}.$$

Similar equations can be proved for $\mathbf{C}_{o1}, \mathbf{C}_{o3}, \mathbf{C}_{o4}$, C_{o5}, C_{o6}, and C_{o7}. Therefore, all the ABFT schemes we proposed can be applied to grouped convolution.

5.3 ABFT for Convolution Back Propagation

Our schemes can also be applied to back propagation together with forward pass so that the convolutional layers can be fully protected in the training phase.

During back propagation, the gradient of kernel $\nabla\mathbf{W}$ is used by methods such as gradient descent in order to update $\mathbf{W}$. The gradient of fmap $\nabla\mathbf{D}$ is used to get $\nabla\mathbf{O}$ of the previous layer. As shown in Fig. 9, the gradients are calculated as $\mathbf{D} \otimes \nabla\mathbf{O} = \nabla\mathbf{W}$ and $\mathbf{W}^T \otimes \nabla\mathbf{O} = \nabla\mathbf{D}$. Checksums for $\nabla\mathbf{O}$ are used in this situation to protect the two convolution operations.

Since CNN models are usually trained in a more stable environment than the inference stage and since the training stage can tolerate some soft errors because of their iterative-convergent nature, we focus our experiments on the inference stage.

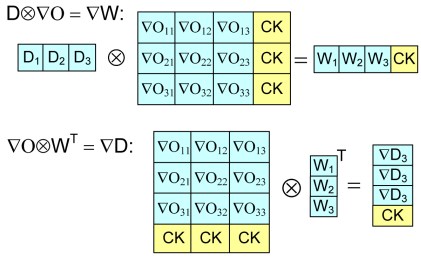


Fig. 9. Demonstration of checksum design for back propagation.

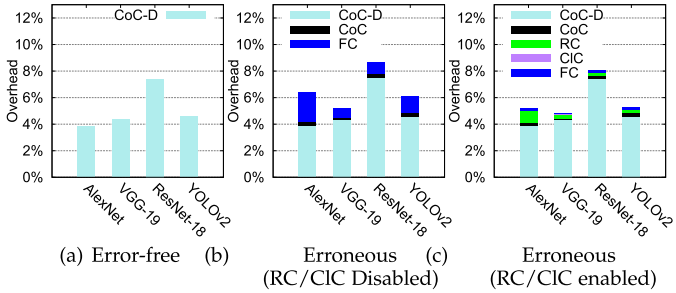


Fig. 10. Runtime overhead with MKL-DNN.

6 Experimental Evaluation

In this section, we evaluate our multischeme workflow using our FT-Caffe fault tolerance CNN framework.

6.1 Experimental Setup

FT-Caffe. Our FT-Caffe framework is based on Intel-Caffe. MKL-DNN is enabled to support dynamic selection of convolution execution. MKL-DNN contains all the convolution implementations we discussed in Section 2.2. It automatically chooses the most suitable implementation to use for each convolutional layer. To compare the runtime overhead of our solution with that of the ABFT designed for matrix-matrix multiplication, we also perform the experiments based on the MM-based convolution implementation.

CNN Models and Dataset. We tested our FT-Caffe with four widely used networks: AlexNet, VGG-19, ResNet-18, and YOLOv2. Pretrained Caffe models are used together with model prototypes for deployment. We adopt the ImageNet validation set, which contains 50k images. The images are preprocessed to smaller size in order to save picture processing time when the program starts. The batch size is set to 64.

Experimental Platforms. We conducted our experiments on the Bebop supercomputer [43] at Argonne National Laboratory using up to 128 nodes. Each node is equipped with 128 GB memory and two Intel Xeon E5-2695 v4 processors (each with 16 cores)

Error injection. To demonstrate the overhead of our fault tolerant solutions, we inject soft errors at the source code level as most ABFT works did [29], [36]. The consequences of one computational fault or memory fault are simulated by randomly corrupting selected row or column of output. We denote the total number of convolutional layers of a CNN model as L. To assess the overhead accurately, we run the experiments for L epochs corresponding to the numbers of convolutional layers of each network (L= 5, 9, 16, 21 for AlexNet, YOLOv2, VGG-19, and ResNet-18, respectively). For the ith epoch, we inject errors to ith convolutional layer. The final overhead is the arithmetic mean of all the inference executions and the standard deviation in our experiments is within 5 percent.

6.2 Experimental Results With MKL-DNN

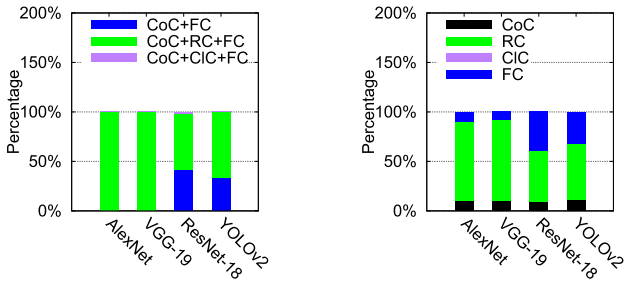
In this section, we present our evaluation results with MKL-DNN . We analyze the results from the perspective of execution time overhead for both error-free cases and erroneous cases.

Error-Free Cases. The experimental results in the error-free cases are presented in Fig. 10a. We can see from the figure that our FT-caffe can protect the inference stage with less than 4, 4.5, 8, and 5 percent overhead for AlexNet, VGG-19, YOLOv2, and ResNet-18, respectively, regardless of the convolution implementation. These results show that our CoC-D error detection scheme has relatively short runtime compared with the convolution execution, which is attributed to the design of avoiding unnecessary computations in our solution (see Section 4.3). The reason ResNet-18 has higher overhead than the other models have is that the ResNet-18 has small convolution kernels (**W** size is $M \times C \times 3 \times 3$) in all the convolutional layers, which have relatively short computing time; thus, the checksum computation and verification time percentage would be relatively large.

Erroneous Cases – RC/ClC Disabled. To show the effectiveness of our layerwise optimization for RC/ClC, we first test our multischeme workflow with RC/ClC disabled in erroneous cases. Fig. 10b demonstrates that the runtime overheads (including both error detection and error correction) of the four CNN models are all below 9 percent. The error detection overhead is higher than the error correction overhead because the error detection scheme is executed for every convolution operation whereas the error correction schemes are invoked only when errors are detected. The full checksum scheme dominates the error correction overhead, thus confirming our analysis in Section 4 that FC has high protection ability and relatively longer runtime.

Erroneous Cases – Layerwise RC/ClC Optimization. Fig. 10c demonstrates the runtime overhead with layerwise optimization enabled. Every layer decides whether to use RC/ClC independently, as described in Section 4.3. Compared with Fig. 10b, the error correction overhead decreases by 40%∼60% (e.g., 1.55% → 0.72% for YOLOv2 as shown in Fig. 10b versus (c)) in all CNN models because of the effectiveness of RC. Fig. 11a shows the distribution of varies workflows that is the result of layerwise RC/ClC optimization. We can see that RC is enabled in all layers of AlexNet and VGG-19, while it is disabled in 30 to 40 percent of layers in ResNet-18 and YOLOv2. The results demonstrate the need for layerwise RC optimization since RC is not suitable for all layers in the same CNN model. Fig. 11b shows the distribution of soft errors by the schemes that correct them. Less than 5 percent of soft errors are corrected by CoC because of the low correction ability of CoC. RC corrects nearly 90 percent of the soft errors in AlexNet and VGG-19 because RC is enabled in all layers of the two CNN models and the probability of soft errors striking a row in **O** is higher than the probability of soft errors striking a column.

Erroneous Cases – Breakdown of Error Correction Overhead by Layer. To better illustrate the overhead of our solution for



(a) Distribution of different workflows (b) Distribution of soft errors corrected by schemes

Fig. 11. Breakdown analysis of multischeme workflow with MKL-DNN.

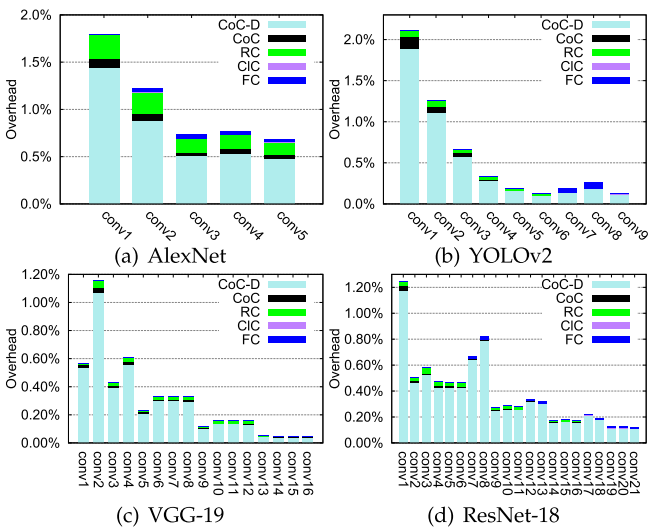


Fig. 12. Breakdown of runtime overhead by layer with MKL-DNN.

each model, we present in Fig. 12 the breakdown of the overhead by layer. The figure demonstrates that layers have diverse overheads of error protection due to the different shapes of **D** and **W**. We also notice that the overhead of RC differs among layers in the same model, thus confirming the functionality of our layerwise RC/ClC optimization.

6.3 Experimental Results With MM-Based Convolution

In this section, we evaluate the runtime overhead of our multischeme workflow and the traditional MM-based ABFT. Since the MM-based ABFT supports only the MM-based convolution implementation, we set the convolution implementation to the MM-based mode in MKL-DNN. We implemented MM-based ABFT rigorously based on [26], which has $\leq$1% overhead for large and square matrices as claimed by the authors of that work. The overhead of the MM-based ABFT in convolution execution is shown in Table 6. The MM-based ABFT incurs up to 60 percent overhead even without error injection for the four CNN models. This result is consistent with our analysis in Section 2.3. Considering that the MM-based ABFT cannot protect the whole process of MM-based convolution and cannot protect other convolution implementations, we conclude that the MM-based ABFT is unsuitable for soft error protection of CNN applications.

Fig. 13 shows the overhead of our multischeme workflow for MM-based convolution. The overhead of our solution is

TABLE 6
Overhead of MM-Based ABFT for MM-Based Convolution, No Error Injection

Model	AlexNet	YOLOv2	VGG-19	ResNet-18
Overhead	27.9%	57.5%	45.8%	61.2%

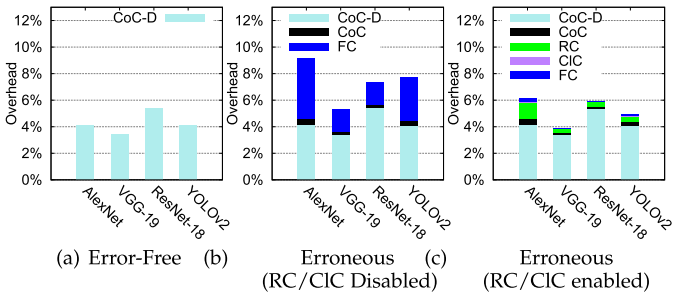
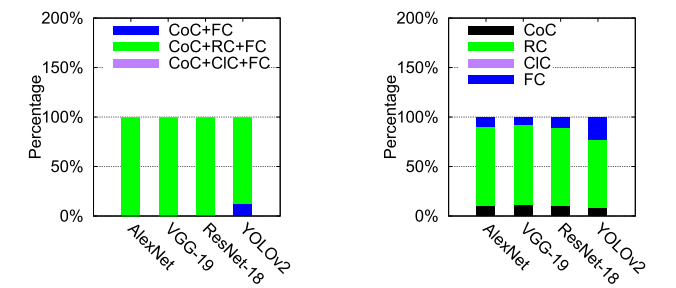


Fig. 13. Runtime overhead with MM-based convolution.



(a) Distribution of different workflows (b) Distribution of soft errors corrected by schemes

Fig. 14. Breakdown analysis of multischeme workflow with MM-based convolution.

below 6 percent in the error-free cases and below 6.5 percent in the cases with injected errors for all CNN models. The layerwise RC/ClC optimization reduces the overhead for error correction by as much as 77 percent. Fig. 14a shows the fractions of different workflows chosen by convolutional layers. Fig. 14b shows the distribution of soft errors that are corrected by different schemes. Compared with the MKL-DNN implementation, more layers adopt RC for error correction in the MM-based convolution (see Fig. 11 versus Fig. 14). The reason is that the relative runtime of RC compared with FC is lower in the MM-based convolution implementation than other implementations.

6.4 Parallel Performance Evaluation

In this section, we present the parallel performance evaluation results of AlexNet, YOLOv2, VGG-19, and ResNet-18. Original images of the ImageNet validation dataset are used without preprocessing in order to better demonstrate the process of parallel CNN inference application. In the beginning of the parallel process, images are distributed to the local disk of each node; then each node starts to do the data processing step first to convert the images to suitable size required by CNN models, and then execute the inference step under the protection of our multischeme workflow.

We conducted the parallel evaluation in both error-free and erroneous cases. However, because of space limits, we

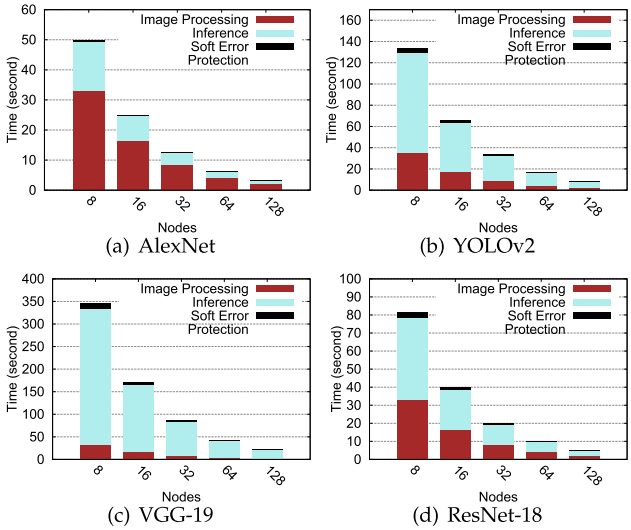


Fig. 15. Parallel performance evaluation of our solution with injected errors on bebop supercomputer.

present only the parallel performance evaluation results in the situation with injected errors (as shown in Fig. 15). In fact, the evaluation results in the error-free situation are similar. Specifically, experiments show that our multischeme workflow has a very good scalability: that is, the soft error protection overhead does not increase with the number of nodes at all. In absolute terms, the overhead stays around 2%∼6% in the erroneous cases and is only 1%∼4% in the error-free cases.

7 Related Work

The importance of fault tolerance for convolution has been emerging in recent years. Guaranteeing the correctness of inference is vital in a safety-critical use case [19]. To achieve better resiliency for CNN networks, researchers have been exploring solutions from different perspectives including hardware, system, and software. For hardware, Kim *et al.* [49] proposed a hardened 3D die-stacked memory based on the fault characteristics in convolutional DNNs. Li *et al.* [19] proposed to add redundant circuits selectively to harden the latches based on analysis of data resiliency. Compared with traditional full-hardware redundancy techniques, those partial-hardware redundancy techniques may not double the power usage. However, hardware modification incurs significant effort considering the varied CNN models and their accelerators. At the system level, other than the DMR/TMR protection, checkpoint/restart (C/R) is also applied to large-scale machine learning systems. Subsequently, Qiao *et al.* proposed a more efficient C/R scheme based on their derived upper bound on extra iteration cost with perturbations [50]. While those C/R techniques are promising to protect model training from soft errors, they are not good fits for inference since one inference execution could be very fast and applying C/R incurs significant overhead. Researchers have therefore pursued lightweight software-level solutions. By applying ABFT techniques for MM-based convolution, Santos *et al.* [20] reported that 50%∼60% of radiation-induced corruptions could be corrected. Unfortunately, the traditional ABFT works only for MM-based convolution, which is inefficient in most cases. In contrast, our solutions can work for any convolution implementations.

8 Conclusion and Future Work

This work focus on extending ABFT to convolution operations in convolutional neural networks. We propose four ABFT schemes and a multischeme workflow to protect the convolutional layer. We further extend our schemes to support bias, grouped convolution, and convolution back propagation. We implement an efficient CNN framework, FT-Caffe, that is resilient to silent data corruption.

Experiments demonstrate that our proposed fault-tolerant solutions incur negligible overhead. In absolute terms, FT-Caffe can acheive less than 8 percent overhead for the most widely used CNN models, including AlexNet, YOLO, VGG-19, and ResNet-18, in both error-free and erroneous cases.

We plan to extend the implementation to more CNN frameworks and to design architecture-specific optimizations for different hardware including GPU, FPGA, and AI accelerators.

Acknowledgments

This work was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations - the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation's exascale computing imperative. The material was supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357. This work was also supported by the National Science Foundation under Grants CCF-1513201, CCF-1619253, and OAC-2034169. The authors would like to acknowledge the computing resources provided on Bebop, which is operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

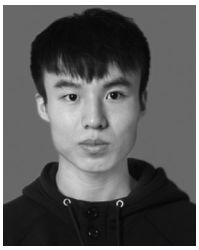
References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.
[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[4] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 7263–7271.
[5] Y. Goldberg, "Neural network methods for natural language processing," *Synthesis Lectures Hum. Lang. Technol.*, vol. 10, no. 1, pp. 1–309, 2017.
[6] S.-C. B. Lo, H.-P. Chan, J.-S. Lin, H. Li, M. T. Freedman, and S. K. Mun, "Artificial convolution neural network for medical image pattern recognition," *Neural Netw.*, vol. 8, no. 7/8, pp. 1201–1214, 1995.
[7] E. Gawehn, J. A. Hiss, and G. Schneider, "Deep learning in drug discovery," *Mol. Informat.*, vol. 35, no. 1, pp. 3–14, 2016.
[8] J. M. Wozniak *et al.*, "Candle/supervisor: A workflow framework for machine learning applied to cancer research," *BMC Bioinf.*, vol. 19, no. 18, 2018, Art. no. 491.
[9] J. J. Zhang *et al.*, "Building robust machine learning systems: Current progress, research challenges, and opportunities," in *Proc. 56th Annu. Des. Autom. Conf.*, 2019, Art. no. 175.
[10] M. T. Le, F. Diehl, T. Brunner, and A. Knol, "Uncertainty estimation for deep neural object detectors in safety-critical applications," in *Proc. 21st Int. Conf. Intell. Transp. Syst.*, 2018, pp. 3873–3878.
[11] S. Burton, L. Gauerhof, and C. Heinzemann, "Making the case for safety of machine learning in highly automated driving," in *Proc. Int. Conf. Comput. Saf. Rel. Secur.*, 2017, pp. 5–16.

[12] M. Snir *et al.*, "Addressing failures in exascale computing," *Int. J. High Perform. Comput. Appl.*, vol. 28, pp. 129–173, 2014.
[13] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, and S. McIntosh-Smith, "Unprotected computing: A large-scale study of DRAM raw error rate on a supercomputer," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 645–655.
[14] L. Tan, S. L. Song, P. Wu, Z. Chen, R. Ge, and D. J. Kerbyson, "Investigating the interplay between energy efficiency and resilience in high performance computing," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 786–796.
[15] J. P. Walters, K. M. Zick, and M. French, "A practical characterization of a NASA SpaceCube application through fault emulation and laser testing," in *Proc. 43rd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2013, pp. 1–8.
[16] A. Geist, "How to kill a supercomputer: Dirty power, cosmic rays, and bad solder," *IEEE Spectrum*, vol. 10, pp. 2–3, 2016.
[17] J. Zhang, K. Rangineni, Z. Ghodsi, and S. Garg, "Thundervolt: Enabling aggressive voltage underscaling and timing error resilience for energy efficient deep learning accelerators," in *Proc. 55th Annu. Des. Autom. Conf.*, 2018, Art. no. 19.
[18] W. Choi, D. Shin, J. Park, and S. Ghosh, "Sensitivity based error resilient techniques for energy efficient deep neural network accelerators," in *Proc. 56th Annu. Des. Autom. Conf.*, 2019, pp. 1–6.
[19] 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.*, 2017, pp. 8:1–8:12.
[20] F. F. D. Santos, L. Draghetti, L. Weigel, L. Carro, P. Navaux, and P. Rech, "Evaluation and mitigation of soft-errors in neural network-based object detection in three GPU architectures," in *Proc. 47th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. Workshops*, 2017, pp. 169–176.
[21] B. Reagen *et al.*, "Ares: A framework for quantifying the resilience of deep neural networks," in *Proc. 55th Annu. Des. Autom. Conf.*, 2018, pp. 17:1–17:6.
[22] R. Salay, R. Queiroz, and K. Czarnecki, "An analysis of ISO 26262: Using machine learning safely in automotive software," 2017, *arXiv: 1709.02435*.
[23] D. Li, Z. Chen, P. Wu, and J. S. Vetter, "Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2013, pp. 1–12.
[24] X. Vera, J. Abella, J. Carretero, and A. González, "Selective replication: A lightweight technique for soft errors," *ACM Trans. Comput. Syst.*, vol. 27, no. 4, 2009, Art. no. 8.
[25] 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.
[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.*, 2016, pp. 31–42.
[27] P. Wu, D. Li, Z. Chen, J. S. Vetter, and S. Mittal, "Algorithm-directed data placement in explicitly managed non-volatile memory," in *Proc. 25th ACM Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2016, pp. 141–152.
[28] L. Tan, S. Kothapalli, L. Chen, O. Hussaini, R. Bissiri, and Z. Chen, "A survey of power and energy efficient techniques for high performance numerical linear algebra operations," *Parallel Comput.*, vol. 40, no. 10, pp. 559–573, 2014.
[29] J. Chen *et al.*, "Fault tolerant one-sided matrix decompositions on heterogeneous systems with GPUs," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2018, pp. 68:1–68:12.
[30] J. Chen *et al.*, "TSM2: Optimizing tall-and-skinny matrix-matrix multiplication on GPUs," in *Proc. ACM Int. Conf. Supercomputing*, 2019, pp. 106–116.
[31] C. Rivera, J. Chen, N. Xiong, S. L. Song, and D. Tao, "TSM2X: High-performance tall-and-skinny matrix-matrix multiplication on GPUs," 2020, *arXiv:2002.03258*.
[32] Z. Chen, "Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods," in *Proc. 18th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2013, pp. 167–176.
[33] D. Tao *et al.*, "New-sum: A novel online ABFT scheme for general iterative methods," in *Proc. 25th ACM Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2016, pp. 43–55.
[34] D. Tao, S. Di, X. Liang, Z. Chen, and F. Cappello, "Improving performance of iterative methods by lossy checkponting," in *Proc. 27th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2018, pp. 52–65.
[35] D. Tao, "Fault tolerance for iterative methods in high-performance computing," PhD dissertation, Dept. Comput. Sci., Univ. California Riverside, Riverside, CA, 2018.
[36] X. Liang *et al.*, "Correcting soft errors online in fast fourier transform," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, pp. 30:1–30:12.
[37] S. Li *et al.*, "FT-iSort: Efficient fault tolerance for introsort," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2019, Art. no. 71.
[38] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *Proc. Int. Conf. Artif. Neural Netw. Mach. Learn.*, 2014, pp. 281–290.
[39] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.
[40] S. Jin, S. Di, X. Liang, J. Tian, D. Tao, and F. Cappello, "DeepSZ: A novel framework to compress deep neural networks by using error-bounded lossy compression," in *Proc. 28th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2019, pp. 159–170.
[41] Z. Hu *et al.*, "Delta-DNN: Efficiently compressing deep neural networks via exploiting floats similarity," in *Proc. 49th Int. Conf. Parallel Process.*, 2020, Art. no. 40.
[42] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image satabase," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.
[43] Bebop supercomputer. 2019. [Online]. Available: https://www.lcrc.anl.gov/systems/resources/bebop
[44] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 4013–4021.
[45] S. Chetlur *et al.*, "cuDNN: Efficient primitives for deep learning," 2014, *arXiv:1410.0759*.
[46] T. Chen *et al.*, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. 19th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2014, pp. 269–284.
[47] NVIDIA, 2019. [Online]. Available: http://nvdla.org.
[48] S. Liu, Q. Wang, and G. Liu, "A versatile method of discrete convolution and FFT (DC-FFT) for contact analyses," *Wear*, vol. 243, no. 1, pp. 101–111, 2000.
[49] J.-S. Kim and J.-S. Yang, "DRIS-3: Deep neural network reliability improvement scheme in 3D die-stacked memory based on fault analysis," in *Proc. 56th Annu. Des. Autom. Conf.*, 2019, pp. 129:1–129:6.
[50] A. Qiao, B. Aragam, B. Zhang, and E. Xing, "Fault tolerance in iterative-convergent machine learning," in *Proc. 36th Int. Conf. Mach. Learn.*, 2019, pp. 5220–5230.

Kai Zhao received the bachelor's degree from Peking University, China, in 2014. He is currently working toward the PhD degree at the University of California, Riverside, Riverside, California. He is a long-term intern at Argonne National Laboratory, Lemont, Illinois. His research interests include high-performance computing, scientific data management and reduction, and resilient machine learning.

Sheng Di (Senior Member, IEEE) received the master's degree from the Huazhong University of Science and Technology, China, in 2007, and the PhD degree from the University of Hong Kong, Hong Kong, in 2011. He is currently a computer scientist at Argonne National Laboratory, Lemont, Illinois. His research interests involve resilience on high-performance computing (such as silent data corruption, optimization checkpoint model, and in-situ data compression) and broad research topics on cloud computing (including optimization of resource allocation, cloud network topology, and prediction of cloud workload/hostload). He is working on multiple HPC projects, such as detection of silent data corruption, characterization of failures and faults for HPC systems, and optimization of multilevel checkpoint models.



Sihuan Li (Student Member, IEEE) received the bachelor's degree in math from the Huazhong University of Science and Technology, China. He is currently working toward the PhD degree in computer science at the University of California, Riverside, Riverside, California. He did a long-term internship at Argonne National Laboratory, Lemont, Illinois. Broadly speaking, his research interests fall into high performance computing. Specifically, he mainly studies algorithm-based fault tolerance (ABFT), lossy compression, and their applications in large scale scientific simulations.

Xin Liang (Member, IEEE) received the bachelor's degree from Peking University, China, in 2014, and the PhD degree from the University of California, Riverside, Riverside, California, in 2019. He is a computer/data scientist at Oak Ridge National Laboratory, Oak Ridge, Tennessee. His research interests include high-performance computing, parallel and distributed systems, scientific data management and reduction, big data analytic, scientific visualization, and cloud computing. He has interned in multiple national laboratories and worked on several exascale computing projects.

Yujia Zhai received the bachelor's degree from the University of Science and Technology of China, China, in 2016, the master's degree from Duke University, Durham, North Carolina, in 2018. He is currently working toward the PhD degree at the University of California, Riverside, Riverside, California. His research interests include high-performance computing, parallel and distributed systems, and numerical linear algebra software.

Jieyang Chen (Member, IEEE) received the bachelor's degree in computer science and engineering from the Beijing University of Technology, China, in 2012, and the master's and PhD degrees in computer science from the University of California, Riverside, Riverside, California, in 2014 and 2019, respectively. He is a computer scientist at Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee. His research interests include high-performance computing, parallel and distributed systems, and big data analytics.

Kaiming Ouyang received the bachelor's degree from the University of Electronic Science and Technology of China, China, and joined the University of California, Riverside SuperLab, Riverside, California, in Fall 2016, where he is currently working toward the PhD degree. He is a long-term intern at Argonne National Laboratory PMRS Group led by Dr. Balaji and supervised by Dr. Si. His research interest includes parallel runtime system.

Franck Cappello (Fellow, IEEE) is the director of the Joint-Laboratory on Extreme Scale Computing gathering six of the leading high-performance computing institutions in the world: Argonne National Laboratory, National Center for Scientific Applications, Inria, Barcelona Supercomputing Center, Julich Supercomputing Center, and Riken AICS. He is a senior computer scientist at Argonne National Laboratory, Lemont, Illinois and an adjunct associate professor with the Department of Computer Science, University of Illinois at Urbana-Champaign, Champaign, Illinois. He is an expert in resilience and fault tolerance for scientific computing and data analytics. Recently he started investigating lossy compression for scientific data sets to respond to the pressing needs of scientist performing large-scale simulations and experiments. His contribution to this domain is one of the best lossy compressors for scientific data set respecting user-set error bounds. He is a member of the editorial board of the *IEEE Transactions on Parallel and Distributed Computing* and of the ACM HPDC, and EEE CCGRID steering committees.

Zizhong Chen (Senior Member, IEEE) received the bachelor's degree in mathematics from Beijing Normal University, China, the master's degree in economics from the Renmin University of China, China, and the PhD degree in computer science from the University of Tennessee, Knoxville, Tennessee. He is a professor of computer science at the University of California, Riverside, Riverside, California. His research interests include high-performance computing, parallel and distributed systems, big data analytics, cluster and cloud computing, algorithm-based fault tolerance, power and energy efficient computing, numerical algorithms and software, and large-scale computer simulations. His research has been supported by National Science Foundation, Department of Energy, CMG Reservoir Simulation Foundation, Abu Dhabi National Oil Company, Nvidia, and Microsoft Corporation. He has received a CAREER Award from the US National Science Foundation and a Best Paper Award from the International Supercomputing Conference. He is a Life Member of ACM. He currently serves as a subject area editor for Elsevier *Parallel Computing Journal* and an associate editor for the *IEEE Transactions on Parallel and Distributed Systems*.