Core Failure Mitigation In Integer Sum-of-Product Computations On Cloud Computing Systems

Ijeoma Anarado and Yiannis Andreopoulos*, *Senior Member, IEEE*

***Abstract*—The decreasing mean-time-to-failure estimates in cloud computing systems indicate that multimedia applications running on such environments should be able to mitigate an increasing number of core failures at runtime. In this paper, we propose a new *roll-forward* failure-mitigation approach for integer sum-of-product computations, with special emphasis on high-performance generic matrix multiplication (GEMM) and convolution/cross-correlation (CONV) routines. Our approach is based on the production of redundant results *within* the numerical representation of the outputs via the use of numerical packing. This differs from all existing roll-forward solutions that require a separate set of checksum (or duplicate) results. Our proposal imposes 37.5% reduction in the maximum output bitwidth supported in comparison to integer GEMM or CONV realizations performed on 32-bit integer representations. However, this bitwidth reduction is comparable to the one imposed due to the checksum elements of traditional roll-forward methods, especially for cases where multiple core failures must be mitigated. Experiments with state-of-the-art GEMM and CONV routines running on a c4.8xlarge (shared-memory, 18-core) compute-optimized instance of Amazon Web Services Elastic Compute Cloud (AWS EC2) demonstrate that the proposed approach is able to mitigate up to one quadcore failure while achieving processing throughput that is: *(i)* comparable to that of the conventional, failure-intolerant, integer GEMM and CONV routines; *(ii)* substantially superior to that of the equivalent roll-forward failure-mitigation method based on checksum streams. Furthermore, when used within an image retrieval framework deployed over a cluster of AWS EC2 spot (i.e., low-cost albeit terminatable) instances, our proposal leads to: *(i)* 16%–23% cost reduction against the equivalent checksum-based method and *(ii)* more than 70% cost reduction against conventional failure-intolerant processing based on AWS EC2 on-demand (i.e., higher-cost albeit guaranteed) instances.**

***Index Terms*—integer matrix products, convolution, core failures, multimedia cloud computing**

I. Introduction

CLOUD computing clusters today provide for significant parallelism possibilities at the cost of decreased mean-time-to-failure (MTTF) characteristics in comparison to conventional desktop multicore systems [1], [2]. For example, high-performance clusters can now be deployed using Amazon Web Services Elastic Compute Cloud (AWS EC2) spot instances with substantially-reduced billing cost [3]. However, AWS reserves the right to terminate EC2 spot instances at any moment with little or no prior notice. In addition, service interruptions may occur at unpredictable intervals, since processor cores in spot instance reservations may not be solely dedicated to the cluster under consideration. More broadly, other types of disruptions, such as vibration, power, or network-induced performance degradations, are also frequently reported in large computing clusters [2], [4]. Such interruptions tend to result in substantial reduction in processing throughput and are therefore extremely detrimental in the performance of high-volume, low-latency, multimedia applications on cloud computing clusters [5], [6]. Beyond such cluster-level threats, at the processor core level, the increased integration density, process variations, hardware component reuse, and inadvertent circuit overclocking or undervolting also contribute to increasingly-lower MTTF per core [7]. Therefore, multimedia applications requiring data-intensive and high-throughput processing (e.g., webpage or multimedia retrieval [8], [9], relevance ranking [10] and object or face recognition in video [11], [12]) on such clusters are now prone to core failures, which incur their data unrecoverable, occurring at increasing rates. Within all these applications, the compute and memory-intensive parts comprise large *sum-of-product computations*, i.e., inner and outer products, generic matrix multiplication (GEMM) [13]–[19], and multidimensional convolution/cross-correlation (CONV) operations [20], [21]. These operations are typically performed using vectorized integer sum-of-product routines, or optimized single/double-precision floating-point libraries [e.g., `sGEMM` and `dGEMM` routines of a mathematics kernel library (MKL)] [13], [14], [22]. Therefore, ensuring the robustness of these operations to core failures is of paramount importance for large-scale multimedia application deployment in cloud computing clusters exhibiting low MTTF characteristics.

A. Summary of Prior Work

Core failures in parallel compute-intensive routines are currently mitigated via *roll-back* or *roll-forward* methods. Roll-back methods are based on periodic checkpointing and recomputation if failures are detected. The vast majority of roll-back methods comprise backward error recovery (BER), where system states are stored periodically and computations are restarted from the last stored state when a failure happens in the computing environment [1], [23]–[27]. Several BER studies show that, depending on the desired level of resilience to core failures, substantial resources may be spent on checkpointing, system state storage/recovery, and recomputation. This has been identified as a major challenge for future exascale systems [1], [28], [29].

Roll-forward methods ensure result recovery from the functioning processor cores *without* recomputation when core

*Corresponding author. The authors are with the Electronic and Electrical Engineering Department, University College London, Roberts Building, Torrington Place, London, WC1E 7JE, UK; tel. +44 20 7679 7303; fax. +44 20 7388 9325 (both authors); email: {ijeoma.anarado.12, i.andreopoulos}@ucl.ac.uk.

failures occur in the system. Examples include: forward error recovery (FER) methods that recover the lost data from pre-established (and stored) input data checksum relationships without repeating computations [30]–[34], algorithm based fault tolerance (ABFT) [35] and modular redundancy [23], [24]. Computations examined in such proposals include matrix products [30], matrix factorization [36], convolution [34], [37] and iterative solvers [38]. The overarching concept of these methods is the production of checksum rows and columns (or entire checksum matrices) so that the performed computation can be applied to the checksum elements alongside the input matrices or vectors. These additional elements can then be used for FER by solving a system of linear equations if core failures are detected. Therefore, all FER approaches incur overhead due to the storage and processing of the checksum vectors or matrices. Moreover, the requirement of additional cores for checksum processing in FER decreases the achievable peak performance, as less cores are dedicated to actual input-data computations. Nevertheless, for failure resilience in compute-intensive routines like GEMM and CONV, roll-forward methods are preferable to roll-back methods, as they achieve higher reliability and can immediately mitigate the effect of failures without service interruption [24], [28], [30], [31], [33], [38].

B. Contribution

The proposed method comprises a novel FER mechanism for integer matrix multiplication and convolution operations that creates redundant results *within* the numerical representation of the output results. This is achieved by packing pairs of inputs within one integer number[1] and carrying out the actual GEMM or CONV operations with packed inputs. Because the packed outputs contain redundant sum-of-products, up to a certain number of failures of consecutive cores can be mitigated based on the available outputs. Importantly, the proposed approach does not require the processing of additional checksum inputs. Therefore, unlike all existing FER approaches, this is the first time a core failure recovery mechanism is proposed that *does not* require additional processor cores (e.g., for checksum computations or modular redundancy) in comparison to the conventional (fault-intolerant) routine.

Unlike our previous work on packing for detection of silent data corruptions (SDCs) in GEMM products [40], this paper focuses on core failure recovery in distributed systems assuming that the underlying parallel programming environment (like FT-MPI [41] and Open MPI [42]) detects such failures. However, if all output streams are received from all cores, our proposal can also be used for the detection of SDCs, since results can be cross-validated from outputs produced by execution in separate processing cores. Therefore, unlike our previous work [40], our current proposal can operate *both* for SDC detection, as well as core failure mitigation. We focus on the latter in this paper, as it has already been deemed as a problem of high relevance to current cloud-computing deployments of multicore multimedia signal processing systems.

Given that our algorithm can only be used for integer data computations, we focus on large-scale, low-latency, multimedia applications that require high-throughput integer-to-integer sum-of-product computations [19]. To quantify the complexity of the proposed approach, we derive its overhead in terms of arithmetic operations in comparison to the equivalent checksum-based method. We also present experimental results on *(i)* an 18-core, shared-memory, AWS EC2 instance[2], and *(ii)* a StarCluster [43] of AWS EC2 spot instances that are terminated and migrated to AWS EC2 on-demand instances for the duration that the spot price exceeds a predetermined threshold. For the former, we demonstrate that the proposed method achieves substantially-higher peak performance against the equivalent FER method based on checksums, both under failure-free and failure-occurring conditions. For the latter, we show that our approach provides for substantial reduction in deployment cost, especially in comparison to the failure-intolerant approach that cannot use spot instances. Finally, the source code for the proposed approach is made available online at https://github.com/NumericalPacking/Core-Failure-Mitigation-Code.

C. Paper Organization

In Section II, we review the basics of checksum-based FER methods for failure recovery in distributed systems and Section III describes the underlying concept of numerical packing. Section IV describes the packing, unpacking and failure-recovery process of the proposed core failure mitigation for both GEMM and CONV computations. Furthermore, we present the theoretical complexity analysis for numerical-packing–based and checksum-based core failure mitigation in comparison to the conventional failure-intolerant algorithm for GEMM computations. Experimental results are presented in Sections V and VI and conclusion and future work aspects can be found in Section VII.

D. Notations

Boldface uppercase and lowercase letters indicate matrices and vectors, respectively. The corresponding italicized lowercase letters indicate their individual elements, e.g., $\mathbf{w}$ and $w[i]$. Superscript T denotes matrix transposition. Notation $\mathbf{A}_{\text{col}}$ indicates the submatrix of $\mathbf{A}$ constructed by retaining the col subset of columns of $\mathbf{A}$, with $\{\text{col}\} \in \{\text{top, bot}\}$; e.g., given an $L \times L$ matrix $\mathbf{A}$, $\mathbf{A}_{\text{top}}$ is the submatrix comprising all rows and only columns with index range between $\left\{0, \ldots, \left\lfloor \frac{L}{2} \right\rfloor - 1\right\}$, while $\mathbf{A}_{\text{bot}}$ is the submatrix comprising all rows and only columns with index range between $\left\{\left\lfloor \frac{L}{2} \right\rfloor, \ldots, L-1\right\}$ of $\mathbf{A}$. Assuming two integers a and b, $a \gg b$

[1] The proposed algorithm is presented for integer representations, with the packed inputs being typecast to floating point in order to utilized optimized mathematics libraries for GEMM and CONV (e.g., sGEMM/dGEMM of Goto or Intel MKL [13], [14]). Even though the usage of floating-point representations for integer GEMM and CONV routines may seem counter-intuitive, it is in fact commonplace today since all processors have native support for floating point [39] and, in the case of CONV, floating point allows for Fourier-domain implementations.

[2] This is a c4.8xlarge AWS EC2 instance composed of multiple dual-nanocore physical processors (Intel Haswell) with hyperthreading.

shifts a by b bits to the right discarding the least-significant bits and $a \ll b$ shifts a by b bits to the left discarding the most-significant bits; $a \leftarrow b$ assigns value b to a. Finally, $\widehat{\mathbf{A}}$ and $\widetilde{\mathbf{A}}$ indicate the packed and extracted values of matrix $\mathbf{A}$ (equivalently for vectors and scalars), respectively.

II. Checksum-based Methods For Core Failures

We present a summary of checksum methods for the mitigation of core failures in integer GEMM products. Similar pre- and post-processing is required for failure-tolerant CONV operations.

Consider L concurrent matrix products carried out on L cores of a multicore computing system. Each core multiplies an $M \times N$ matrix $\mathbf{A}_l$ ($0 \le l < L$) with an $N \times K$ matrix (also called processing kernel) $\mathbf{B}$ in order for the multicore system to produce L output matrices $\mathbf{R}_l$. This is often the case, for example, in covariance-matrix calculations [44], image projections of groups of images [9], [12] (e.g., within a 2D-PCA face recognition system [11], [45]), and several other high-volume multimedia applications running on distributed computing clusters [5], [6]. For our purposes, when F cores ($1 \le F < L$) fail to return results after a predetermined deadline, F core failures are said to have occurred in the system. This could occur due to vibration, network, power, or soft-error–induced disruptions, as described in Section I.

In order to recover from such failures, checksum-based methods produce up to F checksum matrices, $\mathbf{A}_{\mathrm{c}1}, \dots, \mathbf{A}_{\mathrm{c}F}$. The number of checksum matrices is dependent on the number of core failures that should be tolerated in the parallel GEMM execution. The simplest checksum matrix that can be used to recover from a single core failure in the L GEMM products ($0 \le l < L$)

$$\mathbf{R}_l = \mathbf{A}_l \mathbf{B} \tag{1}$$

is

$$\mathbf{A}_{\mathrm{c}1} = \sum_{l=0}^{L-1} \mathbf{A}_l. \tag{2}$$

This checksum matrix undergoes the same GEMM operation, $\mathbf{R}_{\mathrm{c}1} = \mathbf{A}_{\mathrm{c}1}\mathbf{B}$, which requires the usage of an additional core. The dynamic range of $\mathbf{R}_{\mathrm{c}1}$ is increased by L in comparison to GEMM products $\mathbf{A}_l\mathbf{B}$. If the xth core fails, $0 \le x < L$, we recover $\mathbf{R}_x$ by $\mathbf{R}_x = \mathbf{R}_{\mathrm{c}1} - \sum_{\forall l \ne x} \mathbf{R}_l$. A failure occurring in the core that computes the $\mathbf{R}_{\mathrm{c}1}$ checksum is ignored. Additional checksums may be added to mitigate more than one failure by using different linear combinations of the $\mathbf{A}_0, \dots, \mathbf{A}_{L-1}$ matrices, as shown by Stefanidis and Luk [46], [47]. Overall, in order to tolerate F core failures in a parallel computing environment, F additional cores are set aside to compute the results of the linear-checksum matrices [30], [33], [38], $\mathbf{R}_{\mathrm{c}f} = \mathbf{A}_{\mathrm{c}f}\mathbf{B}$, $1 \le f \le F$.

III. Brief Review of Packing for Integer Data Computations

The concept of numerical packing was originally proposed for throughput scaling in multimedia applications [20], [45], [48] and soft-error tolerance in GEMM for fail-continue systems (e.g., due to silent data corruption) [40]. Packed processing stacks multiple small dynamic-range inputs in a standard 32-bit or 64-bit representation using an integer packing factor, k, to avoid overflow (or "invasion"), and thereby performs computation on these inputs simultaneously. Packing can be symmetric, where both matrix (or vector) inputs are packed, or asymmetric, where only one of the input matrices (or vectors) is packed [45]. A simple illustration of symmetric packed processing for the product of two vectors is shown in Fig. 1. In the figure, conventional processing performs a 2×1-by-1×2 product, while, in the packed domain, two 1×1-by-1×1 products ensue following the packing of $\mathbf{b}$ to $\widehat{b}$ and $\mathbf{a}$ to $\widehat{a}_{\mathrm{i}}$ and $\widehat{a}_{\mathrm{j}}$ with $k = 15$, i.e., $r_{\mathrm{i}} = \widehat{a}_{\mathrm{i}}\widehat{b}$ and $r_{\mathrm{j}} = \widehat{a}_{\mathrm{j}}\widehat{b}$. The results are unpacked (extracted) to form the final outputs. Therefore, in packed processing, the overall number of computations required for the outer product is reduced by 50% in comparison to the conventional computation, at the cost of packing and unpacking and the use of higher-bitwidth arithmetic units [20].

More generally, for the computation of the $M \times M$ integer matrix product $\mathbf{R} = \mathbf{AB}$ via symmetric packing (with M even), the packing process creates blocks $\widehat{\mathbf{A}}_{\mathrm{i}}$, $\widehat{\mathbf{A}}_{\mathrm{j}}$, and $\widehat{\mathbf{B}}$ with $\frac{M}{2} \times M$ and $M \times \frac{M}{2}$ coefficients (resp.) given by:

$$\widehat{\mathbf{A}}_{\mathrm{i}} = \left[\left(\mathbf{A}_{\mathrm{bot}}^{\mathrm{T}} \ll k\right) + \mathbf{A}_{\mathrm{top}}^{\mathrm{T}}\right]^{\mathrm{T}} \tag{3}$$

$$\widehat{\mathbf{A}}_{\mathrm{j}} = \left[\left(\mathbf{A}_{\mathrm{top}}^{\mathrm{T}} \ll k\right) + \mathbf{A}_{\mathrm{bot}}^{\mathrm{T}}\right]^{\mathrm{T}} \tag{4}$$

$$\widehat{\mathbf{B}} = \left(\mathbf{B}_{\mathrm{top}} \ll k\right) + \mathbf{B}_{\mathrm{bot}}. \tag{5}$$

with $k \in \mathbb{N}^\star$ and carries out $\widehat{\mathbf{R}}_{\mathrm{i}} = \widehat{\mathbf{A}}_{\mathrm{i}}\widehat{\mathbf{B}}$ and $\widehat{\mathbf{R}}_{\mathrm{j}} = \widehat{\mathbf{A}}_{\mathrm{j}}\widehat{\mathbf{B}}$, followed by unpacking. The utilized value for k depends on the maximum possible value of the matrix product [20], [21], [45]. It has been shown that, to ensure accurate recovery of results after computation [20], [21], [40], [45], [49]:

$$k > \log_2\left(\max_{\forall l} |\mathbf{R}_l|\right) + 1 \text{ and } 3k \le W, \tag{6}$$

with $W \in \{32, 64\}$ if all processing is carried out in 32-bit or 64-bit integer representations and $W \in \{24, 52\}$ if kernel processing is carried out in single or double-precision floating-point representations [20], [40].

IV. Proposed Approach

Let us consider: *(i)* L integer matrices $\mathbf{A}_0, \dots, \mathbf{A}_{L-1}$ ($L \ge 3$), comprising $M \times N$ dimensions each, that must be processed[3] via an $N \times K$ integer kernel matrix $\mathbf{B}$ [50]; *(ii)* L integer input signal vectors $\mathbf{a}_0, , \dots, \mathbf{a}_{L-1}$ ($L \ge 3$), comprising N samples each, that must be filtered by a K-sample integer kernel vector

[3] Supported algorithms include a range of linear and sesquilinear compute-intensive operations including GEMM, Kronecker product, and multidimensional CONV. In this paper we focus on the two illustrative and important cases of matrix products and one-dimensional convolution operations since the remaining operations follow in a straightforward manner from these results.

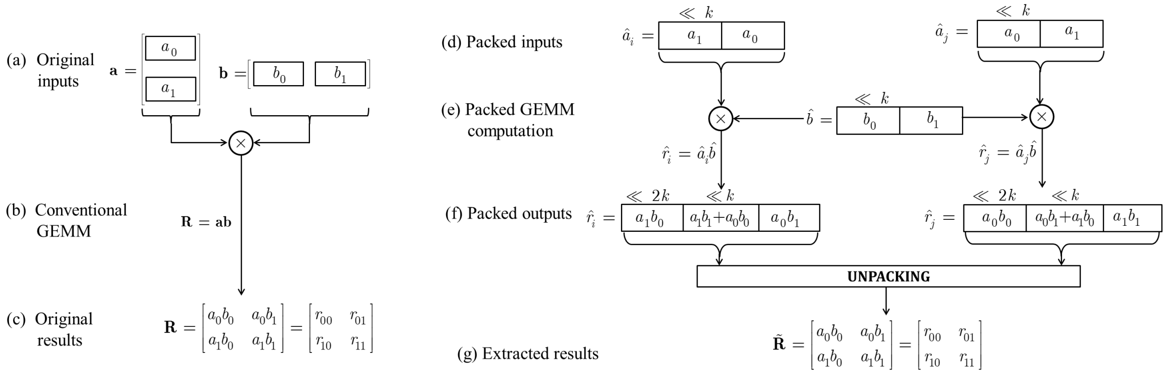
(a) Original inputs
$\mathbf{a} =$
a_0
a_1
$\mathbf{b} =$
b_0
b_1
(b) Conventional GEMM
$\mathbf{R} = \mathbf{ab}$
(c) Original results
$\mathbf{R} = \begin{bmatrix} a_0b_0 & a_0b_1 \\ a_1b_0 & a_1b_1 \end{bmatrix} = \begin{bmatrix} r_{00} & r_{01} \\ r_{10} & r_{11} \end{bmatrix}$
(d) Packed inputs
$\ll k$
$\hat{a}_i =$ a_1 a_0
$\hat{a}_j =$ a_0 a_1
(e) Packed GEMM computation
$\hat{b} =$ b_0 b_1
$\hat{r}_i = \hat{a}_i\hat{b}$
$\hat{r}_j = \hat{a}_j\hat{b}$
(f) Packed outputs
$\ll 2k$
$\hat{r}_i =$ a_1b_0 $a_1b_1+a_0b_0$ a_0b_1
$\hat{r}_j =$ a_0b_0 $a_0b_1+a_1b_0$ a_1b_1
UNPACKING
(g) Extracted results
$\tilde{\mathbf{R}} = \begin{bmatrix} a_0b_0 & a_0b_1 \\ a_1b_0 & a_1b_1 \end{bmatrix} = \begin{bmatrix} r_{00} & r_{01} \\ r_{10} & r_{11} \end{bmatrix}$

Figure 1. (a)–(c) Conventional integer subblock multiplication for the case of a 2 × 1-by-1 × 2 vector product. (d)–(g) Symmetric packing for the same product via an integer representation with packing coefficient $k = 15$. The partitioning within the rectangles shows the location (shifts by k or $2k$ bits) of inputs/outputs when packed within a single number.

b. In both cases, each operation is performed on a different core in an L-core cluster. We shall present an approach that can recover all L GEMM or CONV results even if up to $F = \lfloor \frac{L}{3} \rfloor$ consecutive cores fail, *without* requiring additional checksum inputs. Recovery from up to $\lfloor \frac{L}{3} \rfloor$ consecutive failures is important in the context of computing clusters comprising groups of multicore processors, where a single failure on one multicore board may translate in several consecutive core failures. In order to achieve this, we utilize symmetric packing, where both input and kernel matrices are packed, as elaborated in Section III. Symmetric packing allows for the production of redundant outputs, i.e., akin to the middle zone of k bits in the example of Fig. 1(f). As shown in this paper, if constructed appropriately, these redundant outputs allow for the recovery from core failures.

A. Proposed Packing Method for Failure-tolerant GEMM

The first step of the proposed approach packs pairs from the L input matrices, thereby generating L "packed" input matrices $\widehat{\mathbf{A}}_0, \ldots, \widehat{\mathbf{A}}_{L-1}$ given by ($0 \leq l < L$):

$$\widehat{\mathbf{A}}_l \quad = \quad (\mathbf{A}_l \ll k) + \mathbf{A}_{(F+l) \bmod L} \tag{7}$$

where F is the number of consecutive core failures to be mitigated and k the packing factor introduced in Section III, which is set such that (6) holds with $W = 64$.

For example, for $L = 3$, and $F = 1$, we have

$$\begin{aligned} \widehat{\mathbf{A}}_0 &= (\mathbf{A}_0 \ll k) + \mathbf{A}_1 \\ \widehat{\mathbf{A}}_1 &= (\mathbf{A}_1 \ll k) + \mathbf{A}_2 \\ \widehat{\mathbf{A}}_2 &= (\mathbf{A}_2 \ll k) + \mathbf{A}_0. \end{aligned} \tag{8}$$

Packing of the kernel matrix $\mathbf{B}$ is also performed. This packing produces the $N \times \frac{K}{2}$ matrix[4] $\widehat{\mathbf{B}}$ by:

$$\widehat{\mathbf{B}} = \left(\mathbf{B}_{\text{top}} \ll k\right) + \mathbf{B}_{\text{bot}}. \tag{9}$$

Given that the packing factor k in (7)–(9) will increase the dynamic range of all $\widehat{\mathbf{A}}_l$ and $\widehat{\mathbf{B}}$, we utilize 64-bit representations for $\widehat{\mathbf{A}}_l$ and $\widehat{\mathbf{B}}$.

B. Packed GEMM Computations

All $M \times N$-by-$N \times \frac{K}{2}$ matrix products can be computed concurrently on L processors via the use of L 64-bit GEMM calls (e.g., OpenMP framework with MKL `dGEMM`),

$$\forall l: \ \widehat{\mathbf{R}}_l = \widehat{\mathbf{A}}_l\widehat{\mathbf{B}}, \tag{10}$$

thereby producing all required results, as well as a number of "duplicate" results within the numerical representation of the packed outputs $\widehat{\mathbf{R}}_l$. Fig. 2 illustrates the simple case of three 1×1-by-1×2 matrix products ($L = 3$, $M = N = 1$, $K = 2$) after packing has been carried out via (8) and (9). The contents of the $M \times \frac{K}{2}$ output matrices $\widehat{\mathbf{R}}_0$, $\widehat{\mathbf{R}}_1$ and $\widehat{\mathbf{R}}_2$ can be expressed mathematically by:

$$\begin{aligned} \widehat{\mathbf{R}}_0 &= \widehat{\mathbf{A}}_0\widehat{\mathbf{B}} \\ &= \left(\mathbf{A}_0\mathbf{B}_{\text{top}} \ll 2k\right) + \left[\left(\mathbf{A}_0\mathbf{B}_{\text{bot}} + \mathbf{A}_1\mathbf{B}_{\text{top}}\right) \ll k\right] + \mathbf{A}_1\mathbf{B}_{\text{bot}} \\ &= \left(\widetilde{\mathbf{R}}_{0,\text{top}} \ll 2k\right) + \left[\left(\widetilde{\mathbf{R}}_{0,\text{bot}} + \widetilde{\mathbf{R}}_{1,\text{top}}\right) \ll k\right] + \widetilde{\mathbf{R}}_{1,\text{bot}} \end{aligned} \tag{11}$$

$$\begin{aligned} \widehat{\mathbf{R}}_1 &= \widehat{\mathbf{A}}_1\widehat{\mathbf{B}} \\ &= \left(\mathbf{A}_1\mathbf{B}_{\text{top}} \ll 2k\right) + \left[\left(\mathbf{A}_1\mathbf{B}_{\text{bot}} + \mathbf{A}_2\mathbf{B}_{\text{top}}\right) \ll k\right] + \mathbf{A}_2\mathbf{B}_{\text{bot}} \\ &= \left(\widetilde{\mathbf{R}}_{1,\text{top}} \ll 2k\right) + \left[\left(\widetilde{\mathbf{R}}_{1,\text{bot}} + \widetilde{\mathbf{R}}_{2,\text{top}}\right) \ll k\right] + \widetilde{\mathbf{R}}_{2,\text{bot}} \end{aligned} \tag{12}$$

$$\begin{aligned} \widehat{\mathbf{R}}_2 &= \widehat{\mathbf{A}}_2\widehat{\mathbf{B}} \\ &= \left(\mathbf{A}_2\mathbf{B}_{\text{top}} \ll 2k\right) + \left[\left(\mathbf{A}_2\mathbf{B}_{\text{bot}} + \mathbf{A}_0\mathbf{B}_{\text{top}}\right) \ll k\right] + \mathbf{A}_0\mathbf{B}_{\text{bot}} \\ &= \left(\widetilde{\mathbf{R}}_{2,\text{top}} \ll 2k\right) + \left[\left(\widetilde{\mathbf{R}}_{2,\text{bot}} + \widetilde{\mathbf{R}}_{0,\text{top}}\right) \ll k\right] + \widetilde{\mathbf{R}}_{0,\text{bot}}. \end{aligned} \tag{13}$$

C. Unpacking of the Results

Our key insight is that out of the L packed matrices $\widehat{\mathbf{R}}_0, \ldots, \widehat{\mathbf{R}}_{L-1}$, $L - F$ matrices suffice for the recovery of all L

[4] For simplicity of exposition, we assume that K is even.

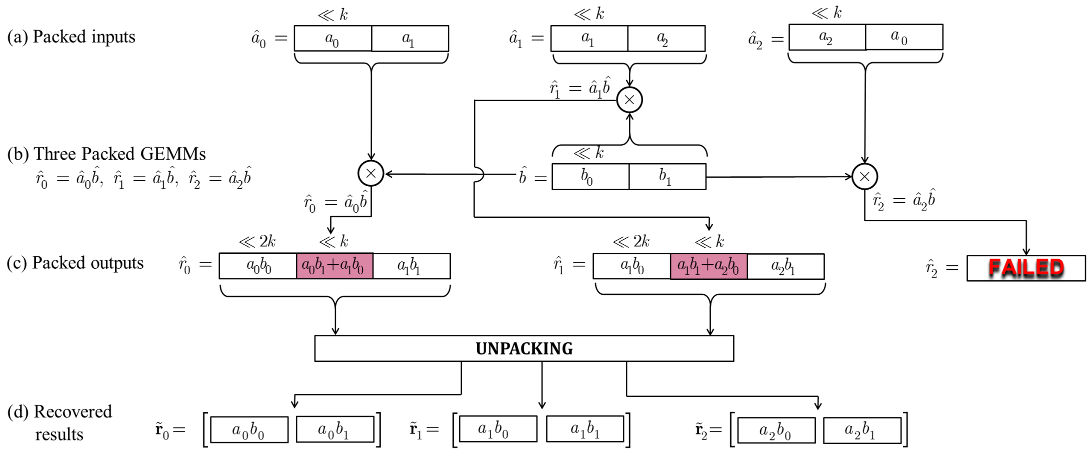
(a) Packed inputs
$\hat{a}_0 =$ a_0 a_1 $\ll k$
$\hat{a}_1 =$ a_1 a_2 $\ll k$
$\hat{a}_2 =$ a_2 a_0 $\ll k$
(b) Three Packed GEMMs
$\hat{r}_0 = \hat{a}_0\hat{b},\ \hat{r}_1 = \hat{a}_1\hat{b},\ \hat{r}_2 = \hat{a}_2\hat{b}$
$\hat{r}_1 = \hat{a}_1\hat{b}$
$\hat{b} =$ b_0 b_1 $\ll k$
$\hat{r}_0 = \hat{a}_0\hat{b}$
$\hat{r}_2 = \hat{a}_2\hat{b}$
(c) Packed outputs
$\hat{r}_0 =$ a_0b_0 $a_0b_1+a_1b_0$ a_1b_1 $\ll 2k$ $\ll k$
$\hat{r}_1 =$ a_1b_0 $a_1b_1+a_2b_0$ a_2b_1 $\ll 2k$ $\ll k$
$\hat{r}_2 =$ FAILED
UNPACKING
(d) Recovered results
$\tilde{\mathbf{r}}_0 =$ a_0b_0 a_0b_1
$\tilde{\mathbf{r}}_1 =$ a_1b_0 a_1b_1
$\tilde{\mathbf{r}}_2 =$ a_2b_0 a_2b_1

Figure 2. Illustration of failure mitigation in integer matrix product for the elementary case of three 1×1-by-1×2 matrix products in an integer representation with $k = 15$. Assuming that the core that computes $\widehat{r}_2$ failed, the results of the other two cores ($\widehat{r}_0$ and $\widehat{r}_1$) are used to produce all three outputs $\widetilde{r}_0$, $\widetilde{r}_1$ and $\widetilde{r}_2$ after unpacking. The partitioning within the rectangles shows the location (shifts by k or $2k$ bits) of inputs/outputs when packed within a single number.

outputs. For example, for $L = 3$ and $F = 1$, any two matrices out of $\widehat{\mathbf{R}}_0$, $\widehat{\mathbf{R}}_1$ and $\widehat{\mathbf{R}}_2$ can produce all outputs, $\widetilde{\mathbf{R}}_0$, $\widetilde{\mathbf{R}}_1$ and $\widetilde{\mathbf{R}}_2$, as illustrated in Fig. 2. For instance, assuming $\widehat{\mathbf{R}}_0$ and $\widehat{\mathbf{R}}_1$ are used for the recovery of $\widetilde{\mathbf{R}}_0$, $\widetilde{\mathbf{R}}_1$ and $\widetilde{\mathbf{R}}_2$, the required steps are given by:

$$\widetilde{\mathbf{R}}_{0,\text{top}} = \widehat{\mathbf{R}}_0 \gg 2k \tag{14}$$

$$\mathbf{T}_0 = \widehat{\mathbf{R}}_0 - \left(\widetilde{\mathbf{R}}_{0,\text{top}} \ll 2k\right) \tag{15}$$

$$\mathbf{T}_1 = \mathbf{T}_0 \gg k. \tag{16}$$

Because of the complement-two arithmetic used in integer representations in commodity hardware, all negative elements (i,j) of $\mathbf{T}_1$ will be found to be larger or equal to 2^{k-1} (maximum positive value within a packed output element) and $\widetilde{R}_{0,\text{top}}[i,j]$ (contained in the most-significant bits of $\widehat{R}_0[i,j]$) will be found to be one less than their correct value. To compensate for these effects of the complement-two arithmetic, we first define the intermediate matrix $\mathbf{E}_0$ to store the signed values of $\mathbf{T}_1$ therein, and adjust the values of $\widetilde{\mathbf{R}}_{0,\text{top}}$ as follows:

- $\forall i,j$: if $T_1[i,j] \geq 2^{k-1}$, then set $E_0[i,j] = T_1[i,j] - 2^k$ (convert to negative number); else set $E_0[i,j] = T_1[i,j]$ (no change);
- $\forall i,j$: if $E_0[i,j] < 0$, then set $\widetilde{R}_{0,\text{top}}[i,j] \leftarrow \left(\widetilde{R}_{0,\text{top}}[i,j] + 1\right)$.

Next, a similar check is performed to extract the signed representation of $\widetilde{\mathbf{R}}_{1,\text{bot}}$ and the values of $\mathbf{E}_0$ are adjusted, i.e.,

$$\mathbf{T}_0 \leftarrow [\mathbf{T}_0 - (\mathbf{T}_1 \ll k)], \tag{17}$$

- $\forall i,j$: if $T_0[i,j] \geq 2^{k-1}$, then set $\widetilde{R}_{1,\text{bot}}[i,j] = \widetilde{R}_{1,\text{bot}}[i,j] - 2^k$; else set $\widetilde{R}_{1,\text{bot}}[i,j] = T_0[i,j]$;
- $\forall i,j$: if $\widetilde{R}_{1,\text{bot}}[i,j] < 0$, then set $E_0[i,j] \leftarrow (E_0[i,j] + 1)$.

Similarly, $\widehat{\mathbf{R}}_1$ undergoes the same processing as $\widehat{\mathbf{R}}_0$ in order to extract $\widetilde{\mathbf{R}}_{1,\text{top}}$, the intermediate matrix $\mathbf{E}_1 = \widetilde{\mathbf{R}}_{1,\text{bot}} + \widetilde{\mathbf{R}}_{2,\text{top}}$ and $\widetilde{\mathbf{R}}_{2,\text{bot}}$. Finally, we perform the following operations to complete the extraction of all results:

$$\widetilde{\mathbf{R}}_{0,\text{bot}} = \mathbf{E}_0 - \widetilde{\mathbf{R}}_{1,\text{top}} \tag{18}$$

$$\widetilde{\mathbf{R}}_{2,\text{top}} = \mathbf{E}_1 - \widetilde{\mathbf{R}}_{1,\text{bot}}. \tag{19}$$

Importantly, the `_mm256_shuffle_epi32` and `_mm256_permute2f128_si256` instructions (which are intrinsically supported in modern SIMD architectures [51]) can further be used to optimize the output matrix reordering using the "top" and "bot" index subsets.

D. Proposed Approach for One-dimensional Convolution

The same process can be used to mitigate core failures within L parallel convolution/cross-correlation (CONV) operations of N-sample input signals $\mathbf{a}_l$ with a K-sample kernel $\mathbf{b}$, producing output vectors $\mathbf{r}_l$ by $(0 \leq i < N+K,\ \ 0 \leq l < L)$:

$$\mathbf{r}_l = \mathbf{a}_l \star \mathbf{b} \iff r_l[i] = \sum_{k=0}^{K-1} a_l[i-k]\,b[k] \ . \tag{20}$$

Specifically, the proposed algorithm packs the signal vectors, $\mathbf{a}_l$, into the packed inputs, $\widehat{\mathbf{a}}_l$, following the same procedure as for (7), while the convolution kernel $\mathbf{b}$ is packed into $\widehat{\mathbf{b}}$ as shown for GEMM in (9).

The convolutions $(0 \leq l < L)$:

$$\forall l:\ \widehat{\mathbf{r}}_l = \widehat{\mathbf{a}}_l \star \widehat{\mathbf{b}} \tag{21}$$

can then be carried out with any optimized library (e.g., Intel's IPP `ippsConvolve_64f` routine [22], Matlab's `conv` function, etc.) in order to produce the $\left(N + \frac{K}{2} - 1\right)$-sample signals $\widehat{\mathbf{r}}_0, \ldots, \widehat{\mathbf{r}}_{L-1}$. In addition, unpacking follows the same procedure as for GEMM, except for an additional summation operation. For example, for $L = 3$, given the unpacked outputs $\widetilde{\mathbf{r}}_{0,\text{top}}$, $\widetilde{\mathbf{r}}_{0,\text{bot}}$, $\widetilde{\mathbf{r}}_{1,\text{top}}$, and $\widetilde{\mathbf{r}}_{1,\text{bot}}$, the "top" vectors comprise output signal samples with indices within $\left[0, N + \frac{K}{2} - 1\right]$, while the "bot" vectors comprise signals with indices within $\left[\frac{K}{2}, N+K-1\right]$. The recovered output signals are obtained by the following concatenation operations ($0 \leq l < 3$):

$$\forall l: \widetilde{\mathbf{r}}_l = \left[\widetilde{\mathbf{r}}_{l,\text{top}} \quad \mathbf{0}_{\frac{K}{2}}\right] + \left[\mathbf{0}_{\frac{K}{2}} \quad \widetilde{\mathbf{r}}_{l,\text{bot}}\right] \tag{22}$$

with $\mathbf{0}_{\frac{K}{2}}$ the $1 \times \frac{K}{2}$ vector of zeros.

E. Summary of Key Results

The proposed method utilizes packing to create L cyclically-duplicated/double-bitwidth descriptions for the inputs of matrix product and convolution operations. The packed inputs that are then used within L 64-bit GEMM or CONV operations on L processing cores. The proposed unpacking process recovers all GEMM or CONV outputs with operations count that depends only on the number of outputs and not on the inner-product dimension of the GEMM or CONV processing. The following four propositions formalize these key points.

Proposition 1. *Let the packing of* (7) *and* (9)*, and the execution of the L GEMM operations of* (10) *on L independent processor cores. The results of* (1) *can be recovered under the failure of any group of up to $\left\lfloor \frac{L}{3} \right\rfloor$ consecutive cores.*

Proof: See Appendix. ■

Proposition 2. *Let the packing of* (7) *and* (9)*, and the execution of the L CONV operations of* (21) *on L independent processor cores. The results of* (20) *can be recovered under the failure of any group of up to $\left\lfloor \frac{L}{3} \right\rfloor$ consecutive cores.*

Proof: See Appendix. ■

If (6) holds, then the unpacking process guarantees that the correct value is obtained for each output under the use of a 64-bit integer representation. However, the conditions of (6) incur certain reduction in the achievable dynamic range in comparison to 32-bit integer representations. This is quantified in the following proposition.

Proposition 3. *The proposed approach incurs loss of approximately* 37.5% *of the dynamic range of the 32-bit integer representation.*

Proof: See Appendix. ■

While the proposed approach (approximately) halves the overall multiply-accumulate (MAC) operations against the conventional approach during GEMM, all operations are performed in 64-bit representations. Under the assumption that 64-bit arithmetic operations require twice the cycles of 32-bit arithmetic operations, which (amongst others) holds for AVX2-based realizations, we can quantify the arithmetic operations (additions and MAC operations) required by the proposed method in comparison to the conventional, checksum-based, core failure mitigation approach based on (2). We focus on the case of GEMM as a reference for analytic comparisons. However, the equivalent results can be derived for the CONV operations following the same approach.

Proposition 4. *In the absence of failures during L concurrent $N \times N$-by-$N \times N$ matrix products, and assuming that a 64-bit operation is equivalent to two 32-bit operations, the number of arithmetic operations required by the proposed approach is:*

$$\frac{2NF + 5F - 8L - 1}{2NL + 2NF - 2F} \times 100\% \tag{23}$$

less than the conventional checksum-based roll-forward method of Section II.

Proof: See Appendix. ■

For example, for $N = 576$, $L = 6$ and $F = \left\lfloor \frac{L}{3} \right\rfloor = 2$, Proposition 4 shows that our approach is 24.59% more efficient than the conventional checksum-based approach.

Proposition 5. *When F core failures occur during L concurrent $N \times N$-by-$N \times N$ matrix products, the number of arithmetic operations required by the proposed approach is up to:*

$$\frac{2NF + 4F - 7L - 1}{2NF + 2NL - 3F + L} \times 100\% \tag{24}$$

less than the conventional checksum-based roll-forward method of Section II.

Proof: See Appendix. ■

For example, for $N = 576$, $L = 6$ and $F = \left\lfloor \frac{L}{3} \right\rfloor = 2$, Proposition 5 shows that numerical packing is 24.62% more efficient than the checksum-based method. We note, however, that this is the worst case scenario for the checksum-based method as some of the failed cores could be cores computing a checksum result, thus requiring no recovery.

In terms of resilience to multiple non-consecutive failures, similar to the checksum approach of Jou and Abraham [52], the proposed approach would either need additional processing cores, i.e., as proposed by Luk, Rexford, *et al.* [47], [53], or a generalized version of the proposed packing mechanism would need to be devised. We may pursue support for this feature in future work.

Table I
GIGA-OPERATIONS PER OUTPUT [AND PERCENTILE OVERHEAD IN COMPARISON TO THE FAILURE-INTOLERANT (CONVENTIONAL) GEMM COMPUTATION] FOR $N \times N$-BY-$N \times N$ GEMM PRODUCTS WHEN MITIGATING $F = 4$ FAILURES IN AN $L = 16$ CORE COMPUTING PLATFORM.

Method	Number of operations per output ($\times 10^9$)		
	$N = 1152$	$N = 4608$	$N = 9216$
Failure-intol.	3.06	195.67	1565.43
Proposed	3.07 (0.33%)	195.82(0.08%)	1566.05(0.04%)
Checksum	4.06 (32.68%)	260.92 (33.35%)	2087.35 (33.34%)

Table I presents examples of the arithmetic complexity per output derived from the calculations of Proposition 4 when $L = 16$ cores are available for data computation. We focus on the fail-free case as a baseline for comparisons, since, under the occurrence of failures, our approach will outperform the checksum-based method with an even higher margin. The results of Table I show that, although the proposed approach requires a limited number of arithmetic operations for packing and unpacking, the production of L matrices using L cores makes its operations' count comparable to the conventional, failure-intolerant, GEMM. On the other hand, the checksum-based method produces only $L - F$ matrices via the L utilized cores, thereby leading to substantially-increased operations-per-output in comparison to both the failure-intolerant GEMM and the proposed approach. On average, Table I shows that numerical packing incurs about 0.15% additional computations per output for packing and unpacking in comparison to the failure-intolerant computation, while the checksum approach requires 33.12% more computations to produce the same number of outputs. The practical overhead incurred by the proposed and the checksum-based approach is investigated via the experiments of the following section.

V. Experimental Results on a Shared-Memory 18-core AWS EC2 Instance

We benchmark our proposal for *core failure* mitigation using a compute-optimized `c4.8xlarge` instance of AWS EC2 (Intel Xeon E5-2666v3 2.9GHz cores, on-demand instance type, Windows Server 2012, Intel C++ 15.0 Compiler, Intel MKL for GEMM operations, 36 EC2 virtual cores corresponding to 18 physical cores with shared memory). Each individual throughput or execution-time result in our experiments corresponds to the average of 1000 runs with randomly selected inputs from the utilized datasets.

A. *Execution Time Comparison for Failure Mitigation in Parallel GEMM Computations*

Given that the optimal performance of the Intel MKL GEMM library is obtained when parallel computations are done in physical cores [54], [55], we set the system affinity to 18 physical cores rather than the 36 EC2 virtual cores. This configuration can accommodate a shared-memory computing cluster comprising four quadcore processors and we can present results for data recovery when all computations in any group of (up to) four consecutive cores are lost (termed as one quadcore failure). All packing, unpacking and checksum generation make use of the parallel computing capability of the computing environment via the OpenMP framework, as well as the increased optimization level offered by AVX2 SIMD instructions.

Table II
Average execution time results (in milliseconds) and percentile overhead in comparison to the failure-intolerant (conventional) GEMM computation when mitigating one quadcore failure in $L = 16$ GEMM computations.

Method	$N = 1152$	$N = 4608$	$N = 9216$
Failure-intol.	3.19	164.13	1255.06
Proposed	4.24 (32.92%)	194.98 (18.8%)	1415.66 (12.8%)
Checksum	4.55 (42.63%)	224.27 (36.64%)	1695.44 (35.09%)

Table II presents the average execution time for all methods when $L = 16$ for the computation of $(0 \leq l < 16)$:

$$\mathbf{R}_l = \mathbf{A}_{l,(N\times N)}\mathbf{B}_{N\times N}. \tag{25}$$

By comparing Table I and Table II, it is evident that the performance ranking of the methods follows the theoretical analysis of Section IV-E. While the proposed approach incurs higher overhead than what is theoretically predicted in Table I (primarily due to variation in execution time between 64-bit and 32-bit memory and arithmetic operations), it is still offering substantial execution-time improvement against the checksum-based failure mitigation approach, particularly as the matrix product dimension increases.

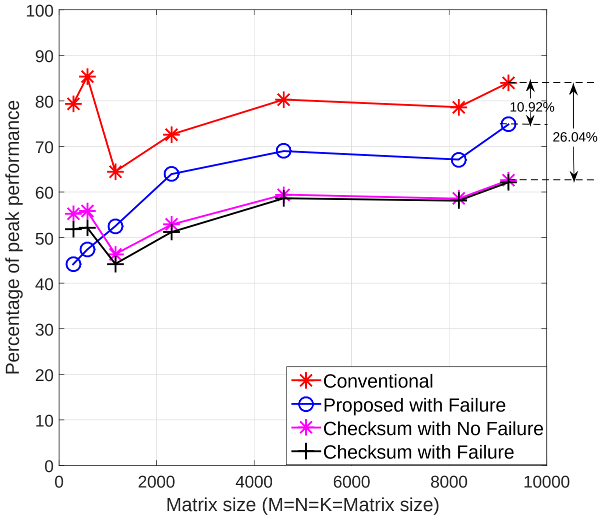

Figure 3. Peak performance achieved by each method in the utilized distributed computing environment for $L = 16$.

Fig. 3 presents the experimental peak performance[5] achieved by each approach including all pre- and post-processing. We note that the peak performance achieved by all methods is slightly reduced because of the overheads of multicore systems with shared memory, i.e., it is well known that all practical GEMM realizations will attain less than 90% of peak performance in such a multicore cluster [13], [54], [55]. Furthermore, since the proposed approach and the checksum-based method require additional operations for failure mitigation, reduced peak performance is expected in comparison to the failure-intolerant (conventional) approach. Our results show that the conventional (failure-intolerant) GEMM achieves 64.50% to 85.31% of peak performance, while the proposed method for core failure mitigation achieves 43.25% to 74.47% of peak performance. On the other hand, checksum-based core failure mitigation achieves only 45.27% to 62.18% of peak performance. Overall, our theoretical analysis and experimental results demonstrate that our proposal offers the same reliability to core failures as conventional checksum-based methods, albeit with peak performance that becomes comparable to the conventional failure-intolerant approach as the inner sum-of-product dimension grows.

[5] Each of the utilized Intel Xeon E5-2666v3 cores achieves $V = 92.8$ GFlop/s (operating at 2.9 GHz with 32 floating-point operations per cycle under AVX2 instructions). The peak performance achieved by each method is calculated by $\frac{U(2N^3-N^2)}{tVL} \times 100\%$, where U is the number of matrices of output results (i.e., $U = L$ for the conventional and proposed methods and $U = L - F$ for the checksum-based approach) and t is the total execution time (in seconds) for each case.

B. Execution Time Comparison for Failure Mitigation in Parallel Cross-correlation Computations

We now benchmark our approach and the checksum-based approach for core failure recovery in the multicore execution of the CONV-based music retrieval algorithm of Ellis *et. al.* [56]. The algorithm identifies cover songs within a music database by extracting beat and tempo data from the query song track and performing cross correlation of the beat and tempo feature vectors with the beat and tempo vectors of each of the songs in the database [56], [57]. The dominant computation within this process comprises two-dimensional cross-correlation between a matrix of audio beat and tempo data and a database of such matrices. Using this algorithm, we perform a music retrieval experiment with the music database comprising beat-tempo data from the Million-song dataset subset available at Columbia University's LabROSA repository [58] and the "1517-Artists" dataset of Seyerlehner *et. al.* [59].

The algorithm tracks beats of each of the input music tracks and generates a twelve-dimensional "chroma" representation specifying the pitches of the twelve distinct semitones in the western octave [56]. We used 100 beats to describe every music track. Thus, each music tack is composed of a 12×100 matrix representing its beat-chroma features. We then modified the Matlab code of Ellis *et. al.* [56] for parallel execution of the music similarity measurement. Specifically, the Matlab `spmd` function was used to set up the parallel computing environment. Given that Matlab uses only the physical cores of the computing unit for parallel processing (see `feature('numCores')` command in Matlab), we distribute the music database amongst 16 cores for the conventional computation and proposed algorithm, while the checksum-based method uses 12 cores for data storage and the other 4 cores for checksum data. All preprocessing for beat and feature extraction from the audio clips is performed offline, i.e., prior to the actual retrieval task, and it is not carried out by the cloud-computing server of our experiment, which is only used for the actual retrieval task.

Table III shows the average execution time (in milliseconds) for the search and retrieval of a music track within various database sizes. The results demonstrate that the performance of the proposed algorithm converges to that of the conventional, failure-intolerant, implementation as the database size increases. On the other hand, the checksum-based method is found to incur considerable (and largely consistent) execution time overhead (33%–41%) across all database sizes.

VI. Image Retrieval Based on Terminatable AWS EC2 Spot Instances

To illustrate the efficacy of our proposal in *instance failure* mitigation when carrying out integer sum-of-product computations with terminatable instances, we performed several medium-scale image retrieval experiments using the state-of-the-art vector of locally aggregated descriptors (VLAD) method of Jegou *et. al.* [60] deployed on a cluster of AWS EC2 spot instances.

A. Application Description

The preprocessing done by the VLAD algorithm produces a compact "signature" for each database image and query image based on [60] [61]: *(i)* the local aggregation of visual features into clusters using K-means clustering; *(ii)* compaction of the aggregated feature vectors into a vector of integers based on normalization, projection and quantization. In order to perform a retrieval task, we compute the inner product between the compacted feature vector of a query image and the compacted feature vector of each of the images in the database. The images corresponding to the top-T highest inner-product values are subsequently returned as the T best matches for the given query.

Given that multiple query images (a.k.a. image "bunch") are matched through the stored database at any given moment (e.g., because of many concurrent users, or due to the use of video that results in multiple feature vectors per query), the matching operations are carried out via GEMM products between the feature vectors of query image bunches and the database images. Following the GEMM, only inner-product values above a predetermined threshold are retained and a sorting algorithm is used in order to find the top-T matches [62]. This post-processing stage has negligible computational cost, thereby leaving the GEMM as the compute-intensive operation being carried out in the cloud computing cluster (the preprocessing stage for the VLAD signature extraction is performed offline for the database and on a local core for the query images).

In our experiments, we use the VLAD descriptors derived from the INRIA Holidays dataset of [60], [63] comprising 1,491 holiday images, together with subsets selected from an additional 110,700 so-called "distractor" images from the INRIA website [64]. Prior to our test, each database image was preprocessed to derive the 8,192-length VLAD signature vector of integers.

B. System Description: StarCluster Comprising AWS EC2 Instances

We examine the cost implication of running the VLAD image retrieval algorithm using a five-instance AWS EC2 cluster based on MIT's open source StarCluster toolkit [43] with the MPICH2 plugin. Each instance is a quadcore `m3.xlarge` instance type running Ubuntu 12.04.2. In order to ensure cluster stability, StarCluster imposes that the master instance is setup as an on-demand instance type, while slave instances can either be spot or on-demand instance types. Unlike the case of the shared-memory cluster of Section V where the image database is in commonly-accessible memory, in this case the image database is equally spread over the four available slave instances, with the exception of the checksum-based approach, where the database is equally spread across three slave instances, with the fourth instance reserved for storing and computing with checksum data.

Firstly, the slave instances of the failure-intolerant implementation are set to run on on-demand instance types with the set price of $0.266 per hour for the `m3.xlarge` instance type [65], thereby ensuring no service interruption albeit at a

Table III

Execution time (in milliseconds) and percentile difference in comparison to the conventional failure-intolerant processing for recovery after a quadcore processor failure in a music retrieval system.

Method Database size	Conventional Failure-intolerant	Proposed Failure-tolerant	Checksum Failure-tolerant
4992	382.26	530.44 (38.8%)	529.68 (38.6%)
7488	490.85	629.41 (28.2%)	692.30 (41.0%)
9984	579.62	726.19 (25.3%)	812.98 (40.3%)
12480	729.13	829.82 (13.8%)	968.92 (32.9%)
17472	926.58	1046.79 (13.0%)	1308.99 (41.3%)

high instance cost. On the other hand, given that the proposed method and the checksum-based implementation of the image retrieval experiment can tolerate instance failures, they run on AWS EC2 spot instances with spot bidding price set the same as for the on-demand instances ($0.266 per hour). We monitor the actual expenditure for spot instances by monitoring the evolution of the spot price via the AWS command line interface (CLI). Although AWS EC2 spot instances are known to be about 60% cheaper than the corresponding on-demand instances, the spot price occasionally spikes above a user's spot bidding price (i.e., above $0.266 in our case), which results in the termination of the spot instances [66].

Fig. 4 shows the spot price history[6] corresponding to the week of our experiment. We superimpose on the figure a predetermined "safety threshold" of $0.100: when spot prices surpass this threshold, a spot price spike is impending and our system is designed to react (as elaborated in the next subsection) before the spot price reaches our $0.266 bid. For the seven-day spot instance history of Fig. 4, the minimum, maximum and average spot prices were found to be $0.034, $0.500 and $0.058 per hour, respectively.

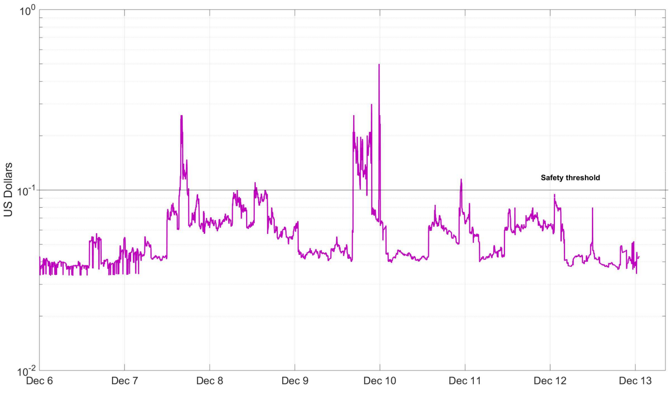

Figure 4. AWS EC2 m3.xlarge spot instance type pricing history for 06/Dec/2015–13/Dec/2015 [availability-zone: us-east-1e, product-description: Linux/UNIX (AWS VPC)].

C. Experiment Description and Results

Following this setup and the spot history of Fig. 4, every time the spot price rose to the safety threshold, three slave instances were migrated to on-demand instances (one at a time[7]) and the fourth instance was simply terminated. Thus, we run three slave instances at the on-demand price for one hour (since billing is carried out in hourly installments). If after one hour the spot price had dropped below the safety threshold, all four slave instances were brought back to spot instances and the on-demand instances were terminated. In this way, within the week reported in Fig. 4, there were 236 timestamps when the spot prices exceeded the safety threshold of $0.100, and they corresponded to 18 hours of on-demand instance usage (i.e., on average, 2 hours 34 minutes per day). Therefore, the total cost of running the failure-tolerant algorithms for the seven-day period (168 hours) is given by:

$$18 \times 3 \times \$0.266 + (168 - 18) \times 4 \times \$0.052 = \$45.564$$

where $0.052 corresponds to the mean of spot prices below the safety threshold for the seven-day period.

On the other hand, the failure-intolerant algorithm run at a flat rate of $0.266-per-instance-per-hour, amounting to $168 \times 4 \times \$0.266 = \$178.752$ for the seven-day time period. Table IV shows the cost savings for the VLAD image retrieval experiment per million images processed, under different database sizes.

The results of Table IV show that, as expected, with increased database size: *(i)* the cost of all approaches per million image queries is increasing; *(ii)* the cost reduction percentage of both approaches exhibits is also increasing. Overall, numerical packing allows for 16%–24% reduction of cost in comparison to the checksum-based method and offers, on average, 71.21% cost reduction (i.e., almost 3.5 times less cost) in comparison to the failure-intolerant realization.

VII. Conclusions

We propose a novel method for core failure mitigation in sum-of-product computations performed in multicore cloud computing platforms, with particular emphasis on integer matrix products and integer convolution/cross-correlation. Our approach inserts redundancy within the numerical representation of the inputs themselves by exploiting the concept of numerical packing. Therefore, our method does not perform any

[6] We pulled the exact spot prices by running the command: `aws ec2 describe-spot-price-history --instance-types m3.xlarge --start-time 2015-12-13T07:08:09 --end-time 2015-12-20T07:08:09 --availability-zone us-east-1e --product-description "Linux/UNIX (Amazon VPC)"` on the AWS CLI [67].

[7] we only need three slave instances in the proposed approach since the results of all four can be derived from three instances if we know no failures are bound to occur (which is the case when switching from spot to on-demand)

Table IV
COST PER MILLION IMAGE QUERIES (ALL IN US DOLLAR CENTS) AND PERCENTILE DIFFERENCE IN COMPARISON TO THE CONVENTIONAL FAILURE-INTOLERANT PROCESSING FOR RECOVERY AFTER A QUADCORE INSTANCE FAILURE IN VLAD-BASED IMAGE RETRIEVAL.

Method	Conventional Failure-intolerant	Proposed Failure-tolerant	Checksum Failure-tolerant
Database size	(×0.01$)	(×0.01$)	(×0.01$)
4992	0.29	0.09 (-68.88%)	0.11 (-61.25%)
11520	0.30	0.09 (-69.15%)	0.11 (-62.22%)
17472	0.37	0.10 (-71.78%)	0.13 (-64.10%)
82944	0.41	0.11 (-72.57%)	0.14 (-65.95%)
112128	0.59	0.16 (-73.66%)	0.19 (-68.64%)

redundant GEMM/CONV computations (akin to checksums) in order to mitigate core failures. We show theoretically and experimentally that this results in significantly-lower overhead in comparison to the equivalent checksum-based core failure mitigation method. Importantly, our approach achieves peak performance results that approach the conventional failure-intolerant computation as the matrix or signal dimensions increase, since the overhead of the required pre- and post-processing diminishes to zero. A deployment of the proposed approach over Amazon Web Services Elastic Compute Cloud (AWS EC2) spot instances that provide for cost savings (but require the mitigation of instance terminations) shows that the proposed method incurs nearly 3.5 times less cost than failure-intolerant processing. Future work could explore the use of higher number of concurrent packings for the mitigation of higher number of core failures (at the cost of decreased dynamic range), as well as the efficacy of the proposed approach on other multimedia applications based on integer sum-of-product computations.

APPENDIX

A. Proof of Proposition 1

Proof: It suffices to show that, given a set of $F = \left\lfloor \frac{L}{3} \right\rfloor$ consecutive core failures with the corresponding failed GEMMs denoted by: $f \in [0, L-1]$, $\forall f' \in \{f, \dots, (f+F-1) \bmod L\}$: $\widetilde{\mathbf{R}}_{f'}$, the proposed algorithm can recover all L GEMM outputs. In addition, for simplicity of exposition, we assume that L is a multiple of 3, with other cases requiring trivial extensions in the proof.

Following the packing algorithm of (7) and (9) and assuming the worst case of all F core failures, i.e. $F = \frac{L}{3}$, we recover the set $\widetilde{\mathbf{R}}_{f'}, \widetilde{\mathbf{R}}_{(f'+F) \bmod L}$ and $\widetilde{\mathbf{R}}_{(f'+2F) \bmod L}$ using only packed outputs, $\widehat{\mathbf{R}}_{(f'+F) \bmod L}$ and $\widehat{\mathbf{R}}_{(f'+2F) \bmod L}$ expressed by:

$$\begin{aligned}
\widehat{\mathbf{R}}_{(f'+F) \bmod L} \;\; = \;\; & \left(\left(\widetilde{\mathbf{R}}_{(f'+F) \bmod L,\text{top}}\right) \ll 2k\right) \\
+ \;\; & \left(\left(\widetilde{\mathbf{R}}_{(f'+F) \bmod L,\text{bot}}\right) \ll k\right) \\
+ \;\; & \left(\left(\widetilde{\mathbf{R}}_{(f'+2F) \bmod L,\text{top}}\right) \ll k\right) \\
+ \;\; & \widetilde{\mathbf{R}}_{(f'+2F) \bmod L,\text{bot}},
\end{aligned} \tag{26}$$

$$\begin{aligned}
\widehat{\mathbf{R}}_{(f'+2F) \bmod L} \;\; = \;\; & \left(\left(\widetilde{\mathbf{R}}_{(f'+2F) \bmod L,\text{top}}\right) \ll 2k\right) \\
+ \;\; & \left(\left(\widetilde{\mathbf{R}}_{(f'+2F) \bmod L,\text{bot}}\right) \ll k\right) \\
+ \;\; & \left(\left(\widetilde{\mathbf{R}}_{f',\text{top}}\right) \ll k\right) + \widetilde{\mathbf{R}}_{f',\text{bot}}.
\end{aligned} \tag{27}$$

First, the output shifted by $2k$ (equivalent to the “top” subset of columns of $\widetilde{\mathbf{R}}_{(f'+F) \bmod L}$), is extracted from the packed GEMM of (26). The obtained result is then removed from (26) in order to obtain the outputs scaled by k and 1 (to be used in subsequent steps) by:

$$\widetilde{\mathbf{R}}_{(f'+F) \bmod L,\text{top}} = \widehat{\mathbf{R}}_{(f'+F) \bmod L} \gg 2k \tag{28}$$

$$\mathbf{T}_0 = \widehat{\mathbf{R}}_{(f'+F) \bmod L} - \left(\widetilde{\mathbf{R}}_{(f'+F) \bmod L,\text{top}} \ll 2k\right) \tag{29}$$

$$\mathbf{T}_1 = \mathbf{T}_0 \gg k. \tag{30}$$

Because of the complement-two arithmetic used in integer representations in commodity hardware, all negative elements (i,j) of $\mathbf{T}_1$ will be found to be larger or equal to 2^{k-1} (maximum positive value within a packed output element) and $\widetilde{R}_{(f'+F) \bmod L,\text{top}}\left[i,j\right]$ (contained in the most-significant bits of $\widehat{R}_{(f'+F) \bmod L}\left[i,j\right]$) will be found to be one less than their correct value. To compensate for these effects of the complement-two arithmetic, we first define the intermediate matrix $\mathbf{E}_0$ to store the signed values of $\mathbf{T}_1$ therein, and adjust the values of $\widetilde{\mathbf{R}}_{(f'+F) \bmod L,\text{top}}$ as follows:

$\forall i,j$: if $T_1[i,j] \geq 2^{k-1}$, then set $E_0\left[i,j\right] = T_1\left[i,j\right] - 2^k$ (convert to negative number); else set $E_0\left[i,j\right] = T_1\left[i,j\right]$ (no change);

$\forall i,j$: if $E_0\left[i,j\right] < 0$, then set $\widetilde{R}_{(f'+F) \bmod L,\text{top}}\left[i,j\right] \leftarrow \left(\widetilde{R}_{(f'+F) \bmod L,\text{top}}\left[i,j\right] + 1\right)$.

Next, a similar check is performed to extract the signed representation of $\widetilde{\mathbf{R}}_{(f'+2F) \bmod L,\text{bot}}$ and the values of $\mathbf{E}_0$ are adjusted, i.e.:

$$\mathbf{T}_0 \leftarrow \left[\mathbf{T}_0 - \left(\mathbf{T}_1 \ll k\right)\right], \tag{31}$$

$\forall i,j$: if $T_0\left[i,j\right] \geq 2^{k-1}$, then set $\widetilde{R}_{(f'+2F) \bmod L,\text{bot}}\left[i,j\right] = \widetilde{R}_{(f'+2F) \bmod L,\text{bot}}\left[i,j\right] - 2^k$; else set $\widetilde{R}_{(f'+2F) \bmod L,\text{bot}}\left[i,j\right] = T_0\left[i,j\right]$;

$\forall i,j$: if $\widetilde{R}_{(f'+2F) \bmod L,\text{bot}}\left[i,j\right] < 0$, then set $E_0\left[i,j\right] \leftarrow \left(E_0\left[i,j\right] + 1\right)$.

Similarly, $\widehat{\mathbf{R}}_{(f'+2F) \bmod L}$ undergoes the same processing as $\widehat{\mathbf{R}}_{(f'+F) \bmod L}$ in order to extract $\widetilde{\mathbf{R}}_{(f'+2F) \bmod L,\text{top}}$, the intermediate matrix $\mathbf{E}_1 = \widetilde{\mathbf{R}}_{(f'+2F) \bmod L,\text{bot}} + \widetilde{\mathbf{R}}_{f' \bmod L,\text{top}}$ and $\widetilde{\mathbf{R}}_{f' \bmod L,\text{bot}}$. Finally, we perform the following operations to complete the extraction of all results:

$$\widetilde{\mathbf{R}}_{(f'+F) \bmod L,\text{bot}} = \mathbf{E}_0 - \widetilde{\mathbf{R}}_{(f'+2F) \bmod L,\text{top}} \tag{32}$$

$$\widetilde{\mathbf{R}}_{f' \bmod L,\text{top}} = \mathbf{E}_1 - \widetilde{\mathbf{R}}_{(f'+2F) \bmod L,\text{bot}}. \tag{33}$$

We have thus shown that all lost $\widetilde{\mathbf{R}}_{f'}$ GEMM outputs, together with the outputs $\widetilde{\mathbf{R}}_{(f'+F)\bmod L}$ and $\widetilde{\mathbf{R}}_{(f'+2F)\bmod L}$, have been recovered. ∎

B. Proof of Proposition 2

Proof: The proof follows the steps of the proof of Proposition 1, with the change of matrices to vectors and is omitted for brevity of description. Once all vectors $\widetilde{\mathbf{r}}_{l,\text{top}}$ and $\widetilde{\mathbf{r}}_{l,\text{bot}}$ have been recovered ($0 \le l < L$), given that the "top" vectors comprise output signal samples with indices within $\left[0, N + \frac{K}{2} - 1,\right]$ while the "bot" vectors comprise signals with indices within $\left[\frac{K}{2}, N + K - 1\right]$, the concatenation operations of (22) are required to obtain the final outputs. ∎

C. Proof of Proposition 3

Proof: Using a 64-bit integer representation for the proposed packing and processing, (6) allows for up to $\pm 2^{19}$ output dynamic range without any approximation. This means that 12 bits of dynamic range are sacrificed in comparison to the 32-bit integer representation, i.e., 37.5% of the bitwidth is sacrificed. ∎

D. Proof of Proposition 4

Proof: In the proposed approach, the packing of the L input matrices $\mathbf{A}_0, \ldots, \mathbf{A}_{L-1}$ and the $\mathbf{B}$ matrix requires $\frac{1}{2}N^2(2L+1)$ addition operations [see (5) and (9) and ignoring all arithmetic shift operations] and the packed matrix products require $\frac{L}{2}\left(2N^3 - N^2\right)$ operations. In terms of recovery, it can be shown via the analysis of Section IV-C that $\frac{7}{2}N^2(L-F)$ operations are required to extract all outputs from the $L - F$ matrices. Given that the proposed method requires double the bitwidth of the conventional failure-intolerant GEMM for its computation, the cycles count of these computations will also be doubled. Therefore, by doubling the sum of all arithmetic operations, the arithmetic operations of the proposed approach are:

$$\text{Cx}\left\{\text{proposed}\right\} = N^2\left(2NL + 8L - 7F + 1\right). \qquad (34)$$

For the mitigation of a single core failure or a group of F consecutive failures using the unweighted checksum-based method, $N^2(L-F)$ additions are required for the checksum generation. Subsequently, $L + F$ GEMMs are computed, which correspond to $\left(2N^3 - N^2\right)(L+F)$ operations. Post-processing is not required for the checksum-based method when no failures are encountered or when the failed cores are solely the cores holding the checksum matrices. In such cases, the arithmetic operations performed by the checksum-based method are:

$$\text{Cx}\left\{\text{no checksum failures}\right\} = N^2(2NF - 2F + 2NL) \qquad (35)$$

Combining (34) and (35) to calculate the percentile reduction in the arithmetic operations stemming from the proposed approach, we reach (23). ∎

E. Proof of Proposition 5

Proof: The proposed approach requires the operations given by (34) regardless of whether core failures occurred or not. However, under the occurrence of a group of F consecutive core failures, the checksum-based method must solve a system of linear equations to recover the lost data. Assuming the failed cores correspond to the cores computing the actual data outputs and not the checksum outputs, it can be shown that $N^2(L-F)$ operations are required for data recovery. Therefore, the overall number of arithmetic operations of the checksum-based roll-forward method when F core failures are encountered is:

$$\text{Cx}\left\{F \text{ checksum failures}\right\} = N^2\left(2NF - 3F + 2NL + L\right) \qquad (36)$$

Combining (34) and (36) to calculate the percentile reduction in the arithmetic operations stemming from the proposed approach, we reach (24). ∎

References

[1] D. Fiala *et al.*, "Detection and correction of silent data corruption for large-scale high-performance computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* IEEE Computer Society Press, 2012, p. 78.

[2] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. 10th Europ. Conf. on Comp. Syst. (Eurosys).* ACM, 2015, p. 18.

[3] Y. Gong, A. C. Zhou, and B. He, "Monetary cost optimizations for HPC applications on Amazon clouds: Checkpoints and replicated execution," *Supercomputing conference, SC'14 (Poster)*, 2014.

[4] C. S. Chan, B. Pan, K. Gross, K. Vaidyanathan, and T. Š. Rosing, "Correcting vibration-induced performance degradation in enterprise servers," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 3, pp. 83–88, 2014.

[5] W. Zhu *et al.*, "Multimedia cloud computing," *IEEE Signal Processing Magazine*, vol. 28, no. 3, pp. 59–69, May 2011.

[6] Y. Wen, X. Zhu, J. Rodrigues, and C. W. Chen, "Cloud mobile media: Reflections and outlook," *IEEE Trans. on Multimedia*, vol. 16, no. 4, pp. 885–902, June 2014.

[7] M. Nicolaidis *et al.*, "Design for test and reliability in ultimate CMOS," in *IEEE Design, Automat. & Test in Europe Conf. & Expo. (DATE), 2012.* IEEE, 2012, pp. 677–682.

[8] B. Carterette *et al.*, "Million query track 2009 overview," in *Proc. TREC'09*, vol. 9, 2009.

[9] Y.-G. Jiang, Q. Dai, T. Mei, Y. Rui, and S.-F. Chang, "Super fast event recognition in internet videos," *IEEE Trans. on Multimedia*, vol. 17, no. 8, pp. 1174–1186, Aug 2015.

[10] L. Page *et al.*, "The PageRank citation ranking: bringing order to the web." 1999.

[11] J. Yang *et al.*, "Two-dimensional PCA: a new approach to appearance-based face representation and recognition," *IEEE Trans. on Patt. Anal. and Machine Intell.*, vol. 26, no. 1, pp. 131–137, 2004.

[12] Z. Zhong, J. Zhu, and S. Hoi, "Fast object retrieval using direct spatial matching," *IEEE Trans. on Multimedia*, vol. 17, no. 8, pp. 1391–1397, Aug 2015.

[13] M. Intel, "Intel math kernel library," 2007.

[14] K. Goto and R. A. Van De Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Soft*, vol. 34, no. 3, p. 12, 2008.

[15] J. Sloan *et al.*, "Algorithmic approaches to low overhead fault detection for sparse linear algebra," in *Proc. 42nd Annual IEEE/IFIP Int. Conf. on Depend. Syst. and Netw. (DSN), 2012*, 2012, pp. 1–12.

[16] C. F. Van Loan, "The ubiquitous kronecker product," *Journal of computational and applied mathematics*, vol. 123, no. 1, pp. 85–100, 2000.

[17] J. Sun, D. Tao, S. Papadimitriou, P. S. Yu, and C. Faloutsos, "Incremental tensor analysis: Theory and applications," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 2, no. 3, p. 11, 2008.

[18] Y. Andreopoulos and I. Patras, "Incremental refinement of image salient-point detection," *IEEE Transactions on Image Processing*, vol. 17, no. 9, pp. 1685–1699, 2008.

[19] Y. Andreopoulos, “Error tolerant multimedia stream processing: There’s plenty of room at the top (of the system stack),” *IEEE Transactions on Multimedia*, vol. 15, no. 2, pp. 291–303, 2013.

[20] D. Anastasia and Y. Andreopoulos, “Linear image processing operations with operational tight packing,” *IEEE Sig. Process. Let.*, vol. 17, no. 4, pp. 375–378, 2010.

[21] A. Kadyrov and M. Petrou, “The" invaders’ algorithm: Range of values modulation for accelerated correlation,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 11, pp. 1882–1886, 2006.

[22] E. Stewart, “Intel integrated performance primitives: How to optimize software applications using Intel IPP,” 2004.

[23] I. P. Egwutuoha *et al.*, “A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems,” *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, 2013.

[24] M. Treaster, “A survey of fault-tolerance and fault-recovery techniques in parallel systems,” *ACM Computing Research Repository (CoRR*, vol. 501002, pp. 1–11, 2005.

[25] C. Wang *et al.*, “A job pause service under LAM/MPI+ BLCR for transparent fault tolerance,” in *IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2007*. IEEE, 2007, pp. 1–10.

[26] G. Bronevetsky *et al.*, “Automated application-level checkpointing of mpi programs,” *ACM Sigplan Notices*, vol. 38, no. 10, pp. 84–94, 2003.

[27] S. Di, E. Berrocal, and F. Cappello, “An efficient silent data corruption detection method with error-feedback control and even sampling for HPC applications,” in *Proc. 15th IEEE/ACM Int. Conf. on Clust., Cloud and Grid Comp. (CCGrid’15)*, 2015.

[28] I. Philp, “Software failures and the road to a petaflop machine,” in *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*, 2005.

[29] J. T. Daly *et al.*, “Application MTTFE vs. platform MTBF: A fresh perspective on system reliability and application throughput for computations at scale,” in *8th IEEE International Symposium on Cluster Computing and the Grid, 2008. CCGRID’08*. IEEE, 2008, pp. 795–800.

[30] Z. Chen and J. Dongarra, “Algorithm-based fault tolerance for fail-stop failures,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 12, pp. 1628–1641, 2008.

[31] C. Engelmann *et al.*, “The case for modular redundancy in large-scale high performance computing systems,” in *Proc. IASTED Internat. Conf.*, vol. 641, 2009, p. 046.

[32] J.-Y. Jou and J. Abraham, “Fault-tolerant matrix arithmetic and signal processing on highly concurrent computing structures,” in *Proc. of the IEEE*, vol. 74, no. 5. IEEE, May 1986, pp. 732,741.

[33] Z. Chen, “Optimal real number codes for fault tolerant matrix operations,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 29.

[34] S. Sundaram and C. N. Hadjicostis, “Fault-tolerant convolution via chinese remainder codes constructed from non-coprime moduli,” *IEEE Transactions on Signal Processing*, vol. 56, no. 9, pp. 4244–4254, 2008.

[35] K.-H. Huang and J. A. Abraham, “Algorithm-based fault tolerance for matrix operations,” *IEEE Trans. on Computers*, vol. 100, no. 6, pp. 518–528, 1984.

[36] P. Du *et al.*, “Algorithm-based fault tolerance for dense matrix factorizations,” *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 225–234, 2012.

[37] P. E. Beckmann and B. R. Musicus, “Fast fault-tolerant digital convolution using a polynomial residue number system,” *IEEE Transactions on Signal Processing*, vol. 41, no. 7, pp. 2300–2313, 1993.

[38] Z. Chen, “Algorithm-based recovery for iterative methods without checkpointing,” in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*. ACM, 2011, pp. 73–84.

[39] N. Firasta *et al.*, “Intel AVX: New frontiers in performance improvements and energy efficiency,” *Intel White paper*, 2008.

[40] I. Anarado *et al.*, “Highly-reliable integer matrix multiplication via numerical packing,” in *Proc. 19th IEEE Internat. On-Line Testing Sympos. (IOLTS’13)*, 2013.

[41] G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, Z. Chen, J. Pjesivac-Grbovic, K. London, and J. J. Dongarra, “Extending the MPI specification for process fault tolerance on high performance computing systems,” in *Proceedings of the International Supercomputer Conference (ICS)*, 2004.

[42] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, “Open mpi: Goals, concept, and design of a next generation mpi implementation,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2004, pp. 97–104.

[43] MIT, “Starcluster,” http://star.mit.edu/cluster/.

[44] H. Tabia and H. Laga, “Covariance-based descriptors for efficient 3d shape matching, retrieval, and classification,” *IEEE Trans. on Multimedia*, vol. 17, no. 9, pp. 1591–1603, Sept 2015.

[45] D. Anastasia and Y. Andreopoulos, “Throughput-distortion computation of generic matrix multiplication: Toward a computation channel for digital signal processing systems,” *IEEE Trans. on Signal Processing*, vol. 60, no. 4, pp. 2024–2037, 2012.

[46] V. K. Stefanidis and K. G. Margaritis, “Algorithm based fault tolerance: Review and experimental study,” in *Proc. Int. Conf. of Numer. Anal. and Appl. Math.* IEEE, 2004.

[47] F. T. Luk, “Algorithm-based fault tolerance for parallel matrix equation solvers,” *SPIE Real-Time Signal processing VIII*, vol. 564, pp. 631–635, 1985.

[48] M. A. Anam and Y. Andreopoulos, “Throughput scaling of convolution for error-tolerant multimedia applications,” *IEEE Transactions on Multimedia*, vol. 14, no. 3, pp. 797–804, 2012.

[49] D. Anastasia and Y. Andreopoulos, “Software designs of image processing tasks with incremental refinement of computation,” *IEEE Trans. on Image Processing*, vol. 19, no. 8, pp. 2099–2114, 2010.

[50] S. Ohshima, K. Kise, T. Katagiri, and T. Yuba, “Parallel processing of matrix multiplication in a cpu and gpu heterogeneous environment,” in *High Performance Computing for Computational Science-VECPAR 2006*. Springer, 2007, pp. 305–318.

[51] Intel, “Intel intrinsics guide,” http://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[52] J.-Y. Jou and J. A. Abraham, “Fault-tolerant matrix operations on multiple processor systems using weighted checksums,” in *Proc. 28th Annual Tech. Symp.* International Society for Optics and Photonics, 1984, pp. 94–101.

[53] J. Rexford and N. Jha, “Algorithm-based fault tolerance for floating-point operations in massively parallel systems,” in *Proc. IEEE Int. Symp. on Circ. and Syst.*, vol. 2. IEEE, May 1992, pp. 649,652.

[54] K. A. (Intel), “Recommended settings for calling intel MKL routines from multi-threaded applications,” https://software.intel.com/en-us/articles/recommended-settings-for-calling-intel-mkl-routines-from-multi-threaded-applications, 2011.

[55] C. Y. (Intel), “Intel MKL 10.x threading,” https://software.intel.com/en-us/articles/intel-math-kernel-library-intel-mkl-intel-mkl-100-threading, 2010.

[56] D. P. Ellis, C. V. Cotton, and M. I. Mandel, “Cross-correlation of beat-synchronous representations for music similarity,” in *IEEE International Conference on Acoustics, Speech and Signal Processing, 2008. ICASSP 2008*. IEEE, 2008, pp. 57–60.

[57] M. A. Anam, P. Whatmough, and Y. Andreopoulos, “Precision–energy–throughput scaling of generic matrix multiplication and convolution kernels via linear projections,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 24, no. 11, pp. 1860–1873, 2014.

[58] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere, “The million song dataset,” in *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[59] K. Seyerlehner, G. Widmer, and T. Pohle, “Fusing block-level features for music similarity estimation,” in *Proc. of the 13th Int. Conference on Digital Audio Effects (DAFx-10)*, 2010, pp. 225–232.

[60] H. Jégou, M. Douze, C. Schmid, and P. Pérez, “Aggregating local descriptors into a compact image representation,” in *2010 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2010, pp. 3304–3311.

[61] G. Amato, P. Bolettieri, F. Falchi, and C. Gennaro, “Large scale image retrieval using vector of locally aggregated descriptors,” in *Similarity Search and Applications*. Springer, 2013, pp. 245–256.

[62] G. Tolias, Y. Avrithis, and H. Jégou, “To aggregate or not to aggregate: Selective match kernels for image search,” in *2013 IEEE International Conference on Computer Vision (ICCV)*. IEEE, 2013, pp. 1401–1408.

[63] H. Jégou, F. Perronnin, M. Douze, J. Sanchez, P. Perez, and C. Schmid, “Aggregating local image descriptors into compact codes,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 9, pp. 1704–1716, 2012.

[64] J. H. (INRIA), “INRIA holidays dataset,” http://lear.inrialpes.fr/~jegou/data.php#holidays, 2015.

[65] Amazon, “Amazon EC2 pricing,” https://aws.amazon.com/ec2/pricing/.

[66] ——, “New EC2 spot instance termination notices,” https://aws.amazon.com/blogs/aws/new-ec2-spot-instance-termination-notices/.

[67] ——, “Amazon EC2 spot price history,” http://docs.amazonaws.cn/cli/latest/reference/ec2/describe-spot-price-history.html.

Ijeoma Anarado is currently pursuing the Ph.D. degree at the Department of Electronic and Electrical Engineering, University College London, U.K. Her research interests include the design of system level algorithms for fault tolerance in data computations and throughput acceleration in signal processing tasks. Her PhD is funded by the Federal Government of Nigeria under the PRESSID Scheme.

Yiannis Andreopoulos (M'00-SM'14) is Reader (Assoc. Professor) in Data and Signal Processing Systems in the Department of Electronic and Electrical Engineering of University College London (U.K.). His research interests are in wireless sensor networks, error-tolerant computing and multimedia systems. He received the 2007 Most-Cited Paper Award from the Elsevier EURASIP Signal Processing: Image Communication journal and a best paper award from the 2009 IEEE Workshop on Signal Processing Systems. He was Special Sessions Co-Chair of the 10th International Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2009) and Programme Co-Chair of the 18th International Conference on Multimedia Modeling (MMM 2012) and the 9th International Conference on Body Area Networks (BODYNETS 2014). He has been an Associate Editor of the IEEE Transactions on Multimedia, the IEEE Signal Processing Letters and Image and Vision Computing (Elsevier).