# Mitigating Silent Data Corruptions In Integer Matrix Products: Toward Reliable Multimedia Computing On Unreliable Hardware

Ijeoma Anarado*, Mohammad Ashraful Anam, Fabio Verdicchio, and Yiannis Andreopoulos

***Abstract*—The generic matrix multiply (GEMM) routine comprises the compute and memory-intensive part of many information retrieval, machine learning and object recognition systems that process integer inputs. Therefore, it is of paramount importance to ensure that integer GEMM computations remain robust to silent data corruptions (SDCs), which stem from accidental voltage or frequency overscaling, or other hardware non-idealities. In this paper, we introduce a new method for SDC mitigation based on the concept of numerical packing. The key difference between our approach and all existing methods is the production of redundant results *within* the numerical representation of the outputs, rather than as a separate set of checksums. Importantly, unlike well-known algorithm-based fault tolerance (ABFT) approaches for GEMM, the proposed approach can reliably detect the locations of the vast majority of all possible SDCs in the results of GEMM computations. An experimental investigation of voltage-scaled integer GEMM computations for visual descriptor matching within state-of-the-art image and video retrieval algorithms running on an Intel i7- 4578U 3GHz processor shows that SDC mitigation based on numerical packing leads to comparable or lower execution and energy-consumption overhead in comparison to all other alternatives.**

***Index Terms*—integer matrix multiplication, dependable systems, fault tolerance, soft errors, voltage scaling**

## I. Introduction

THE increase of integration density and the power wall of future CMOS technologies now require increased levels of resilience to transient arithmetic, memory or logic faults caused by process variations and other soft errors (e.g., caused by particle strikes,circuit overclocking or voltage overscaling) [43], [53], [58]. Voltage overscaling in particular is emerging as one of the most effective methods for reducing power and energy consumption in processor designs [2], [58]. However, to ensure that processors remain reliable, strong error detection and correction mechanisms are required, especially for the case of *silent data corruptions* (SDCs), i.e., soft errors (occurring primarily on DRAM [2], [55]) that remain undetected because they do not lead to a system halt or crash. This is now a challenge of increasing importance for prevalent applications in mobile, desktop and high-performance multimedia systems [20], [40], [56], [59], [62], where SDCs affecting critical data sections [38], [55] can result in incorrect outputs, or cause the system to behave unexpectedly. Therefore, reliability measures, such as caches with hardware-supported error-correction codes (ECC) are now becoming the norm in most processor designs. However, the constantly increasing SDC rates along with the complex corruption patterns (e.g., burst errors and multibit upsets) found in aggressively-scaled systems require increasingly-sophisticated ECC methods, which are known to incur very substantial area, performance and power overheads [3], [45], [48]. This is why the International Technology Roadmap for Semiconductors [34] put forward that mitigation of SDC bursts should also be based on fault-tolerant software/algorithmic designs rather than solely depending on expensive circuit-level techniques [34]. This has led to a growing number of research proposals for developing cross-layer system reliability designs [14], [45], which ensure that systems remain reliable under unreliable hardware.

Mitigation techniques for SDCs are especially important for *matrix products* performed during descriptor matching, power iterations, backpropagation, transform decompositions, random projections, kernel methods, covariance matrix calculations and block Lanczos iterations within information retrieval, object detection and tracking, machine learning, and classification applications [8]–[10], [12], [15], [16], [20], [29], [35], [40], [52], [56], [59], [62]. This is because: *(i)* matrix-product computations comprise the bulk of processing in such applications;*(ii)* subsequent processing stages apply low-complex-yet-noise-sensitive thresholding, sorting and clustering operations [29], [33], [38], [44] and any undetected SDCs in the matrix-product stage can have severe repercussions in the final results. Many such matrix products utilize integer inputs and are typically performed with an integer generic matrix multiply (GEMM) routine, or the single- or double-precision floating-point GEMM (sGEMM or dGEMM) routines of a high-performance mathematics kernel library (MKL) [22], [28]. Thus, ensuring that integer matrix products remain reliable against SDCs is of paramount importance for such multimedia systems.


*Corresponding author. IA, MAA, YA 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 (all authors); email: {ijeoma.anarado.12, mohammad.anam.10, i.andreopoulos}@ucl.ac.uk. FV is with the School of Engineering, University of Aberdeen, King's College, Fraser Noble building, Aberdeen, AB24 3UE, UK; tel. +44 1224 273665; email: fverdicc@abdn.ac.uk. This work was supported by EPSRC, project EP/M00113X/1. The work of I. Anarado was supported by the Federal Government of Nigeria under the PRESSID Scheme.

### A. Summary of Prior Work

Existing techniques that can ensure reliability to SDCs comprise two categories: *(i)* algorithm-based fault tolerance[1] (ABFT)—i.e., methods using checksums specifically tailored to the algorithm under consideration—that can reliably detect (and possibly correct) up to a limited number of SDCs [13], [17], [19], [25], [39], [46], [47], [60]; *(ii)* systems with dual modular redundancy (DMR), where all non-coinciding SDCs can be detected if the same operation is duplicated in two separate processors (or threads) that cross-validate their results [21], but SDCs cannot be corrected without using triple modular redundancy (TMR) [23].

It is well known that DMR/TMR solutions lead to substantial processing overhead in hardware/software systems and incur increased energy consumption [21]. On the other hand, while ABFT in GEMM comes with limited implementation overhead [25], it can only detect the locations of a limited number of SDCs (typically up to one or two) for an entire GEMM computation. To address this issue, modified ABFT (mABFT) schemes were proposed [13], [17], [30], [31], [39], [41], [47], [54], [61], which are designed to locate and correct more SDCs per GEMM. However, mABFT approaches incur considerable implementation overhead and significant reduction in available bitwidth for integer GEMM results [13], [17], [30], [31], [39], [41], [47], [54], [61]. In conjunction with recent studies indicating that SDCs in large cache memories tend to increase drastically under voltage overscaling [2], [43], [45], these observations indicate that ABFT and mABFT may ultimately not be the most efficient approaches to detect *arbitrary SDC patterns* occurring in GEMM computations under increased rates of transient hardware faults.

### B. Contribution

The proposed method detects any number of SDCs in integer matrix products by creating redundant results *within* the numerical representation of the output results. This is achieved by packing pairs of inputs within one integer number and performing two complementary quarter-size GEMM calls. Beyond the desired outputs, these products *spontaneously create numerically-entangled pairs of outputs within the numerical representation of the results*. All outputs are then extracted and, in conjunction with a very low-complex row-column ABFT check, are cross-validated with the numerically-entangled outputs to detect arbitrary SDCs with very high probability.

Calculations of the computational complexity, as well as experimental results on an Intel Sandy Bridge processor, demonstrate that the average execution time overhead of the proposed method against high-performance, fault-intolerant, GEMM realizations under SDC-free conditions is limited to $11.50\%$. Therefore, the reliability and execution time efficiency of our method is comparable to that of ABFT and, unlike ABFT, our method is capable of detecting the locations of the vast majority of all possible SDCs. This makes our method superior to ABFT, mABFT and DMR under the occurrence of multiple SDCs in memory. Finally, via an SDC-injection campaign using an open source low-level virtual machine (LLVM) based fault injection tool, the Kontrollable Utah LLVM Fault Injector (KULFI) [49], [50], we demonstrate that our proposal paves the way for a new class of SDC mitigation methods for integer matrix products that are fast, energy efficient, and highly reliable.

The remainder of the paper is organized as follows. In Section II, we summarize the operation of high-performance GEMM routines with different variants of ABFT and highlight their differences with the proposed approach. In Section III, we describe the proposed approach starting from the previously-proposed concept of numerical packing. Sections IV and V-B provide performance results and results with a voltage-scaled platform and Section VI concludes the paper.

## II. High-performance GEMM, ABFT and mABFT

Consider the GEMM design depicted in Figure 1(a), which follows the general structure found in optimized MKL designs [22]. The application calls GEMM for an $M \times K$ by $K \times N$ matrix multiplication which is further subdivided into $L \times L$ "inner-kernel" matrix products. For our approach, $L$ is specified by ($k \in \mathbb{N}^{\star}$):

$$L = 2k \times \frac{\text{SIMD}_{\text{bits}}}{b_{\text{repr}}} \tag{1}$$

with: $\text{SIMD}_{\text{bits}}$ the number of bits of each SIMD register ($\text{SIMD}_{\text{bits}} = 256$ in this work); $b_{\text{repr}} \in \{32, 64\}$ the number of bits for floating-point or integer representations. The inner-kernel result[2] $\mathbf{R}_{2,1}$ of the example shown in Figure 1(a) comprises the sum of multiple subblock multiplications $\mathbf{A}_{2,l}\mathbf{B}_{l,1}$:

$$\mathbf{R}_{2,1} = \sum_{l=0}^{\frac{K}{L}-1} \mathbf{A}_{2,l}\mathbf{B}_{l,1}. \tag{2}$$

If the matrices' dimensions are not multiples of $L$, some "cleanup" code [22] is applied at the borders to complete the inner-kernel results of the overall matrix multiplication. This separation into top-level processing and subblock-level processing is done for efficient cache utilization. Specifically, during the initial data access of GEMM for top-level processing, data in matrix $\mathbf{A}$ and $\mathbf{B}$ is reordered into block major format: for each $L \times L$ pair of subblocks $\mathbf{A}_{i,l}$ and $\mathbf{B}_{l,j}$ multiplied to produce inner-kernel result $\mathbf{R}_{i,j}$, $0 \leq l < \frac{K}{L}$, $0 \leq i < \frac{M}{L}$, $0 \leq j < \frac{N}{L}$, the input data within $\mathbf{A}_{i,l}$ and $\mathbf{B}_{l,j}$ is reordered in rowwise and columnwise raster manner, respectively. Thus, sequential data accesses are performed during each subblock matrix multiplication and this enables the use of SIMD instructions, thereby leading to significant acceleration.

[1] In this paper, ABFT refers to Huang and Abraham's original proposal [25], while mABFT is used to refer to various modifications and extensions, i.e., including the use of weighted checksums and subblock partitioning.

[2] Notations: Boldface uppercase and lowercase letters indicate matrices and vectors, respectively. The corresponding italicized lowercase indicate their individual elements, e.g., $\mathbf{A}$ and $a_{m,n}$. All indices are integers. Assuming two integers $a$ and $b$, $a \gg b$ 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 $a$. The packed and extracted value of $x$ is indicated by $\hat{x}$ and $\tilde{x}$, respectively.

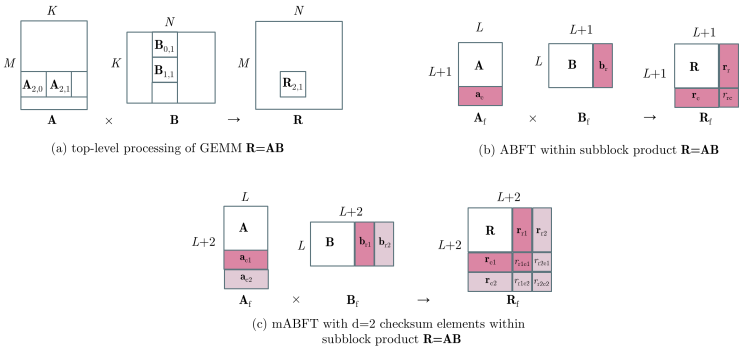


Figure 1: (a) Top-level processing of GEMM highlighting the input subblocks used for the computation of the subblock result $\mathbf{R}_{2,1}$; (b) ABFT within a single subblock of GEMM via checksum vectors; (c) mABFT within a single subblock with two checksum vectors.

From this description it is evident that the top-level processing simply administers the computation (and, optionally, error detection) at the subblock level and the core operations are performed within each subblock independently, before being aggregated to produce the final results. Therefore, in the remainder of the paper we present ABFT, mABFT and our approach in reference to a single subblock matrix product. For notational simplicity, we remove the indices from each subblock product $\mathbf{A}_{i,l}\mathbf{B}_{l,j}$. Moreover, despite the block-major format reordering, we retain the 2D indexing in the equations to facilitate intuitive understanding of the proposed method.

### A. Algorithm-based Fault Tolerance

All ABFT methods specifically tailored for GEMM computations [13], [17], [25], [60], [61] append the input subblocks with (redundant) checksum vectors (rows or columns), denoted by $\mathbf{a}_\mathrm{c}$, $\mathbf{b}_\mathrm{r}$ in Figure 1(b) and highlighted in color.

In the ABFT approach proposed by Huang and Abraham [25], a single checksum row and column per subblock is used for error detection and correction. Specifically, checksum vector $\mathbf{a}_\mathrm{c}$ is the sum across each column of $\mathbf{A}$. Checksum vector $\mathbf{b}_\mathrm{r}$ is the sum across each row of $\mathbf{B}$.

Under the described configuration of Figure 1(b), the result of the checksum-appended subblock GEMM, $\mathbf{R}_\mathrm{f} = \mathbf{A}_\mathrm{f}\mathbf{B}_\mathrm{f}$, can be checked for the detection and correction of any single SDC within the subblock product $\mathbf{R}$ [13], [17], [25], [61]. Specifically: *(i)* the sum of each row and each column of $\mathbf{R}$ (which is contained within $\mathbf{R}_\mathrm{f}$) is validated against the corresponding checksum elements of $\mathbf{r}_\mathrm{r}$ and $\mathbf{r}_\mathrm{c}$; *(ii)* the sum of the checksum vectors themselves is validated against $r_\mathrm{rc}$. If these validations fail for a certain row and column, the index of this row and column indicates the location of the SDC within $\mathbf{R}_\mathrm{f}$.

However, when multiple SDCs occur within a subblock multiplication, ABFT cannot pinpoint their exact locations and requires recomputation of results, which is termed as "rollback ABFT" [1]. For example, given the $\pm\varepsilon$ and $\delta$ error pattern shown in Figure 2, ABFT flags locations $r_{3,1}$ and $r_{3,3}$ as erroneous, while two other erroneous locations, $r_{1,1}$ and $r_{1,3}$ go undetected due to the cancellation effect of the given error pattern. Therefore, when multiple SDCs are detected, the only reliable method for recovery via ABFT is to recompute (i.e., "rollback") the entirety of the rows and columns that have been detected as erroneous. If a substantial number of SDCs is detected, this will result in entire GEMM subblock recomputation, i.e., complete execution rollback, incurring significant execution time penalty.

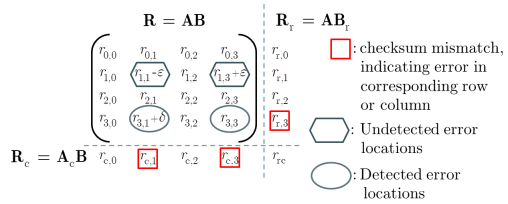


Figure 2: Errors in ABFT that require row and column recomputation (rollback ABFT [1]).

Finally, due to the fact that the checksum inputs $\mathbf{a}_\mathrm{c}$ and $\mathbf{b}_\mathrm{r}$ are the columnwise or rowwise summation of inputs, the elements of the row and column checksums, $\mathbf{r}_\mathrm{r}$ and $\mathbf{r}_\mathrm{c}$, will have increased dynamic range by $\log_2 L$ bits for an $L \times L$ by $L \times L$ integer GEMM. In addition, given that $r_\mathrm{rc}$ is the sum of the elements of $\mathbf{r}_\mathrm{r}$ and $\mathbf{r}_\mathrm{c}$, this element will require $2\log_2 L$ additional bits. Therefore, in comparison to the conventional (fault-intolerant) GEMM computation and for $L = 288$ and 32-bit integer representations, ABFT incurs loss of

$$\left.\frac{\log_2 L}{15.5}\right|_{L=288} \times 100\% \simeq 0.53 \tag{3}$$

of output dynamic range in order to accommodate all checksum results.

### B. Modified Algorithm-based Fault Tolerance

More recently, modified ABFT methods were proposed that make use of *additional* checksums, constructed with specific weight vectors. The number of additional checksums and their dynamic range is dependent on the desired detectability.

For example, Jou, Anfison *et. al.* [11], [30] extended ABFT with the introduction of $d$ checksum rows and columns, $d > 1$. All checksum elements beyond the first one are generated via inner products of rows and columns with specially-constructed weight vectors. It is then shown that, under suitable choice of

weight vectors, this extension can guarantee the detection of up to $d$ SDCs and correction of up to $\lfloor \frac{d}{2} \rfloor$ SDCs [11], [30], [47] per row and column of a subblock product. This comes at the cost of increasing the size of the checksum matrices by $d$ times. An illustration of a subblock product with $d = 2$ is given in Figure 1(c).

In more detail, the first checksum of each column of $\mathbf{A}$ and row of $\mathbf{B}$ is the one of the original ABFT; we denote this as the *unweighted checksum*, $\mathbf{a}_{c1}$ and $\mathbf{b}_{r1}$, respectively. For an mABFT approach with $d = 2$, the second checksum of each column of $\mathbf{A}$ and row of $\mathbf{B}$ (denoted by $\mathbf{a}_{c2}$ and $\mathbf{b}_{r2}$, respectively) is *the inner product of the column* of $\mathbf{A}$ (resp. *inner product of the row* of $\mathbf{B}$). Jou and Abraham [30] proposed the weight vector $\mathbf{w}_{exp}$ given by:

$$\mathbf{w}_{exp} = \begin{bmatrix} 2^0 & 2^1 & \dots & 2^{L-1} \end{bmatrix}. \tag{4}$$

However, due to the exponential increase of the dynamic range of $\mathbf{a}_{c2}$ and $\mathbf{b}_{r2}$, the use of $\mathbf{w}_{exp}$ will cause overflow problems under 32-bit or 64-bit integer representations. Such an mABFT approach is therefore mostly of theoretical interest, rather than of practical relevance to conventional numerical representations used in programmable processors.

As a remedy to the dynamic range expansion caused by the weight vector of (4), Rexford and Jha [47] proposed the further partitioning of input subblocks into smaller blocks. Complementary to this approach, Luk *et. al.* [39], [54] proposed the weights:

$$\mathbf{w}_{linear} = \begin{bmatrix} 1 & 2 & \dots & L \end{bmatrix} \tag{5}$$

instead of $\mathbf{w}_{exp}$ of (4), in order to allow for quadratic increase of dynamic range in $\mathbf{a}_{c2}$ and $\mathbf{b}_{r2}$ and quartic increase of dynamic range for the second row-column checksum element, $r_{r2c2}$. Specifically, the quartic increase in the latter is by factor $\left(\sum_{l=1}^{L} l\right)^2$, or $2\left\lceil \log_2\left(L^2 + L\right) - 1 \right\rceil$ bits. Their work shows that, under the use of $\mathbf{w}_{linear}$ and for $d = 2$, up to two SDCs per row or column of a subblock can be reliably detected and up to one SDC can be reliably corrected. Therefore, the combination of partitioning of $L \times L$ subblocks into $P \times P$ blocks with: $P = 16$, $d = 2$ and the length-$P$ linear weights $\mathbf{w}_{linear}$ (with $L$ replaced by $P$) allows for mABFT to operate with 32-bit representations. However, this comes at the loss of

$$\left. \frac{\log_2\left(P^2 + P\right) - 1}{15.5} \right|_{P=16} \times 100\% \simeq 45.7\% \tag{6}$$

of output dynamic range for integer representations in comparison to the conventional (fault-intolerant) GEMM computation.

## III. Proposed Fault-tolerant Numerical Packing in GEMM

Packed processing has been proposed for throughput-precision scaling of GEMM computations in our previous work [6]. Since the proposed method is built upon the concept of packing [6], this is briefly summarized here. To facilitate the exposition, we shall be referring to the illustrations of Figure 3(d)–(g) and Figure 4, which present the steps of the described methods for the elementary case of a $2 \times 2$ matrix

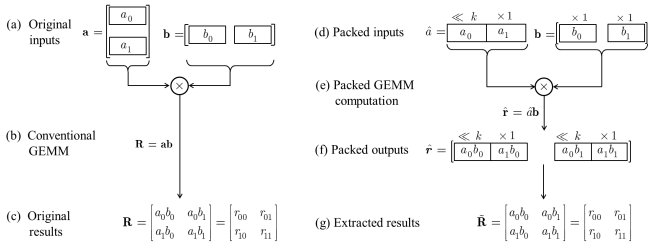


Figure 3: (a)–(c) Conventional integer subblock multiplication for the case of a $2 \times 1$ by $1 \times 2$ vector product. (d)–(g) Packing for the same product with packing coefficient $k = 10$. The partitioning within the rectangles shows the location (shifts by $k$ bits) of inputs/outputs when packed within a single number.

produced via a $2 \times 1$ by $1 \times 2$ vector product. In addition, the conventional (fault-intolerant) approach is illustrated in Figure 3(a)–(c). Finally, beyond this elementary case, our analytic exposition will be based on the general case of an $L \times L$ subblock product.

### A. Review of Packing in Integer Subblock Matrix Multiplication

In packing for integer subblock multiplication $\mathbf{R} = \mathbf{AB}$, only one input subblock is packed, i.e., either $\mathbf{A}$ or $\mathbf{B}$. This selection does not affect the performance in the case of GEMM, as both subblocks have been reordered in block-major format. Assuming $\mathbf{A}$ is chosen, the packing process creates block $\hat{\mathbf{A}}$ with $\frac{L}{2} \times L$ coefficients given by ($\forall m, n: \ 0 \le m, n < L, \ \hat{m} = \left\lfloor \frac{m}{2} \right\rfloor$):

$$\hat{a}_{\hat{m},n} = \left(a_{2\hat{m},n} \ll k\right) + a_{2\hat{m}+1,n} \tag{7}$$

where $k$ is the utilized packing coefficient , $k \in \mathbb{N}^{\star}$. The utilized value for $k$ depend on the maximum possible value of the matrix product, as it will be elaborated in the following. An illustration of the packing of (7) for the two elements of a $2 \times 1$ vector $\mathbf{a}$ is given in Figure 3(d), i.e., for $n = \hat{m} = 0$.

Notice that (7) operates along the columns of $\hat{\mathbf{A}}$ in order to pack rows $2\hat{m}$ and $2\hat{m}+1$ together. This means that, in order to use integer SIMD instructions for accelerated computation of (7), we can group $\frac{\text{SIMD}_{\text{bits}}}{b_{\text{repr}}}$ consecutive elements of each row and apply each multiply-accumulate (MAC) operation of (7) to them with one SIMD instruction. Once (7) is completed, processing occurs via $\hat{\mathbf{R}} = \hat{\mathbf{A}}\mathbf{B}$ ($\forall m, n: \ 0 \le m, n < L, \ \hat{m} = \left\lfloor \frac{m}{2} \right\rfloor$):

$$\begin{aligned} \hat{r}_{\hat{m},n} &= \sum_{j=0}^{L-1} \hat{a}_{\hat{m},j} b_{j,n} \\ &= \left( \sum_{j=0}^{L-1} a_{2\hat{m},j} b_{j,n} \ll k \right) + \sum_{j=0}^{L-1} a_{2\hat{m}+1,j} b_{j,n} \end{aligned} \tag{8}$$

The packed output of (8) contains both rows of the results, $\hat{r}_{2\hat{m},n}$ and $\hat{r}_{2\hat{m}+1,n}$, packed together. An example is shown in Figure 3(f) for $n = \hat{m} = 0$. Importantly, if they do not overlap in the packed representation and the numerical representation

used can accommodate both packed outputs, both rows can be computed concurrently via (8). For the packed processing of two inputs shown in (7) and (8), these two conditions are met when the packing coefficient satisfies the below constraint [4]–[6]:

- For 32/64-bit integer representation: $k > \log_2\left(\max_{\forall m,n}|r_{m,n}|\right) + 1$ and $2k \leq W$, with $W \in \{32, 64\}$.

Any high-performance $\frac{L}{2} \times L$ by $L \times L$ subblock code for 32/64-bit GEMM (integer or floating-point) can be used for the computation of (8). Following the completion of the processing, unpacking of the results can be performed by the following process [4]–[6] ($\forall m, n: \ 0 \leq m, n < L, \ \hat{m} = \left\lfloor \frac{m}{2} \right\rfloor$).

The first output, $\widetilde{r}_{2\hat{m}+1,n}$, which is packed at the most significant bits of $\hat{r}_{\hat{m},n}$, is extracted by:

$$\widetilde{r}_{2\hat{m},n} = \hat{r}_{\hat{m},n} \gg k \tag{9}$$

The extracted output is then removed from $\hat{r}_{\hat{m},n}$:

$$\widetilde{r}_{2\hat{m}+1,n} = \hat{r}_{\hat{m},n} - \left[\left(\hat{r}_{\hat{m},n} \gg k\right) \ll k\right], \tag{10}$$

The result is then converted into its signed representation. Specifically, if $\widetilde{r}_{2\hat{m}+1,n} > 2^{k-1}$, then $\widetilde{r}_{2\hat{m}+1,n} \leftarrow \left(\widetilde{r}_{2\hat{m},n} - 2^k\right)$.

Finally if $\widetilde{r}_{2\hat{m}+1,n} < 0$, then the result extracted from the most-significant bits must be incremented by one, i.e.: $\widetilde{r}_{2\hat{m},n} \leftarrow (\widetilde{r}_{2\hat{m},n} + 1)$.

### B. Proposed Fault-tolerant Packing

We utilize the packing concept in order to provide highly-reliable integer matrix products within each GEMM call. Our method uses standard 64-bit integer representations and off-the-shelf GEMM subblock kernels to process 32-bit inputs, as detailed in the following.

*1) Packing Process:* The proposed approach performs **two packings** of matrix $\mathbf{A}$ into $\hat{\mathbf{A}}_i$ and $\hat{\mathbf{A}}_j$ and one packing of $\mathbf{B}$ into $\hat{\mathbf{B}}$, as shown in the example of Figure 4(a). The figure illustrates only the case of an elementary $2 \times 1$ by $1 \times 2$ vector product with $k = 10$ and, due to the small input vector dimensions, the packed matrices are in fact scalars $\hat{a}_i$, $\hat{a}_j$ and $\hat{b}$. For the general case of an $L \times L$ by $L \times L$ subblock product, the process is identical for all other pairs of inputs within $\mathbf{A}$ and $\mathbf{B}$ and is expressed by ($\forall m, n: \ 0 \leq m, n < L, \ \hat{m} = \left\lfloor \frac{m}{2} \right\rfloor \ \hat{n} = \left\lfloor \frac{n}{2} \right\rfloor$):

$$\hat{a}_{i,\hat{m},n} = \left(a_{2\hat{m},n} \ll k\right) - a_{2\hat{m}+1,n} \tag{11}$$

$$\hat{a}_{j,\hat{m},n} = \left(a_{2\hat{m}+1,n} \ll k\right) - a_{2\hat{m},n} \tag{12}$$

$$\hat{b}_{m,\hat{n}} = \left(b_{m,2\hat{n}} \ll k\right) + b_{m,2\hat{n}+1} \tag{13}$$

Each packing "stacks" two inputs together in a single 64-bit integer number (assuming 32-bit inputs) and can be done during initial reading and reordering of each subblock in GEMM [6], [22]. Notice that, in comparison to conventional packing presented in the previous section [and illustrated in Figure 3(d)–(g)]:

- (11) and (12) invert the sign of the elements of $\mathbf{A}$ that are left shifted by $k$-bits;
- $\mathbf{A}$ is packed twice with reverse "ordering", as shown in (11) and (12);
- $\mathbf{B}$ is packed as well.

This setup enables packed processing to be used for error detection.

*2) Packed GEMM computations:* Two $\left(\frac{L}{2} \times L\right) \times \left(L \times \frac{L}{2}\right)$ subblock multiplications ensue via the use of two **standard subblock GEMM calls** (using either 64-bit integer or 64-bit floating point representation), producing all required results, as well as a number of "entangled" results within the numerical representation of the packed outputs. An example for the two packed products of elementary $2 \times 1$ by $1 \times 2$ vectors is shown in Figure 4(b)–(c). For the general case of an $L \times L$ by $L \times L$ subblock product, the elements packed within $\hat{\mathbf{R}}_i$ and $\hat{\mathbf{R}}_j$ are expressed mathematically by ($\forall m, n: \ 0 \leq m, n < L, \ \hat{m} = \left\lfloor \frac{m}{2} \right\rfloor, \ \hat{n} = \left\lfloor \frac{n}{2} \right\rfloor$):

$$\begin{aligned}
\hat{r}_{i,\hat{m},\hat{n}} &= \sum_{k=0}^{L-1} \hat{a}_{i,\hat{m},k}\hat{b}_{k,\hat{n}} \\
&= \left(\sum_{k=0}^{L-1} a_{2\hat{m},k}b_{k,2\hat{n}} \ll 2k\right) - \left(\sum_{k=0}^{L-1} a_{2\hat{m}+1,k}b_{k,2\hat{n}} \ll k\right) \\
&\quad + \left(\sum_{k=0}^{L-1} a_{2\hat{m},k}b_{k,2\hat{n}+1} \ll k\right) - \sum_{k=0}^{L-1} a_{2\hat{m}+1,k}b_{k,2\hat{n}+1} \\
&= \widetilde{r}_{2\hat{m},2\hat{n}} \ll 2k + \left(\widetilde{r}_{2\hat{m},2\hat{n}+1} - \widetilde{r}_{2\hat{m}+1,2\hat{n}}\right) \ll k \\
&\quad - \widetilde{r}_{2\hat{m}+1,2\hat{n}+1}
\end{aligned} \tag{14}$$

$$\begin{aligned}
\hat{r}_{j,\hat{m},\hat{n}} &= \sum_{k=0}^{L-1} \hat{a}_{j,\hat{m},k}\hat{b}_{k,\hat{n}} \\
&= \left(\sum_{k=0}^{L-1} a_{2\hat{m}+1,k}b_{k,2\hat{n}} \ll 2k\right) - \left(\sum_{k=0}^{L-1} a_{2\hat{m},k}b_{k,2\hat{n}} \ll k\right) \\
&+ \left(\sum_{k=0}^{L-1} a_{2\hat{m}+1,k}b_{k,2\hat{n}+1} \ll k\right) - \sum_{k=0}^{L-1} a_{2\hat{m},k}b_{k,2\hat{n}+1} \\
&= \widetilde{r}_{2\hat{m}+1,2\hat{n}} \ll 2k + \left(\widetilde{r}_{2\hat{m}+1,2\hat{n}+1} - \widetilde{r}_{2\hat{m},2\hat{n}}\right) \ll k \\
&\quad - \widetilde{r}_{2\hat{m},2\hat{n}+1}
\end{aligned} \tag{15}$$

*3) Unpacking of the Results:* Subsequently, unpacking occurs within each element of $\hat{\mathbf{R}}_i$ and $\hat{\mathbf{R}}_j$ based on the process presented below, which extends the unpacking of the previous section in order to handle the three packed results within each output $\hat{r}_{i,\hat{m},\hat{n}}$ and $\hat{r}_{j,\hat{m},\hat{n}}$, illustrated in Figure 4(d).

We extract the components packed within $\hat{r}_i$ following the unpacking process described in the previous subsection, with $\tilde{\varepsilon}_{(k_0,k_1),(k_2,k_3)}$ indicating the entanglement (i.e., superposition within the representation) of elements $\widetilde{r}_{k_0,k_1}$ and $\widetilde{r}_{k_2,k_3}$. The first output, $\widetilde{r}_{2\hat{m},2\hat{n}}$, which is packed at the most-significant bits of $\hat{r}_i$, is extracted by:

$$\widetilde{r}_{2\hat{m},2\hat{n}} = \hat{r}_i \gg (2k), \tag{16}$$

The extracted output is then removed from $\hat{r}_i$ and relevant bitwise operations ensue to determine the sign of the entangled output, $\tilde{\varepsilon}_{(2\hat{m},2\hat{n}+1),(2\hat{m}+1,2\hat{n})}$. We present the details of this sign check by defining two parameters, $t_{i,1}$ and $t_{i,2}$ as outlined below:

$$t_{i,1} = \hat{r}_i - \left[\widetilde{r}_{2\hat{m},2\hat{n}} \ll (2k)\right], \tag{17}$$

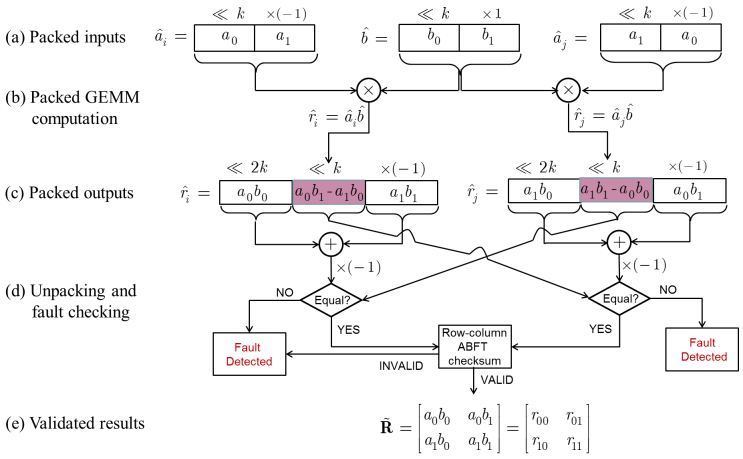


Figure 4: Illustration of highly-reliable integer subblock multiplication via numerical packing for the case of a $2 \times 1$ by $1 \times 2$ vector product with packing coefficient $k = 10$. The partitioning within the rectangles shows the location (shifts by $k$ or $2k$ bits) of inputs/outputs when packed within a single number.

$$t_{i,2} = t_{i,1} \gg k, \tag{18}$$

If $t_{i,2} \geq 2^{k-1}$, then $\tilde{\varepsilon}_{(2\hat{m},2\hat{n}+1),(2\hat{m}+1,2\hat{n})} = t_{i,2} - 2^k$, else $\tilde{\varepsilon}_{(2\hat{m},2\hat{n}+1),(2\hat{m}+1,2\hat{n})} = t_{i,2}$.
If $\tilde{\varepsilon}_{(2\hat{m},2\hat{n}+1),(2\hat{m}+1,2\hat{n})} < 0$, then : $\widetilde{r}_{2\hat{m},2\hat{n}} \leftarrow (\widetilde{r}_{2\hat{m},2\hat{n}} + 1)$.
Subsequently, the intermediate result extracted by (18) is also removed from the packed representation and a similar check is performed in order to extract the signed representation of $\widetilde{r}_{2\hat{m}+1,2\hat{n}+1}$, i.e.:

$$t_{i,1} \leftarrow [t_{i,1} - (t_{i,2} \ll k)] \tag{19}$$

If $t_{i,1} \geq 2^{k-1}$, then $\widetilde{r}_{2\hat{m}+1,2\hat{n}+1} = 2^k - t_{i,1}$, else $\widetilde{r}_{2\hat{m}+1,2\hat{n}+1} = -t_{i,1}$. Finally, similarly as before, if $\widetilde{r}_{2\hat{m}+1,2\hat{n}+1} < 0$, then the the entangled output, $\tilde{\varepsilon}_{(2\hat{m},2\hat{n}+1),(2\hat{m}+1,2\hat{n})}$ is incremented by one: $\tilde{\varepsilon}_{(2\hat{m},2\hat{n}+1),(2\hat{m}+1,2\hat{n})} \leftarrow \left(\tilde{\varepsilon}_{(2\hat{m},2\hat{n}+1),(2\hat{m}+1,2\hat{n})} + 1\right)$. We extract the components packed within $\hat{r}_j$ via the following process, which is identical to the process described for $\hat{r}_i$:

$$\widetilde{r}_{2\hat{m}+1,2\hat{n}} = \hat{r}_j \gg (2k), \tag{20}$$

$$t_{j,1} = \hat{r}_j - [\widetilde{r}_{2\hat{m}+1,2\hat{n}} \ll (2k)], \tag{21}$$

$$t_{j,2} = t_{j,1} \gg k, \tag{22}$$

If $t_{j,2} \geq 2^{k-1}$, then $\tilde{\varepsilon}_{(2\hat{m}+1,2\hat{n}+1),(2\hat{m},2\hat{n})} = t_{j,2} - 2^k$, else $\tilde{\varepsilon}_{(2\hat{m}+1,2\hat{n}+1),(2\hat{m},2\hat{n})} = t_{j,2}$. If $\tilde{\varepsilon}_{(2\hat{m}+1,2\hat{n}+1),(2\hat{m},2\hat{n})} < 0$, then $\widetilde{r}_{2\hat{m}+1,2\hat{n}} \leftarrow (\widetilde{r}_{2\hat{m}+1,2\hat{n}} + 1)$. Subsequently:

$$t_{j,1} \leftarrow [t_{j,1} - (t_{j,2} \ll k)]. \tag{23}$$

If $t_{j,1} \geq 2^{k-1}$, then $\widetilde{r}_{2\hat{m},2\hat{n}+1} = 2^k - t_{j,1}$, else $\widetilde{r}_{2\hat{m},2\hat{n}+1} = -t_{j,1}$. If $\widetilde{r}_{2\hat{m},2\hat{n}+1} < 0$, then $\tilde{\varepsilon}_{(2\hat{m}+1,2\hat{n}+1),(2\hat{m},2\hat{n})} \leftarrow \left(\tilde{\varepsilon}_{(2\hat{m}+1,2\hat{n}+1),(2\hat{m},2\hat{n})} + 1\right)$.
If:

- sufficient spacing is provisioned via the packing coefficient $k$ so that no overlapping (or "invasion") of results occurs within the numerical representation [4]–[6], and
- all three packed results fit within the utilized numerical representation,

then the results are guaranteed to be recoverable.

Relating to the second requirement, if $3k \leq W$, then the unpacking process determines the correct value of each output. This condition imposes that:

- $k \leq 21$ for $W = 64$ in 64-bit integer GEMM. Given that the sum of two outputs must be accommodated for the packed results shifted by $k$-bits (i.e., the entangled results) and one bit must be provided to allow for the sign information to be preserved within the packed integer representation, this 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 32-bit integer GEMM that allows for up to $\pm 2^{31}$ bits, i.e., loss of 37.5% of the bitwidth. This loss is substantially smaller than the corresponding bitwidth loss of ABFT and mABFT, reported in (3) and (6), respectively. Since the computational complexity and error detectability of the proposed algorithm is independent of the packing factor, we set $k$ to the maximum achievable value, i.e, $k = 20$ and $k = 9$ for 64 and 32-bit integer processing, respectively. Further details on the mathematics of the analytic calculation of the maximum packing factor are found in related work [4]–[6].

*4) Error Detection by Post-Entanglement followed by Row-Column ABFT Checksum Validation:* The advantage of the proposed packing-based GEMM is that, we not only obtain the results, **but we can also validate them by post-entangling**, i.e., doing $\widetilde{r}_{2\hat{m},2\hat{n}+1} - \widetilde{r}_{2\hat{m}+1,2\hat{n}}$ and $\widetilde{r}_{2\hat{m}+1,2\hat{n}+1} - \widetilde{r}_{2\hat{m},2\hat{n}}$ and comparing these with the entangled results $\tilde{\varepsilon}_{(2\hat{m},2\hat{n}+1),(2\hat{m}+1,2\hat{n})}$ and $\tilde{\varepsilon}_{(2\hat{m}+1,2\hat{n}+1),(2\hat{m},2\hat{n})}$, respectively. An example is shown in Figure 4(d)–(e) for $\hat{m} = \hat{n} = 0$. If differences are detected, the higher level routine (top-level processing) can be notified and a decision can be made by the application on whether to recompute the erroneous results or not.

One weakness of the post-entanglement check is that,

however unlikely, it is still possible that an SDC occurs in a location in the packed results $\hat{\mathbf{R}}_i$ and $\hat{\mathbf{R}}_j$ in a way that the operation $\widetilde{r}_{2\hat{m},2\hat{n}+1} - \widetilde{r}_{2\hat{m}+1,2\hat{n}}$ or $\widetilde{r}_{2\hat{m}+1,2\hat{n}+1} - \widetilde{r}_{2\hat{m},2\hat{n}}$ does not detect it. Specifically, this would be the case if *both* the extracted results $\widetilde{r}_{2\hat{m},2\hat{n}+1}$ and $\widetilde{r}_{2\hat{m}+1,2\hat{n}}$ were affected by an additive noise term $\delta$, with $\delta \in \mathbb{Z}^{\star}$. To detect the occurrence of such pathological cases, we simply utilize the conventional row-column ABFT checksum by calculating a single inner product between the sum of the columns of $\mathbf{A}$ with the sum of the rows of $\mathbf{B}$:

$$r_{\text{rc}} = \sum_{l=0}^{L-1} \left( \sum_{i=0}^{L-1} a_{i,l} \cdot \sum_{j=0}^{L-1} b_{l,j} \right), \tag{24}$$

which, similar to ABFT, can be derived during the input subblock reordering. Then, after checking for SDCs based on the post-entanglement check (and recalculating all detected SDCs), we can check if $r_{\text{rc}}$ produced by (24) agrees with the sum of all the elements of $\mathbf{R}$: $\sum_{i=0}^{L-1} \sum_{j=0}^{L-1} r_{i,j}$. It is straightforward to derive the latter when the output matrix is returned via a raster-scan and summation. This checksum ensures no such SDCs remain undetected.

The overall ABFT-based row-column checksum process requires producing only a single checksum per integer matrix product, i.e., $r_{\text{rc}}$ of (24), and simply adding the results in the order they are returned to the top-level processing of GEMM. Our experiments will demonstrate that the overall error checking process (packing, unpacking, post-entanglement check and row-column ABFT checksum) incurs execution time overhead similar to that of ABFT.

### C. Theoretical Analysis

The following two propositions summarize the arithmetic operations (addition, multiplication and comparison) for the computation of the proposed method for SDC mitigation in GEMM in reference to ABFT, mABFT and DMR. Note that the operation counts for the proposed approach have been scaled by two to account for the fact that the use of double-bitwidth representations (due to the use of packing) reduces the processing throughput by a factor of two.

**Proposition 1.** *In the absence of SDCs within an $L \times L$ GEMM output, the number of arithmetic operations of the proposed method is:*
*(i):* $\left(\frac{11L^2}{2L^3+7L^2-1}\right) \cdot 100\%$ *more than that of ABFT;*
*(ii):* $\frac{34L^3-433L^2+64}{162L^3+719L^2} \cdot 100\%$ *less than that of mABFT;*
*(iii):* $\frac{2L^3-19L^2+1}{4L^3-L^2} \cdot 100\%$ *less than that of DMR.*

*Proof:* See Appendix I. ∎

For example, for $L = 576$, Proposition 1 shows that numerical packing is only $0.95\%$ *less* efficient than ABFT, while remaining $20.37\%$ and $49.20\%$ *more* efficient than mABFT and DMR respectively.

**Proposition 2.** *Given $x$ SDCs in output GEMM results, the number of arithmetic operations of the proposed method for the GEMM computation followed by error detection and correction, and averaged between the best and worst case of SDC patterns, is:*

*(i):* $\frac{L^3+L^2\left(2\sqrt{x}+26.5\right)+L\left(\sqrt{x}-6x\right)-30x-\sqrt{x}}{3L^3+L^2\left(2\sqrt{x}+8.5\right)+L\sqrt{x}-\sqrt{x}-1.5} \cdot 100\%$ *less than that of ABFT;*

*(ii):* $\frac{34L^3-433L^2-366Lx+1912x+64}{162L^3+719L^2+18Lx-8x}) \cdot 100\%$ *less than that of mABFT;*

*(iii):* $\frac{2L^3-19L^2-3Lx-31.5x+1}{4L^3-L^2+3xL-1.5x} \cdot 100\%$ *less than that of DMR.*

*Proof:* See Appendix I. ∎

By setting $L = 576$ and $x = 10$, Proposition 2 estimates that numerical packing is $32.86\%$, $20.36\%$ and $49.19\%$ more efficient than ABFT, mABFT and DMR respectively. Overall, the complexity analysis of Propositions 1 and 2 indicates that our proposal is expected to incur small penalty when no SDCs happen, and become highly-beneficial when multiple SDCs occur in GEMM.

## IV. Experimental Results

In this section, we present execution time results with an Intel i7- 3632QM 2.2GHz processor (Ivy Bridge architecture with AVX support, Ubuntu 14.04.1 LTS, Clang 3.2 compiler). The BLAS routine of the ATLAS math library was used for all GEMM calls (sGEMM and dGEMM for 64-bit)[3]. After experimenting with various subblock sizes, we selected six subblock sizes as representative cases, ranging from $32 \times 32$ (below which the use of sGEMM or dGEMM is not beneficial) to $1152 \times 1152$. We opted for subblock sizes that are multiples of the subblock size settings for sGEMM and dGEMM within most open-source MKLs (e.g., ATLAS and GOTO) and avoid the use of cleanup code for the borders.

The ABFT approach follows Huang and Abraham [25], while we implemented the mABFT following Rexford and Jha [47], with the linear weighted checksum vector [39], [54] of (5). We opted for partition block size of $16 \times 16$ within each subblock in order to achieve output dynamic range comparable to the proposed approach and at the same time limit the overhead caused by the two additional checksum rows and columns per block[4]. For the ABFT implementation, due to the detection limitations analyzed in Section II-A, entire rows and columns are recomputed when multiple SDCs are detected. In addition, when ten SDCs are detected in different rows and columns, the error detection process stops and entire GEMM subblock recomputation (i.e., complete rollback) is carried out, as this turns out to be less costly than complete detection and multi-row & column recomputation for the chosen subblock sizes.

[3]Since all optimized math kernel libraries today support only 32/64bit floating point number representations, we cast all integer inputs into floating point data-type after preprocessing while casting to integer data type in order to perform error-check and correction if required.

[4]Specifically, we found that, under $P = 8$, the overhead caused by mABFT is more than $150\%$ in comparison to the fault-intolerant original GEMM and $P = 32$ or higher leads to unacceptable loss of dynamic range, as illustrated by (6).

Table I: Average execution times (in milliseconds) and percentile comparisons against the fault-intolerant (conventional) ATLAS BLAS sGEMM for all methods. All fault-tolerant approaches were able to detect and correct all SDCs.

| Component measured | Fault-Intolerant sGEMM | Proposed | % incr. in exec. time | ABFT sGEMM [25] | % incr. in exec. time | mABFT sGEMM [18], [30] [39], [47], [54] | % incr. in exec. time | DMR-based sGEMM [21] | % incr. in exec. time |
|---|---|---|---|---|---|---|---|---|---|
| subblock size: $32 \times 32$ | | | | | | | | | |
| GEMM calls only | 0.0166 | 0.0082 | -50.60 | 0.0181 | 9.04 | 0.0211 | 27.11 | 0.033 | 98.80 |
| with EC* & SDC count = 0 | | 0.0125 | -24.70 | 0.0193 | 16.27 | 0.0287 | 72.89 | 0.0362 | 118.07 |
| with EC & SDC count = 1 | | 0.0128 | -22.89 | 0.0194 | 16.87 | 0.0297 | 78.92 | 0.0366 | 120.48 |
| with EC & SDC count = 1 row | | 0.0170 | 2.41 | 0.0375 | 125.90 | 0.0467 | 181.33 | 0.0446 | 168.67 |
| subblock size:144×144 | | | | | | | | | |
| GEMM calls only | 0.419 | 0.3293 | -21.41 | 0.4455 | 6.32 | 0.5431 | 29.62 | 0.8348 | 99.24 |
| with EC & SDC count = 0 | | 0.4075 | -2.74 | 0.4699 | 12.15 | 0.6988 | 66.78 | 0.8918 | 112.84 |
| with EC & SDC count = 1 | | 0.4087 | -2.46 | 0.4700 | 12.17 | 0.7033 | 67.85 | 0.8930 | 113.13 |
| with EC & SDC count = 1 row | | 0.4540 | 8.35 | 0.9223 | 120.12 | 1.0134 | 141.86 | 0.9935 | 137.11 |
| subblock size: $288 \times 288$ | | | | | | | | | |
| GEMM calls only | 2.45 | 2.77 | 13.05 | 2.61 | 6.46 | 3.27 | 33.34 | 5.02 | 104.80 |
| with EC & SDC count = 0 | | 3.51 | 43.20 | 2.87 | 17.01 | 3.88 | 58.37 | 5.11 | 108.67 |
| with EC & SDC count = 1 | | 3.52 | 43.78 | 2.87 | 17.04 | 3.91 | 59.38 | 5.14 | 109.58 |
| with EC & SDC count = 1 row | | 3.67 | 49.94 | 5.47 | 123.43 | 4.02 | 64.20 | 5.47 | 123.15 |
| subblock size: $384 \times 384$ | | | | | | | | | |
| GEMM calls only | 5.64 | 6.11 | 8.38 | 5.97 | 6.00 | 7.14 | 26.61 | 11.41 | 102.48 |
| with EC & SDC count = 0 | | 7.28 | 29.13 | 6.45 | 14.41 | 8.29 | 47.02 | 11.59 | 105.58 |
| with EC & SDC count = 1 | | 7.28 | 29.25 | 6.45 | 14.45 | 8.34 | 47.95 | 11.61 | 106.02 |
| with EC & SDC count = 1 row | | 7.62 | 35.19 | 12.31 | 118.38 | 8.49 | 50.68 | 12.31 | 118.34 |
| subblock size: $576 \times 576$ | | | | | | | | | |
| GEMM calls only | 18.51 | 18.92 | 2.20 | 19.00 | 2.66 | 23.19 | 25.29 | 37.34 | 101.76 |
| with EC & SDC count = 0 | | 21.80 | 17.80 | 20.40 | 10.21 | 25.23 | 36.31 | 37.43 | 102.23 |
| with EC & SDC count = 1 | | 21.83 | 17.92 | 20.40 | 10.24 | 25.30 | 36.70 | 37.46 | 102.38 |
| with EC & SDC count = 1 row | | 22.60 | 22.11 | 41.60 | 124.74 | 26.18 | 41.43 | 38.59 | 108.52 |
| subblock size: $1152 \times 1152$ | | | | | | | | | |
| GEMM calls only | 148.13 | 148.39 | 0.18 | 150.86 | 1.84 | 182.98 | 23.53 | 298.63 | 101.59 |
| with EC & SDC count = 0 | | 157.50 | 6.32 | 155.62 | 5.06 | 194.32 | 31.18 | 300.15 | 102.63 |
| with EC & SDC count = 1 | | 158.08 | 6.71 | 155.69 | 5.10 | 194.71 | 31.44 | 300.18 | 102.64 |
| with EC & SDC count = 1 row | | 161.83 | 9.25 | 310.28 | 109.46 | 196.86 | 32.89 | 308.19 | 108.05 |

*EC stands for Error Correction

### A. SDC Injection Technique

In order to investigate the performance of all presented approaches for the detection and correction of soft errors, we perform an artificial fault injection campaign using the open source LLVM [37] based fault injection tool, KULFI [50]. KULFI emulates faults occurring in CPU state elements, which usually manifest as bit flips in a random computational state chosen at run time by injecting faults at the intermediate representation (IR) code of LLVM. Because the LLVM IR code preserves variable and function names and supports program analysis and transformations, it allows for controlled fault injections to be performed at specific program points, and into specific instructions. Given that such flips are somewhat rare in computing systems today, KULFI injects a single transient fault during every execution under investigation, albeit at a random time instance. By selecting fault injection parameters[5] as outlined in the author's example code [49], we observe and characterize the effect of the injected faults during GEMM computation into three classes:

- Benign errors: errors that have no effect on the GEMM output.
- Single SDC: errors that affect a data element that are not reused during the GEMM computation.
- Single-row SDC: errors that affect a data element that is used to compute an entire row/column of GEMM.

For example, a typical error summary when performing 100 fault injection executions (excluding segmentation and out-of-bound faults) of an $N \times N$-by-$N \times N$ GEMM is given by:

```
Single-row SDCs:16, Single SDC:34, Benign errors:50
```

In order to keep track of the number of injected errors that manifest as SDCs, KULFI performs two executions of a given source code: error-free and error-prone executions. We design the error-free execution to implement optimized GEMM calls (using the `cblas_sgemm` and `cblas_dgemm` function of the ATLAS library), while the error-prone GEMM implementation is written in plain C code in such a way as to allow for error injection during GEMM. In this way, GEMM execution time reported in Table I is measured from the error-free implementation while the performance for pre- and post processing (including error correction) is measured from the error-prone execution.

### B. Results under SDCs

The average execution time out of several independent runs (each using randomly-generated inputs) is presented in Table I. For all the reported experiments, all methods detected all incurred errors (i.e., the detection rate was 1.0 for

[5]Namely, the following parameters are used for all our fault injection executions: `iteration=100; byte_position=random; expected_fault_count=10; total_fault_count=100; inject_once flag=1; static_fault/dynamic_fault=1; inject_pointer_error=0; inject_data_error=1; print_fault_site=0`

all approaches). Therefore, the reported results compare the different methods with respect to execution time. To present a detailed breakdown of the execution time performance of the components of the proposed approach, four subcases are shown: *(i)* only the cost of dGEMM calls (i.e., not the cost of packing, unpacking or error checking); *(ii)* everything when no SDCs occur; *(iii)* everything, including error checking and correction when one SDC occurs; *(iv)* everything, including the detection and correction of one row of SDCs injected using the KULFI tool.

Theoretically, the execution time for GEMM computation (without error tolerance) is expected to be equal for the conventional sGEMM and the two quatersize dGEMMs perfomed by the proposed approach (cf. proof of Section III-C). In practice, due to the internal kernel optimizations of the ATLAS MKL for different matrix inner blocks, the results of Table I show both positive and negative GEMM execution time overhead for the proposed approach, which subsequently affects the overall overhead for error tolerance for the presented matrix sizes. Overall, the results show that, against fault-intolerant (conventional) GEMM design, the proposed approach incurs execution time overhead between $-24.70\%$ and $43.20\%$ when no SDCs occur and $2.41\%$ and $49.94\%$ when mitigating up to $N$ SDCs in an $N \times N$ GEMM output. On average, $12.05\%$ to $21.21\%$ overhead is incurred by the proposed scheme for tolerating up to $N$ SDCs in GEMM for the presented matrix sizes. Similarly, Table I shows that ABFT incurs an average execution time overhead[6] of $12.64\%$ to $120.34\%$ for the same level of fault tolerance. Overall, it is evident from the results of Table I that the proposed method incurs comparable overhead to ABFT when no errors occur and this overhead is (approximately) $18.04\%$ and $46.37\%$ less than that of mABFT and DMR-based GEMM. These results are in line with the theoretical predictions of Proposition 1 and, as expected by the theoretical analysis of Section III-C, the percentile overhead of all methods tends to decrease with increased subblock length (with some fluctuation for small subblock sizes due to internal kernel optimizations of the utilized ATLAS library).

For execution time overhead when correcting SDCs, the results of Table I show that the performance of ABFT could be worse than modular redundancy in multiple SDC scenarios. Specifically, while the proposed algorithm, mABFT and DMR incurs very low overhead for the correction of detected SDCs, ABFT requires (on average) more than $50\%$ additional overhead for error correction. This is because, under multiple detected SDCs in GEMM, ABFT recomputes several rows and columns of the result, or indeed the entire GEMM subblock when ten erroneous rows/colums are detected ("rollback ABFT" [1]). This significant increase in the incurred overhead is also evident in the theoretical analysis of Proposition 2. It is also evident that the additional overhead for tolerating multiple SDCs decreases considerably for all approaches, with the exception of ABFT, as the matrix size increases. For example, numerical packing requires $53.01\%$ additional overhead to tolerate multiple SDCs for the $32 \times 32$ matrix, while requiring only $9.07\%$ to tolerate the same proportion of SDCs in the $1152 \times 1152$ matrix. This property is shared amongst all exact error-location algorithms (like numerical packing and DMR) and is beneficial as the requirements for low-cost SDC correction techniques are more significant for large matrix sizes, where entire GEMM recomputation would lead to substantial performance degradation.

In terms of error detection, by injecting IUD bit flips in all the outputs of the two GEMM calls of the proposed approach (in the integer case) under an extensive SDC campaign of more than 10 trillion combinations, we verified experimentally that the **locations** of all SDCs were indeed detectable by the proposed approach. On the contrary, as detailed in Section II, ABFT can reliably detect and correct only up to a single SDC within each subblock product. ABFT requires recomputation of entire rows and columns to ensure no SDCs remain uncorrected, as discussed in the example of Figure 2. This is circumvented via the use of mABFT, which, under the utilized settings, can reliably detect the locations of up to 32 SDCs per GEMM subblock, albeit at the cost of substantial execution time overhead.

Overall, our theoretical analysis and experimental results demonstrate that our proposal offers very high accuracy and reliability in the detection of the locations of SDCs, while it comes with runtime overhead that is similar to that of ABFT when no SDCs occur.

## V. Application in Energy-aware Computing Systems under Voltage Scaling for Visual Descriptor Matching in Image and Video Retrieval

Voltage overscaling has been proposed as the means to improve battery life in portable devices, albeit at the risk of exposing the execution to transient faults in memory [32], [42], [63]. For example, Narayanan and Xie [42] report that dropping the operating voltage of a 4Mbit-SRAM memory for one hour brings significant energy savings but also increases the number of memory errors from 57 to 658. Our approach can be used as the fault detection and mitigation framework when applying such a scenario within integer matrix products of multimedia applications. It may be argued that, because multimedia applications are inherently error tolerant [7], SDCs would always have a benign effect on the output results. However, recent studies [24], [38], [44] show that there exist "critical" sections of multimedia applications where mitigation of soft errors is imperative, especially when outputs from such sections are reused for subsequent computations. In view of this, in the next two subsections we present such an example within the context of state-of-the-art image and video retrieval algorithms and we also showcase the corresponding energy

[6]The performance results for ABFT are in line with previously-reported ABFT benchmarks in GEMM. For example Bosilca *et. al.* [13], report that checkpointing the system state to detect a single SDC per process (and rolling back to a previous state when SDCs are detected) leads to $9\% \sim 34\%$ time overhead for an implementation using up to $484$ processes. Similarly, Chen and Dongarra [17] report that, for detecting a single SDC per subblock of a large matrix operation, $4\% \sim 9\%$ execution time overhead is incurred in a ScaLAPACK implementation over a distributed computing system. Finally, Wunderlich *et. al.* [61] report that ABFT incurs $18\% \sim 45\%$ execution time overhead versus GEMM on medium to large matrix dimensions under a GPU implementation.

Table II: Average number of discrepancies in the VLAD matches under one row of computed similarity measures being affected by SDCs.

| Image bunch size $N$ | No. of wrong VLAD matches |
|---|---|
| 288 | 84.2 (28.61%) |
| 384 | 111.4 (29.01%) |
| 576 | 165.2 (28.68%) |
| 1152 | 343.1 (29.78%) |

savings obtained when SDC tolerant algorithms are used in a vulnerable voltage-overscaled environment.

### A. Effect of SDCs on the Results of an Image or Video Retrieval Task

Consider a large image database preprocessed using the state-of-the-art vector of locally aggregated descriptors (VLAD) method of Jegou *et. al.* [29] to produce a compact descriptor for each database image comprising $N$ integers, with $N \in [288, 1152]$. The database images could be standalone pictures, or consist of frames of several video sequences. In order to match a given query image (or video frame) with the database images, an inner product between the compact descriptor of the query image and the descriptor of each of the images in the database is computed [29]. 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 descriptors of query image bunches and the ones of the database images.

In order to see the impact of SDCs in such a framework, we consider matching a query bunch comprising $N$ image descriptors within 1000 segments of $N$ database descriptors each (i.e., 1000 $N \times N$-by-$N \times N$ GEMMs), with the descriptors extracted from the Holidays dataset [29]. The experiment is set to establish the differences occurring in the retrieved images when running under SDC-free conditions versus when running under KULFI's single-row SDC case of Table I. We present an illustration of returned matches in both error free and erroneous cases in Fig. 5. The average difference in the obtained results for various bunch sizes is presented in Table II. It is evident that the compact nature of the image descriptors (which makes them sensitive to errors) leads to nearly 30% discrepancy between the error-free and the SDC execution of the descriptors. This results in irrelevant images being retrieved, as shown in Fig. 5. Therefore, in such image and video retrieval experiments, mitigation of SDCs is imperative in order to maintain reliable system operation.

### B. Application in Energy-aware Computing Systems under Voltage Scaling Incurring High SDC Rates in Data Cache Memory

We present an example of the efficacy of our proposal as a power saving technique within systems where it is imperative to ensure reliable performance at reduced energy consumption.

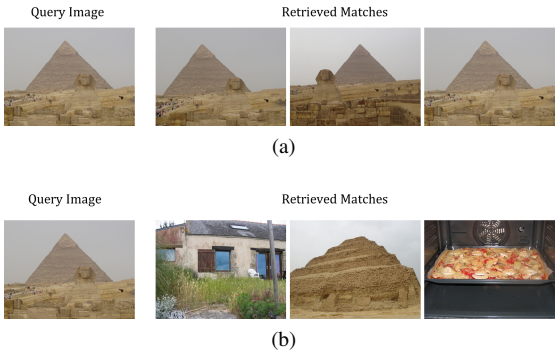


Figure 5: Indicative image matches (from the Holidays dataset [29]) in VLAD-based image retrieval when using the query image on the left for *(a)* an error free case; *(b)* a single-row erroneous case.

Voltage scaling is commonly found today within power-aware computing for multimedia systems [26]. Recently, *voltage overscaling*

### C. Application Context

We performed experiments where ABFT, mABFT, DMR and the proposed approach are used to detect SDCs within the GEMM operations of the image retrieval task described in Section V-A for database segment sizes of 576 and 1152 images, and each image in the database represented by a descriptor comprising 576 and 1152 integers, respectively.

### D. System Description and SDC Injection

The experiments were performed on an Intel i7-4578U 3GHz processor (Windows 8.1, Intel C++ Compiler 15.0.1). The Intel Extreme Tuning Utility was used for all voltage adjustments in our experiments and the Intel Power Gadget API [27] was used to measure the dynamic CPU power dissipation for the different approaches. We selected $V_{\text{safe}}$ = 1.37 V for SDC-free operation and $V_{\text{error}}$ = 0.98 V for overscaling. The chosen value for $V_{\text{error}}$ is below the recommended voltage for our system, but was still found to be providing for safe operation. Further overscaling causes actual SDCs in our setup, but does not allow for any control in our experimental conditions. Therefore, in common with related work [33], [36], [51], [57], we opted to operate our platform at a safe voltage level and then inject SDCs artificially with probabilities of $10^{-4}$ and $10^{-5}$ per data cache access[7]. Thus, on average, 1 SDC occurs every $10,000$ and $100,000$ accesses during each GEMM computation, respectively. This ensures that all methods undergo the same SDC injection campaign. In summary, the chosen experimental setup emulates the case where the fault-tolerant parts of the application (operating at $V_{\text{error}}$ V) are exposed to data-cache SDCs that are detected and corrected at runtime.

[7]Related work [2], [33], [36], [51], [57] has carried out experiments and simulation studies with similar SDC rates to the ones utilized here and we have verified that KULFI produces similar rates.

Table III: Average CPU energy (mJ) and percentile comparison against the fault-intolerant (conventional) sGEMM operating at $V_{\text{safe}}$V for GEMM computations for matching a bunch of $N$ = 576 and $N$ = 1152 images against a database of images consisting of $N$ = 576 and $N$ = 1152 images respectively. For each case, each image is represented by an $N$-element VLAD.

| Approach | SDC rate | Measured Energy (mJ) | % decrease in Energy | SDC rate | Measured Energy (mJ) | % decrease in Energy |
|---|---|---|---|---|---|---|
| $N$ = 576 | | | | | | |
| sGEMM at $V_{\text{safe}}$ | $10^{-5}$ | 133.92 | 0 | $10^{-4}$ | 133.92 | 0 |
| Proposed | | 127.07 | 5.11 | | 133.15 | 0.57 |
| ABFT [25] | | 238.30 | -77.94 | | 235.86 | -76.12 |
| mABFT [18], [30], [39], [47], [54] | | 148.20 | -10.66 | | 150.35 | -12.27 |
| DMR [21] | | 168.04 | -25.47 | | 176.00 | -31.42 |
| $N$ = 1152 | | | | | | |
| sGEMM at $V_{\text{safe}}$ | $10^{-5}$ | 1124.28 | 0 | $10^{-4}$ | 1124.28 | 0 |
| Proposed | | 730.86 | 34.99 | | 1030.32 | 8.36 |
| ABFT [25] | | 2148.78 | -91.13 | | 2154.37 | -91.62 |
| mABFT [18], [30], [39], [47], [54] | | 1349.31 | -20.02 | | 1381.31 | -22.86 |
| DMR [21] | | 1875.17 | -66.79 | | 2168.82 | -92.91 |

Table IV: Summary of features of different methods for SDC mitigation in integer matrix products.

| Feature \ Method | Proposed Fault-tolerant Numerical Packing | ABFT [25] | mABFT [18], [30], [39], [47], [54] | Dual Modular Redundancy [21] |
|---|---|---|---|---|
| Memory overhead | 1 checksum element | 1 additional row & col | 12.5% more rows & cols | 100% increase |
| Dynamic range loss | 37.5% of bitwidth | equation (3) | equation (6) | 0 |
| Error correction capability | Recomputes erroneous pairs of locations | Corrects 1 SDC; recomputes rows & columns for >1 SDC | Corrects multiple SDCs | Recomputes erroneous pairs of locations |
| Average execution overhead vs. fault-intolerant GEMM under no SDCs | 11.50 | 12.52% | 52.09% | 108.34 |
| Average execution overhead vs. fault-intolerant GEMM for "single-row" of SDCs | 21.21% | 120.34% | 85.40% | 127.31 |
| Energy decrease with voltage overscaling and SDC rate of $10^{-5}$ | 20.05% | −84.54% | −15.34% | −46.13% |

### *E. Energy Consumption Results*

Table III shows the average CPU energy for the different approaches. Due to the increased number of recomputations of ABFT, its performance drops with increased SDC rates. Table III shows that the proposed approach provides for up to 35% reduction in energy consumption against the fault-intolerant GEMM and up to 96% reduction against the other alternatives. This is because:

1) like DMR, it achieves highly-reliable detection of the locations of the vast majority of all the erroneous computations, while ABFT requires row and column recomputations given that, beyond the case of a single SDC per GEMM subblock, it cannot pin-point the SDC locations precisely (Figure 2);
2) it has substantially-reduced overhead in comparison to DMR and mABFT-based GEMM and its processing cycles are comparable to the fault-intolerant GEMM.

It can thus be considered as the best approach for SDC mitigation under aggressive voltage scaling in integer GEMM operations within energy-aware computing systems.

## VI. Conclusions

We proposed a new class of methods for highly-reliable integer matrix products. Our approach inserts redundancy within the numerical representation of the inputs by exploiting the concept of numerical packing. Analysis and validation of our proposal using high-performance generic matrix multiply (GEMM) routines demonstrated that high reliability comes at the cost of certain bitwidth loss and limited execution time overhead in comparison to the equivalent fault-intolerant GEMM. A summary of the features, requirements and performance of the proposed method in reference to ABFT, mABFT and DMR is given in Table IV. The summary shows that, on average: *(i)* under no SDCs in the GEMM execution, our approach incurs only 2.51% higher overhead than algorithm-based fault-tolerance (ABFT), and is 18.04% and 46.37% more efficient than modified ABFT (mABFT) and dual modular redundancy (DMR), respectively; *(ii)* under a single-row data corruption emanating from the use of a fault injection tool, KULFI, the proposed approach is 17% to 47% more efficient than all other methods, as it can pin-point the locations of all detected SDCs. In terms of energy consumption requirements, based on an emulated SDC-injection campaign incurred in data cache memory under voltage overscaling, we show that the proposed approach performs best in comparison to all other approaches. Future work will investigate alternative forms of numerical packing and their feasibility for SDC mitigation within broader classes of numerical computations.

## Appendix

### Proof of Proposition 1

*Proof.* Given two $L \times L$ matrices $\mathbf{A}$ and $\mathbf{B}$, the proposed method requires $\frac{3L^2}{2}$ addition/subtraction operations (see (11) ~ (13) ignoring all arithmetic shift operations) to generate the $\frac{L}{2} \times \frac{L}{2}$ packed matrices $\hat{\mathbf{A}}_i$, $\hat{\mathbf{A}}_j$ and $\hat{\mathbf{B}}$ since each element of

the packed inputs is computed with an addition and a bit-shift operation with the packing factor, $k$.

The GEMM computation of $\hat{\mathbf{R}}_i = \hat{\mathbf{A}}_i\hat{\mathbf{B}}$ and $\hat{\mathbf{R}}_j = \hat{\mathbf{A}}_j\hat{\mathbf{B}}$ subsequently requires $\frac{1}{2}\left(2L^3 - L^2\right)$ operations as both $\hat{\mathbf{R}}_i$ and $\hat{\mathbf{R}}_j$ are $\frac{L}{2} \times \frac{L}{2}$ matrices.

Assuming no SDCs occur within output GEMM results, each packed output $\hat{r_i}$ or $\hat{r_j}$ requires 11 arithmetic operations (including comparison operations) for unpacking [as elaborated in equations (16) – (23)] and an additional subtract and compare operation for error check (cf. Section III-B3). Therefore, $\frac{11L^2}{2}$ operations are required for the unpacking of the two $\frac{L}{2} \times \frac{L}{2}$ packed outputs, $\hat{\mathbf{R}}_i$ and $\hat{\mathbf{R}}_j$, while $L^2$ operations are required for error checking. The final Row-Column ABFT validation utilized by the proposed approach for the detection of certain pathological SDC cases would require $2L^2 - 2L$ and $L^2 + 2L - 1$ operations for pre-processing and error check respectively. In summary, assuming no SDCs occur, $\frac{1}{2}(2L^3 + 15L^2)$ double bitwidth and $3L^2 - 1$ single bitwidth arithmetic operations are required. By doubling the number of operations computed using the double bitwidth number representation, the number of operations is given by:

$$\text{Cx}\left\{\text{no error in proposed}\right\} = 2L^3 + 18L^2 - 1 \tag{25}$$

To achieve same protection for GEMM outputs using traditional ABFT, the row-column checksum would require $2L^2 - 2L$ addition operations for it's computation, while the GEMM itself will be computed with $2L^3 + 3L^2 - 1$ operations and the ABFT row & column checksum will be validated with $2L^2 + 2L$ addition and comparisons. Therefore, assuming no SDCs occurred, the operations of traditional ABFT are given by:

$$\text{Cx}\left\{\text{no error in ABFT}\right\} = 2L^3 + 7L^2 - 1 \tag{26}$$

For mABFT, given that two checksum rows (and columns) are generated per partition size of 16, each unweighted checksum element would require 15 add operations for it's computation while the corresponding weighted checksum element is computed with 31 arithmetic operations. Therefore, $46L$ operations are required for the checksum computation of each partition, and $\frac{L}{16}$ partitions exist for each of the input matrices. In total, pre-processing for mABFT would require $46L \cdot \frac{L}{16} \cdot 2 = \frac{23}{4}L^2$ arithmetic operations for the calculation of the row & column checksums. Performing GEMM between the $\frac{9L}{8} \times L$ and $L \times \frac{9L}{8}$ matrices of mABFT will subsequently require $\frac{81}{64}\left(2L^3 - L^2\right)$ operations and, in order to check for SDCs within each $18 \times 18$ partition of the resulting $\frac{9L}{8} \times \frac{9L}{8}$ output matrix, 16 (or 32) operations will be required for each unweighted row/column checksum error check (or each weighted row/column error check). This implies that mABFT requires $48 \cdot 2 \cdot \left(\frac{9L}{8}\right)^2 \cdot \left(\frac{1}{18}\right) = \frac{27L^2}{4}$ operations for error check. Therefore, the overall complexity of mABFT (assuming no SDCs occurred) is:

$$\text{Cx}\left\{\text{no error in mABFT}\right\} = \frac{1}{64}\left(162L^3 + 719L^2\right) \tag{27}$$

Finally, dual modular redundancy does not require any pre or post processing for its computation. However, $4L^3 - 2L^2$ and $L^2$ arithmetic operations are performed for the computation of GEMM and error checking, respectively. Therefore, assuming no SDCs occurred, the complexity of DMR is:

$$\text{Cx}\left\{\text{no error in DMR}\right\} = 4L^3 - L^2 \tag{28}$$

Combining (25)–(28) leads to the ratios reported in Proposition 1.

## Proof of Proposition 2

*Proof.* Given the detection of $x$ SDCs within the output GEMM results, the number of computations required by all algorithms for error correction is dependent on their location. We therefore calculate the operations required for error correction as the average of the "worst-case" and "best-case" SDC distribution in output GEMM results, where "worst-case" refers to an SDC distribution requiring the highest number of operations and "best-case" SDC distribution requires the least number of operations.

For numerical packing, the "best-case" refers to the case of detection of an SDC in an output $r_i$ and its corresponding output $r_j$ , i.e, $i = j$. In such a case, our method of error correction re-computes $\frac{x}{2}$ locations for each of $\hat{\mathbf{R}}_i$ and $\hat{\mathbf{R}}_j$, thereby requiring $x\left(2L - 1\right)$ operations for all erroneous location re-computations, while $2 \cdot 11 \cdot \frac{x}{2}$ operations would be required for the unpacking of all $x$ locations in both matrices (as each unpacking requires 11 operations).

On the other hand, the "worst-case" SDC distribution occurs when all $x$ SDCs happen within either of $\hat{\mathbf{R}}_i$ or $\hat{\mathbf{R}}_j$. Given that any SDC detected for each $r_i$ or $r_j$ requires the re-computation of both $\hat{r}_i$ and $\hat{r}_j$ of the failed location, $2x$ re-computations and unpackings will be required for this scenario, i.e., $x\left(4L + 20\right)$ arithmetic operations.

By averaging between the two cases, the proposed approach requires $\frac{x}{2}\left(6L + 21\right)$ operations to correct $x$ detected SDCs. By doubling this result:

$$\text{Cx}\left\{x\text{ errors in proposed}\right\} = 2L^3 + 18L^2 - 1 + 6xL + 30x \tag{29}$$

Concerning ABFT, assuming (for simplicity of exposition) that $\sqrt{x}$ is integer, the "best-case" SDC distribution for ABFT is having the SDCs located within a $\sqrt{x} \times \sqrt{x}$ square in the matrix. Therefore, provided the SDCs are all detectable, ABFT would flag $\sqrt{x}$ rows and columns as erroneous. When multiple SDCs are detected via ABFT, entire rows and columns that fail the checksum test are recomputed. The "best-case" SDC distribution of ABFT would therefore require a re-computation of $\sqrt{x}$ rows and $\sqrt{x}$ columns involving $2\sqrt{x}\left(2L^2 + L - 1\right)$ MAC operations.

On the other hand, if $x$ is large (e.g., $x \geq 16$), entire GEMM re-computation is more beneficial in practice in comparison to selective recomputation of multiple rows and columns. This is because the data access irregularity, in conjunction with the high percentage of outputs that are recomputed, ends up incurring higher execution time penalty in comparison

to simply recomputing the GEMM. Therefore, the required operations for this case are:

$$\frac{1}{2}\left[2L^3+3L^2-1+2\sqrt{x}\left(2L^2+L-1\right)\right].$$

By averaging between the two cases:

$$\begin{aligned}\text{Cx}\left\{x \text{ errors in ABFT}\right\} &= 2L^3+7L^2+0.5\cdot\left[2L^3+3L^2-1\right]\\ &\quad+\left[\sqrt{x}\left(2L^2+L-1\right)\right]-1\\ &= 3L^3+L^2\left(2\sqrt{x}+8.5\right)\\ &\quad+L\sqrt{x}-\sqrt{x}-1.5\end{aligned} \tag{30}$$

The mABFT implementation presented in the paper can detect one SDC and correct two SDCs for each row or column within a partition. The "best-case" SDC distribution for this implementation would be having $x$ SDCs spread in such a way that only one SDC occurs in a row or column of a partition. Since mABFT requires one division operation for error location and one subtraction operation for error correction [19], [54] , $2x$ operations would be required in order to correct the $x$ detected SDCs.

On the other hand, the "worst-case" error location in mABFT occurs when $4$ SDCs occur within a $2\times2$ sub-block in a partition such that two SDCs are detected in each of the two rows and columns corresponding to the sub-block error location. Since the mABFT implementation in this work can not correct more than one SDC per row/column, entire erroneous rows and columns of the affected partitions are re-computed. Thus, $4\cdot18\left(2L-1\right)$ arithmetic operations are required for the re-computation of 2 erroneous rows and 2 erroneous columns assumed to be spread across $\frac{x}{4}$ different locations. Therefore, the overall complexity of mABFT including the correction of $x$ SDCs is:

$$\begin{aligned}\text{Cx}\left\{x \text{ errors in mABFT}\right\} &= \frac{1}{64}\left(162L^3+719L^2\right)\\ &\quad+x\left(18L-8\right)\end{aligned} \tag{31}$$

Finally, performing same analysis for DMR, the "best-case" SDC distribution would refer to two SDCs occurring exactly at corresponding positions of the two GEMMs. This would require re-computation of $\frac{x}{2}$ output locations. On the other hand, the "worst-case" SDC here refers to all $x$ SDCs occurring at different locations of one of the GEMM outputs. In such a case, $x$ locations would need to be re-computed for each of the output matrices. Therefore, the overall complexity of DMR (including the correction of $x$ SDCs) is:

$$\text{Cx}\left\{x \text{ errors in DMR}\right\} = \quad 4L^3-L^2+3xL-1.5x \tag{32}$$

Combining the last four equations leads to the ratios reported in Proposition 2.

## REFERENCES


[1] A. A. Al-Yamani, N. Oh, and E. J. McCluskey. Performance evaluation of checksum-based abft. In *Proc. IEEE Int. Symp. Defect and Fault Tol. in VLSI Syst., DFT'01*, pages 461–466, 2001.

[2] A. R. Alameldeen et al. Energy-efficient cache design using variable-strength error-correcting codes. In *Proc. IEEE Int. Symp. on Computer Archit. (ISCA'11)*, pages 461–471, 2011.

[3] P. Ampadu, M. Zhang, and V. Stojanovic. Breaking the energy barrier in fault-tolerant caches for multicore systems. In *Proc. IEEE Int. Conf. on Des., Autom. and Test in Europe, DATE'13*, pages 731–736, 2013.

[4] D. Anastasia and Y. Andreopoulos. Linear image processing operations with operational tight packing. *IEEE Sig. Process. Let.*, 17(4):375–378, 2010.

[5] D. Anastasia and Y. Andreopoulos. Software designs of image processing tasks with incremental refinement of computation. *IEEE Trans. on Image Processing*, 19(8):2099–2114, 2010.

[6] 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*, 60(4):2024–2037, 2012.

[7] Y. Andreopoulos. Error tolerant multimedia stream processing: There's plenty of room at the top (of the system stack). *IEEE Trans. on Multimedia*, 15(2):291–303, 2013.

[8] Y. Andreopoulos et al. A local wavelet transform implementation versus an optimal row-column algorithm for the 2d multilevel decomposition. In *Proc. IEEE Int. Conf. on Image Process., ICIP 2001*, volume 3, pages 330–333. IEEE, 2001.

[9] Y Andreopoulos et al. A new method for complete-to-overcomplete discrete wavelet transforms. In *Proc. 14th IEEE Int. Conf. on Digital Signal Process., DSP 2002*, volume 2, pages 501–504. IEEE, 2002.

[10] Y. Andreopoulos and M. van der Schaar. Incremental refinement of computation for the discrete wavelet transform. *IEEE Trans. on Signal Process.*, 56(1):140–157, 2008.

[11] C.J. Anfinson and F.T. Luk. A linear algebraic model of algorithm-based fault tolerance. *IEEE Trans. on Computers*, 37(12):1599–1604, 1988.

[12] E. Bingham and H. Mannila. Random projection in dimensionality reduction: applications to image and text data. In *Proceedings of the 7th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 245–250, 2001.

[13] G. Bosilca et al. Algorithm-based fault tolerance applied to high performance computing. *Elsevier J. Paral. and Distrib. Comput.*, 69(4):410–416, 2009.

[14] N. P. Carter, H. Naeimi, and D. S. Gardner. Design techniques for cross-layer resilience. In *Proc. IEEE Int. Conf. on Des., Autom. and Test in Europe, DATE'10*, pages 1023–1028, 2010.

[15] A. Chadha and Y. Andreopoulos. Region-of-interest retrieval in large image datasets with Voronoi VLAD. In *Computer Vision Systems ICVS*, pages 218–227, 2015.

[16] Y. Chauvin and D. E. Rumelhart. *Backpropagation: theory, architectures, and applications*. Psychology Press, 2013.

[17] Z. Chen et al. Fault tolerant high performance computing by a coding approach. In *Proc. ACM SIGPLAN Symp. on Princ. and Pract. of Paral. Program.*, pages 213–223, 2005.

[18] T. Davies et al. High performance linpack benchmark: a fault tolerant implementation without checkpointing. In *Proc. Int. Conf. on Supercomputing*, pages 162–171, 2011.

[19] P. Du et al. Soft error resilient QR factorization for hybrid system with GPGPU. *J. of Computat. Sci.*, 4(6):457–464, 2013.

[20] S. Dubois et al. Decomposition of dynamic textures using morphological component analysis. *IEEE Trans. on Circ. and Syst. for Video Technol.*, 22(2):188–201, 2012.

[21] C. Engelmann et al. The case for modular redundancy in large-scale high performance computing systems. In *Proc. IASTED Internat. Conf.*, volume 641, page 046, 2009.

[22] K. Goto and R. A. Van De Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Soft*, 34(3):12, 2008.

[23] K. Han, G. Lee, and K. Choi. Software-level approaches for tolerating transient faults in a coarse-grainedreconfigurable architecture. *IEEE Trans. on Depend. and Secure Comp.*, 11(4):392–398, 2014.

[24] A. Heinig et al. Improving transient memory fault resilience of an H. 264 decoder. In *8th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010*, pages 121–130, 2010.

[25] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. on Computers*, 100(6):518–528, 1984.

[26] Y. Ikenaga et al. A 27% active-power-reduced 40-nm CMOS multimedia SoC with adaptive voltage scaling using distributed universal delay lines. *IEEE J. Solid-State Circ.*, 47(4):832–840, 2012.

[27] Intel. Intel Power Gadget 3.0 (Windows 64-bit), 2014.

[28] MKL Intel. Intel math kernel library, 2007.

[29] H. Jégou et al. Aggregating local image descriptors into compact codes. *IEEE Trans. Pat. Anal. and Machine Intel.*, 34(9):1704–1716, 2012.

[30] J.-Y. Jou and J.A. Abraham. Fault-tolerant matrix arithmetic and signal processing on highly concurrent computing structures. In *Proc. of the IEEE*, volume 74, pages 732,741, May 1986.

[31] J.Y Jou and J. A Abraham. Fault-tolerant matrix operations on multiple processor systems using weighted checksums. In *Proc. 28th Annual Tech. Symp.*, pages 94–101, 1984.

[32] W. Kim et al. Performance comparison of dynamic voltage scaling algorithms for hard real-time systems. In *8th IEEE Real-Time and Embedded Technology and Applications Symposium, 2002.*, pages 219–228, 2002.

[33] V. B. Kleeberger et al. A cross-layer technology-based study of how memory errors impact system resilience. *IEEE Micro*, 33(4):46–55, 2013.

[34] P. Kogge et al. Exascale computing study: Technology challenges in achieving exascale systems. *DARPA Information Processing Techniques Office*, 2008.

[35] N. Kontorinis et al. Statistical framework for video decoding complexity modeling and prediction. *IEEE Trans. on Circ. and Syst. for Video Technol.*, 19(7):1000–1013, 2009.

[36] F. J. Kurdahi et al. Low-power multimedia system design by aggressive voltage scaling. *IEEE Trans. Very Large Scale Integration (VLSI) Syst.*, 18(5):852–856, 2010.

[37] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. IEEE Int. Symp. Code Generation and Optimization, CGO 2004*, pages 75–86, 2004.

[38] K. Lee et al. Mitigating soft error failures for multimedia applications by selective data protection. In *Proc. ACM Int. Conf. on Comp., Arch. and Synth. for Embed. Syst.*, pages 411–420, 2006.

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

[40] Y. Mu et al. Video de-fencing. *IEEE Trans. on Circ. and Syst. for Video Technol.*, 24(7):1111–1121, 2014.

[41] V.S.S Nair and J.A. Abraham. General linear codes for fault tolerant matrix operations on processor arrays. In *Proc. Int. Symp. Fault Tolerant Comput.*, pages 180–185, 1988.

[42] V. Narayanan and Y. Xie. Reliability concerns in embedded system designs. *IEEE Computer Magazine*, 39(1):118–120, 2006.

[43] M. Nicolaidis et al. Design for test and reliability in ultimate CMOS. In *Proc. IEEE Int. Conf. on Des., Autom. and Test in Europe, DATE'12*, pages 677–682, 2012.

[44] I. Polian et al. Low-cost hardening of image processing applications against soft errors. In *Proc. IEEE Int. Symp. Defect and Fault Tol, in VLSI Syst., DFT'06*, pages 274–279. IEEE, 2006.

[45] H. M Quinn et al. CCC visioning study: system-level cross-layer cooperation to achieve predictable systems from unpredictable components. Technical report, Los Alamos National Laboratory (LANL), 2011.

[46] G. R. Redinbo. Wavelet codes for algorithm-based fault tolerance applications. *IEEE Trans. on Depend. and Secure Comp.*, 7(3):315–328, 2010.

[47] J. Rexford and N.K. Jha. Algorithm-based fault tolerance for floating-point operations in massively parallel systems. In *Proc. IEEE Int. Symp. on Circ. and Syst.*, volume 2, pages 649,652, May 1992.

[48] D. Rossi et al. Error correcting code analysis for cache memory high reliability and performance. In *Proc. IEEE Int. Conf. on Des., Autom. and Test in Europe, DATE'11*, pages 1–6, 2011.

[49] V. C. Sharma et al. Kulfi. https://github.com/soarlab/KULFI/.

[50] V. C. Sharma et al. Towards formal approaches to system resilience. In *Dependable Computing (PRDC), 2013 IEEE 19th Pacific Rim International Symposium on*, pages 41–50, 2013.

[51] J. Sloan, R. Kumar, and G. Bronevetsky. An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance. In *Proc. IEEE/IFIP Int. Conf. on Depend. Syst. and Net. (DSN'13)*, pages 1–12, 2013.

[52] V. Spiliotopoulos et al. Quantization effect on vlsi implementations for the 9/7 dwt filters. In *Proc. IEEE Int. Conf. on Acoust., Speech, and Signal Process., ICASSP'01*, volume 2, pages 1197–1200. IEEE, 2001.

[53] V. Sridharan et al. Reducing data cache susceptibility to soft errors. *IEEE Trans. on Depend. and Secure Comp.*, 3(4):353–364, 2006.

[54] 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.*, 2004.

[55] I. Stefanovici et al. Battling borked bits. *IEEE Spectrum*, 52(12):34–53, 2015.

[56] L. Sun and T. Shibata. Unsupervised object extraction by contour delineation and texture discrimination based on oriented edge features. *IEEE Trans. on Circ. and Syst. for Video Technol.*, 24(5):780–788, 2014.

[57] A. Suresh and J. Sartori. Automated algorithmic error resilience for structured grid problems based on outlier detection. In *Proc. Annual IEEE/ACM Int. Symp. on Code Gen. and Opt.*, page 240, 2014.

[58] A. Timor et al. Using underutilized CPU resources to enhance its reliability. *IEEE Trans. on Depend. and Secure Comp.*, 7(1):94–109, 2010.

[59] J. Wen et al. Joint video frame set division and low-rank decomposition for background subtraction. *IEEE Trans. on Circ. and Syst. for Video Technol.*, 24(12):2034–2048, 2014.

[60] P. Wu et al. On-line soft error correction in matrix–matrix multiplication. *J. of Comput. Sci.*, 4(6):465–472, 2013.

[61] H.-J. Wunderlich et al. Efficacy and efficiency of algorithm-based fault-tolerance on GPUs. In *IEEE Internat. On-Line Testing Symp., 2013*, pages 240–243, 2013.

[62] 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.*, 26(1):131–137, 2004.

[63] B. Zhao, H. Aydin, and D. Zhu. Reliability-aware dynamic voltage scaling for energy-constrained real-time embedded systems. In *IEEE International Conference on Computer Design, 2008. ICCD 2008.*, pages 633–639, 2008.


**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.

**Mohammad Ashraful Anam** obtained the PhD in Electronic Engineering from University College London (Lombardi Prize for the Best PhD thesis in Electronic Engineering) and is currently postdoctoral research associate in the Department of Electronic and Electrical Engineering, University College London, London, UK. His research interests are in error tolerant computing, and reliable cloud computing

**Fabio Verdicchio** is a Lecturer in the School of Engineering within the College of Physical Sciences at the University of Aberdeen. His research interests include Internet video streaming, applications of microcontrollers to sensing and design of video-processing algorithms

**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.