Reduced-precision Algorithm-based Fault Tolerance for FPGA-implemented Accelerators

James J. Davis and Peter Y. K. Cheung

Imperial College London, London, SW7 2AZ, United Kingdom
{james.davis06, p.cheung}@imperial.ac.uk

Abstract. As the threat of fault susceptibility caused by mechanisms including variation and degradation increases, engineers must give growing consideration to error detection and correction. While the use of common fault tolerance strategies frequently causes the incursion of significant overheads in area, performance and/or power consumption, options exist that buck these trends. In particular, algorithm-based fault tolerance embodies a proven family of low-overhead error mitigation techniques able to be built upon to create self-verifying circuitry.

In this paper, we present our research into the application of algorithm-based fault tolerance (ABFT) in FPGA-implemented accelerators at reduced levels of precision. This allows for the introduction of a previously unexplored tradeoff: sacrificing the observability of faults associated with low-magnitude errors for gains in area, performance and efficiency by reducing the bit-widths of logic used for error detection. We describe the implementation of a novel checksum truncation technique, analysing its effects upon overheads and allowed error. Our findings include that bit-width reduction of ABFT circuitry within a fault-tolerant accelerator used for multiplying pairs of 32×32 matrices resulted in the reduction of incurred area overhead by 16.7% and recovery of 8.27% of timing model $f_{\max}$. These came at the cost of introducing average and maximum absolute output errors of 0.430% and 0.927%, respectively, of the maximum absolute output value under transient fault injection.

1 Introduction

ABFT relies upon the augmentation of data with additional information—checksums formed from that data—to provide post-operation verification of results with low overheads compared to alternatives including modular redundancy. While previous fixed-point ABFT-related work has assumed all data and checksums to be n -bit integer (i.e. modulo- 2^n), it is possible to break this relationship and consider data and checksum precision independently. By making informed decisions regarding exactly which information to discard when forming and manipulating checksums, the incurred overheads can be reduced at the cost of accepting some data error tolerance. The methods introduced in this work lend themselves to FPGAs thanks to their efficient simultaneous implementation of multiple arbitrary-precision datapaths. Here, as a case study for the

investigation into reduced-precision ABFT, we use hardware-accelerated matrix multiplication: a benchmark for which the ABFT operation is straightforward yet that is representative of commonly hardware-accelerated operations.

The novel contributions of this work are: (1) the first consideration of distinct data and checksum bit-widths within ABFT-protected operations, which we call *reduced-precision* or *RP-ABFT*, (2) an implementation of circuitry incorporating RP-ABFT for resilience against hardware faults, (3) analysis of the costs and benefits of applying RP-ABFT at various levels of precision and (4) insight into the fault tolerability of RP-ABFT.

2 Application-level Fault Tolerance

Tailoring fault tolerance to particular applications can facilitate drastic overhead reduction versus general-purpose methods. ABFT represents a methodology for achieving such reduction while maintaining high fault detectability. A subclass of linear algebra operations exists that can be protected by ABFT; amongst these are matrix operations (multiplication, addition, LU decomposition, etc.) [4] and Fourier transformations [7]. Linear filtering operations can be protected when considered in state-space form [4]. These operations are also highly suited to hardware acceleration thanks to their inherent parallelism. Beyond low area overhead, ABFT has two further key advantages: (1) its application requires no fundamental changes to the datapaths used for performing mathematical operations and (2) output and error-indicating data are produced simultaneously.

While ABFT has traditionally been used to protect fixed-point operations, the methods are compatible with floating-point arithmetic as well. Of particular relevance to this work are the errors introduced by floating-point operations, which necessitate error bounding to distinguish them from those caused by other mechanisms [6]. Recent work [1] sought to lower the required bounds in a GPU-accelerated floating-point benchmark by analysing input data a priori.

ABFT-protected accelerators implemented in FPGAs have been the focus of several recent publications. Jacobs et al. implemented algorithmic protection of several matrix multiplication architectures [5]. ABFT was called upon for error detection of the same operator in more recent work, with resource reallocation performed using additional logic [2] and dynamic partial reconfiguration [3] in order to avoid faulty components at runtime. Area overheads for accelerators capable of multiplying pairs of 32×32 matrices were found to be 17.3% and 10.1%, respectively, therein. While error correction is not the focus of this work, previously published fault avoidance strategies [2] [3] are directly compatible.

3 Principles of ABFT

The mechanics of ABFT checksumming are described here [4]. Any $m \times n$ data matrix $\mathbf{D}$ can be supplemented with an additional row of column-wise checksums to produce an $(m + 1) \times n$ *column checksum-encoded* matrix $\mathbf{D}_c$. The transformation, achieved with *generation* matrix $\mathbf{G}_c$, is $\mathbf{D}_c = \mathbf{G}_c \mathbf{D} = \begin{pmatrix} \mathbf{I}_{m \times m} \\ \mathbf{1}_{1 \times m} \end{pmatrix} \mathbf{D}$.

Note that $\mathbf{I}$ is the identity matrix and $\mathbf{1}$ a vector of ones. Similarly, row-wise checksums can be added within an additional column to form an $m \times (n+1)$ *row checksum-encoded* matrix $\mathbf{D}_r$ with generation matrix $\mathbf{G}_r$ by performing $\mathbf{D}_r = \mathbf{D}\mathbf{G}_r = \mathbf{D}(\mathbf{I}_{n \times n} \mathbf{1}_{n \times 1})$. An $(m+1) \times (n+1)$ *full checksum-encoded* matrix $\mathbf{D}_f$ can be produced by performing $\mathbf{D}_f = \mathbf{G}_c \mathbf{D} \mathbf{G}_r$.

Following storage, transmission or computation that preserves the form of checksum-encoded matrices, data integrity can be verified by producing a *discrepancy* vector δ . Column- and row-wise discrepancy vectors can be produced, using *verification* vectors v_c and v_r , by performing $\delta_c = v_c \mathbf{D}_c = (\mathbf{1}_{1 \times m} \mathbf{1}_{n \times 1}) \mathbf{D}_c$ and $\delta_r = \mathbf{D}_r v_r = \mathbf{D}_r (\mathbf{1}_{n \times 1})$. Non-zero elements indicate the presence, locations and magnitudes of errors within checksum-encoded matrices. A full checksum-encoded matrix $\mathbf{D}_f$ can be verified by independently producing both δ_c and δ_r .

Consider $\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ and $\mathbf{B} = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$. For simplicity and clarity, we assume the data matrices used to always be square with dimensions $s \times s$, although this is not a requirement. Since matrix multiplication is a *checksum-preserving* operation, $\mathbf{C} = \mathbf{A}\mathbf{B}$ can be protected by forming $\mathbf{A}_c$ and $\mathbf{B}_r$ as explained in Sect. 3 and then performing $\mathbf{C}_f = \mathbf{A}_c \mathbf{B}_r$. The transformations and subsequent computation are shown in (1), with column- and row-wise checksums shown in red and blue, respectively. The result's corner element is shown in magenta to indicate that it is both column- *and* row-wise checksum. Note that the data present in (1)'s unprotected result is preserved in its protected result. The protected result can be verified by calculating discrepancy vectors δ_c and δ_r , as shown in (2).

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 4 & 6 \end{pmatrix} \begin{pmatrix} 5 & 6 & 11 \\ 7 & 8 & 15 \end{pmatrix} = \begin{pmatrix} 19 & 22 & 41 \\ 43 & 50 & 93 \\ 62 & 72 & 134 \end{pmatrix}. \quad (1)$$

$$(1 \ 1 \ -1) \begin{pmatrix} 19 & 22 & 41 \\ 43 & 50 & 93 \\ 62 & 72 & 134 \end{pmatrix} = (0 \ 0 \ 0) \ , \quad \begin{pmatrix} 19 & 22 & 41 \\ 43 & 50 & 93 \\ 62 & 72 & 134 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}. \quad (2)$$

4 Principles of RP-ABFT

To reduce overheads while maintaining sensitivity to faults that cause high-magnitude errors, truncation can be performed from the least significant bits (LSBs) of data elements ‘upwards’ when forming and manipulating checksums. In this paper, all input data elements are n -bit signed integers and we call the number of bits of precision removed from each during checksum generation the *truncation width*, represented by r . Output data elements are always $2n$ -bit. We label input and output data elements within ABFT-protected matrix multiplication as d_{in} and d_{out} , respectively. cs_{in} and cs_{out} are input and output checksums, while *corner* checksum $cs_{out, c}$ is special, being formed exclusively from cs_{in} elements. We use $\vee(\cdot)$ to represent *maximum absolute value*, while $\epsilon(\cdot)$ is the *maximum absolute error* introduced by truncation.

$\vee(d_{\text{in}})$ is 2^{n-1} . The r -bit truncation of a d_{in} element, performed with bitwise shifts as $(d_{\text{in}} \gg r) \ll r$, is represented as $\lfloor d_{\text{in}} \rfloor_r$ since rounding, for both positive and negative values, is towards negative infinity. Note that $\vee(\lfloor d_{\text{in}} \rfloor_r) = \vee(d_{\text{in}})$; the maximum negative value, for which truncation by any $0 \leq r < n$ will have no effect, also represents the maximum absolute value. $\epsilon(\lfloor d_{\text{in}} \rfloor_r) = 2^r - 1$. Each input checksum element, cs_{in} , is formed from s independently truncated d_{in} elements. $\vee(cs_{\text{in}})$ and $\epsilon(cs_{\text{in}})$ are therefore simply $s2^{n-1}$ and $s(2^r - 1)$, respectively.

Output checksum elements, cs_{out} , are comprised of s multiplied pairs of d_{in} and cs_{in} . Since the d_{in} element used within each multiplication is not truncated, it does not introduce error: this comes purely from each cs_{in} , so $\epsilon(cs_{\text{out}}) = s\vee(d_{\text{in}})\epsilon(cs_{\text{in}}) = s^2(2^r - 1)2^{n-1}$. The corner output checksum element, $cs_{\text{out}, c}$, is formed of s multiplied pairs of cs_{in} elements. Unlike for each cs_{out} element, therefore, error can be introduced by both of the multiplicands within each product. Consequently, $\epsilon(cs_{\text{out}, c}) = s(\vee(cs_{\text{in}})\epsilon(cs_{\text{in}}) + \epsilon(cs_{\text{in}})\vee(cs_{\text{in}}) + \epsilon(cs_{\text{in}})^2) = s^3(2^r - 1)(2^n + 2^r - 1)$.

5 Implementation

The datapath of the fault-tolerant matrix multiplication accelerator used in this work is shown in Fig. 1. At its core lie $s + 1$ identical multiply-accumulators (MACs), each responsible for calculating the values of elements between exactly one column of output matrix $\mathbf{C}_f$. All data is signed fixed-point, with n -bit input elements and $2n$ -bit outputs. Wide— ns -bit input and $2n(s + 1)$ -bit output—RAMs prevent starvation, allowing complete matrix rows to be accessed on a cycle-by-cycle basis. When $r = 0$, the paths for $\mathbf{A}_c$ and $\mathbf{B}_r$ are $n + \log_2 s$ bits per element: this prevents overflow within the input checksums, allowing output checksums to be valid up to the required $2n$ bits.

Fig. 1. Datapath

Checksum generation and verification logic, shown in Fig. 2, serves to perform the ABFT procedures described in Sect. 3. Rows of $\mathbf{B}$ are first fetched in turn such that the checksums in $\mathbf{B}_r$ can be calculated. The adder, narrow register and cs_r RAM are used for this purpose. Multiplication proceeds thereafter: $\mathbf{A}$'s first row is stored in the wide register, then the rows of $\mathbf{B}_r$ are presented in turn

$\frac{s^3(2^r-1)(2^n+2^r-1)}{2^{2r}} \approx s^3 2^{n-r}$. Note that the per-element paths for $\mathbf{A}_c$ and $\mathbf{B}_r$ are each $n + \max(\log_2 s - r, 0)$ -bit to optimally fit the single largest element.

All hardware shown in Figs. 1 and 2 was implemented in the programmable logic portion of a Xilinx Zynq-7000 XC7Z020 system-on-chip. Supporting hardware, formed of Xilinx IP cores, included BRAM and direct memory access controllers for facilitating data transfer between BRAM and off-chip dynamic RAM. One of the XC7Z020's two hard ARM CPU cores was used as a controller to trigger memory transfers and accelerator runs. The CPU is not integral to the functionality of the developed hardware.

6 Area and Performance Overheads

Designs were compiled using Xilinx Vivado 2014.4 for each combination of $s \in \{2, 4, 8, 16, 32\}$ and $r \in \{0, 4, 8, 12, 16, 20, 24\}$. A set of baseline designs without ABFT protection was also produced, and n was 32 in all cases. Figure 3 shows the total area—calculated as $\mu(\text{LUT} (\%), \text{FF} (\%), \text{BRAM} (\%), \text{DSP} (\%))$ —overhead versus the equivalently sized design without ABFT. The matrix size s was limited to 32 by the FPGA targetted.

Fig. 3. Resource usage and $f_{\max}$ vs unprotected design

Area overhead initially increased for $r > 0$ in all cases other than $s = 32$. This was primarily due to the introduction of the subtractors shown in Fig. 2. Gains were realised in the $s = 16$ case for $r \geq 8$, $r \geq 16$ in the $s = 8$ case and $r \geq 20$ in the remaining two. The maximum area gain, again for $s = 32$ and $r = 28$, was 23.8%. It should be noted that BRAM and DSP usage are independent of r since truncation affects only the checksum generation and verification logic, which is devoid of multipliers and contain only small, distributed memories.

The reported timing model $f_{\max}$ of each design was also recorded. Changes versus the equivalently sized unprotected designs are captured in Fig. 3. To overcome the effects of CAD noise, linear regressions are included for each plot, shown as dashed lines. Note that the value of r chosen does not affect the (clock cycle) latency of a design versus its standard ABFT equivalent, allowing $f_{\max}$ to be used for performance comparison directly. $f_{\max}$ reductions begin significantly:

for $s = 32$ and $r = 0$, $f_{\max}$ dropped by 40.8%. Designs with $r > 0$ exhibited relatively small performance improvements: for $s = 32$, a drop in frequency impact of 7.23% was found. Although trends for smaller s are actually negative, those for larger s are positive. This is a result of the lack of severe output truncation and the introduction of additional logic. Nevertheless, frequency gains were realised for larger designs, with $s = 32$ showing increasing gains for $r \neq 28$.

7 Fault Observability

Functional simulations were performed to assess the fault observability of the proposed designs across the range of implementation variables used in Sect. 6 under the presence of both permanent and transient faults. The fault model applied was that of individually targetted stuck-at-one accumulator bits. Such faults were chosen since they are representative of a range of phenomena under different conditions, e.g. worn transistors or bridged interconnects in the case of permanent faults and register or memory upsets in the case of transients. Accumulator outputs were manipulated since these components lie at the ends of the datapaths of interest, reducing the probability of logical masking and representing somewhat of a worst-case operating scenario. For each combination of implementation variables, the following steps were repeated 1,024,000 times:

1. Generate two $s \times s$ matrices, $\mathbf{A}$ and $\mathbf{B}$, and populate their elements with n -bit signed integer data selected randomly from a uniform distribution.
2. Add checksumming to $\mathbf{A}$ and $\mathbf{B}$ to form $\mathbf{A}_c$ and $\mathbf{B}_r$ as explained in Sect. 3.
3. Perform $\mathbf{C}_f = \mathbf{A}_c \mathbf{B}_r$, element-wise modulo- 2^{2n} .
4. Perform $\mathbf{C}'_f = \mathbf{A}_c \mathbf{B}_r$, element-wise modulo- 2^{2n} , with fault emulation:
  - For a permanent fault, select a (column, bit) combination from a uniform distribution. During all accumulation steps, force this bit high.
  - For a transient fault, select a (row, column, step, bit) combination from a uniform distribution. Force this bit high during computation.
5. If comparison of data and checksums within $\mathbf{C}_f$ and $\mathbf{C}'_f$ reveals that the fault was missed, record the maximum absolute error of $\mathbf{C}'_f$'s data elements.

Figure 4 shows the means of errors encountered within results flagged as false negative; those that were missed. Assuming that unmissed errors are able to be corrected, Fig. 4's results therefore represent the average expected worst-element errors introduced by RP-ABFT. They indicate that RP-ABFT allows only relatively small errors to propagate, particularly when r is small. It is around the first inflection seen in the permanent fault plots in Fig. 4 that the detection logic starts to become ineffective. Consider $s = 32$ in Sect. 4. Setting $\epsilon(cs_{out, c}) = 2^{63}$, i.e. $\vee(d_{out})$ for $n = 32$, reveals that at $r \approx 16$ corner checksums cease to be effective. Similarly, setting $\epsilon(cs_{out}) = 2^{63}$ for the same s and n shows that *all* checksumming is rendered useless at $r \approx 22$.

Fig. 4. Means of maximum absolute errors encountered within false negative results

8 Conclusion

In this paper, we introduced reduced-precision algorithm-based fault tolerance, or RP-ABFT. RP-ABFT with LSB-first checksum truncation was theorised and implemented in hardware using matrix multiplication as a case study. Our results showed that meaningful overhead reduction can be achieved by sacrificing some fault observability. Our future work on RP-ABFT will explore the false positive-to-false negative tradeoffs achievable through the manipulation of output error threshold values. We will also explore enhancements to the checksumming logic, particularly for performance, as well as output-only truncation to introduce additional tradeoff data points.

The authors acknowledge the support of the EPSRC-funded PRiME project (<http://www.prime-project.org>); grant number EP/K034448/1.

References

1. Braun, C., et al.: A-ABFT: Autonomous Algorithm-based Fault Tolerance for Matrix Multiplications on Graphics Processing Units. In: International Conference on Dependable Systems and Networks (DSN) (2014)
2. Davis, J.J., et al.: Datapath Fault Tolerance for Parallel Accelerators. In: International Conference on Field-Programmable Technology (FPT) (2013)
3. Davis, J.J., et al.: Achieving Low-overhead Fault Tolerance for Parallel Accelerators with Dynamic Partial Reconfiguration. In: International Conference on Field-programmable Logic and Applications (FPL) (2014)
4. Huang, K.H., et al.: Algorithm-based Fault Tolerance for Matrix Operations. *IEEE Transactions on Computers* C-33(6) (1984)
5. Jacobs, A., et al.: Overhead and Reliability Analysis of Algorithm-based Fault Tolerance in FPGA systems. In: International Conference on Field Programmable Logic and Applications (FPL) (2012)
6. Rexford, J., et al.: Algorithm-based Fault Tolerance for Floating-point Operations in Massively Parallel Systems. In: International Symposium on Circuits and Systems (ISCAS). vol. 2 (1992)
7. Wang, S.J., et al.: Algorithm-based Fault Tolerance for FFT Networks. *IEEE Transactions on Computers* 43(7) (1994)