Imperial College London
Department of Electrical and Electronic Engineering

Low-overhead Fault-tolerant Logic for Field-programmable Gate Arrays

James J. Davis

November 2015

Supervised by Professor Peter Y. K. Cheung

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Electrical and Electronic Engineering of Imperial College London
and the Diploma of Imperial College London

Licence

Statement of Originality

The work in this thesis is my own, except where it has been appropriately referenced and attributed.

Abstract

While allowing for the fabrication of increasingly complex and efficient circuitry, transistor shrinkage and count-per-device expansion have major downsides: chiefly increased variation, degradation and fault susceptibility. For this reason, design-time consideration of faults will have to be given to increasing numbers of electronic systems in the future to ensure yields, reliabilities and lifetimes remain acceptably high. Many mathematical operators commonly accelerated in hardware are suited to modification resulting in datapath error detection and correction capabilities with far lower area, performance and/or power consumption overheads than those incurred through the utilisation of more established, general-purpose fault tolerance methods such as modular redundancy. Field-programmable gate arrays are uniquely placed to allow further area savings to be made thanks to their dynamic reconfigurability.

The majority of the technical work presented within this thesis is based upon a benchmark hardware accelerator—a matrix multiplier—that underwent several evolutions in order to detect and correct faults manifesting along its datapath at runtime. In the first instance, fault detectability in excess of 99% was achieved in return for 7.87% additional area and 45.5% extra latency. In the second, the ability to correct errors caused by those faults was added at the cost of 4.20% more area, while 50.7% of this—and 46.2% of the previously incurred latency overhead—was removed through the introduction of partial reconfiguration in the third. The fourth demonstrates further reductions in both area and performance overheads—of 16.7% and 8.27%, respectively—through systematic data width reduction by allowing errors of less than ±0.5% of the maximum output value to propagate.

Acknowledgements

My thanks, firstly, go to my supervisor, Peter Cheung. I could not have asked for a better supervisory experience: Peter has allowed me the freedom to explore and develop largely automomously while always remaining able to provide fresh insight and offer support when it was needed. As the lead academic of the weekly 'Reliability Club' meetings and supervisor for my current research assistantship, George Constantinides has played a pivotal role in my previous, and ongoing, research, including much of the content of this thesis. Both David Thomas and Christos Bouganis have also provided perspective for the work, for which I am grateful.

To all other members of the Reliability Club, both past and present—Rosella Arcucci, Sam Bayliss, David Boland, Sumanta Chaudhuri, Rui Duarte, Zhenyu Guan, Eddie Hung, Josh Levine, Karthick Parashar, Kan Shi, Ed Stott, Michail Vavouras, Justin Wong and Rong Ye—I owe thanks for guidance and suggestions, without which much of the work in this thesis would not have been possible. Peter Ogden's advice and assistance has also been invaluable.

Members of the Circuits and Systems group, many of whom have already been mentioned, have provided help and relief from the stresses of work over the past few years, for both of which I am thankful. In particular, Wiesia Hsissen has been instrumental in allowing me to complete my research while never knowingly missing a deadline.

Special thanks go to my friends Ed Stott and Cate Slade for their hospitality over the recent months. Their kindness and generosity have allowed me to complete this thesis safe in the knowledge that I have a warm and happy home to return to each evening.

Many thanks to lifelong friends Pete & Carol Miller for their boundless encouragement. My friends across the Atlantic—particularly Dan & Lisa Albright, Clark & Kelly Grace, Jo Dee & Carl Schultz and Joe Smith—as well as my sister, Louise, have done a fantastic job of keeping me entertained during periods of—and please pardon the pun—downtime.

I am of course grateful to my parents, Chris and Mary, for everything that they have done, and continue to do, for me. I would not be who or where I am today without their

encouragement and support.

Last but by no means least, my thanks go to my partner and best friend, Melanie Albright, for her unwavering support and love.

Thank you.

Contents

List of Tables

List of Figures

List of Symbols & Abbreviations

a	accumulator latency
$\lceil\ \rceil$	ceiling
$\boldsymbol{cs}_{\mathrm{c}}$	column-wise checksum vector
$\boldsymbol{cs}_{\mathrm{r}}$	row-wise checksum vector
cs_{in}	input checksum element
cs_{out}	output checksum element
$cs_{\mathrm{out,\ c}}$	corner output checksum element
d	distance
Δ	change in
d_{in}	input data element
d_{out}	output data element
ϵ	maximum absolute error
f	number of simultaneous faults
$\lfloor\ \rfloor$	floor
f_{max}	timing model-inferred maximum operating frequency
m	multiplier latency
μ	mean
n	data width
r	truncation width
s	square matrix size
θ	output checksum element error threshold
θ_{c}	corner output checksum element error threshold

$\vee$	maximum absolute value
ABFT	algorithm-based fault tolerance
AMD	Advanced Micro Devices
ARM	Acorn reduced instruction set computer machine
ASIC	application-specific integrated circuit
AXI	Advanced eXtensible Interface
BIST	built-in self-test
BRAM	block random-access memory
BUT	block-under-test
CAD	computer-aided design
CLK	clock
CPU	central processing unit
CTR	counter
CUT	circuit-under-test
DFT	discrete Fourier transform
DMA	direct memory access
DPR	dynamic partial reconfiguration
DRAM	dynamic random-access memory
DSP	digital signal processing block
EEPROM	electronically erasable programmable read-only memory
EM	electromigration
FF	flip-flop
FFT	fast Fourier transform
FIR	finite impulse response
FPGA	field-programmable gate array
FSM	finite state machine
GPU	graphics processing unit
HCI	hot carrier injection

HDL	hardware description language
I/O	input/output
IIR	infinite impulse response
ISE	Integrated Synthesis Environment
LB	logic block
LR	launch register
LSB	least-significant bit
LU	lower- and upper-triangular
LUT	look-up table
MAC	multiply-accumulator
MOS	metal oxide semiconductor
MSB	most-significant bit
MUX	multiplexer
NBTI	negative-bias temperature instability
NMOS	n-type metal oxide semiconductor
OpenCL	Open Computing Language
ORA	output response analyser
PBTI	positive-bias temperature instability
PCAP	processor configuration access port
PL	programmable logic
PLB	programmable logic block
PMOS	p-type metal oxide semiconductor
PS	processor subsystem
PUT	paths-under-test
RAM	random-access memory
RP-ABFT	reduced-precision algorithm-based fault tolerance
SA	set of healthy spare logic cells
SA0	stuck-at-zero

SA1	stuck-at-one
SG	sample register
SM	set of logical functions assigned to faulty logic cells
SoC	system-on-chip
SP	signal path
SRAM	static random-access memory
STAR	self-testing area
TAC	transition activity counter
TCG	test clock generator
TDDB	time-dependent dielectric breakdown
TMR	triple modular redundancy
TPA	transition probability analyser
TPG	test pattern generator
TVG	test vector generator
VHDL	very high-speed integrated circuit hardware description language
XOR	exclusive-OR

1 Introduction

Aggressive process scaling leads to increasing uncertainty in the behaviour of metal oxide semiconductor (MOS) transistors, in turn decreasing their reliability. Above transistor level, such phenomena and the faults they induce can lead to reduced yield, decreased system reliability and, in extreme cases, total failure after a period of successful operation. Although error detection and correction are almost always considered for highly sensitive and susceptible applications such as those to be deployed in space or the battlefield, they are often overlooked for other, more general-purpose applications; this is likely to have to change in the future as the effects caused by such scaling continue to worsen. Self-testing and -repairing reconfigurable circuits may well present themselves to be viable facilitators for overcoming the reliability problems that transistor scaling causes.

Modern field-programmable gate arrays (FPGAs) are often exploited for their ability to realise high-performance hardware, typically through parallelisation and pipelining, without the high setup costs associated with application-specific integrated circuits (ASICs). As devices built from many millions or even billions of transistors, FPGAs are at least as susceptible to the mechanisms degrading the switching performance, eventually leading to the inoperability, of those devices as ASICs. Furthermore, FPGAs' reliance upon random-access memory (RAM) for configuration storage intensifies the potential for damage from radiation-induced upsets, since such occurrences can alter circuit-defining configuration in addition to corrupting data. They do, however, make ideal platforms upon which to develop and verify fault-tolerant techniques: their resource abundances, hierarchical structures and runtime reconfigurability present many exciting possibilities not only for the creation of elaborate configurations, but also for the ability to prevent, detect, analyse and/or correct faults (from) occurring during their lifetimes.

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 (ABFT) embodies a proven family of low-overhead error mitigation techniques able to be built upon to create

self-verifying circuitry. ABFT protection can be applied to a wide range of linear algebraic mathametical operations—commonly accelerated in hardware—making it a suitable basis for the hardening of custom logic.

This thesis is representative of several years' research into the design, implementation and verification of 'bolt-on' error detection logic used to facilitate runtime fault tolerance of FPGA-implemented hardware accelerators. ABFT is shown to be an effective error detection tool through a case study, which is expanded upon to allow for the reduction of algorithmic parallelisation in order to maintain accurate operation under faulty conditions. Further case studies are considered when analysing the fault observability of ABFT-protected operators. The sacrification of some detectability, particularly of faults associated with low-magnitude errors, is explored in order to achieve additional gains in area and power efficiency.

1.1 Contributions

The original contributions of the work presented in this thesis are:

- The implementation of a complete hardware-software platform for the verification of ABFT protection of a benchmark matrix multiplication accelerator.
- A quantitative analysis of the overheads—in terms of area and performance—incurred through the incorporation of ABFT protection within that benchmark circuit.
- Insight into the hardware fault tolerance of ABFT upon that benchmark.
- The first implementation of custom logic for error correction in the presence of faulty resources guided by an ABFT error detection mechanism.
- The first implementation of ABFT-protected hardware using dynamic partial reconfiguration (DPR) for recovery.
- A quantitative analysis of the overheads—of resources, performance and memory—incurred through the incorporation of those error correction strategies into a benchmark hardware accelerator.
- A software framework for fault simulation in hardware-accelerated linear algebra operators protected with ABFT.

- A thorough analysis of the fault tolerance of three benchmark ABFT-protected operators.
- The first consideration of distance-x, for $x > 2$, ABFT application in custom logic.
- The first consideration of distinct data and checksum bit-widths within ABFT-protected operations: reduced-precision algorithm-based fault tolerance (RP-ABFT).
- The first implementation of circuitry incorporating RP-ABFT for resilience against hardware faults.
- Analysis of the costs and benefits of applying two forms of RP-ABFT to various precisions.
- Insight into the hardware fault tolerance of RP-ABFT.

1.2 Publications

The original contributions made in this thesis and related work have been published as peer-reviewed conference papers in the following publications:

- **J. J. Davis** and P. Y. K. Cheung, "Datapath Fault Tolerance for Parallel Accelerators," in *International Conference on Field-programmable Technology (FPT)*, 2013.
- **J. J. Davis** and P. Y. K. Cheung, "Reducing Overheads for Fault-tolerant Datapaths with Dynamic Partial Reconfiguration," in *IEEE International Symposium on Field-programmable Custom Computing Machines (FCCM)*, 2014.
- **J. J. Davis** and P. Y. K. Cheung, "Achieving Low-overhead Fault Tolerance for Parallel Accelerators with Dynamic Partial Reconfiguration," in *International Conference on Field-programmable Logic and Applications (FPL)*, 2014.
- **J. J. Davis** and P. Y. K. Cheung, "Reduced-precision Algorithm-based Fault Tolerance for FPGA-implemented Custom Accelerators," in *International Workshop on Applied Reconfigurable Computing (ARC)*, 2016.

1.3 Outline

The remainder of this thesis is organised as follows. Chapter 2 primarily consists of a thorough literature review, with research having been focussed upon schemes proposed

to achieve fault tolerance—testing methods, techniques designed to mitigate degradation and fault-correction schemes—with particular attention paid to those for achieving runtime fault tolerance in FPGAs. Chapter 2 also introduces ABFT, the error detection mechanism of which represents the foundation for much of this thesis' technical content. In Chapter 3, an implementation for the detection of runtime data errors occurring within an ABFT-protected hardware accelerator is presented. This work is expanded upon in Chapter 4 in order to create a fault-tolerant hardware platform: logic capable of detecting faults within its datapath, and autonomously acting to rearrange itself in order to route around them, at runtime. Chapter 5 forms an in-depth investigation into the observability of faults within ABFT-protected datapaths, a software simulation framework having been designed to ascertain fault observabilities for three hardened mathematical operations. Chapter 6, the final technical chapter, presents research into the application of ABFT at lower levels of precision than in prior work, introducing a previously unexplored area-to-allowed error tradeoff. Concluding remarks are presented in Chapter 7.

2 Background

2.1 Introduction

In order to establish the current state of the art, and to identify gaps and promising directions in the literature to date, a thorough review was completed. Research focussed upon the three aspects vital to the exploitation of runtime reconfiguration for fault-tolerance in field-programmable gate arrays (FPGAs): testing methods, techniques designed to mitigate degradation and fault correction schemes.

2.1.1 Outline

The remainder of this chapter is organised as follows. Section 2.2 contains discussion of the various types of fault that affect metal oxide semiconductor (MOS) transistors, while Section 2.3 gives an overview of the mechanisms that contribute to their degradation over time. Section 2.4 contains details of methods that can be employed for the detection of faults and monitoring of degradation effects, grouped into either offline (Section 2.4.1), online via roving scan (Section 2.4.2) or online via health monitoring (Section 2.4.3) categories, as appropriate. Fault mitigation—pre-emptive action—is discussed in Section 2.5, while error correction—reactive action—is considered in Section 2.6. Error correction methods are classified depending on whether they require compile-time provisioning (Section 2.6.1) or are free to provision fully at runtime (Section 2.6.2). Section 2.7 introduces algorithm-based fault tolerance (ABFT); the subsections within it detail the ABFT checksumming procedures (Section 2.7.1) and demonstrate their application to applicable operators (Section 2.7.2) by numerical example. A method for classifying results based upon the locations of incorrectly computed elements is presented in Section 2.7.3. Concluding remarks are given in Section 2.8.

2.2 Faults & Errors

The distinction between various types of faults and the errors they cause is central to the concepts presented in this thesis.

Broadly speaking, faults fall into one of two categories: *permanent* and *transient*. Permanent faults are those present as a result of manufacturing defects or, potentially, that manifest after a period of time due to the effects of degradation. Stuck-at-zero (SA0) or stuck-at-one (SA1), respectively describing nets that become stuck at low and high logic levels, and open and short circuits are the permanent faults most often modelled. Timing faults—slow-switching transistors being the obvious example—represent a subcategory of permanent fault whereby errors appear only under certain timing-related conditions, such as at certain operating frequencies. Transient faults are those usually caused by radiation, generally by the 'flipping' of bits within memories. Such faults may lead to knock-on errors occurring until they are corrected, for example by overwriting values stored in affected memory locations.

The nature of FPGAs blurs the line between permanent and transient faults, since configuration memory upsets—traditionally considered to be transient—can cause continuing incorrect circuit functionality; classically a symptom of permanent faults. For this reason, the concept of *hard* and *soft* faults is introduced: hard faults are those that change system behaviour via physical circuit alteration, while soft faults are those that do so through configuration changes.

In the work presented in the technical chapters of this thesis, data errors (i.e. incorrectly computed values) are used to infer the presence and location of the faults that cause them, thus allowing for corrective action to be taken to prevent those faults causing errors in the future. In terms of error correction, the focus is generally on permanent hard faults since, as outlined in Sections 2.5 and 2.6, fault tolerance strategies exist that are considered to combat other types more adequately.

2.3 Degradation

Five main mechanisms contribute to the physical degradation of transistors in FPGAs and other MOS-based devices [10] [11]: *negative-bias temperature instability (NBTI)* and *positive-bias temperature instability (PBTI)*, *hot carrier injection (HCI)*, *electromigration (EM)* and *time-dependent dielectric breakdown (TDDB)*. All are predicted to result in

worsening effects, shortening device lifetime, as feature sizes decrease due to corresponding increases in gate field strengths and current densities [12].

NBTI and PBTI, respectively applicable to p-type metal oxide semiconductor (PMOS) and n-type metal oxide semiconductor (NMOS) transistors, along with HCI, cause charges to become trapped in their gate-channel interfaces, increasing threshold voltages and reducing channel mobilities [13]. These effects result in decreased switching speeds, leading to timing faults. EM can lead to bridging faults (short circuits) caused by the movement of metal ions within and between transistor interconnects, while TDDB gives rise to the creation of shorts across transistors via a trapping of charges in gate oxides, resulting in increased gate leakage that promotes further charge trapping, and so on in a cyclical fashion [10].

NBTI and PBTI are considered to be the dominant mechanisms causing degradation in modern FPGAs [13] [14]; this is particularly true of those with high-K gate dielectrics and metal gates. NBTI promoted by static-zero logic inputs has been shown to cause the most rapid degradation in the look-up tables (LUTs) of Altera Cyclone III [15] FPGAs [5].

2.4 Fault Detection

Before any type of error correction can be applied, the faults causing those errors must be detected and the offending resources located. The following three subsections describe testing methods suitable for the detection and location of all fault types, both with regards to logical function, i.e. correct output, and timing. Schemes can be considered to be either *offline* or *online* depending on whether they are carried out independently to an FPGA's application configuration (offline) or not (online). Online methods can be further classified as operating either in tandem with, but distinct from, the application (*roving scans*), or as part of the application configuration itself (*health monitoring*). Note that many of the methods presented as offline could be adapted to become online roving schemes and vice-versa.

Although academic research into the detection of hard fault-causing defects is mature (focus has now shifted towards the detection of timing faults and process variation, while some defect-tolerant schemes have been adopted commercially), they are discussed here since they are still relevant to mitigation and correction schemes that require hard fault detection and location.

2.4.1 Offline

During system start-up, or in applications where periodic downtime of an FPGA is possible, one or more dedicated testing configurations can be loaded onto a device in order to test its resources. Self-contained offline testing methods, i.e. ones which operate without external test hardware, are known as built-in self-test (BIST) schemes. Typically, logic is configured as test pattern generators (TPGs), circuits-under-test (CUTs) and output response analysers (ORAs), with placements of these often exchanged across test phases to verify all resources. Offline testing methods are able to achieve high fault coverage due to the complete flexibility presented by having an unused array to test, impose no restrictions upon the application configurations that can be implemented on the FPGAs they are designed to test and can locate dormant faults, i.e. those that exist in resources not used by the application configuration. They do, however, require application circuitry to be taken entirely offline and, where periodic testing is employed, fault detection latencies will be limited by the frequency of that testing.

Programmable logic block (PLB) functional testing is a highly researched area [1] [16] [17]. Groups of logic are often cascaded to form paths-under-test (PUTs) in order to lower testing times by reducing the numbers of reconfigurations required [16] [17]. Figure 2.1 shows one such test arrangement [1]: C groups of m-bit TPGs, where m is the number of inputs to each block-under-test (BUT), are used to drive C groups of n BUTs connected in parallel. The O outputs of each BUT is then fed into the corresponding O groups of n ORAs in order to compare behaviours across the C groups of BUTs. Testing covering entire FPGAs has been achieved with detection granularities down to one in five LUTs [17].

Functional testing specific to the location of interconnect faults has also been discussed [18] [19] [20], with some originally presented methods [18] extended [21] to allow fault locations to be established once they have been detected. Hierarchical techniques suitable for testing cluster-based FPGAs have also been proposed [19]. Recent work [20] has resulted in a testing scheme for both interconnect and PLBs that obtains 100% fault coverage.

Consideration to timing fault BIST in PLBs has been given [22], and it has been shown that LUT propagation delays are dependent upon both the functions they implement and the input patterns applied to them. The delay measurement method presented by Wong *et al.* [2] allows the maximum operating frequency of an arbitrary combinatorial

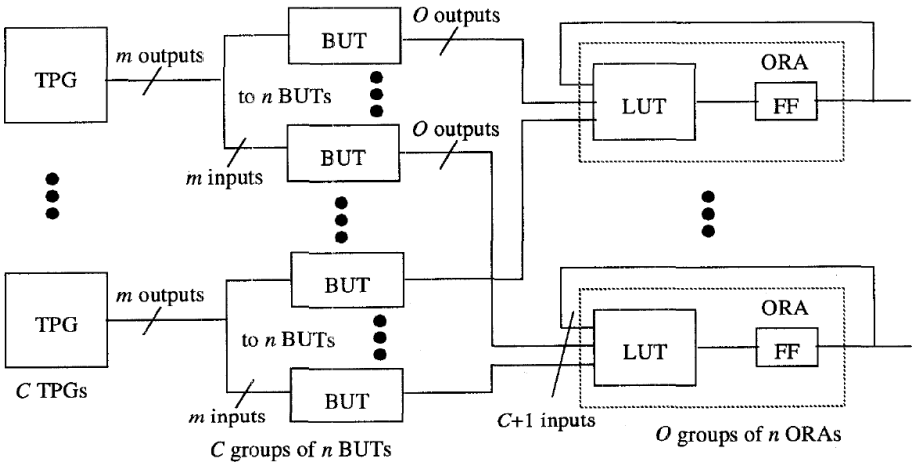


Figure 2.1: Test arrangement used by Stroud *et al.* [1], exemplifying the functional testing of PLBs with TPGs, CUTs and ORAs.

and/or sequential circuit to be established by recording the transition probabilities, i.e. likelihoods that logic levels change between cycles, at its output as the clock frequency is swept. Steps in plots of transition probability versus frequency allow rising and falling edge propagation delays to be estimated. Figure 2.2 shows the required arrangement of measurement circuitry around the arbitrary logic being tested: an N-bit test vector generator (TVG) is used to supply inputs to the CUT, sandwiched between registers clocked at varying frequencies by a test clock generator (TCG). The M outputs of the CUT are then analysed by the transition activity counter (TAC) and transition probability analyser (TPA) that follow. Frequency estimates have been shown to be within 12% of those obtainable using an alternative, exhaustive testing procedure [23].

Timing fault testing of FPGA interconnect has also been explored [24] [25] [26]. Two test configurations are presented by Wang *et al.* [25]: the first using feedback to alleviate clock skew and the second for validating that skew across PLBs, with a combination of the two reported to achieve defect coverage of over 98%. A more recent method [26] was shown to achieve 100% diagnostic resolution in the presence of individual, and almost 100% with pairs of, defects.

Offline testing methods have also been proposed for testing less frequently considered FPGA hardware. BIST techniques for embedded multipliers have been presented [27] [23], while work addressing testing of digital signal processing blocks [28] and embedded memories [29] has also been completed. Recent publications have addressed timing fault testing in clock networks [30] [31] and input/output hardware [32].

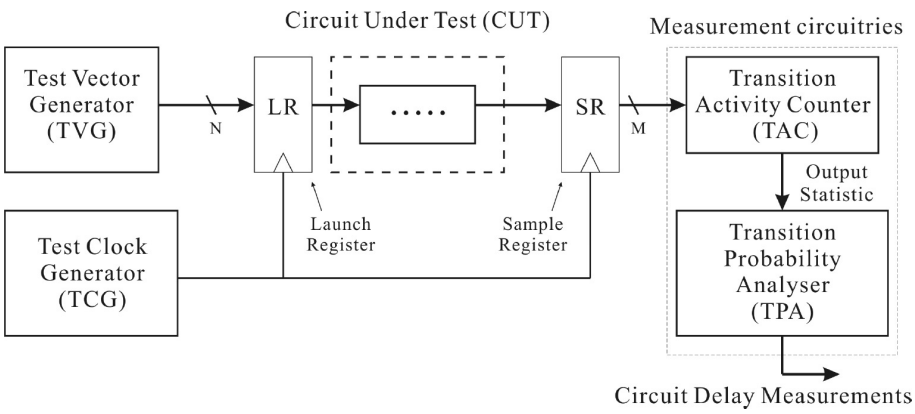


Figure 2.2: Delay measurement method proposed by Wong *et al.* [2] that allows the maximum operating frequency of an arbitrary combinatorial and/or sequential circuit to be established by recording the transition probabilities at its output as the clock frequency is swept.

2.4.2 Online (Roving)

Dynamic partial reconfiguration (DPR) can be exploited to allow portions of an FPGA to be tested while the remainder continues to perform its normal functions. Once verification of particular areas has been completed, test hardware is moved to allow different areas to be verified. While an entire array does not need to be reconfigured to facilitate testing, as in offline testing methods, the requirement to introduce temporary testing areas into designs limits maximum resource usage and forces clock slowdown due to path-lengthening effects: required slowdowns of around 2.5–15.1% have been reported [3]. Roving scans present opportunities to detect dormant faults, but often introduce significant fault detection latency: chip-wise scans have been reported to take 850ms [3], implying worst-case fault detection latencies of the same amount.

Abramovici *et al.* presented a roving scheme [3] that tests PLB and interconnect functionality by configuring a chip-wide row and column of PLBs as self-testing areas (STARs). Complete rows and columns are used to allow testing of horizontal and vertical global routing resources, respectively. Each PLB within each STAR is used as part of either a TPG, CUT or ORA, with these roles rotated regularly to allow each PLB to be tested in all modes. Over time, the STARs are moved in order to test all PLB and interconnect resources. Figure 2.3 shows the roving principle for a column of resources: application functionality is shifted to a neighbouring column such that the column-under-test is moved by one place. Improvements to the diagnosis techniques originally used have been pro-

posed [33] [34] [35]. Changes have been suggested that allow individual or pairs of faulty PLBs to be identified without reconfiguration, reducing the overall diagnosis time [33]. A new testing architecture has been presented [34], reportedly able to identify 96% of faulty PLBs in an FPGA with up to 10% of randomly distributed faulty logic resources: a 38% improvement over the original diagnosis scheme. Focussed upon the location of interconnect faults, another modified testing scheme [35] reduces detection latency by an order of magnitude over its predecessor by greatly reducing the number of reconfigurations required. A divide-and-conquer approach is used to identify points of failure and achieve high diagnosability, including in the presence of multiple faults: 99.3% fault coverage was reported with a fault density of 10%.

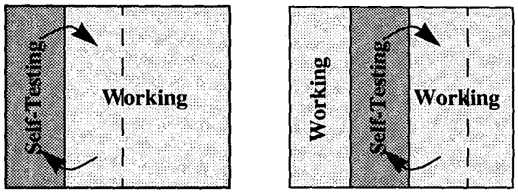


Figure 2.3: Visualisation of the roving test scheme proposed by Abramovici *et al.* [3], showing the roving principle for a column of resources: application functionality is shifted to a neighbouring column such that the column-under-test is moved by one place.

Architectural modifications to facilitate online PLB functional fault detection have been proposed [36]: a roving scan scheme that reserves a column of resources at design-time to facilitate testing was suggested. Detection latencies are low—hundreds or thousands of chip-wise scans per second were reported to be possible—but an additional memory and multiplexer must be added to each PLB.

The roving STAR scheme [3] was used as an online testing framework for a timing fault identification method [37] which proposed configuring logic and routing resources to allow propagation delays between pairs of PUTs to be compared.

2.4.3 Online (Health Monitoring)

While offline and roving methods configure FPGA resources with temporary testing circuitry to exercise them, health monitoring schemes use permanent, additional hardware to monitor the state of the application. Faults of all types can be detected in monitored hardware, and such techniques offer the lowest possible detection latencies at the expense

of consuming varying amounts of additional area and/or causing the incursion of various levels of throughput reduction.

Modular redundancy methods involve the duplication of logic for specific operations, allowing them to be performed, in parallel, more than once. Voting circuitry is used at the output of the processing logic to detect discrepancies indicative of faults. Triple modular redundancy (TMR) [38], in which operations are each performed three times, is particularly popular. Employing modular redundancy allows faults to be detected almost immediately, and delays added to signal paths by the voting logic are small. Diagnostic resolution is, however, limited by the scale of the functional block being replicated, since a fault can only be attributed to a particular instance of it. Resolution can be improved by employing redundancy multiple times on subsections of a block, however area overheads increase rapidly: TMR requires over 200% extra area compared to a single instance of the same functional logic. Tradeoffs between resolution and area overhead are therefore necessary.

Recent works sought to lower the area penalties of modular redundancy through the use of reduced-precision replicated modules [39] [40] and the application of differing numbers of them—either none, one (for duplicate-with-compare) or two (for TMR)—depending on real-time upset rates [41].

Work by de Lima *et al.* [42] aimed to reduce hardware redundancy by introducing time redundancy: operations are each carried out twice, serially, by the same logic, with the inputs for every second computation encoded in order to utilise the hardware differently. Outputs are buffered, decoded and checked for discrepancies. In the sample application presented, the technique was reported to consume 2.3% less area than an equivalent TMR approach while introducing 8% extra latency per replicated computation.

Concurrent error detection techniques generally require the addition of less logic than redundancy methods. Rather than having operations performed multiple times, error coding information, e.g. parity, is added to data buses, memories, etc., which can then be verified by testing hardware to detect errors. Such schemes often suffer from confounding problems: multiple results may have the same error code values, which can mask faults. A two-rail scheme for combinational logic was introduced [43] to facilitate error detection. Boolean functions are split into expressions with no more than four inputs each, with these then mapped to predefined product and/or sum logic cells, each with normal and complemented outputs, implemented in PLBs. Arbitrarily complex designs can be constructed

using such cells, and a two-rail checker cell is provided for testing individual or multiple logic cells' outputs: matching logic levels on normal and complementary output pairs indicate faults. Area overheads for this method are high: 76% more area was consumed, on average, than with direct implementation in PLBs.

Methods normally used in offline testing were employed for online testing in work by Karri *et al.* [44], in which TPGs and ORAs are added to functional units to be tested. Clock cycles known to be otherwise unused by those units are then used to apply test inputs to them, with their outputs checked for invalid results. Detection latencies of a few milliseconds were reported, but overheads were high: around 25–35% extra area was consumed, including 50% more registers.

Levine *et al.* presented a method for the measurement of timing slack in circuit paths between registers [4], represented diagrammatically in Figure 2.4. For each path under monitoring (PUM) required, an additional, 'shadow' register (S) is added to its terminus in parallel with the existing register (P), thereby creating two signal paths (SP1 and SP2). The shadow register is clocked by a phase-shifted version (S_CLK) of the system clock (M_CLK), with the phase swept over time: discrepancies between the registers' outputs indicate timing faults. Pass/fail data recorded with varying amounts of phase shift can be used to infer the path's maximum frequency and, if it exists, the current margin between its actual and minimal propagation delays: the timing slack. Both an error counter and first-fail recorder, which latches once the first timing error has been encountered, are provided. A maximum error in delay measurement of 1.2% was reported, with a 0.28% negative speed impact incurred by the addition of the monitoring circuitry. An average PLB overhead of 2.7% was given.

Current monitoring circuits [45] can be added to designs as suggested by Nicolaidis [46] in order to detect power usage anomalies indicative of hard faults such as stuck-ats. Usage of such methods may necessitate periodic slowdown of the application hardware in order to acquire accurate current readings, however.

2.5 Fault Mitigation

The ability to reconfigure FPGAs at runtime presents opportunities to slow degradation and/or conceal its effects. The wear-levelling approaches detailed by Stott *et al.* [5] aim to improve FPGA reliability by periodically loading new configurations onto the target device. At design-time, alternative configurations for the application are computed that,

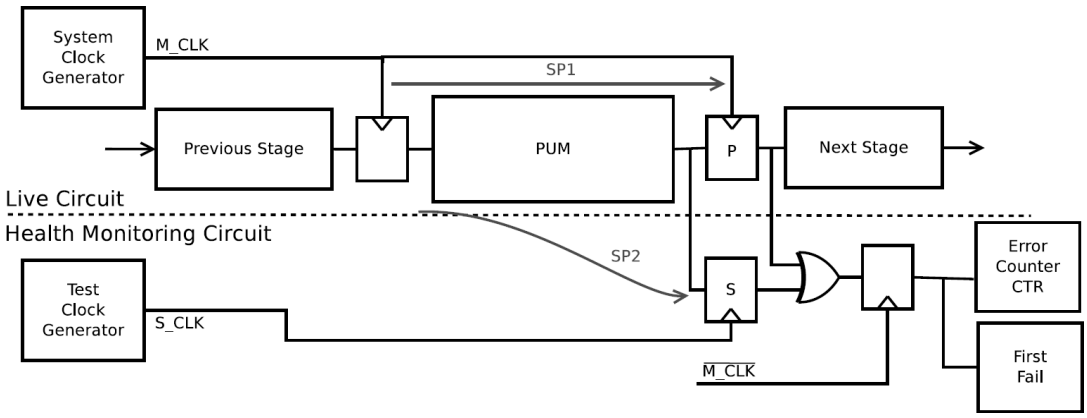


Figure 2.4: Timing slack measurement method proposed by Levine *et al.* [4]. Discrepancies between the registers' outputs, the secondary register being clocked by a phase-shifted version of the primary's, indicate timing faults.

while exhibiting the same functionality, exercise resources differently such that, when applied alternately at runtime, they aim to mitigate degradation by minimising electrical hotspots. Examples of the three classes of wear-levelling presented are shown in Figure 2.5. *Alternative mapping* involves the inversion of nets within the application, *spare resources* involves the swapping of functionality between used and unused PLBs and *alternative placement* involves the shifting of functionality between used PLBs. Reductions in timing degradation over the course of accelerated-life tests in hardware equivalent to five years of normal usage of over 20% were reported.

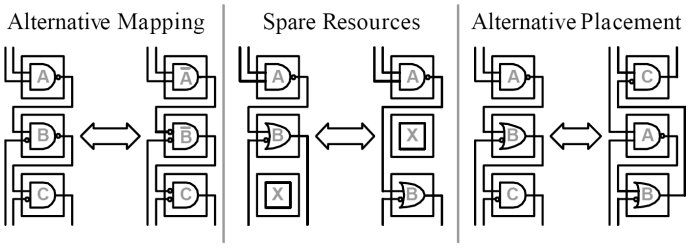


Figure 2.5: Wear-levelling strategies proposed by Stott *et al.* [5]. Alternative configurations for the application are computed that, while exhibiting the same functionality, exercise resources differently such that, when applied alternately at runtime, they aim to mitigate degradation by minimising electrical hotspots.

2.6 Error Correction

2.6.1 Compile-time Provisioning

It is possible for systems to respond to fault detection and location information without resorting to often expensive, both in terms of time and computational resources, replacement and -routing. Schemes that avoid such computation at runtime generally offer lower fault tolerance than their fully dynamic counterparts, described in Section 2.6.2.

A repair strategy involving the application of precompiled alternative configurations for particular sections of an FPGA, referred to as tiles, was presented by Lach *et al.* [6]. Alternatives are intended to be computed such that they utilise different resources within the tiles in order that, when a faulty resource is identified, an alternative that does not rely upon its functionality can be selected for that tile. In order to achieve this, at least one resource within each tile must be reserved as spare and configurations that each avoid the use of at least one of the resources within it computed. The principle operation is exemplified in Figure 2.6: four alternative configurations, each exhibiting identical functionality and featuring the same and consistently placed inputs (A to D) and output (Y), are shown, with each of the alternatives designed such that a different resource within the tile is left unused. Multiple failures within a single tile cannot be tolerated without yet more spare resource reservation and the design-time computation and runtime storage of more configurations.

A fault-tolerant technique based upon cluster shifting in 'chains' has also been presented [7]. Here, functionality in faulty logic is shifted to neighbouring fault-free clusters, some of which are reserved at design-time, and the strategy for reserved interconnect ensures that rerouting is not required and extra delay to data paths is not added following reconfiguration. An example of this is shown in Figure 2.7: if the cells containing functions A and E are found to be faulty, all cell functionality is shifted right by one place, with routing reconfigured to suit the shift, in order to prevent the use of the non-operational resources. Once the shift has taken place, the rightmost cells—originally reserved as spares—are then occupied by functions D and H.

An FPGA architecture drawing inspiration from biology, reported to exhibit self-repair and -healing properties, has been described [47], with analysis of its reliability presented in later work [48]. Array elements are analogous to biological cells, each storing the functional configuration for the entire device, and perform certain binary functions dependent upon

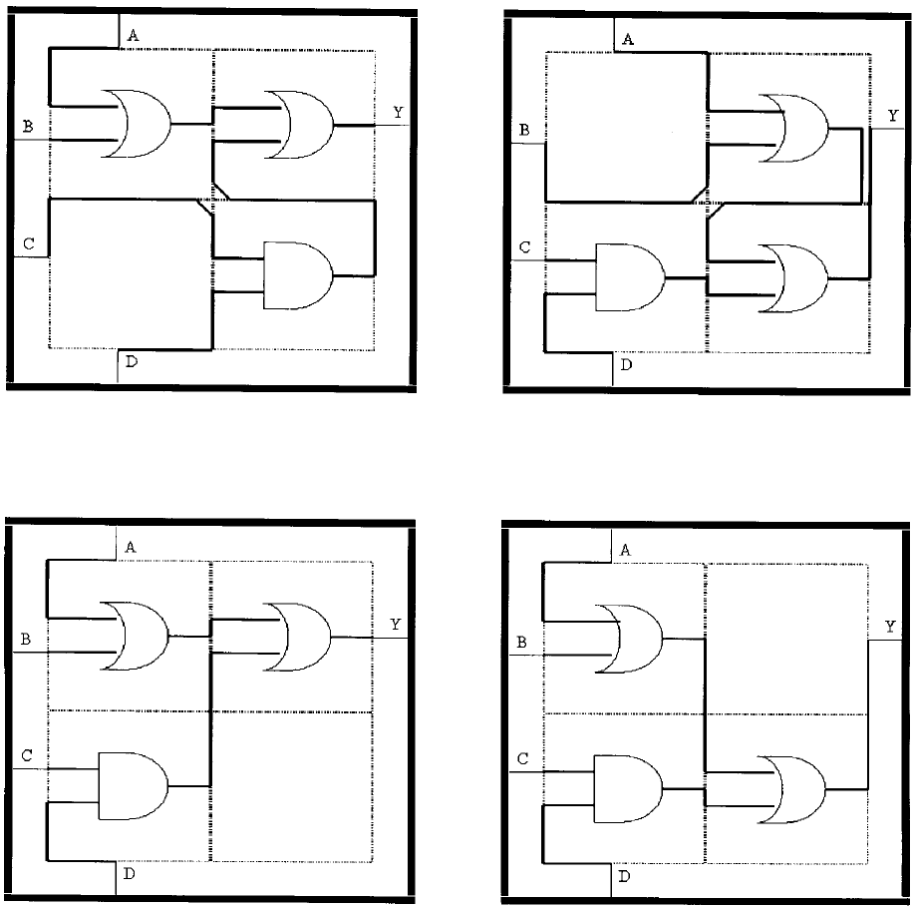


Figure 2.6: Visualisation of the repair scheme proposed by Lach *et al.* [6], in which four alternative configurations, each exhibiting identical functionality and featuring the same and consistently placed inputs (A to D) and output (Y), are shown, with each of the alternatives designed such that a different resource within the tile is left unused.

their position. It was proposed that fault combating should be attempted via shifting of functionality between cells, with circuitry added to allow those found to be faulty to be bypassed: a minimal level of reconfiguration would therefore be required to overcome faults. Significant area overhead is introduced by the requirement to store a functional description for the entire array within each of its elements, however.

2.6.2 Runtime Provisioning

Repair schemes that require placement and/or routing to be performed at runtime, while dependent upon the availability of often significantly powerful reconfiguration controllers, offer the most potential for repair since they are amongst the least restricted in terms of

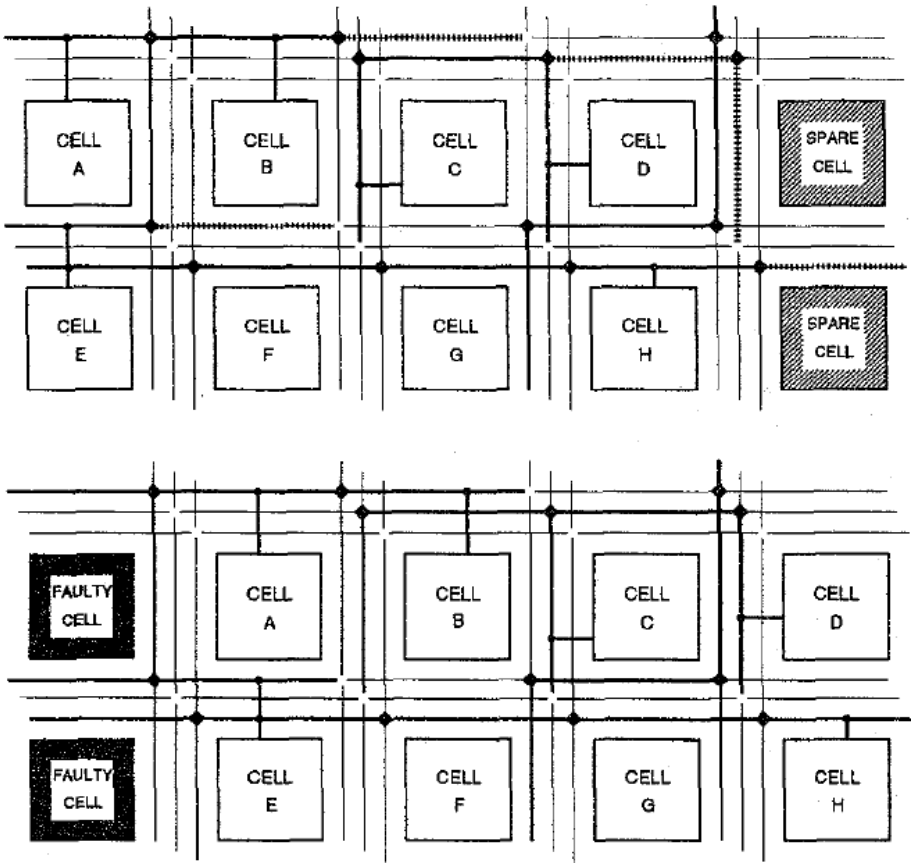


Figure 2.7: Visualisation of the repair scheme proposed by Hanchek *et al.* [7]. Functionality in faulty logic is shifted to neighbouring fault-free clusters, some of which are reserved at design-time, and the strategy for reserved interconnect ensures that rerouting is not required and extra delay to data paths is not added following reconfiguration.

resource utilisation.

Computationally expensive and time-consuming chip-wide placement and routing can be avoided in cases where faults are contained, and repair can be successfully completed, within a single logic cluster [49]. Although the application of such schemes does not have drastic effects upon system timing, many faults cannot be tolerated and they are reliant upon spare resources being available within each cluster for use during repair.

The concept of pebble-shifting [50] was introduced as a means to bypass faulty clusters while minimising timing degradation by shifting functionality between them, consuming fault-free spares where necessary, while attempting to keep interconnect lengths between clusters as short as possible. Average timing degradation of 0.13% after such shifting had taken place was reported [51].

The roving STAR testing architecture has also been used as the basis for repair [3]. A combination of in-cluster reconfiguration and pebble shifting, with preference of application in that order, is used, and the proposed fault-tolerant techniques also allow for, in some cases, the use of partially defective resources. A worst-case figure for the proportion of time the system clock must be frozen—halting the application—in order to test for, diagnose and correct repairable faults of 6.25% was given.

Faults in PLBs can also be dealt with at the cluster level [8]: functionality in faulty clusters is moved to fault-free resources, while interconnect faults are corrected via rip-up and rerouting. A representation of such a repair is presented in Figure 2.8: if the leftmost cluster is found to be faulty, functionality is moved in order to restore correct operation while keeping the wire lengths between clusters as short as possible. This is achieved, in this case, by shifting each of the three functions one place to the right. Configuration memory readback is exploited to avoid having to store a netlist for the application, however significant computational resources are required for the compilation of new configurations.

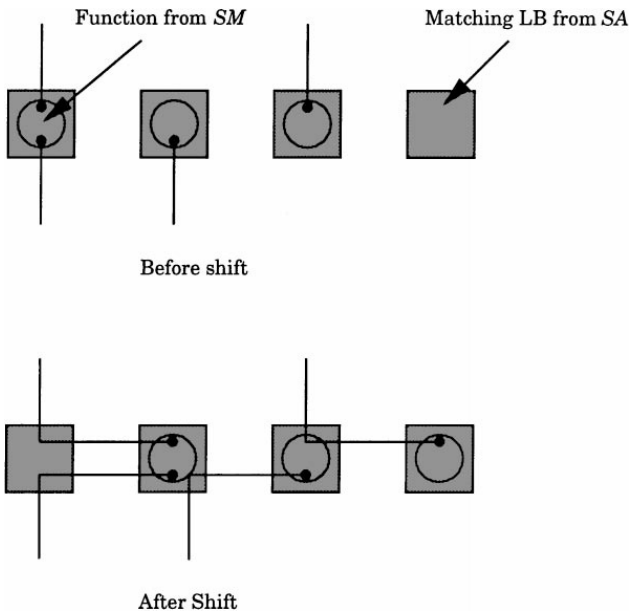


Figure 2.8: Visualisation of the repair scheme proposed by Emmert *et al.* [8]. Functionality in faulty clusters is moved to fault-free resources, while interconnect faults are corrected via rip-up and rerouting.

An evolutionary approach to repair has also been taken [9]: here, pairs of alternative configurations of the same functional module, taken from a group of competing candidate configurations, are tested against each other. When particular instances of a module

are found to produce incorrect output, they are randomly mutated and readded to the group for analysis. Figure 2.9 represents the proposed layout: the positions of the pair of active configurations are shown, along with the surrounding control logic. Note that each competing configuration contains circuitry—a discrepancy checker—to compare its output to its neighbour's; this is done such that errors within the testing circuitry itself can be detected and potentially repaired. Although such systems may be able to 'discover' novel repair strategies, their random nature makes them entirely indeterministic: there is no guarantee of finding suitable repairs within particular times nor, indeed, that such repairs even exist.

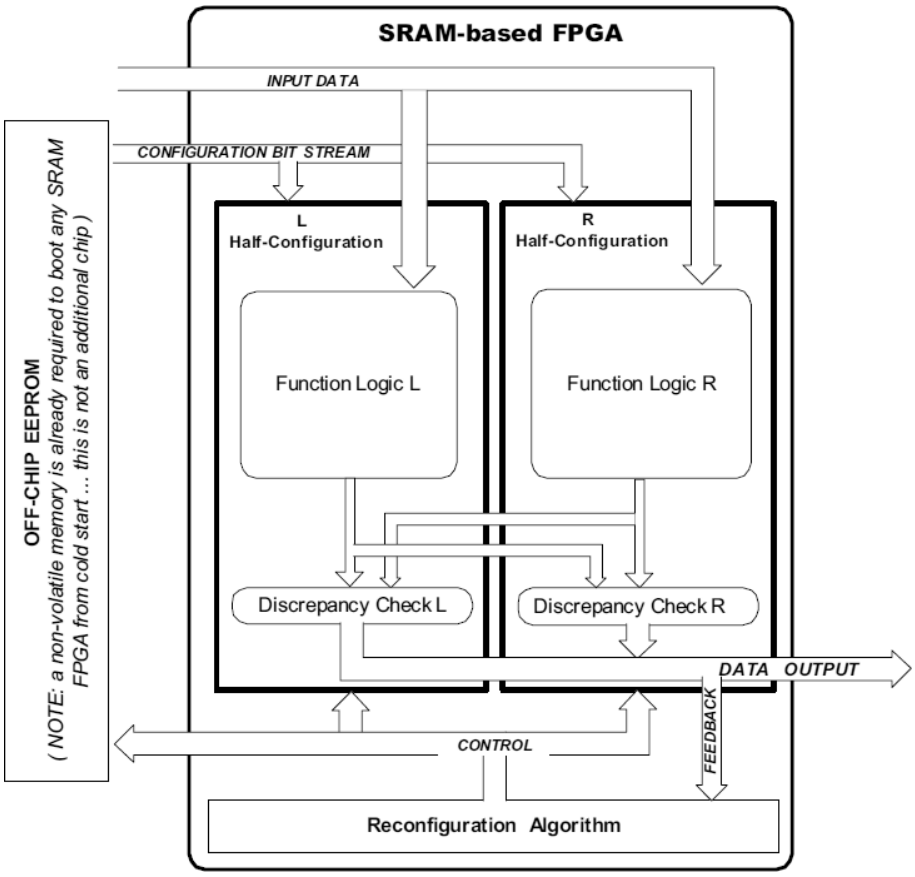


Figure 2.9: Evolutionary repair scheme proposed by DeMara *et al.* [9]. When particular instances of a module are found to produce incorrect output, they are randomly mutated and readded to a group of competing candidate configurations for analysis through being tested against each other.

Fault-tolerant architectures based around microprocessor cores residing on FPGAs have been analysed [52] [53]. In work by Girau *et al.* [52], an array of 156 simple, identical cores was proposed, with each core able to detect faults within itself and its neighbours;

functionality within faulty cores is simply moved to fault-free spares. A software framework allowing faulty hardware peripherals to be replaced with soft-core equivalents has also been presented [53].

2.7 Algorithm-based Fault Tolerance

By applying fault-tolerant techniques at a level above transistors, gates or small circuits—at the algorithmic layer—it is possible to produce robust designs capable of detecting the presence of faults during their normal operation with low impacts upon both area and performance. A wide range of linear algebra operators, including matrix operations and Fourier transformations, can be protected with *ABFT* techniques [54] [55]. ABFT was recently applied to matrix multiplication in FPGAs [56] with promising results: the authors reported a 99% decrease in design vulnerability at the expense of 25% area overhead.

While ABFT has traditionally been used to protect fixed-point operations, the methods are compatible with floating-point arithmetic as well. Of relevance to Chapter 6 are the errors, in this case introduced by floating-point operations, which necessitate error bounding to distinguish them from those caused by other mechanisms [57]. Recent work [58] sought to lower the required bounds in a graphics processing unit (GPU)-accelerated floating-point benchmark by analysing input data prior to each computation.

2.7.1 Matrix Encoding & Decoding

Any $m \times n$ *data* matrix, $\boldsymbol{D}$, can be augmented with distance (d) additional rows of column-wise checksums to produce an $(m+d) \times n$ *column checksum-encoded* matrix, $\boldsymbol{D_c}$. This is achieved by performing $\boldsymbol{D_c} = \boldsymbol{G}_\mathrm{c}\boldsymbol{D}$, where *generator* matrix $\boldsymbol{G}_\mathrm{c}$ is constructed as shown in Equation 2.1. $\boldsymbol{I}$ is the identity matrix.

$$\boldsymbol{G}_\mathrm{c} = \begin{pmatrix} & \boldsymbol{I}_{m\times m} & & \\ & & & \\ 2^0 & 2^0 & \cdots & 2^0 \\ 2^0 & 2^1 & \cdots & 2^{m-1} \\ \vdots & \vdots & \ddots & \vdots \\ 2^0 & 2^{d-1} & \cdots & 2^{(d-1)(m-1)} \end{pmatrix} \tag{2.1}$$

The final d rows of $\boldsymbol{G}_\mathrm{c}$ are linearly independent [59]; thus, they represent a distance-

$(d+1)$ code and are consequently capable of facilitating the detection of at least d errors per column. An example of $\boldsymbol{G}_\mathrm{c}$'s application, in which $m = n = d = 2$, is given in Equation 2.2.

$$\boldsymbol{D}_\mathrm{c} = \boldsymbol{G}_\mathrm{c}\boldsymbol{D} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 4 & 6 \\ 7 & 10 \end{pmatrix} \tag{2.2}$$

Note that $\boldsymbol{D}$ itself is a sub-matrix within $\boldsymbol{D}_\mathrm{c}$, occupying the uppermost $m \times n$ elements. The added column-wise checksums are shown in red.

Row- rather than column-wise checksums can be added to a data matrix by performing the complementary operation, $\boldsymbol{D}_\mathrm{r} = \boldsymbol{D}\boldsymbol{G}_\mathrm{r}$, where generator matrix $\boldsymbol{G}_\mathrm{r}$ is as shown in Equation 2.3.

$$\boldsymbol{G}_\mathrm{r} = \begin{pmatrix} & 2^0 & 2^0 & \cdots & 2^0 \\ & 2^0 & 2^1 & \cdots & 2^{d-1} \\ \boldsymbol{I}_{n\times n} & \vdots & \vdots & \ddots & \vdots \\ & 2^0 & 2^{n-1} & \cdots & 2^{(n-1)(d-1)} \end{pmatrix} \tag{2.3}$$

$\boldsymbol{G}_\mathrm{r}$'s application to data matrix $\boldsymbol{D}$ will lead to the addition of d columns of row-wise checksums, producing an $m \times (n+d)$ *row checksum-encoded* matrix $\boldsymbol{D}_\mathrm{r}$. The linear independence property of the final d rows of $\boldsymbol{G}_\mathrm{c}$ also applies to the final d columns of $\boldsymbol{G}_\mathrm{r}$. Equation 2.4 contains an example of $\boldsymbol{G}_\mathrm{r}$'s application in which $m = n = d = 2$.

$$\boldsymbol{D}_\mathrm{r} = \boldsymbol{D}\boldsymbol{G}_\mathrm{r} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 5 \\ 3 & 4 & 7 & 11 \end{pmatrix} \tag{2.4}$$

Row-wise checksums are shown in green. In parallel to the application of $\boldsymbol{G}_\mathrm{c}$, note that $\boldsymbol{D}$ itself occupies the leftmost $m \times n$ elements of $\boldsymbol{D}_\mathrm{r}$.

An $(m+d) \times (n+d)$ *full checksum-encoded* matrix $\boldsymbol{D}_\mathrm{f}$ can be formed by performing column and row checksum generation simultaneously: $\boldsymbol{D}_\mathrm{f} = \boldsymbol{G}_\mathrm{c}\boldsymbol{D}\boldsymbol{G}_\mathrm{r}$. For the same data matrix $\boldsymbol{D}$ used in Equations 2.2 and 2.4 with $d = 2$, the corresponding $\boldsymbol{D}_\mathrm{f}$ is as shown in Equation 2.5.

$$\boldsymbol{D}_{\mathrm{f}} = \boldsymbol{G}_{\mathrm{c}}\boldsymbol{D}\boldsymbol{G}_{\mathrm{r}} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 5 \\ 3 & 4 & 7 & 11 \\ 4 & 6 & 10 & 16 \\ 7 & 10 & 17 & 27 \end{pmatrix} \tag{2.5}$$

Note that the elements of $\boldsymbol{D}$ appear as before, and that column- and row-wise checksums present in $\boldsymbol{D}_{\mathrm{f}}$ are identical to those obtained in Equations 2.2 and 2.4, respectively. The additional elements, shown in yellow, are both column- *and* row-wise checksums since they can be formed from elements in either dimension.

Decoding a checksum-encoded matrix of any type is a trivial process, requiring simply stripping off the final d rows (for a column checksum-encoded matrix), columns (for row) or both (for full) to leave the data sub-matrix only.

Following storage, transmission or computation that preserves the form of checksum-encoded matrices, integrity can be verified by comparing the checksum elements within a checksum-encoded matrix to those produced from the data elements. Non-zero differences are indicative of the presence, locations and magnitudes of errors within checksum-encoded matrices. Column and row checksum-encoded matrices, respectively, can be multiplied by *verification* matrices $\boldsymbol{V}_{\mathrm{c}}$ and $\boldsymbol{V}_{\mathrm{r}}$, shown in Equations 2.6 and 2.7, to produce *discrepancy* matrices $\boldsymbol{\Delta}_{\mathrm{c}}$ and $\boldsymbol{\Delta}_{\mathrm{r}}$ for this purpose.

$$\boldsymbol{V}_{\mathrm{c}} = \begin{pmatrix} 2^0 & 2^0 & \cdots & 2^0 \\ 2^0 & 2^1 & \cdots & 2^{m-1} \\ \vdots & \vdots & \ddots & \vdots \\ 2^0 & 2^{d-1} & \cdots & 2^{(d-1)(m-1)} \end{pmatrix} \quad -\boldsymbol{I}_{d\times d} \tag{2.6}$$

$$\boldsymbol{V}_{\mathrm{r}} = \begin{pmatrix} 2^0 & 2^0 & \cdots & 2^0 \\ 2^0 & 2^1 & \cdots & 2^{d-1} \\ \vdots & \vdots & \ddots & \vdots \\ 2^0 & 2^{n-1} & \cdots & 2^{(n-1)(d-1)} \\ & & & \\ & -\boldsymbol{I}_{d\times d} & & \end{pmatrix} \tag{2.7}$$

$\boldsymbol{\Delta}_{\mathrm{c}}$ is produced by performing $\boldsymbol{\Delta}_{\mathrm{c}} = \boldsymbol{V}_{\mathrm{c}}\boldsymbol{D}_{\mathrm{c}}$, yielding a $d \times n$ discrepancy matrix, while performing $\boldsymbol{\Delta}_{\mathrm{r}} = \boldsymbol{D}_{\mathrm{r}}\boldsymbol{V}_{\mathrm{r}}$ produces an $m \times d$ discrepancy matrix $\boldsymbol{\Delta}_{\mathrm{r}}$. Equations 2.8 and 2.9 respectively show the verification process for the checksum-encoded matrices $\boldsymbol{A}_{\mathrm{c}}$ and $\boldsymbol{A}_{\mathrm{r}}$ obtained in Equations 2.2 and 2.4.

$$\boldsymbol{\Delta}_{\mathrm{c}} = \boldsymbol{V}_{\mathrm{c}}\boldsymbol{A}_{\mathrm{c}} = \begin{pmatrix} 1 & 1 & -1 & 0 \\ 1 & 2 & 0 & -1 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 4 & 6 \\ 7 & 10 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \tag{2.8}$$

$$\boldsymbol{\Delta}_{\mathrm{r}} = \boldsymbol{A}_{\mathrm{r}}\boldsymbol{V}_{\mathrm{r}} = \begin{pmatrix} 1 & 2 & 3 & 5 \\ 3 & 4 & 7 & 11 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \tag{2.9}$$

Verification of a full checksum-encoded matrix is done by considering column- and row-wise checksums independently, producing two discrepancy matrices: column-wise $\boldsymbol{\Delta}_{\mathrm{f,\,c}}$ and row-wise $\boldsymbol{\Delta}_{\mathrm{f,\,r}}$. For the full checksum-encoded matrix obtained in Equation 2.5, $\boldsymbol{\Delta}_{\mathrm{f,\,c}}$ and $\boldsymbol{\Delta}_{\mathrm{f,\,r}}$ are calculated as shown in Equations 2.10 and 2.11, respectively.

$$\boldsymbol{\Delta}_{\mathrm{f,\,c}} = \boldsymbol{V}_{\mathrm{c}}\boldsymbol{D}_{\mathrm{f}} = \begin{pmatrix} 1 & 1 & -1 & 0 \\ 1 & 2 & 0 & -1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 & 5 \\ 3 & 4 & 7 & 11 \\ 4 & 6 & 10 & 16 \\ 7 & 10 & 17 & 27 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{2.10}$$

$$\boldsymbol{\Delta}_{\mathrm{f,\,r}} = \boldsymbol{D}_{\mathrm{f}}\boldsymbol{V}_{\mathrm{r}} = \begin{pmatrix} 1 & 2 & 3 & 5 \\ 3 & 4 & 7 & 11 \\ 4 & 6 & 10 & 16 \\ 7 & 10 & 17 & 27 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \tag{2.11}$$

2.7.2 Application to Arithmetic Operations

Matrix-matrix Multiplication

Assuming dimensional compatibility, the multiplication of a column checksum-encoded matrix $\boldsymbol{A}_{\mathrm{c}}$ by a row checksum-encoded matrix $\boldsymbol{B}_{\mathrm{r}}$ will yield a full checksum-encoded matrix $\boldsymbol{C}_{\mathrm{f}}$. As an example, consider the matrix-matrix multiplication in Equation 2.12.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 4 & 6 \\ 7 & 10 \end{pmatrix} \begin{pmatrix} 5 & 6 & 11 & 17 \\ 7 & 8 & 15 & 23 \end{pmatrix} = \begin{pmatrix} 19 & 22 & 41 & 63 \\ 43 & 50 & 93 & 143 \\ 62 & 72 & 134 & 206 \\ 105 & 122 & 227 & 349 \end{pmatrix} \tag{2.12}$$

Matrix Addition

The addition of two encoded matrices of identical type will produce a result of the same form. For example, consider the matrix addition shown in Equation 2.13.

$$\begin{pmatrix} 1 & 2 & 3 & 5 \\ 3 & 4 & 7 & 11 \\ 4 & 6 & 10 & 16 \\ 7 & 10 & 17 & 27 \end{pmatrix} + \begin{pmatrix} 5 & 6 & 11 & 17 \\ 7 & 8 & 15 & 23 \\ 12 & 14 & 26 & 40 \\ 19 & 22 & 41 & 63 \end{pmatrix} = \begin{pmatrix} 6 & 8 & 14 & 22 \\ 10 & 12 & 22 & 34 \\ 16 & 20 & 36 & 56 \\ 26 & 32 & 58 & 90 \end{pmatrix} \tag{2.13}$$

Matrix-vector Multiplication

The multiplication of a column checksum-encoded matrix $\boldsymbol{A}_{\mathrm{c}}$ by a column vector $\boldsymbol{b}$ will produce a checksum-encoded column vector $\boldsymbol{c}_{\mathrm{c}}$. Note that the multiplication of a row vector by a row checksum-encoded matrix will produce a checksum-encoded row vector. For example, consider the matrix-vector multiplication shown in Equation 2.14.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 4 & 6 \\ 7 & 10 \end{pmatrix} \begin{pmatrix} 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 17 \\ 39 \\ 56 \\ 95 \end{pmatrix} \tag{2.14}$$

Matrix-scalar Multiplication

The scalar multiplication of any type of encoded matrix will yield a result of identical form. Consider Equation 2.15 as an example.

$$5\begin{pmatrix} 1 & 2 & 3 & 5 \\ 3 & 4 & 7 & 11 \\ 4 & 6 & 10 & 16 \\ 7 & 10 & 17 & 27 \end{pmatrix} = \begin{pmatrix} 5 & 10 & 15 & 25 \\ 15 & 20 & 35 & 55 \\ 20 & 30 & 50 & 80 \\ 35 & 50 & 85 & 135 \end{pmatrix} \tag{2.15}$$

LU Decomposition

Any matrix $\boldsymbol{A}$ decomposable into lower- and upper-triangular (LU) matrices $\boldsymbol{L}$ and $\boldsymbol{U}$ can be decomposed into checksum-encoded matrices $\boldsymbol{L}_{\mathrm{c}}$ (with column-wise checksums) and $\boldsymbol{U}_{\mathrm{r}}$ (row-wise) with identical information content if $\boldsymbol{A}$ is first transformed into its full checksum-encoded equivalent $\boldsymbol{A}_{\mathrm{f}}$. As an example, consider the LU decomposition shown in Equation 2.16.

$$\begin{pmatrix} 4 & 5 & 9 & 14 \\ 8 & 28 & 36 & 64 \\ 12 & 33 & 45 & 78 \\ 20 & 61 & 81 & 142 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 2 & 3 \\ 3 & 3 \\ 5 & 6 \end{pmatrix} \begin{pmatrix} 4 & 5 & 9 & 14 \\ 0 & 6 & 6 & 12 \end{pmatrix} \tag{2.16}$$

Transposition

Checksums of all types are preserved through the transposition of a matrix. A full checksum-encoded matrix will remain as such, while a row or column checksum-encoded matrix will become the opposite type. For example, consider the transposition shown in Equation 2.17.

$$\begin{pmatrix} 1 & 2 & 3 & 5 \\ 3 & 4 & 7 & 11 \\ 4 & 6 & 10 & 16 \\ 7 & 10 & 17 & 27 \end{pmatrix}^{\mathrm{T}} = \begin{pmatrix} 1 & 3 & 4 & 7 \\ 2 & 4 & 6 & 10 \\ 3 & 7 & 10 & 17 \\ 5 & 11 & 16 & 27 \end{pmatrix} \tag{2.17}$$

Linear Filtering

The result shown in Section 2.7.2 is of particular significance since any one-dimensional linear filter—finite impulse response (FIR), infinite impulse response (IIR), discrete Fourier transform (DFT) (including fast Fourier transform (FFT)), etc.—can be represented in state-space form as a matrix-vector multiplication [54].

2.7.3 Result Classification

Once operations have been performed upon checksum-encoded matrices, the positions of incorrectly computed elements allow the classification of each result into one of a number of categories. Consider the outcomes for operations performed that result in the generation of distance-2 full checksum-encoded 2×2 matrices shown in Equation 2.18.

$$\begin{pmatrix} \checkmark & \checkmark & \checkmark \\ \checkmark & \checkmark & \checkmark \\ \checkmark & \checkmark & \checkmark \end{pmatrix}, \begin{pmatrix} \checkmark & \checkmark & \checkmark \\ \checkmark & \checkmark & \checkmark \\ \times & \checkmark & \checkmark \end{pmatrix}, \begin{pmatrix} \times & \checkmark & \checkmark \\ \checkmark & \checkmark & \checkmark \\ \checkmark & \checkmark & \checkmark \end{pmatrix}, \begin{pmatrix} \times & \checkmark & \times \\ \checkmark & \checkmark & \checkmark \\ \times & \checkmark & \times \end{pmatrix} \tag{2.18}$$

Assuming that one or more faults occurred along the datapath that generated each result, they respectively represent examples of *masked*, *false positive*, *false negative* and *detected* faults. Note that the same classification holds for all other checksumming types and distances discussed in Section 2.7.1.

Masked faults are those that have no observable external effect. The addition of two even integers with an adder whose least-significant output bit is stuck at zero represents a trivial example for such a result. False positives are those that affect only checksum elements, leaving information unaffected. Faults that manifest within circuitry used solely for checksumming, not within the associated dathapath, are responsible for their occurrence. While undesirable since they will force unnecessary corrective action, such results are safe since their information elements remain valid. False negatives, on the other hand, are unsafe since they represent results in which incorrectly computed information elements are undetectable due to checksums being valid. Faults occurring along datapath logic are always responsible for false negative result occurrence. In the case of a single distance-2 checksum, a single information element of value one higher than expected, along with equal checksum discrepancies, represents a simple numerical example of a false negative result. Faults are considered to be successfully detected when at least one information

element, and at least one checksum, are invalid.

2.8 Conclusion

This chapter presented an overview of the current state of the art in relation to fault tolerance, with discussion focussing upon techniques applicable to, and developed for, FPGAs. Distinctions between various types of faults and the errors they cause was presented first, with discussion of degradation following. Methods for fault detection—both offline and online—were presented, with fault mitigation and correction—methods both involving and avoiding re-placement and -routing at runtime—discussed afterwards. The final section focussed upon ABFT; an online fault tolerance technique tailored to particular algorithms. The mathematical prerequesites for ABFT were given, followed by examples of its application to various linear algebraic operators. Finally, a method of classifying results obtained from ABFT-protected operations was presented.

As a result of this background research, ABFT was identified as a promising area for further exploration, with the high-level aim of keeping overheads—in terms of area, performance and power—low while maintaining sensitivity, and the ability to react, to faults high. To validate this decision, Table 2.1 places ABFT within a side-by-side comparison of competing families of fault detection strategies, comparing their respective abilities to detect certain types and proportions of potential faults. ABFT can be seen to compare favourably to its competitors by striking a balance between these factors that the others cannot: mixing high fault coverage with low detection latency while keeping application overheads—both in terms of area and latency—relatively low at the cost of being tailored to a limited number of specific algorithms.

Method	Fault type(s) targetted	Fault coverage	Detection latency	Application overheads		Other limitations
				Area	Latency	
BIST	Permanent	Moderate–high	High	None	None	Requires downtime
Roving scans	Permanent	High	High	Low–moderate	Low	
Modular redundancy	Both	High	Low	High	Low	
Re-execution	Transient	Low–moderate	Moderate	Moderate	High	
Concurrency	Both	Low	Low	Low	Low	Datapath-unsuitable
Cycle-stealing	Permanent	Moderate–high	Low–moderate	Moderate–high	Low	
ABFT	Both	High	Low–moderate	Low–moderate	Low	Algorithm-specific

Table 2.1: Comparison of fault detection methods, showing ABFT to compare favourably to its competitors by striking a balance between factors that the others cannot: mixing high fault coverage with low detection latency while keeping application overheads—both in terms of area and latency—relatively low at the cost of being tailored to a limited number of specific algorithms.

3 Algorithm-tailored Low-overhead Online Error Detection

3.1 Introduction

In order to assess the impacts of applying algorithm-based fault tolerance (ABFT) to a benchmark accelerator, an application circuit to which ABFT could be applied was required. Matrix multiplication was chosen as the case study for the work described in this thesis since the operator is used in many hardware-accelerated applications and because the adaptation of its operation for ABFT is straightforward. This chapter details the design, implementation and evaluation of an ABFT-protected matrix multiplication accelerator running in hardware on an field-programmable gate array (FPGA).

The findings herein include that, for the largest-implemented accelerator tested, datapath fault detectability in excess of 99% was achieved in return for area and performance overheads of 7.87% and 45.5%, respectively, demonstrating that the achievement of high fault observability does not necessitate the incursion of, in particular, huge area overheads.

Note that the focus of this chapter is intended to be upon design and implementation. With regards to observability testing, in paticular, analysis is relatively brief and focusses upon permanent faults only; a far more detailed analysis of this operator's robustness, including against transient faults, is presented in Chapter 5.

3.1.1 Contributions

The original contributions of the work presented in this chapter are:

- The implementation of a complete hardware-software platform for the verification of ABFT protection of a benchmark matrix multiplication accelerator.
- A quantitative analysis of the overheads—in terms of area and performance—incurred through the incorporation of ABFT protection within that benchmark

circuit.

- Insight into the hardware fault tolerance of ABFT upon that benchmark.

3.1.2 Publications

The work presented in this chapter has been peer-reviewed and appeared in the 2013 proceedings of the International Conference on Field-programmable Technology (FPT) [60].

3.1.3 Outline

The remainder of this chapter is organised as follows. Section 3.2 details the development of, primarily, hardware needed to evaluate the fault-hardening of an application circuit. Section 3.2.1 describes the platform, while Section 3.2.2 focusses upon the design and operation of the chosen matrix multiplication benchmark. Section 3.2.3 explains the steps taken to modify the benchmark to achieve fault tolerance, with Sections 3.2.4 and 3.2.5 giving implementational details concerning fault location inference and fault injection, respectively. Section 3.3 deals with the encountered overheads for the fault-tolerant design over its unprotected equivalent, with Section 3.3.1 focussing upon area and Section 3.3.2 upon performance. An analysis of the hardened circuit's fault observability is presented in Section 3.4, and concluding remarks are given in Section 3.5.

3.2 Implementation

A platform was required upon which custom hardware could be implemented and tested quickly. Particularly for the testability requirement, as well as for the potential of realising hybrid hardware-software solutions to fault tolerance in the future due to the availability of hard central processing unit (CPU) cores, a relatively new system-on-chip (SoC) was used: a member of the Xilinx Zynq [61] family. The baseline or reference accelerator was first to be designed, with hardware operation confirmed using one of the available CPUs for oversight. Following this, ABFT logic was added to the baseline design, and the CPU was further used for targetted error injection to exercise the error detection circuitry and ensure accurate fault location.

3.2.1 Hardware-software Platform

Figure 3.1 gives a high-level overview of the developed platform. All boxed components shown are contained within a Xilinx Zynq-7000 XC7Z020 SoC [62].

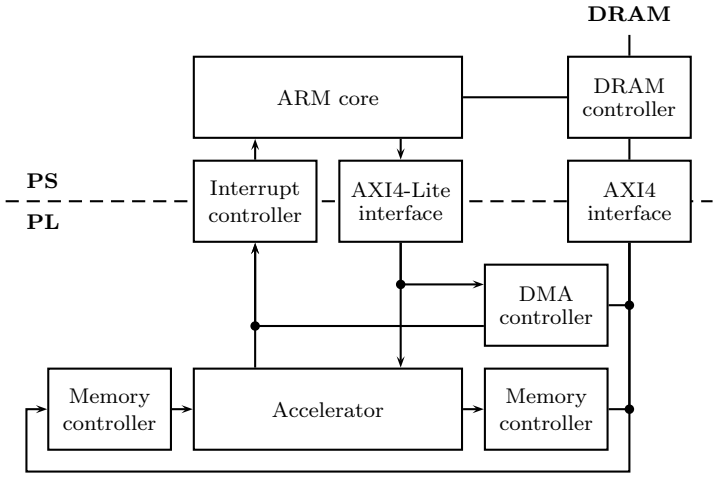


Figure 3.1: Top-level system block diagram. All boxed components shown are contained within a Xilinx Zynq-7000 XC7Z020 SoC [62], with a custom-designed matrix multiplication accelerator wrapped by several Xilinx IP blocks on the PL side of the device.

The Zynq SoC is split into two distinct halves: the *processor subsystem (PS)*, housing a pair of hard Acorn reduced instruction set computer machine (ARM) Cortex-A9 CPU cores, and *programmable logic (PL)*: a modestly sized (53,200-look-up tables (LUTs)) FPGA. Many available hard peripherals on the device, such as memory and high-speed input/output (I/O) controllers, are multiplexed so as to be accessible to either the ARM cores or custom logic, while others are tied directly to the PS or PL. High-speed configurable interconnect is provided for facilitating PS-PL communication.

Throughout the hardware development and testing described in this thesis, a single CPU core was used in 'bare-metal' (without operating system) fashion, primarily as a controller for the FPGA-implemented logic but also for test vector generation, result verification and latency measurement. The hard dynamic random-access memory (DRAM) controller on the PS side was configured in order to allow fast shared memory access by both the ARM core and soft logic.

Several Xilinx IP blocks—a direct memory access (DMA) controller [63], two memory controllers [64] and interfacing logic to service data transfers [65] and interrupts [66] across the PS-PL boundary—sit to control and feed data into and out of a custom-designed

matrix multiplication accelerator, described in Section 3.2.2. Software-accessible control registers within the accelerator are connected to the CPU with a low-bandwidth Advanced eXtensible Interface (AXI)-Lite bus, while data transfers are achieved via a high-bandwidth AXI bus operated in burst mode. Interrupts are triggered by status register changes within the accelerator.

3.2.2 Baseline Architecture

The architecture developed for accelerating matrix multiplication is shown in Figure 3.2. It, along with all other hardware developed as part of the work presented in this thesis, was written in platform-independent very high-speed integrated circuit hardware description language (VHDL). The following parameters are customisable:

- Square matrix size (s).
- Data width (n) (bits).
- Data memory resource type.
- Multiplier resource type.
- Multiplier latency (m) (cycles).
- Accumulator latency (a) (cycles).

Throughout the hardware development conducted as described in this thesis, matrices are always square with dimensions $s \times s$, however this is not a requirement imposed by the fault tolerance methods presented.

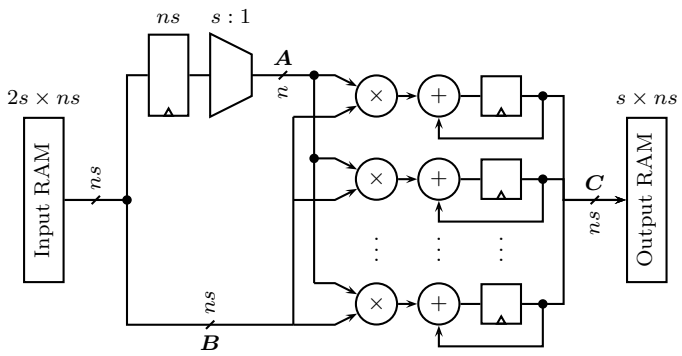


Figure 3.2: Baseline datapath for matrix multiplication. A full matrix row's contingent of MACs are used to create an efficient inner loop-unrolled architecture.

Multiplier pipelining is achieved automatically via retiming of register chains instantiated on the multipliers' outputs by the computer-aided design (CAD) tools used for compilation; registers are 'pushed backwards' through each multiplier to balance combinatorial logic latency between stages, increasing timing model-inferred maximum operating frequency ($f_{\max}$), without affecting cycle latency. Accumulator pipelining must be implemented more explicitly since feedback is required. To allow accumulator pipelining, the architecture shown in Figure 3.3 was developed, allowing latency to be increased from 1 to a and maximum adder widths to be reduced from n to $\lceil n/a \rceil$. Adder widths are chosen to be as close to optimal (i.e. n/a) as possible, with wider adders used first, if necessary. From all but the final stage, an overflow (bit $\lceil n/a \rceil$) signal is tapped off the adder to feed into the subsequent stage. A shift register—not shown in Figure 3.3—bubbles reset signals from input to output, allowing consecutive accumulations to occur with only a single cycle used for reset, regardless of the value of a. The finite state machine (FSM) controlling the datapath—not shown in Figure 3.2—adjusts itself to accommodate for changes in m and a.

Before the accelerator runs, input data is transferred, triggered by a command from the CPU, from DRAM to FPGA fabric random-access memory (RAM). FPGA RAM words represent full matrix rows (ns bits each) to increase data parallelism: for the largest-implemented matrix size ($s = 32$), with $n = 32$ as mentioned previously, 1024-bit words were used. A single RAM was used on the input side rather than two (one per input matrix) since the increased memory transfer times were found to outweigh potential speedup. The input RAM therefore consists of $2s$ ns-bit words, while the output RAM has s ns-bit words. To allow connection to the memory controllers for PS access, the RAMs are asymmetric: regardless of n or s, memory access ports presented to the external memory controllers are always 64-bit to match the maximum width of the high-performance AXI interconnect available on the target device [67]. To achieve this, address decoding drives either byte write enables (for the write port of the dual-ported input RAM) or read data multiplexing (for the read port of the dual-ported output RAM).

With reference to Figure 3.2 and the accompanying pseudocode, Algorithm 1, computation proceeds as follows following an input data transfer. Once the first row of matrix $\boldsymbol{A}$ is fetched (Line 2) and buffered into a register (Line 3) and the multiply-accumulators (MACs) are reset (Line 4), each row of matrix $\boldsymbol{B}$ is fetched from RAM in turn (Line 7) and presented to a full row's contingent of MACs (Lines 8 to 11), along with

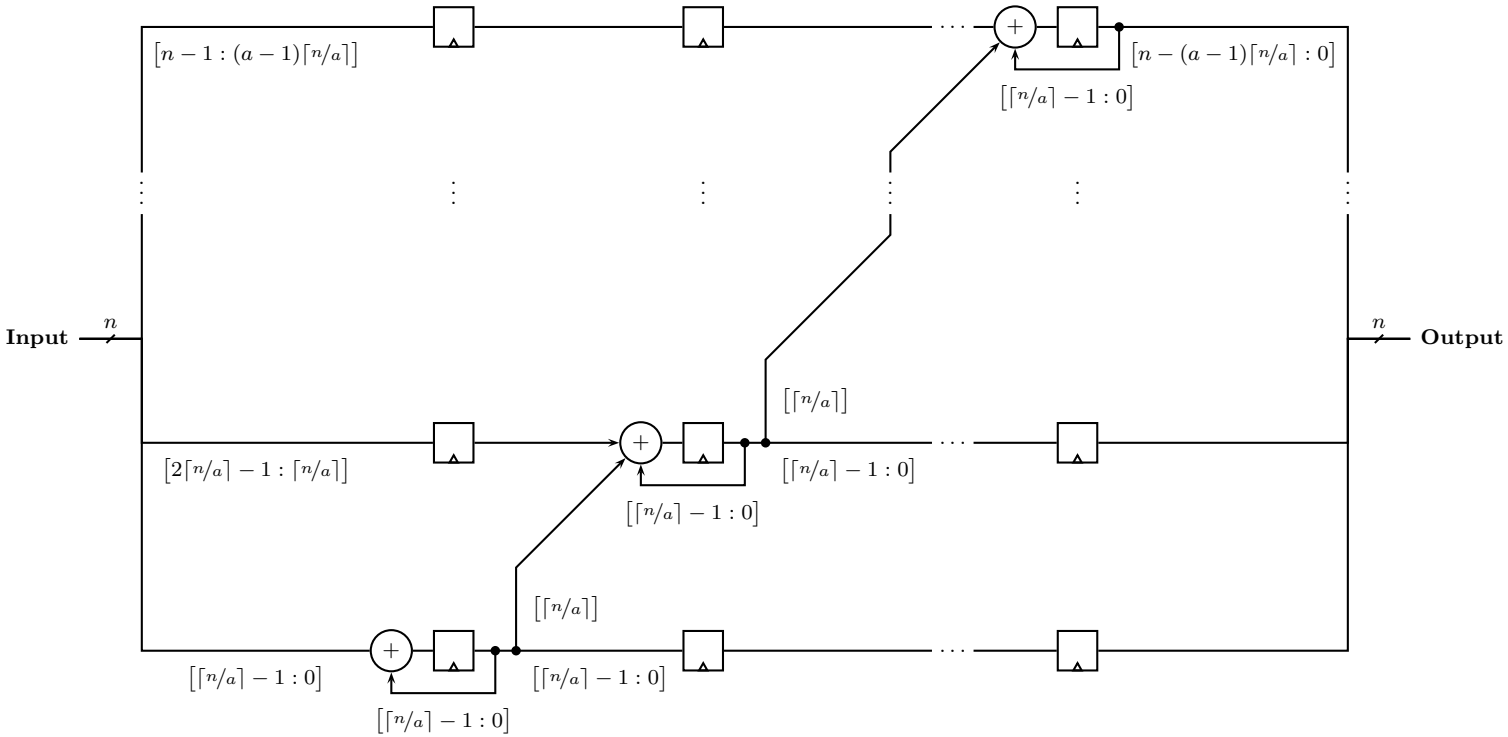


Figure 3.3: Pipelined accumulator, in which a adders each up to $\lceil n/a \rceil$ bits wide are used to achieve pipelining at the expense of increased latency.

the corresponding element of $\boldsymbol{A}$ (Line 6), for computation. Once the final row of $\boldsymbol{B}$ has been consumed, the computation of $\boldsymbol{C}$'s first row is complete: it is stored into the output RAM (Line 13) and the process is repeated for the remaining rows of $\boldsymbol{A}$ and $\boldsymbol{C}$. Once a multiplication has been completed in its entirety, an interrupt occurs, triggering a second DMA transfer to copy data back from the FPGA's RAM to DRAM.

Algorithm 1 Baseline matrix multiplication

1: **for** $i = 0$ **to** $s - 1$ **do**
2: **fetch** $\boldsymbol{A}[i]$
3: **buffer** $\boldsymbol{A}[i]$
4: $\boldsymbol{C}[i] \leftarrow \begin{pmatrix} 0 & 0 & \cdots & 0 \end{pmatrix}$
5: **for** $j = 0$ **to** $s - 1$ **do**
6: **select** $\boldsymbol{A}[i][j]$
7: **fetch** $\boldsymbol{B}[j]$
8: $\boldsymbol{C}[i][0] \leftarrow (\boldsymbol{C}[i][0] + \boldsymbol{A}[i][j] \times \boldsymbol{B}[j][0])$ **mod** 2^n
9: $\boldsymbol{C}[i][1] \leftarrow (\boldsymbol{C}[i][1] + \boldsymbol{A}[i][j] \times \boldsymbol{B}[j][1])$ **mod** 2^n
10: $\cdots$
11: $\boldsymbol{C}[i][s-1] \leftarrow (\boldsymbol{C}[i][s-1] + \boldsymbol{A}[i][j] \times \boldsymbol{B}[j][s-1])$ **mod** 2^n
12: **end for**
13: **store** $\boldsymbol{C}[i]$
14: **end for**

3.2.3 Checksum Generation & Verification

To support error detection, several additions and modifications were made to the accelerator presented in Section 3.2.2; these are shown in Figures 3.4, 3.5 and 3.6 and described algorithmically in Algorithms 2, 3 and 4. Aside from lengthening accelerator latency, the changes made have no effect upon normal operation: data transferred between DRAM and FPGA RAM, and vice-versa, is identical in both quantity and form to that moved previously.

The checksum generation logic shown sandwiched between the input RAM and MACs in Figure 3.4 is detailed in Figure 3.5. The input buffer and multiplexer (MUX) serve the same purpose as those shown in Figure 3.2 and described in Section 3.2.2, while the adder, register, RAMs and two additional MUXes form the logic responsible for the transformation of $\boldsymbol{A}$ and $\boldsymbol{B}$ into checksum-encoded matrices $\boldsymbol{A}_{\mathrm{c}}$ and $\boldsymbol{B}_{\mathrm{r}}$. Within the datapath itself, an extra MAC is added to mirror the expansion of the matrices being multiplied: what was an $s \times s$ multiplication is now effectively $(s+1) \times (s+1)$. On the output side, a buffering register is added to hold rows of $\boldsymbol{C}_{\mathrm{f}}$ as they are computed; this is shown in Figure 3.4. The checksum verification logic shown in Figure 3.6, consisting

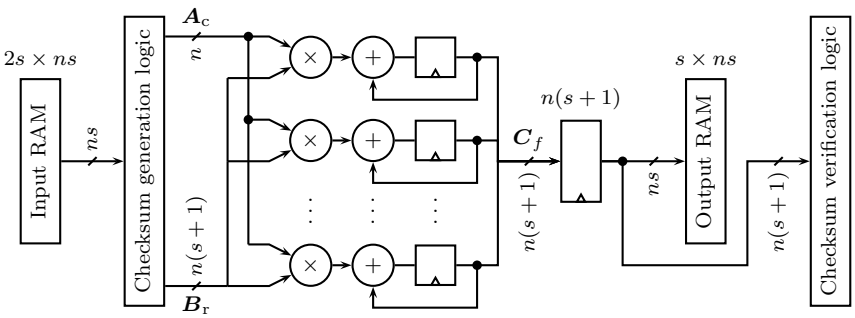


Figure 3.4: ABFT-protected matrix multiplication datapath. Aside from lengthening accelerator latency, the changes made have no effect upon normal operation: data transferred between DRAM and FPGA RAM, and vice-versa, is identical in both quantity and form to that moved previously. An extra MAC is added to mirror the expansion of the matrices being multiplied: what was an $s \times s$ multiplication is now effectively $(s+1) \times (s+1)$.

of a MUX, two adders, a register, RAM and two comparators, are added following the buffer. The results generated by this logic are used to inform the accelerator's controller of any checksum discrepancies that occur during computation. The resource used for the implementation of small RAMs needed for checksum generation and verification is also parameterisable.

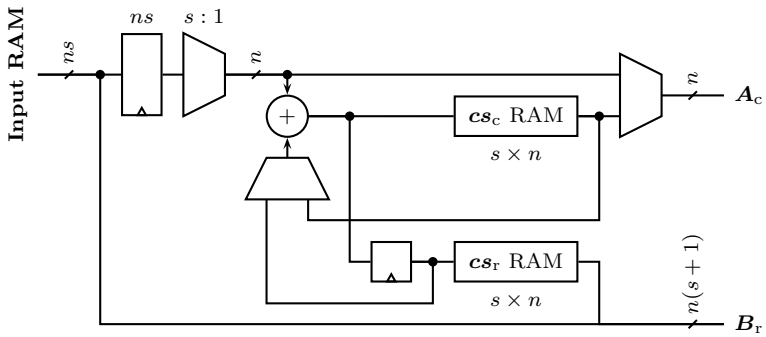


Figure 3.5: Checksum generation logic for matrix multiplication accelerator, responsible for the transformation of $\boldsymbol{A}$ and $\boldsymbol{B}$ into checksum-encoded matrices $\boldsymbol{A}_\mathrm{c}$ and $\boldsymbol{B}_\mathrm{r}$.

The operations performed during each multiplication are largely unchanged following the addition of this ABFT logic: results are computed row-by-row, as before, with input checksums generated as input data is accessed and output checksums verified as output data is computed. The exception to this is that, due to the need to access complete rows of $\boldsymbol{B}_\mathrm{r}$ on a one-per-cycle basis, its checksums are computed before multiplication begins.

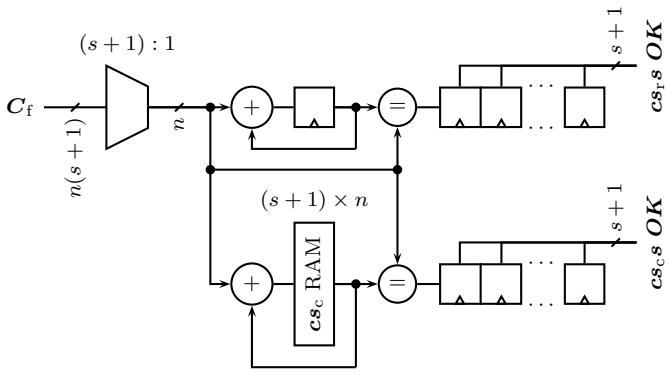


Figure 3.6: Checksum verification logic for matrix multiplication accelerator, responsible for checking checksums contained within $\boldsymbol{C}_{\mathrm{f}}$. The results generated by this logic are used to inform the accelerator's controller of any checksum discrepancies that occur during computation.

Algorithm 2 details the operation of Figure 3.5's logic during this precomputation. Rows of $\boldsymbol{B}$ are fetched from RAM (Line 2) and buffered in turn, with their elements selected sequentially by a MUX (Line 6) in order to be accumulated (Line 7). As each row's checksum is computed, it is stored (Line 9) in the row-wise checksum vector ($\boldsymbol{cs}_{\mathrm{r}}$) RAM. While some performance penalty results from the decision to have row checksums precomputed, the area overhead saving is significant since an s-input adder would otherwise be required to complete those computations at speed.

Algorithm 2 ABFT-protected matrix multiplication $\boldsymbol{B}_{\mathrm{r}}[0 \cdots s-1][s]$ precomputation

1: **for** $i = 0$ **to** $s-1$ **do**
2: **fetch** $\boldsymbol{B}[i]$ **as** $\boldsymbol{B}_{\mathrm{r}}[i][0 \cdots s-1]$
3: **buffer** $\boldsymbol{B}_{\mathrm{r}}[i][0 \cdots s-1]$
4: $\boldsymbol{B}_{\mathrm{r}}[i][s] \leftarrow 0$
5: **for** $j = 0$ **to** $s-1$ **do**
6: **select** $\boldsymbol{B}_{\mathrm{r}}[i][j]$
7: $\boldsymbol{B}_{\mathrm{r}}[i][s] \leftarrow (\boldsymbol{B}_{\mathrm{r}}[i][s] + \boldsymbol{B}_{\mathrm{r}}[i][j])$ **mod** 2^n
8: **end for**
9: **store** $\boldsymbol{B}_{\mathrm{r}}[i][s]$
10: **end for**

Only one adder is needed in the checksum generation logic since its function switches to $\boldsymbol{A}_{\mathrm{c}}$ column checksum generation once $\boldsymbol{B}_{\mathrm{r}}$'s row checksums have been generated. Algorithm 3 details the operation of both the checksum generation logic and main datapath, shown in Figure 3.4, following $\boldsymbol{B}_{\mathrm{r}}$ precomputation. Here, the MAC component of the algorithm (Lines 15 to 18) covers the 'additional' $\boldsymbol{B}_{\mathrm{r}}$ and $\boldsymbol{C}_{\mathrm{f}}$ column, s, not present in either $\boldsymbol{B}$ nor $\boldsymbol{C}$. As rows of $\boldsymbol{A}[i]$ are fetched from RAM (Line 3) then buffered and their

elements selected (Line 9), checksums are computed using Figure 3.5's adder and column-wise checksum vector ($\boldsymbol{cs}_\mathrm{c}$) RAM as an accumulator (Lines 20 to 26). A RAM is required here rather than a register since the particular column's checksum being calculated changes every cycle. Rows of $\boldsymbol{B}_\mathrm{r}$ must be sourced from both the input RAM (Line 13) and $\boldsymbol{cs}_\mathrm{r}$ RAM (Line 14) simultaneously. The final row of $\boldsymbol{A}_\mathrm{c}$ is treated differently (Line 11) since it originates from the $\boldsymbol{cs}_\mathrm{c}$ RAM rather than the input RAM. Sections of $\boldsymbol{C}_\mathrm{f}$ that form part of $\boldsymbol{C}$, i.e. all but the final row and column, are stored in the output RAM (Line 31) after buffering.

Algorithm 3 ABFT-protected matrix multiplication main computation

1: **for** $i = 0$ **to** s **do**
2: **if** $i < s$ **then**
3: **fetch** $\boldsymbol{A}[i]$ **as** $\boldsymbol{A}_\mathrm{c}[i]$
4: **buffer** $\boldsymbol{A}_\mathrm{c}[i]$
5: **end if**
6: $\boldsymbol{C}_\mathrm{f}[i] \leftarrow \begin{pmatrix} 0 & 0 & \cdots & 0 \end{pmatrix}$
7: **for** $j = 0$ **to** $s - 1$ **do**
8: **if** $i < s$ **then**
9: **select** $\boldsymbol{A}_\mathrm{c}[i][j]$
10: **else**
11: **fetch** $\boldsymbol{A}_\mathrm{c}[s][j]$
12: **end if**
13: **fetch** $\boldsymbol{B}[j]$ **as** $\boldsymbol{B}_\mathrm{r}[j][0 \ \cdots \ s-1]$
14: **fetch** $\boldsymbol{B}_\mathrm{r}[j][s]$
15: $\boldsymbol{C}_\mathrm{f}[i][0] \leftarrow (\boldsymbol{C}_\mathrm{f}[i][0] + \boldsymbol{A}_\mathrm{c}[i][j] \times \boldsymbol{B}_\mathrm{r}[j][0])$ **mod** 2^n
16: $\boldsymbol{C}_\mathrm{f}[i][1] \leftarrow (\boldsymbol{C}_\mathrm{f}[i][1] + \boldsymbol{A}_\mathrm{c}[i][j] \times \boldsymbol{B}_\mathrm{r}[j][1])$ **mod** 2^n
17: $\cdots$
18: $\boldsymbol{C}_\mathrm{f}[i][s] \leftarrow (\boldsymbol{C}_\mathrm{f}[i][s] + \boldsymbol{A}_\mathrm{c}[i][j] \times \boldsymbol{B}_\mathrm{r}[j][s])$ **mod** 2^n
19: **if** $i < s$ **then**
20: **if** $i = 0$ **then**
21: $\boldsymbol{A}_\mathrm{c}[s][j] \leftarrow 0$
22: **else**
23: **fetch** $\boldsymbol{A}_\mathrm{c}[s][j]$
24: **end if**
25: $\boldsymbol{A}_\mathrm{c}[s][j] \leftarrow (\boldsymbol{A}_\mathrm{c}[s][j] + \boldsymbol{A}_\mathrm{c}[i][j])$ **mod** 2^n
26: **store** $\boldsymbol{A}_\mathrm{c}[s][j]$
27: **end if**
28: **end for**
29: **buffer** $\boldsymbol{C}_\mathrm{f}[i]$
30: **if** $i < s$ **then**
31: **store** $\boldsymbol{C}_\mathrm{f}[i][0 \ \cdots \ s-1]$ **as** $\boldsymbol{C}[i]$
32: **end if**
33: **end for**

Checksum verification occurs in parallel with $\boldsymbol{A}_\mathrm{c}$ checksum generation and normal com-

putation. With reference to Figure 3.6 and Algorithm 4, verification is completed as follows. As rows of $\boldsymbol{C}_\mathrm{f}$ become available (Line 2), their elements are selected in turn (Line 5) and accumulated in both row- (Line 7) and column-wise (Line 17) fashions simultaneously. Once the final column (for row-wise checksums, Line 9) or row (for column-wise checksums, Line 21) of $\boldsymbol{C}_\mathrm{f}$ is reached, the appropriate checksum is compared with its corresponding element. These results are stored in software-accessible registers for analysis by the accelerator's driver. Note that the verification of $\boldsymbol{C}_\mathrm{f}$'s column-wise checksums requires an accumulator constructed using a RAM, rather than a register, since the column index changes once per cycle.

Algorithm 4 ABFT-protected matrix multiplication $\boldsymbol{C}_\mathrm{f}$ verification

```
1:  for i = 0 to s do
2:     wait until C_f[i] available
3:     cs_r[i] ← 0
4:     for j = 0 to s do
5:        select C_f[i][j]
6:        if j < s then
7:           cs_r[i] ← (cs_r[i] + C_f[i][j]) mod 2^n
8:        else
9:           cs_rs OK[i] ← cs_r[i] = C_f[i][s]
10:       end if
11:       if i < s then
12:          if i = 0 then
13:             cs_c[j] ← 0
14:          else
15:             fetch cs_c[j]
16:          end if
17:          cs_c[j] ← (cs_c[j] + C_f[i][j]) mod 2^n
18:          store cs_c[j]
19:       else
20:          fetch cs_c[j]
21:          cs_cs OK[j] ← cs_c[j] = C_f[s][j]
22:       end if
23:    end for
24: end for
```

3.2.4 Fault Location

Consider the simple ABFT-protected matrix multiplication shown in Equation 3.1. Had the multiplication resulted in, for example, one of the alternative outputs shown in Equation 3.2 instead, the positioning of incorrect checksum values would have revealed location information regarding the MACs that caused the errors. Each of these three cases is syn-

onymous with a single MAC register's least-significant bit (LSB) experiencing a stuck-at-one (SA1) fault. Elements that have been calculated incorrectly are shown in **bold**, while *italics* mark error-indicating checksum values. Note that column checksum mismatches relate one-to-one with faulty MACs, since each MAC is responsible for computing exactly one output column's elements. Simultaneous faults occurring both within an individual MAC and across multiple MACs would yield equally informative results: a single column checksum mismatch in the former case and multiple in the latter.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 4 & 6 \end{pmatrix} \begin{pmatrix} 5 & 6 & 11 \\ 7 & 8 & 15 \end{pmatrix} = \begin{pmatrix} 19 & 22 & 41 \\ 43 & 50 & 93 \\ 62 & 72 & 134 \end{pmatrix} \tag{3.1}$$

$$\begin{pmatrix} \mathbf{21} & 22 & \mathit{41} \\ \mathbf{45} & 50 & \mathit{93} \\ \mathbf{\mathit{63}} & 72 & \mathit{134} \end{pmatrix}, \begin{pmatrix} 19 & \mathbf{23} & \mathit{41} \\ 43 & \mathbf{51} & \mathit{93} \\ 62 & \mathbf{\mathit{73}} & \mathit{134} \end{pmatrix}, \begin{pmatrix} 19 & 22 & \mathbf{\mathit{43}} \\ 43 & 50 & \mathbf{\mathit{95}} \\ 62 & 72 & \mathbf{\mathit{135}} \end{pmatrix} \tag{3.2}$$

3.2.5 Error Injection

Rather than opting to directly cause faults, whether by purposefully upsetting configuration bitstreams or otherwise, it was instead chosen to emulate datapath faults by causing data errors at MAC outputs. This is accomplished by issuing error injection instructions on the controlling CPU, which cause one or more specified bits of one or more particular MAC outputs to be flipped using an array of exclusive-OR (XOR) gates in hardware. While not particularly representative of any real-world fault type, this simple scheme was chosen for hardware testing since it allows different errors to be injected tens of thousands of times per second without causing confounding, or masking, issues to occur which would skew performance results by allowing errors to go undetected.

It should be emphasised that, as is also the case for the remaining hardware and software (fault observability) testing described later in this thesis, errors emulated at MAC outputs are not indicative of faults occurring merely at those locations. Occurrences of faults at any point along the datapath, whether in logic or routing, are liable to produce errors at MAC outputs since those are the points through which all data must travel prior to storage.

3.3 Overheads

Experiments were performed to assess the impacts of adding ABFT protection to the baseline accelerator in terms of area and performance. All designs were compiled using version 14.7 of Xilinx's Integrated Synthesis Environment (ISE) toolchain. The following range of implementation variables was used:

- Target device: Xilinx Zynq-7000 XC7Z020.
- s: {2, 4, 8, 16, 32}.
- n (bits): 32 (signed, fixed-point).
- Data memory resource type: block random-access memory (BRAM).
- Checksum memory resource type: distributed RAM.
- Multiplier resource type: digital signal processing block (DSP).
- m (cycles): 15.
- a (cycles): 1.

Since triplets of DSPs—each optimally pipelined when they absorb five register stages [68]—were required to implement each two-input 32-bit multiplier, 15-stage multipliers were used consistently. Achieved $f_{\max}$ were found to be highest when accumulators were not pipelined, allowing DSP block absorption, so single-stage accumulators were used throughout.

3.3.1 Area

Table 3.1 contains the raw resource usage figures obtained for all implementations. Percentages of the total number of each of these resources for the target device are also included, along with means of the individual proportions—each calculated as the mean (μ) of LUT (%), flip-flop (FF) (%), BRAM (%) and DSP (%)—to give an indication of the overall resource utilisation. Figure 3.7 presents a visual summary of the combined resource usage data. Trendlines, shown dashed, have been included to counter the effects of CAD tool noise.

It can be seen from Table 3.1 that BRAM and DSP overheads are fixed while register and LUT overheads increase with s. Fixed BRAM overheads are due to the requirement for three small (s, s and $s + 1$ 32-bit words), separate dual-port RAMs for checksum

Matrix size s	ABFT enabled	Resource type LUT	FF	BRAM	DSP	Total
2	✗	239 (0.449%)	210 (0.197%)	2 (0.714%)	6 (2.73%)	1.02%
	✓	618 (1.16%)	476 (0.447%)	5 (1.79%)	9 (4.09%)	1.69%
4	✗	441 (0.829%)	406 (0.382%)	8 (2.86%)	12 (5.45%)	2.38%
	✓	757 (1.42%)	749 (0.704%)	11 (3.93%)	15 (6.82%)	3.22%
8	✗	604 (1.14%)	794 (0.746%)	16 (5.71%)	24 (10.9%)	4.63%
	✓	877 (1.65%)	1286 (1.21%)	19 (6.79%)	27 (12.3%)	5.49%
16	✗	613 (1.15%)	1566 (1.47%)	30 (10.7%)	48 (21.8%)	8.79%
	✓	1159 (2.18%)	2352 (2.21%)	33 (11.8%)	51 (23.2%)	9.85%
32	✗	2115 (3.98%)	3105 (2.92%)	58 (20.7%)	96 (43.6%)	17.8%
	✓	3072 (5.77%)	4472 (4.20%)	61 (21.8%)	99 (45.0%)	19.2%

Table 3.1: Baseline & ABFT-protected accelerator resource usage, containing the raw resource usage figures obtained for all implementations. Percentages of the total number of each of these resources for the target device are also included, along with means of those proportions to give an indication of the overall resource utilisation.

storage, while the same number of extra DSPs is required for any s due to the need for a single additional MAC in all cases. While additional register and LUT requirements both increase with s, proportionately they decrease.

The impact upon area incurred through the introduction of checksum generation and verification circuitry clearly shows it to be one of the most attractive properties of ABFT. For the largest-implemented design, capable of multiplying pairs of 32×32 matrices with 32-bit data elements, the overall area overhead was just 7.87%. Of this, the majority of the overhead lies in the least-used resources, LUTs and FFs: more minimal overheads of 5.17% and 3.13% were encountered for scarcer BRAMs and DSPs resources, respectively. Applications involving linear algebraic operations efficiently implemented on modern FPGAs tend to be DSP-heavy, so the minimal DSP overhead is particularly beneficial.

3.3.2 Performance

Impacts in terms of performance are presented in Table 3.2. 10,000 tests were performed for each value of s, with operating frequency kept at 50MHz throughout, and results

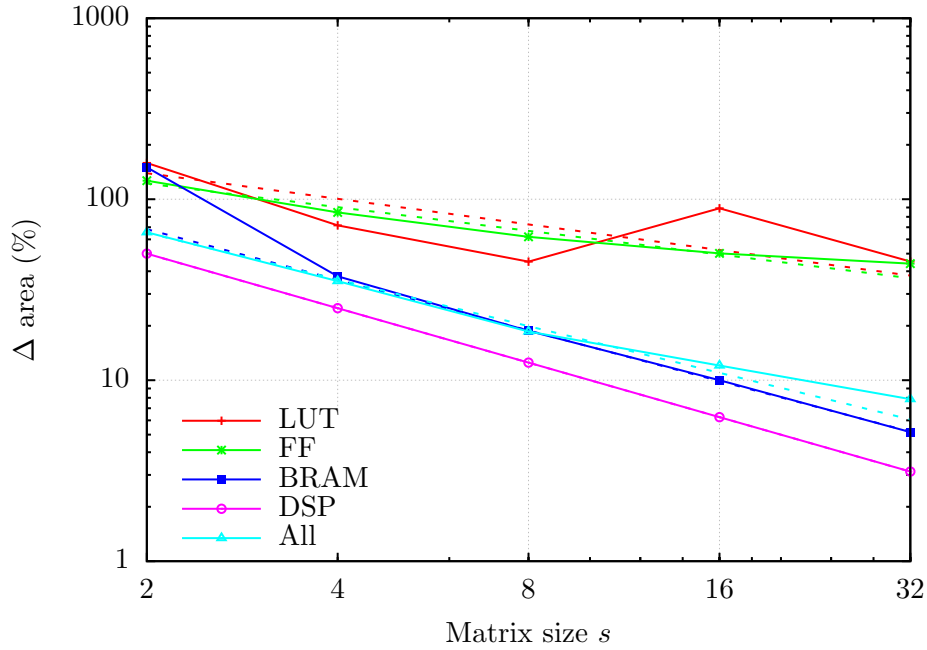


Figure 3.7: ABFT-protected accelerator resource usage overhead versus baseline, showing the changes in resource utilisation for each design versus its equivalently sized unprotected, baseline implementation.

were averaged across all tests completed. Figure 3.8 presents a visual summary of the encountered latency overheads.

Slowdown over the baseline design is caused by the requirement to precompute row checksums before multiplication itself takes place, while the column checksum generation and checksum validation have no impact upon performance since they operate in parallel with the MACs. The trend-reversal seen after $s = 16$ can be attributed to data transfer throttling: once s passes 16, memory copies begin to dominate accelerator execution for proportional runtime. Changes in $f_{\max}$ are largely unmeaningful and can likely be attributed to the stochastic nature of the CAD tools used for compilation.

The latency overheads encountered were, while not huge, also not insignificant. For the largest-implemented design, with $s = 32$ and $n = 32$, a latency penalty of 45.5% was measured. A clear tradeoff exists across the spectrum of hardware fault tolerance techniques: for example, while triple modular redundancy (TMR) necessitates very large ($> 200\%$) area overhead in comparison to ABFT's small figures ($< 10\%$ for reasonably sized matrices), its application forces virtually no latency penalty.

Matrix size s	ABFT enabled	Execution time (μs)	$f_{\max}$ (MHz)
2	✗	254	90.2
	✓	272	77.9
4	✗	314	80.1
	✓	339	81.7
8	✗	348	76.9
	✓	546	77.1
16	✗	497	50.5
	✓	1350	60.7
32	✗	3100	58.2
	✓	4510	65.2

Table 3.2: Baseline & ABFT-protected accelerator performance. Averaged execution times and maximum operating frequencies achieved are shown for each design to allow side-by-side comparison of unprotected and protected implementations.

3.4 Fault Observability

In order to assess the fault observability of the chosen detection method, fault injection simulations were performed to ascertain the hardware's ability to correctly detect and locate faults. Detectable faults are those that result in one or more checksum mismatches—in one or more rows, columns or both—while those that are locatable cause mismatches within the columns corresponding to the MACs they have affected.

The fault model chosen was that of individually targetted SA1 accumulator output bits. Such faults were chosen since they are representative of a range of phenomena, e.g. worn transistors or bridged interconnects. In terms of location, accumulator outputs were chosen since these components lie at the ends of the datapaths of interest, affording them maximal opportunity to impact the results. Note that the choice of fault model was different to that used during hardware testing as explained in Section 3.2.5; while that used for hardware testing used MAC output bit inversion in order to avoid masking, software simulation uses stuck-ats since such avoidance was not desirable.

Results gleaned from this testing were, as is the case in the remainder of the fault observability testing performed in this thesis, independent of fault rate, area and latency: they demonstrate the *proportions of total accelerator executions* that should be expected to result in particular classes of outputs under *fixed fault conditions*. They cannot be used to directly ascertain the expected rates of certain output classes' occurrence, although this can be achieved by scaling the proportions to take fault rate, area and/or latency, as required, into account.

Simulations were performed in software rather than in hardware since hundreds of mil-

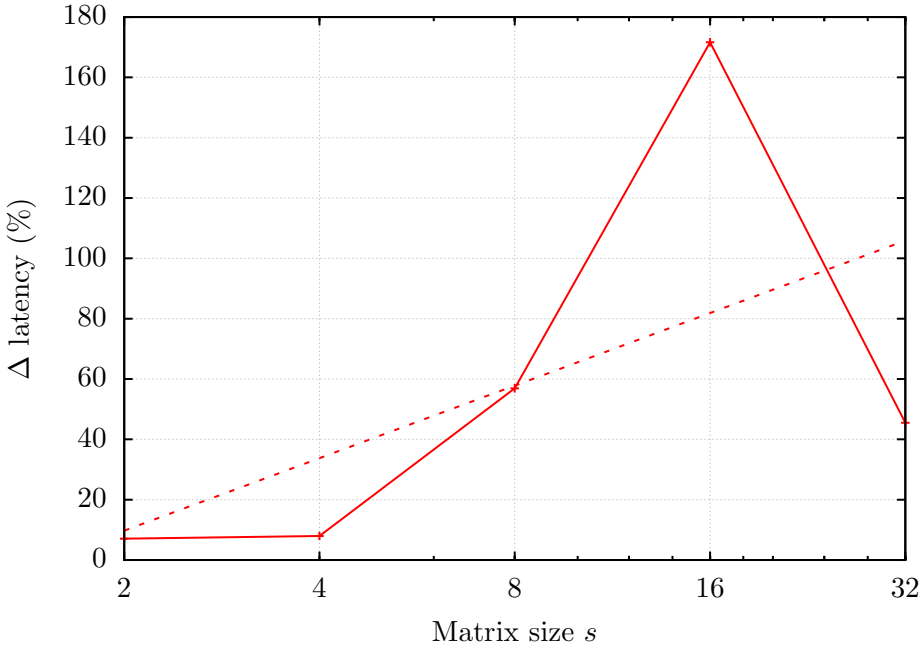


Figure 3.8: ABFT-protected accelerator latency overhead versus baseline, showing the change in execution time for each design versus its equivalently sized unprotected, baseline implementation.

lions of tests could be run within a reasonable time period across a wide range of implementations with relatively little up-front effort (versus hardware design) and minimal intervention required at runtime. Since the faults emulated affect the operation being performed at an algorithmic level, and that the fault tolerance being verified also acts at that level, fault injection experimentation in hardware was considered to be unnecessary and software simulation sufficient.

Algorithm 5 details the steps executed for each simulation. Arrays for $s \times s$ matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ are provisioned first (Lines 1 and 2), with the elements of each filled with random n-bit signed values selected from a uniform distribution (Lines 3 and 4). Matrices $\boldsymbol{A}_{\mathrm{c}}$ and $\boldsymbol{B}_{\mathrm{r}}$ are created (Lines 5 and 6), with $\boldsymbol{A}$ and $\boldsymbol{B}$ encoded to form them as explained in Section 2.7.1 (Lines 7 and 8) and the output array, representing matrix $\boldsymbol{C}_{\mathrm{f}}$, is provisioned (Line 9). An $(s+1)$-element bit mask array is created (Line 10) to represent the faults present during the simulation. Across the array's $s+1$ n-bit elements, number of simultaneous faults (f) bits, at random locations, are non-zero. Matrix multiplication to generate $\boldsymbol{C}_{\mathrm{f}}$'s values—performed modulo-2^n to represent overflow—proceeds as normal but, both before (Line 13) and during (Line 15) each multiply-accumulate step, the corresponding

column's bit mask is 'or'ed with the (intermediate) result to emulate either one or two SA1 MAC register output bits. A result was recorded as detected (Line 19) if one or more of the checksums present within $\boldsymbol{C}_{\mathrm{f}}$ contained mismatches, and as located (Line 20) if, additionally, those mismatches corresponded to the columns chosen for fault injection.

Algorithm 5 Fault injection simulation

1: **create** $s \times s$ matrix $\boldsymbol{A}$
2: **create** $s \times s$ matrix $\boldsymbol{B}$
3: $\boldsymbol{A}$.rand_fill()
4: $\boldsymbol{B}$.rand_fill()
5: **create** $(s+1) \times s$ matrix $\boldsymbol{A}_{\mathrm{c}}$
6: **create** $s \times (s+1)$ matrix $\boldsymbol{B}_{\mathrm{r}}$
7: $\boldsymbol{A}_{\mathrm{c}} \leftarrow \boldsymbol{A}$.add_cs('col')
8: $\boldsymbol{B}_{\mathrm{r}} \leftarrow \boldsymbol{B}$.add_cs('row')
9: **create** $(s+1) \times (s+1)$ matrix $\boldsymbol{C}_{\mathrm{f}}$
10: $bit\_mask,\ faulty\_cols \leftarrow$ generate_bit_mask($s,\ n,\ f$)
11: **for** $i = 0$ **to** s **do**
12: **for** $j = 0$ **to** s **do**
13: $\boldsymbol{C}_{\mathrm{f}}[i][j] \leftarrow bit\_mask[j]$
14: **for** $k = 0$ **to** $s - 1$ **do**
15: $\boldsymbol{C}_{\mathrm{f}}[i][j] \leftarrow \big((\boldsymbol{C}_{\mathrm{f}}[i][j] + \boldsymbol{A}_{\mathrm{c}}[i][k] \times \boldsymbol{B}_{\mathrm{r}}[k][j])$ **mod** $2^n\big)$ **bitwise or** $bit\_mask[j]$
16: **end for**
17: **end for**
18: **end for**
19: $detected \leftarrow$ **not** $\boldsymbol{C}_{\mathrm{f}}$.check_cs()
20: $located \leftarrow detected$ **and** $\boldsymbol{C}_{\mathrm{f}}$.diagnose_cs() $= faulty\_cols$

Algorithm 6 details the procedure used to generate the bit mask called as `generate_bit_mask()` in Algorithm 5. An array for the bit mask (Line 1) and another to represent faulty columns (Line 2) are provisioned, with target column and bit pairs randomly selected (Lines 7 and 8) and checked for uniqueness (Line 9) before being set high (Line 10). The column affected is also flagged (Line 11).

The simulation framework was written in Python and threaded [69] to allow efficient parallel execution on a 64-core Advanced Micro Devices (AMD) Opteron [70]-based server. The test steps described were completed 1,024,000 times for each combination of the following variables:

- s: {2, 4, 8, 16, 32}.
- n (bits): {2, 4, 8, 16, 32} (signed, fixed-point).
- Fault type: permanent.
- f: {1, 2}.

Algorithm 6 `generate_bit_mask()` procedure used in fault injection simulation

Require: $s,\ n,\ f$

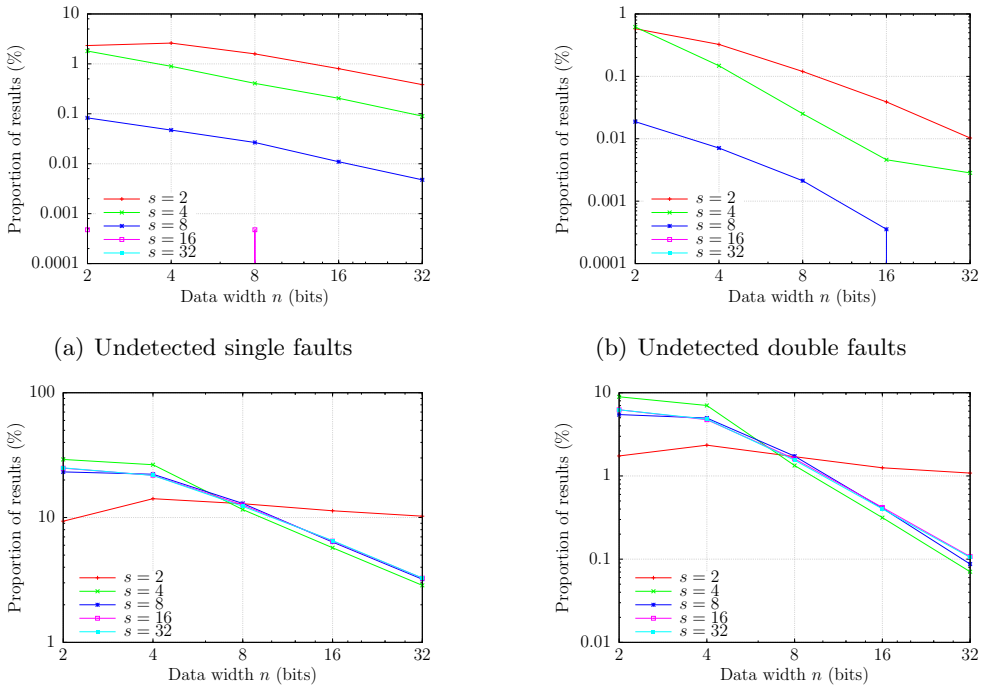
1: **create** $s+1$ vector bit_mask
2: **create** $s+1$ vector $faulty\_cols$
3: bit_mask.zero_fill()
4: $faulty\_cols$.zero_fill()
5: **for** $i = 0$ **to** $f-1$ **do**
6: **repeat**
7: $col = \text{rand}(0$ **to** $s)$
8: $bit = \text{rand}(0$ **to** $n-1)$
9: **until not** $bit\_mask[col]$ **bitwise and** 2^{bit}
10: $bit\_mask[col] \leftarrow bit\_mask[col]$ **bitwise or** 2^{bit}
11: $faulty\_cols[col] \leftarrow$ **true**
12: **end for**
13: **return** $bit\_mask, faulty\_cols$

This testing represented a total of several days of computional effort. Proportions of detected and located faults for each value of s, n and f were averaged across all simulations performed. The results are presented in Figure 3.9.

For single fault injection, in all cases except for $s = 2,\ n = 2$, the proportion of undetected faults dropped off with both s and n. For $s \geq 16$, undetectable fault proportions fell below 0.1% for all data widths and, for $s = 32$, undetectable faults ceased to be encountered. As expected, proportions of unlocatable faults were higher than those that were undetectable due to the lack of redundancy in checksums used for location. In all cases except for $s = 2,\ n = 2$, however, the proportion of unlocated faults observed dropped with increasing data width for each value of s. For larger s, the locatability of faults is largely independent of s itself.

Similar trends were seen for double fault injection testing. The rates of both undetected and unlocated faults were all lower, however, for each combination of s and n. This is expected of undetected faults since the likelihood of errors being masked in multiple columns simultaneously decreases as the number of affected columns increases. The proportions of unlocatable double faults encountered were again significantly higher than those which were undetectable but, for all cases except for $s = 2,\ n = 2$, dropped off with increasing data width for all s.

It should be noted that the results shown in Figure 3.9 do not take area into account; that is, designs are subjected to single (or double) faults regardless of their physical size. While time was not explicitly considered in the fault observability testing, it is nevertheless accurate to say that designs of different area experience different fault rates under oth-



(a) Undetected single faults

(b) Undetected double faults

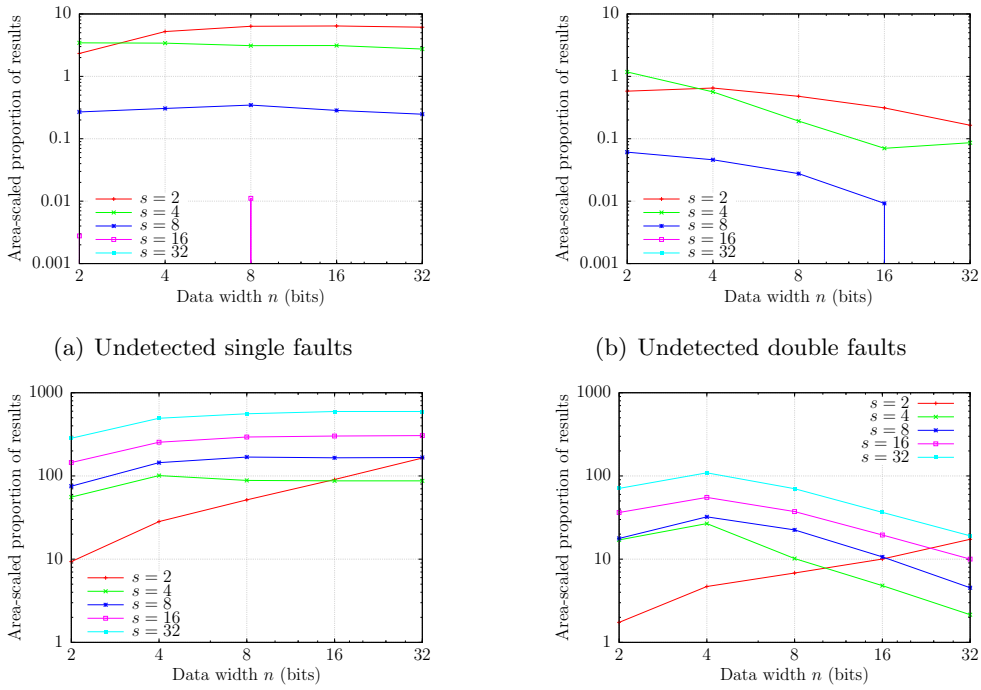
(c) Unlocated single faults

(d) Unlocated double faults

Figure 3.9: Fault proportions for ABFT-protected matrix multiplication. Each figure shows the proportion of matrix multiplication results, for a particular s and n, for which a particular outcome was observed.

erwise identical conditions. For this reason, additional plots, shown in Figure 3.10, were produced to attempt to capture the effect of area upon likelihood of fault manifestation, thereby scaling the previously seen fault proportions. Both s and n have an impact upon area. For s, total ABFT-enabled resource usage figures from Table 3.1 were used to scale the results, with $s = 2$ taken as the baseline. For example, an otherwise-equivalent accelerator with $s = 4$ consumes $^{3.22}/_{1.69} = 1.91\times$ the area of that with $s = 2$, and is therefore considered likely to experience $1.91\times$ the fault rate. For n, linear scaling was used, with $n = 2$ taken as the baseline, for further scaling. The latter tends to penalise designs with larger n since a proportion of logic, particularly that for the FSM, is independent of n, however this was considered to be a minor concern.

The results presented in Figure 3.10 tend to show similar but flatter curves than those in Figure 3.9. Area-scaled detectability results are promising, with proportions falling as s rose in value under both single and double fault injection. Unfortunately, area-scaled locatability was found to decrease as s increased, however the effect upon locatability of increases in n were found to be either minimal (for single fault testing), with all curves



(a) Undetected single faults

(b) Undetected double faults

(c) Unlocated single faults

(d) Unlocated double faults

Figure 3.10: Fault proportions for ABFT-protected matrix multiplication scaled by area, with total resource usage figures used to scale for s and linear scaling used for n.

flattening off around $n = 8$ apart from that for $s = 2$, or positive (for double fault testing).

3.5 Conclusion

This chapter detailed the design, implementation and evaluation of an ABFT-protected matrix multiplication accelerator running in hardware on a hybrid CPU-FPGA SoC platform. Specifics of the hardware-software platform developed were presented first, followed by the baseline accelerator and modifications made to harden the design against faults. Architectural details governing the inference of fault locations and the method of error injection were also given. Results were presented with an emphasis on implementational overhead in terms of area, frequency and latency, and a fault injection simulation method was described and used to evaluate the fault observability of the developed system.

The results showed significant promise for ABFT's application in hardware: for the physically largest-implemented ($s = 32$, $n = 32$) accelerator tested, the area overhead incurred was 7.87% averaged across all resources, comparing favourably with competing

fault tolerance techniques such as TMR. For the same s and n, the latency penalty incurred was 45.5%, and fault injection simulation suggested single datapath fault locatability of 96.7% with detectability well in excess of 99%.

The work presented in this chapter represents a solid foundation upon which further enhancements could, and can still, be made. In particular, the ABFT implementation for error detection and fault location, as well as the simulation framework for fault injection, completed represent significant output upon which the work presented in Chapters 4, 5 and 6 is based.

4 Error Correction via Runtime Resource Reallocation

4.1 Introduction

In this chapter, the foundational work presented in Chapter 3, which focussed upon error detection and fault location inference, is built upon to allow those errors to be corrected using two distinct strategies. A combination of 'bolt-on' algorithm-based fault tolerance (ABFT) error detection logic and resource reallocation serve to provide low-overhead datapath fault tolerance at runtime. Initially, the latter is achieved through the use of additional logic which, guided by information gleaned from ABFT, reduces algorithmic parallelisation at runtime in order to maintain accurate operation. Field-programmable gate arrays (FPGAs) are uniquely placed to allow further area savings to be made when incorporating error correction mechanisms thanks to their dynamic reconfigurability; dynamic partial reconfiguration (DPR) is therefore called upon for the purpose of error correction as well. For ease of comparison, the benchmark, platform and design tools used for the work described in this chapter are identical to those in Chapter 3.

The results in this chapter demonstrate that rapid yet accurate fault diagnoses along with low hardware (area), performance (latency) and, where necessary, software (memory storage) penalties are achievable through algorithm-tailored error correction strategies. The results shown in this chapter include that, for the largest-implemented circuit able to detect, diagnose and correct errors within its datapath, area overheads of 12.4% (with additional logic for error correction) or 10.1% (with DPR) are achievable in return for latency penalties as low as 24.5% under fault-free operation.

4.1.1 Contributions

The original contributions of the work presented in this chapter are:

- The first implementation of custom logic for error correction in the presence of faulty resources guided by an ABFT error detection mechanism.
- The first implementation of ABFT-protected hardware using DPR for recovery.
- A quantitative analysis of the overheads—of resources, performance and memory—incurred through the incorporation of those error correction strategies into a benchmark hardware accelerator.

4.1.2 Publications

The work presented in this chapter has been peer-reviewed and appeared in the 2014 proceedings of the IEEE International Symposium on Field-programmable Custom Computing Machines (FCCM) [71] and International Conference on Field-programmable Logic and Applications (FPL) [72].

4.1.3 Outline

The remainder of this chapter is organised as follows. Section 4.2 details the development and functionality of two alternative hardware error correction strategies, with Section 4.2.1 considering the use of additional logic to achieve this aim and Section 4.2.2 making use of DPR to achieve the same goal. Overheads of the two methods are analysed in Section 4.3, with area considered in Section 4.3.1 and performance in Section 4.3.2. For DPR-facilitated error correction, an additional overhead—memory utilisation—is studied. Concluding comments are given in Section 4.4.

4.2 Implementation

4.2.1 Additional Logic

Once one or more errors have been detected in the accelerator described in Chapter 3, and the resources causing those errors identified, the matrix multiplication can be rerun in a modified fashion such that the faulty resources are bypassed. Initially, a data-shifting strategy was employed: by effectively reducing the level of parallelism, i.e. not making use of all previously available multiply-accumulators (MACs), and dynamically reallocating data to the remaining resources, correct computation can be achieved at the expense of elongated computation time. To achieve this, the datapath shown in Figure 3.4 was

modified to that shown in Figure 4.1: the two are identical save for the two circular shifters, shown in Figure 4.3, present in the latter.

Figure 4.2 demonstrates the operation of this data-shifting strategy to route around a single faulty MAC in an $s = 2$ matrix multiplier. In each case, $\boldsymbol{B}_{\mathrm{r}}$ input data—signified by boxed numbers which correspond to their original places—is captured in the circular shifter inserted onto the path for $\boldsymbol{B}_{\mathrm{r}}$ as shown in Figure 4.1. This additional logic is capable of rotating input data 'downwards' by x places in x clock cycles before it is fed to the MACs—signified by circled numbers—for processing. No modifications to the flow of $\boldsymbol{A}_{\mathrm{c}}$ input data are required since the same values of $\boldsymbol{A}_{\mathrm{c}}$ are presented to all MACs simultaneously. Post-computation, the second circular shifter, this one configured to rotate data 'upwards,' replaces the buffering register shown in Figure 3.4 that captures rows of $\boldsymbol{C}_{\mathrm{f}}$. In the presence of a single faulty MAC, a single-place data shift is all that is required to bypass it.

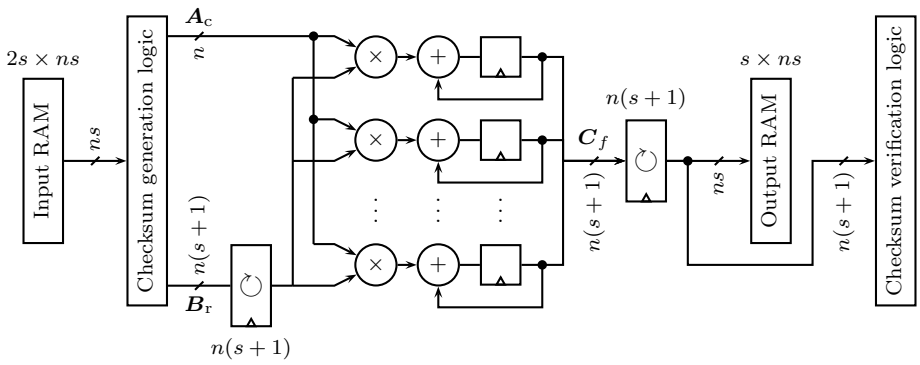


Figure 4.1: ABFT-enabled datapath with circular shifters, allowing the dynamic reallocation of data to resources.

Note that in all three cases the operation is the same: during the first execution, input data is rotated downwards by one place, computed, and output data rotated upwards by one place to correct for the input shift. During the second execution, no shifting is required, however only the result from the MAC directly above that found to be faulty is stored, overwriting the value outputted during the first execution. These steps remain the same for any value of s.

While the current control hardware is only able to work around single faulty MACs, the same shifting logic is capable of performing data rotations to prevent the use of any number of faulty MACs up to s, i.e. all-but-one unavailable. Again, only latency would be affected by such occurrences. Latency would scale proportionately to the number of

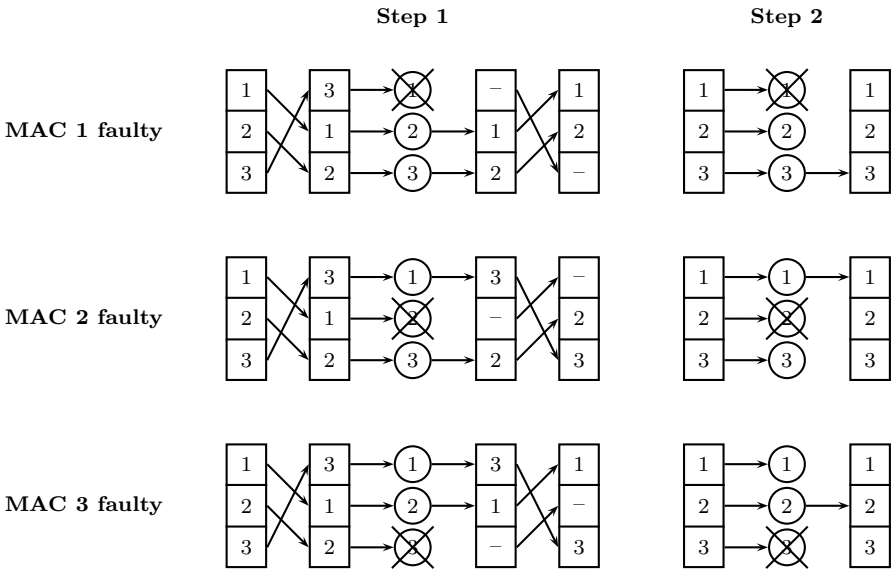


Figure 4.2: Resource reallocation for $s = 2$ with single fault using circular shifters. $\boldsymbol{B}_{\mathrm{r}}$ input data—signified by boxed numbers which correspond to their original places—is captured in the circular shifter inserted onto the path for $\boldsymbol{B}_{\mathrm{r}}$, rotated and fed to the MACs—signified by circled numbers—for processing. The mirrored equivalent occurs on the output side for $\boldsymbol{C}_{\mathrm{f}}$.

faults that need to be tolerated, since multiple faults will require multiple data shifts to be routed around.

It should be emphasised that the accelerator does not retain state between computations. For this reason, the hardware defaults to its normal, fault-unaware mode at the beginning of every computation: only if one or more errors are detected does it run in a fault-bypassing state. A more practical implementation might retain error counts for each MAC and, once one or more of those counts cross a predetermined threshold value, prevent any use of the associated MACs from that point onward, thus decreasing the average computation time.

4.2.2 Partial Routing Reconfiguration

While the modified accelerator described in Section 4.2.1 made use of additional logic to dynamically reallocate data to the MACs in order to bypass faults, in this section the means to achieve the same end result with partial routing reconfiguration are presented. During accelerator executions in which at least one error is detected, fault location data is sent back to the controlling software driver in order to facilitate corrective action. Based upon the locations of faults observed and, conversely, the locations of remaining functional

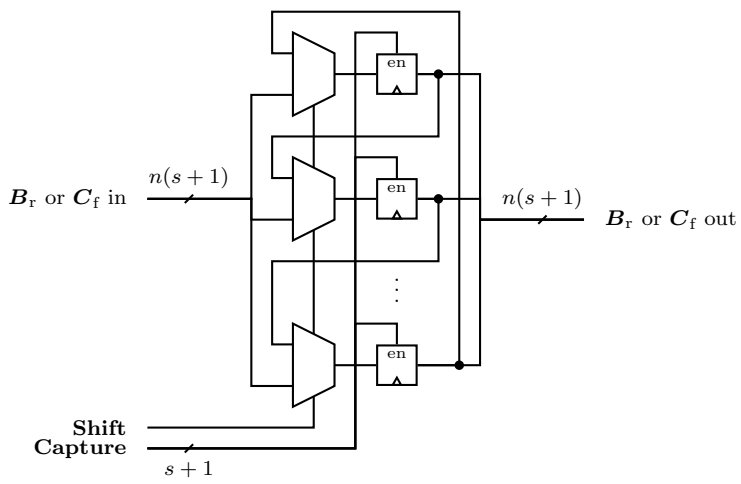


Figure 4.3: Circular shifter, capable of rotating data by x places in x clock cycles and used to allow dynamic resource reallocation.

MACs, one or more rounds of routing reconfiguration followed by accelerator executions, together called 'corrective executions,' can be performed in order to re-establish accurate operation.

Routing reconfiguration, rather than dynamic relocation of MACs themselves, was chosen for three reasons:

- The datapath (MACs) represents the vast majority—well over 90% in designs with higher s—of the area consumed by the accelerator.
- Routing bitstreams are small, as is quantified in Section 4.3.3, and so can be applied more quickly (and, consequently, frequently) than those for MAC reconfiguration could be.
- The reservation of regions of fabric (to accommodate replacement MACs) is rendered unnecessary, allowing full use of the available resources.

In order to facilitate routing reconfiguration via DPR, several modifications needed to be made to the system described in Section 3.2 and accompanying block diagram, Figure 3.1. These are shown in Figure 4.4. A region of the accelerator—represented by a dashed rectangle—is made reconfigurable. Reconfiguration is handled by one of the hard Acorn reduced instruction set computer machine (ARM) central processing unit (CPU) cores on the Zynq system-on-chip (SoC) through the processor configuration access port (PCAP) [73].

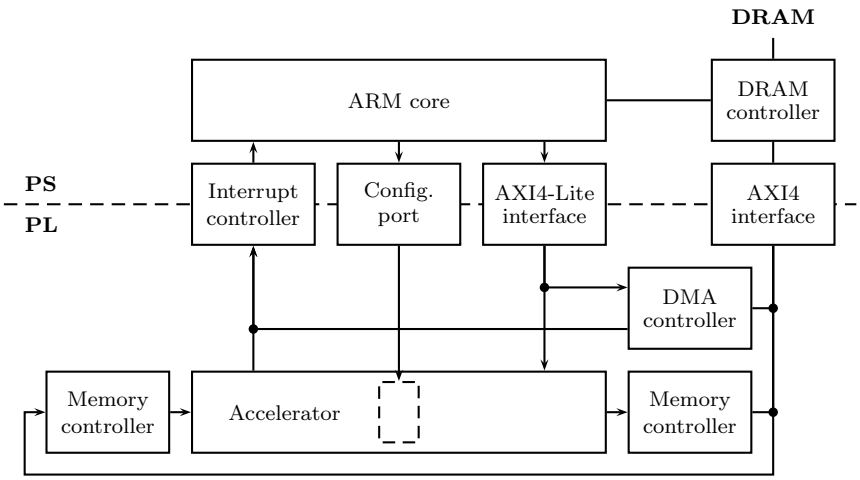


Figure 4.4: System block diagram with DPR. A region of the accelerator—represented by a dashed rectangle—is made reconfigurable. Reconfiguration is handled by one of the hard ARM CPU cores on the Zynq SoC through the PCAP [73].

Reconfiguration is considerably slower than modifying multiplexer (MUX) addressing—even for small, partial bitstreams—due to the need to set up and execute a memory transfer each time a new configuration is to be loaded. For this reason, the datapath shown in Figure 4.1 was modified further to, as shown in Figure 4.5, relocate the checksum verification logic such that it took its source from the output random-access memory (RAM) rather than the (buffered) MAC outputs. With this arrangement, partially faulty outputs need only be partially overwritten, thus saving reconfiguration cycles. Whereas previously an $s = 2$ accelerator required six changes in MUX addressing to route around a single fault—two changes per row computed—during an execution, only two reconfigurations are needed with this arrangement as the routing can stay the same while correcting all errors caused by that fault before being reset prior to the next computation.

The relocation of checksum verification logic also necessitated the expansion of the output RAM. Whereas until now the output RAM only needed to store the $s \times s$ output matrix $\boldsymbol{C}$, not the output checksum elements, the output RAM shown in Figure 4.5 needs to store the full $(s+1) \times (s+1)$ output matrix $\boldsymbol{C}_{\mathrm{f}}$. To achieve this with zero impact upon the software, which expects to receive only $\boldsymbol{C}$ as output, a hybrid RAM was designed. Within it, the first s rows and columns are stored in a dual-ported RAM, intended to be implemented in block random-access memory (BRAM) as before, whose read port is accessible from the processor subsystem (PS) via a memory controller. Two small RAMs, intended to be implemented in distributed RAM, were added to store the elements within

the $(s+1)^{\text{th}}$ row and column of the output matrix. Control logic was added such that, from the accelerator side, all three RAMs appeared to be a single, contiguous storage block for the entitiry of $\boldsymbol{C}_{\text{f}}$.

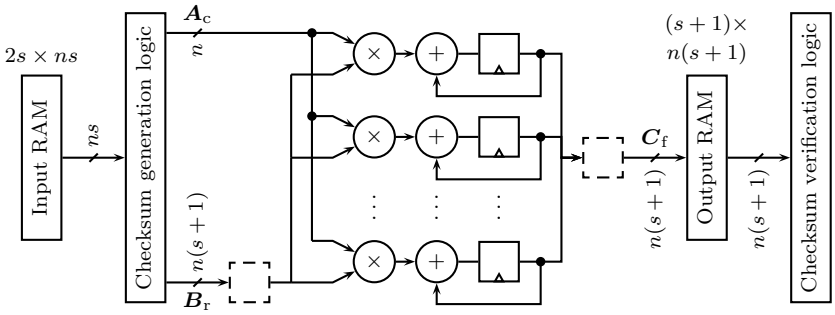


Figure 4.5: ABFT-enabled datapath with DPR. The input and output halves of the reconfigurable partition are shown as dashed rectangles. The movement of checksum verification logic from the input to output side of the output RAM is also shown, where the output RAM now stores the full $(s+1) \times (s+1)$ output matrix $\boldsymbol{C}_{\text{f}}$.

At compile-time, routing configurations representing different amounts of data-shifting are compiled along with the rest of the design, which remains static. Nets $\boldsymbol{B}_{\text{r}}$ and $\boldsymbol{C}_{\text{f}}$—shown in Figure 4.5—are broken and routed via a single reconfigurable partition, which then dictates the data connections on both the input and output sides of the accelerator's datapath. Figure 4.6 shows the configurations available for the multiplier when $s = 2$. Circled numbers represent MACs, while the input and output halves of the reconfigurable partition are shown as dashed rectangles. In each case, the output shifting arrangement mirrors that on the input side.

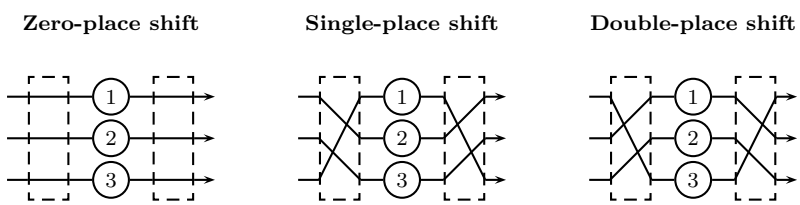


Figure 4.6: Routing configurations available for $s = 2$. Circled numbers represent MACs, while the input and output halves of the reconfigurable partition are shown as dashed rectangles. In each case, the output shifting arrangement mirrors that on the input side.

When routing reconfiguration is required, the accelerator's driver initiates a partial bitstream transfer, via the PCAP, from dynamic random-access memory (DRAM) to

the FPGA fabric. In order to lower the total number of configurations required, only configurations with equal-place shifting per MAC are generated at compile-time. The number of configurations stored for each accelerator is therefore $s + 1$.

The driver supports two levels of safety for error correction. When operating in the safer mode, all incorrectly computed columns of $\boldsymbol{C}_{\mathrm{f}}$ are recalculated, after which checksum verification is repeated to confirm successful correction. In the less safe mode, the ABFT mechanism is essentially turned off: the $(s + 1)^{\mathrm{th}}$ MAC becomes a usable spare and the output is assumed to be accurate once all corrective executions complete. As a consequence of this, faults that affect only the $(s + 1)^{\mathrm{th}}$ MAC are ignored in the less safe mode. The choice made between these modes as part of a larger application would be based upon the likelihood of additional faults developing in different MACs during the time it takes to complete a corrective execution. Note that re-transfer of input data and input checksum regeneration are not required in either mode.

Figure 4.7 demonstrates the application of routing reconfiguration in order to bypass a single faulty MAC—labelled 2—when $s = 2$. Intuitively, one corrective execution is required to overwrite the second column's elements. A single-place shift allows MAC 3 to perform the recalculation required.

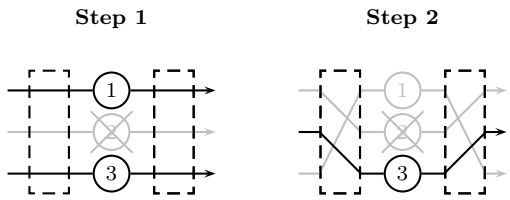


Figure 4.7: Resource reallocation for $s = 2$ with single fault using DPR, demonstrating the application of routing reconfiguration in order to bypass a single faulty MAC—labelled 2—when $s = 2$. A single-place shift allows MAC 3 to perform the recalculation required.

In cases of multiple faults, differing amounts of data-shifting are required. This is exemplified in Figure 4.8, in which six different combinations of double-fault locations are shown for $s = 4$. In the three leftmost cases, one single-place shift is required, while in the three rightmost cases, one double-place shift is required. Curved arrows represent the reallocation of resources necessary during a corrective execution.

Intuition may suggest that the number of corrective executions required is only dependant upon the ratio of faulty to functional MACs. When $s \in \{2, 4\}$, this is indeed true, but for $s \geq 8$ the situation is more complicated since there are cases in which a configuration

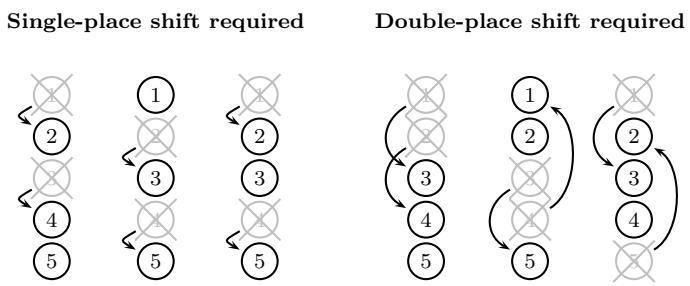


Figure 4.8: Example double-fault routing reconfigurations when $s = 4$. In the three leftmost cases, one single-place shift is required, while in the three rightmost cases, one double-place shift is required. Curved arrows represent the reallocation of resources necessary during a corrective execution.

with an equal-place shift per MAC can no longer match all faulty MACs to remaining functional ones. In Figure 4.9, where $s = 8$, six combinations of quadruple-fault locations are shown. In the three leftmost cases, only a single corrective execution is required; in the three rightmost cases, however, two are needed: resource reallocations which cannot be performed in the first execution are represented by dashed lines.

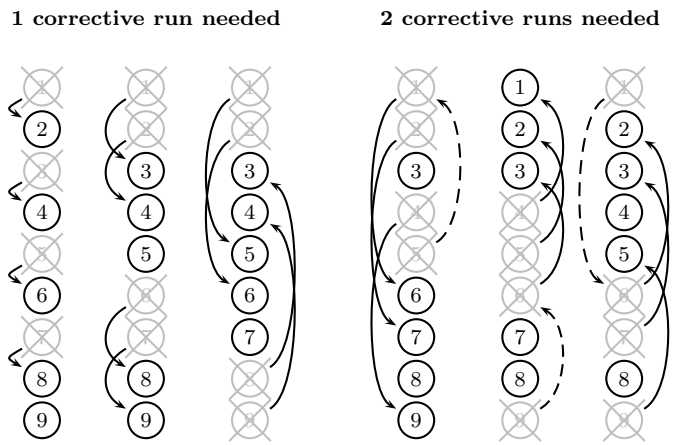


Figure 4.9: Example quadruple-fault routing reconfigurations when $s = 8$, demonstrating cases where the same number of faults require differing numbers of corrective executions to be bypassed.

4.3 Overheads

Experiments were performed to assess the impacts of adding ABFT-informed error correction, implemented both with additional logic and DPR, to the fault-intolerant, baseline accelerator in terms of area and performance. All designs were compiled using version

14.7 of Xilinx's Integrated Synthesis Environment (ISE) toolchain. The following range of implementation variables was used:

- Target device: Xilinx Zynq-7000 XC7Z020.
- s: {2, 4, 8, 16, 32}.
- Data width (n) (bits): 32 (signed, fixed-point).
- Data memory resource type: BRAM.
- Checksum memory resource type: distributed RAM.
- Multiplier resource type: digital signal processing block (DSP).
- Multiplier latency (m) (cycles): 15.
- Accumulator latency (a) (cycles): 1.

4.3.1 Area

Table 4.1 contains the raw resource usage figures obtained for all implementations—including the fault-intolerant, fault-tolerant via additional logic and fault-tolerant via DPR versions of the accelerator—per resource type. Percentages of the total number of each of these resources for the target device are also included, along with a mean of the individual proportions to give an indication of the overall resource utilisation. Figures 4.10 and 4.11 present a visual summary of the resource usage data, with the former showing overheads of individual resources and the latter overheads across all resource types.

With the exception of $s = 16$, it is clear from Figure 4.11 that the DPR-shifting accelerator performs better than its additional logic-shifting counterpart for overall resource usage across the range of s tested. Since BRAM and DSP usage are identical between the two versions, look-up table (LUT) and flip-flop (FF) counts are responsible for all differences in utilisation. As expected, FF overhead for the DPR design decreases proportionally as s increases thanks to the elimination of the circular shifters present in the additional logic-shifting version. Conversely, LUT overhead tends to increase slightly; this is due to LUTs being used to implement distributed RAMs for output checksum storage as described in Section 4.2.2. For the largest-tested design, $s = 32$, the DPR-shifting accelerator achieved an overall area overhead of 10.1%—17.7% lower than its additional logic-shifting equivalent. Between these two fault-tolerant designs, FF overhead decreased by 77.5% while LUT overhead increased by 7.7%.

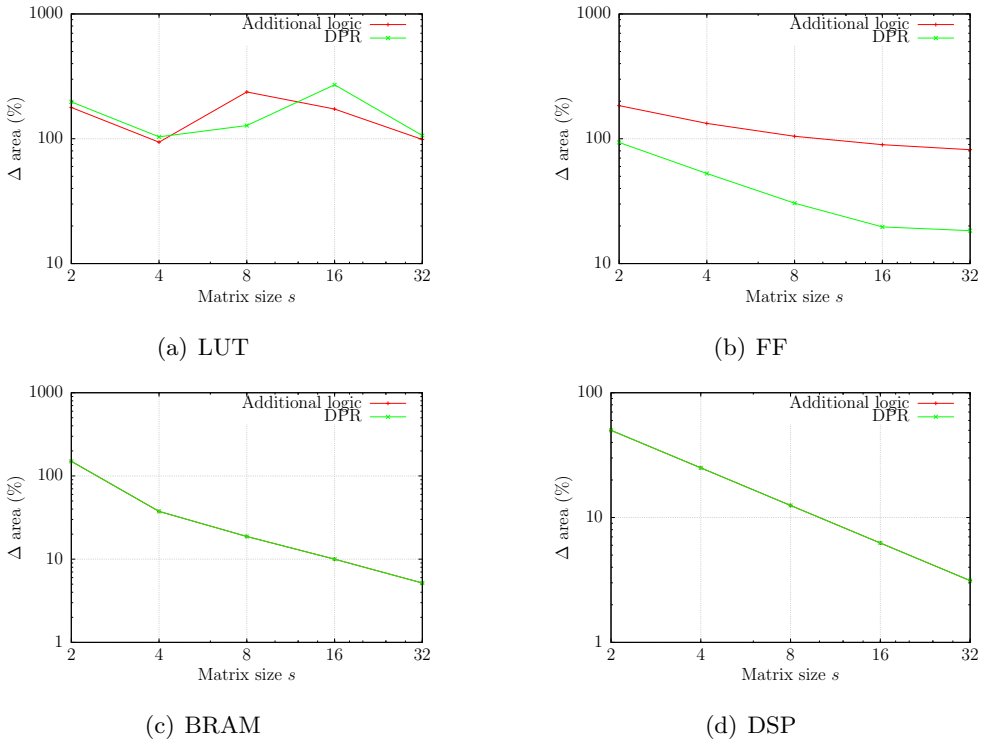


Figure 4.10: ABFT-protected accelerator with additional logic & DPR error correction resource usage overhead versus baseline, showing the change in individual resource utilisations for each design versus its equivalently sized unprotected, baseline implementation.

4.3.2 Performance

Testing was performed on the hardware in order to measure its impact upon performance under a number of conditions. Table 4.2 summarises the results of all of these performance tests. Each test was completed 10,000 times; the mean of these executions is given in all cases. Prior to each test, new uniformly distributed random input data was generated to form $\boldsymbol{A}$ and $\boldsymbol{B}$. Execution times were measured using a cycle-accurate ARM timer peripheral. In all cases, the FPGA fabric was clocked at 50MHz. Included in Table 4.2 are execution times for the fault-intolerant multiplier, the fault-tolerant via additional logic-shifting accelerator and DPR-enabled version running in both of its operating modes. Where appropriate, latency increases relative to the equivalently sized fault-intolerant design are given for comparison. Execution times are given for the occurrences of singular and double MAC failures—the former for both the additional logic-shifting and DPR hardware, and the latter for the DPR version only. Permanent faults were emulated through the targetted inversion of a single accumulator output bit within either one or

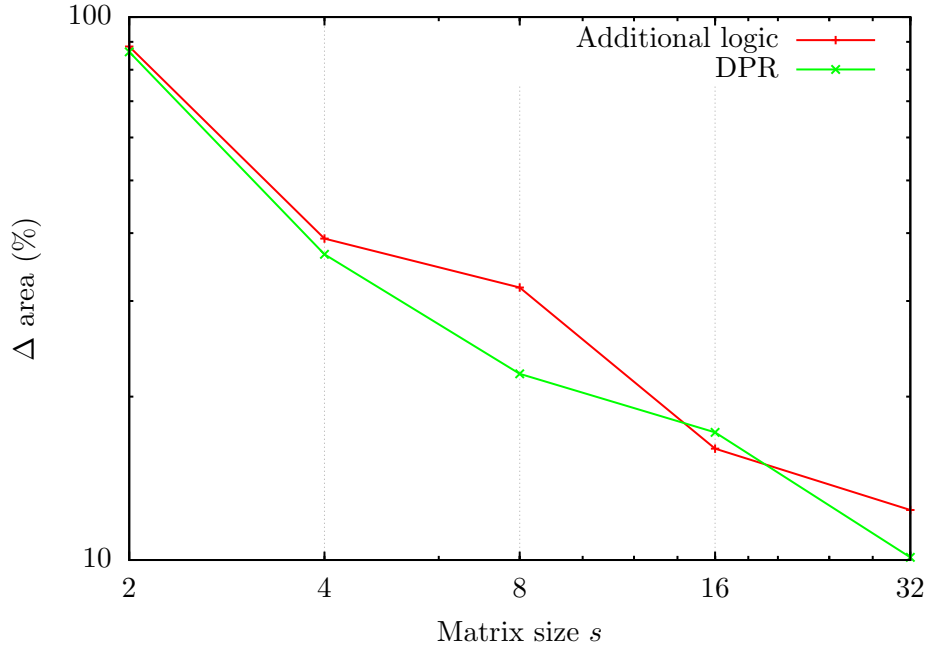
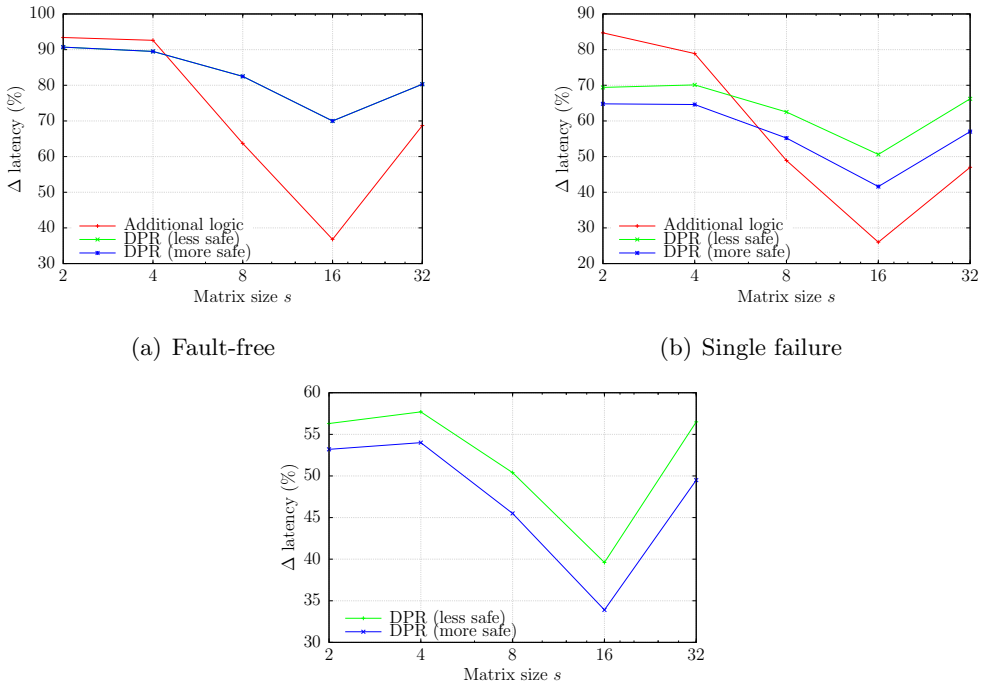


Figure 4.11: ABFT-protected accelerator with additional logic & DPR error correction combined resource usage overhead versus baseline, showing the change in combined resource utilisation for each design versus its equivalently sized unprotected, baseline implementation.

two MACs per execution, with fault locations also randomly chosen. Plots of the latency increases over the fault-intolerant hardware under fault-free, singly and doubly faulty conditions are given in Figure 4.12.

The results show that, for all $s > 4$, the DPR-shifting accelerator outperforms the additional logic-shifting version under normal, fault-free operation as well as that in the presence of a single failure. When comparing the performance of the two error correction implementations side-by-side, recall that the routing's construction is not the only implementational difference them. The lower penalties seen during fault-free operation are due to the relocation of the checksum verification logic from the input to the output side of the output RAM described in Section 4.2.2, allowing return programmable logic (PL)-to-PS data transfers to begin (and end) sooner than they had previously, while gains under single failure mode are realised for larger s as reconfiguration times proportionately fall. The relationship between the performance plots for the DPR-shifting version working in its two modes demonstrates the near-fixed performance cost paid by operating more safely. The trend-reversal seen on all plots after $s = 16$ can be attributed to data transfer throttling:



(a) Fault-free

(b) Single failure

(c) Double failure

Figure 4.12: ABFT-protected accelerator with additional logic & DPR error correction latency overhead versus baseline, showing the change in execution time for each design versus its equivalently sized unprotected, baseline implementation under a range of fault conditions.

once s passes 16, memory copies begin to dominate accelerator execution for proportional runtime. Performance impacts arising from the use of partial reconfiguration are negligible due to the bitstreams' small size and infrequent application per accelerator execution. For the largest-tested design, $s = 32$, the DPR-shifting accelerator incurred a 24.5% latency penalty under fault-free operation—46.1% lower than its additional logic-shifting equivalent.

Timing model-inferred maximum operating frequency ($f_{\max}$) changes are not well correlated, likely due to the stochastic nature of the placement and routing tools used, although decreases between the additional logic- and DPR-shifting designs, likely due to path-lengthening incurred through the reconfigurable partition, are seen for larger s.

4.3.3 Memory

From a software perspective, the primary overhead of the DPR-based fault tolerance strategy is partial bitstream storage. Since accelerator data and bitstream transfers, as well as

accelerator executions, are interrupt-driven, their impacts upon CPU performance are negligible. Table 4.3 summarises the DRAM storage requirements for each value of s tested. The size of each partial bitstream is given along with the total storage requirement for that value of s. The memory occupation is also expressed, for each s, as a proportion of the DRAM available (512MB) on the development board used, an Avnet ZedBoard [74].

4.4 Conclusion

In this chapter, two error correction strategies using additional logic and DPR to achieve runtime resource reallocation were presented, both of which are capable of routing data around resources found to be faulty at runtime by ABFT error detection circuitry. Implementational details were described first, followed by results of experiments performed to assess the hardware's overheads in terms of area, performance and, for the fault-tolerant accelerator using DPR, memory utilisation.

For the largest-implemented design, capable of multiplying pairs of 32×32 matrices with an inner loop-unrolled accelerator, area overheads of 12.4% and 10.1% were encountered through the use of additional logic and DPR for achieving resource reallocation, respectively; a 22.8% reduction for the latter. The additional logic-shifting design with the same s experienced a 45.5% latency penalty during fault-free operation, while its DPR-shifting version achieved 24.5%; a 46.2% reduction, suggesting that DPR betters the use of additional logic in terms of both area and performance for larger-sized accelerators. Again for $s = 32$, mean performance penalties found under faulty conditions were 51.0% and 77.1% over the baseline, fault-free execution time for single and double faults, respectively, for operating in the less safe mode. When operating in the more safe mode, those figures rose to 75.5% and 102%, respectively.

The work presented in this chapter has demonstrated that error detection and correction based upon ABFT, and in particular the combination of ABFT and DPR, is a powerful contendor for low-overhead hardware fault tolerance. The application of error detection at lower levels of precision, thereby introducing an area-to-allowed error tradeoff, is explored in Chapter 6.

Matrix size s	ABFT enabled	Fault avoidance	Resource type LUT	FF	BRAM	DSP	Total
2	✗		239 (0.449%)	210 (0.197%)	2 (0.714%)	6 (2.73%)	1.02%
	✓	Additional logic	665 (1.25%)	597 (0.561%)	5 (1.79%)	9 (4.09%)	1.92%
		DPR	711 (1.34%)	406 (0.382%)	5 (1.79%)	9 (4.09%)	1.90%
4	✗		441 (0.829%)	406 (0.382%)	8 (2.86%)	12 (5.45%)	2.38%
	✓	Additional logic	855 (1.61%)	945 (0.888%)	11 (3.93%)	15 (6.82%)	3.31%
		DPR	898 (1.69%)	620 (0.583%)	11 (3.93%)	15 (6.82%)	3.25%
8	✗		604 (1.14%)	794 (0.746%)	16 (5.71%)	24 (10.9%)	4.63%
	✓	Additional logic	2037 (3.83%)	1625 (1.53%)	19 (6.79%)	27 (12.3%)	6.10%
		DPR	1375 (2.58%)	1036 (0.974%)	19 (6.79%)	27 (12.3%)	5.65%
16	✗		613 (1.15%)	1566 (1.47%)	30 (10.7%)	48 (21.8%)	8.79%
	✓	Additional logic	1674 (3.15%)	2970 (2.79%)	33 (11.8%)	51 (23.2%)	10.2%
		DPR	2273 (4.27%)	1874 (1.76%)	33 (11.8%)	51 (23.2%)	10.3%
32	✗		2115 (3.98%)	3105 (2.92%)	58 (20.7%)	96 (43.6%)	17.8%
	✓	Additional logic	4203 (7.90%)	5643 (5.30%)	61 (21.8%)	99 (45.0%)	20.0%
		DPR	4363 (8.20%)	3675 (3.45%)	61 (21.8%)	99 (45.0%)	19.6%

Table 4.1: Baseline & ABFT-protected accelerator with additional logic & DPR error correction resource usage, containing the raw resource usage figures obtained for all implementations. Percentages of the total number of each of these resources for the target device are also included, along with means of those proportions to give an indication of the overall resource utilisation.

Matrix size s	ABFT enabled	Fault avoidance	Execution time (μs) Fault-free	Single failure	Double failure	$f_{\max}$ (MHz)
2	✗		254			90.204
	✓	Additional logic	272	300		88.992
		DPR (less safe)	280	366	451	95.712
		DPR (more safe)	280	392	477	95.712
4	✗		314			80.103
	✓	Additional logic	339	398		88.168
		DPR (less safe)	351	448	544	82.102
		DPR (more safe)	351	486	581	82.102
8	✗		348			76.941
	✓	Additional logic	546	712		77.042
		DPR (less safe)	422	557	690	85.918
		DPR (more safe)	422	631	764	85.918
16	✗		497			50.495
	✓	Additional logic	1350	1910		56.850
		DPR (less safe)	710	982	1254	52.062
		DPR (more safe)	710	1195	1467	52.062
32	✗		3100			58.156
	✓	Additional logic	4510	6600		55.857
		DPR (less safe)	3860	4680	5490	53.101
		DPR (more safe)	3860	5440	6260	53.101

Table 4.2: ABFT-protected accelerator with additional logic & DPR error correction performance. Averaged execution times and maximum operating frequencies achieved are shown for each design to allow side-by-side comparison of unprotected and the range of protected implementations.

Matrix size s	Bitstream size (kB) Each	Total
2	15.2	45.7 (0.00871%)
4	29.4	147 (0.0281%)
8	43.6	393 (0.0749%)
16	87.1	1480 (0.282%)
32	158	5220 (0.995%)

Table 4.3: ABFT-protected accelerator with DPR error correction bitstream storage requirements, summarising the DRAM partial bitstream storage requirements for each s tested. The size of each partial bitstream is given along with the total storage requirement, absolute and proportional, for that value of s.

5 Fault Observability for Matrix & DSP Operations

5.1 Introduction

In this chapter, work completed to generalise the fault observability testing introduced in Chapter 3 is presented. Three common matrix manipulation algorithms, each highly suited to hardware acceleration—one of which being representative of linear filtering operations—are studied in detail from an implementational perspective to gauge their susceptibility to, observability of and recoverability from faults occurring within their datapaths. Results presented herein capture the impacts of differing operating conditions along with the range of parameters available to be specified by a hardware designer. Rather than by hand, they could equally be used by an automated design tool capable of creating low-overhead fault-tolerant hardware from high-level functional descriptions, thus allowing informed decisions to be made. In all cases in this chapter, an inner loop-unrolled (i.e. $(s + d)$ parallel multiply-accumulators (MACs) or adders, depending on the operator) hardware architecture is assumed.

5.1.1 Contributions

The original contributions of the work presented in this chapter are:

- A software framework for fault simulation in hardware-accelerated linear algebra operators protected with algorithm-based fault tolerance (ABFT).
- A thorough analysis of the fault tolerance of three benchmark ABFT-protected operators.
- The first consideration of distance-x, for $x > 2$, ABFT application in custom logic.

5.1.2 Outline

The remainder of this chapter is organised as follows. Section 5.2 details the fault injection simulation method devised for ascertaining fault observability. Sections 5.3, 5.4 and 5.5 describe the results of the fault observability testing performed for three target operations: matrix-matrix multiplication (Section 5.3), matrix addition (Section 5.4) and matrix-vector multiplication (Section 5.5). In each of the latter three sections, results are presented and analysed across a wide range of variables. The chapter is summarised in Section 5.6.

5.2 Method

Software simulations were performed to assess the fault observabilities of several operators across the following range of variables:

- Operator: {matrix-matrix multiplication, matrix addition, matrix-vector multiplication}.
- Fault type: {permanent, transient}.
- Number of simultaneous faults (f): {1, 2, 3}.
- d: {1, 2, 3}.
- s (or vector length): {2, 4, 8, 16, 32}.
- Data width (n) (bits): {2, 4, 8, 16, 32}.

For each combination of these, the steps detailed in either Algorithm 7 (for matrix-matrix multiplication), 10 (for matrix-matrix addition) or 11 (for matrix-vector multiplication) were repeated 960,000 times.

In all cases, the fault model applied was that of individually targetted stuck-at-one (SA1) accumulator output (for multiplicative operations) or adder (for additional) bits. For permanent faults, such SA1s are representative of, for example, worn transistors or bridged interconnects, while they mimic effects including register and memory upsets in the case of transients.

Results gleaned from this testing were, as is the case in the all of the fault observability testing performed in this thesis, independent of fault rate, area and latency: they demonstrate the *proportions of total accelerator executions* that should be expected to result in

particular classes of outputs under *fixed fault conditions*. They cannot be used to directly ascertain the expected rates of certain output classes' occurrence, although this can be achieved by scaling the proportions to take fault rate, area and/or latency, as required, into account.

Algorithm 7 details the steps performed for each matrix-matrix multiplication simulation. The procedure is largely similar to that shown in Algorithm 5 in Section 3.4, but expanded to support transient as well as permanent faults—selectable via the $fault\_type$ variable—and generalised for d. Changes were also made to allow for more detailed result classification. Following the creation of randomly filled matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, reference matrix $\boldsymbol{C}_{\text{ref}}$—containing the correct result—is created (Line 5) and calculated (Line 6). Checksumming is performed on $\boldsymbol{A}$ and $\boldsymbol{B}$ to create $\boldsymbol{A}_{\text{c}}$ and $\boldsymbol{B}_{\text{r}}$ (Lines 9 and 10) with d additional rows (for $\boldsymbol{A}_{\text{c}}$) or columns (for $\boldsymbol{B}_{\text{r}}$) added as shown in Section 2.7.1. Once $\boldsymbol{C}_{\text{f}}$ is provisioned (Line 11), control splits depending on whether $fault\_type$ indicates that permanent (from Line 12) or transient (from Line 22) fault emulation is required. For permanent faults, a one-dimensional (vector) bit mask is created (Line 13) to represent the faults. This is an $(s + d)$-element array of n-bit zero-initialised values with f '1's randomly scattered throughout. Computation proceeds with column j's bit mask applied both before (Line 16) and during (Line 18) each multiply-accumulation. Transient faults each affect a single bit during a single multiply-accumulation step only, so a larger bit mask is created (Line 23) to represent them. This is an $\big((s+d)\times(s+d)\times(s+d)\big)$-element three-dimensional array; hence the 3 in the procedure call. Calculation proceeds with the bit mask value for the particular row, column and multiply-accumulation step applied before (Line 26) and during (Line 28) each iteration. Note that the zeroth elements in the outermost dimension correspond to faults that occur during the resetting of variables before accumulation commences. Following fault-emulated computation, the data elements in $\boldsymbol{C}_{\text{f}}$ are compared with $\boldsymbol{C}_{\text{ref}}$ (Line 33) and checksums verified (Line 34) in order to classify the result by calling procedure `classify_result()` (Line 35). Finally, checksums are compared to the faulty columns array created in order to determine whether or not faults were successfully located (Line 36).

Procedure `gen_bit_mask()` is detailed in Algorithm 8. Depending on whether *dimensions* is 1, 2 or 3, a one-, two- or three-dimensional array of n-bit values, with $s + d$ elements per dimension, is created. Initially these values are zero, but f '1's are scattered randomly throughout them to represent faults. In multiply-accumulation

with permanent fault emulation, for example, each '1' is representative of a single bit of a single MAC's registers remaining high throughout a complete multiplication. For transient fault emulation with the same operator, however, each '1' represents a single bit of a MAC register being forced high for a single clock cycle.

Algorithm 9 details the steps involved in procedure `classify_result()`, which are independent of the operation performed. This procedure serves to classify the result, as explained in Section 2.7.3, as either detected, erroneous (false positive), missed (false negative) or undetectable (masked).

Algorithm 10 shows the steps taken to emulate faulty matrix addition. The layout is identical to that in Algorithm 7, with relatively minor changes made to suit the different operator. Element-wise addition is performed modulo-2^n (Line 6) to produce $\boldsymbol{C}_{\text{ref}}$, with d rows and columns of checksums added to *both* $\boldsymbol{A}$ and $\boldsymbol{B}$ to form $\boldsymbol{A}_{\text{f}}$ and $\boldsymbol{B}_{\text{f}}$ (Lines 9 and 10), as exemplified in Section 2.7.2. Under transient fault injection, a two-, rather than three-, dimensional bit mask array is created (Line 20) and applied (Line 23) since matrix addition is a two-, rather than three-, loop procedure.

The simulation of matrix-vector multiplication is detailed in Algorithm 11. The steps here are also similar to those in Algorithms 7 and 10, with changes made to suit the checksumming steps exemplified in Section 2.7.2. s-element vector $\boldsymbol{a}$ is created (Line 1) rather than a matrix, with checksumming—row-wise—performed only upon $\boldsymbol{B}$ to create $\boldsymbol{B}_{\text{r}}$ (Line 8). Fault-free and -prone outputs $\boldsymbol{c}_{\text{ref}}$ and $\boldsymbol{c}_{\text{r}}$, created on Lines 5 and 9, respectively, are also vectors rather than matrices. Following multiplication, results are classified (Line 29) as normal but fault location is not determined since $\boldsymbol{c}_{\text{r}}$, a row vector, does not contain any column-wise checksums from which to determine location; faults are therefore only able to be detected within a matrix-vector multiplication.

5.3 Matrix-matrix Multiplication

Figure 5.1 summarises the results of matrix-matrix multiplication simulations performed under the influence of permanent faults. Proportions of detected faults are displayed relative to s, f and d; n was found to have little impact upon the results in this case, so those obtained were averaged across the range of n tested. The plots of detected faults show that their proportions increase with s and f but decrease with d.

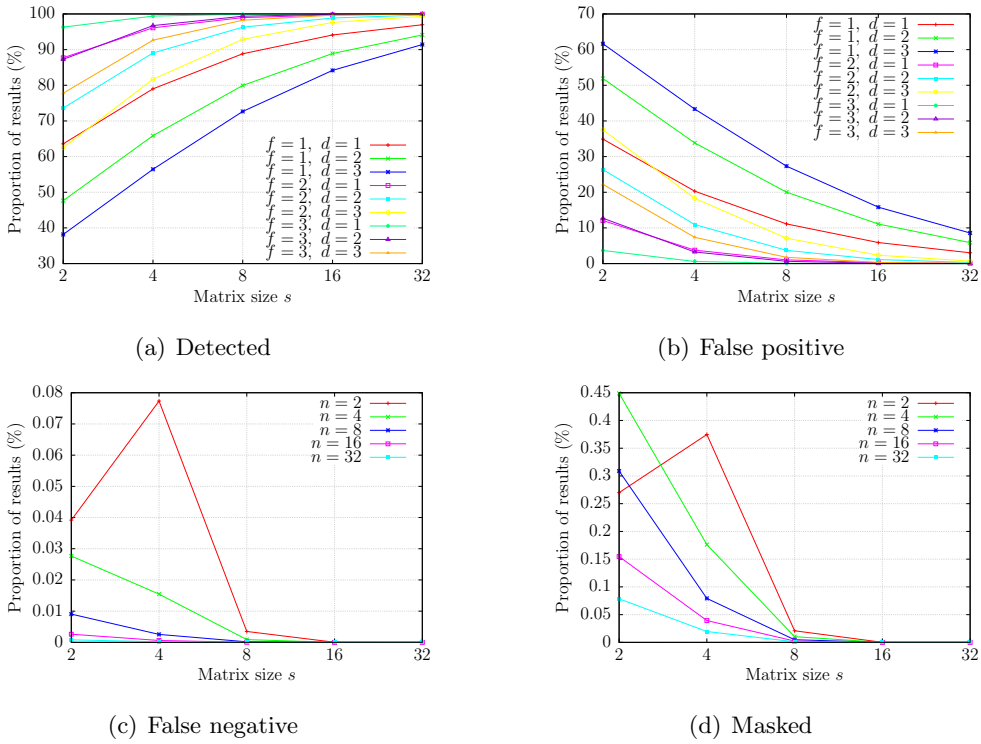
False positives account for the majority of results not classified as detected. n was also found to be of little significance to these results; it is therefore not shown here. The

prevalence of false positives is largely controlled by the ratio of checksum to total elements within each matrix row (or column). In the $s = 2,\ f = 1,\ d = 1$ case, for example, this is $1/3$—approximately the value of the corresponding point in Figure 5.1. Since d increases this ratio, false positives become more common as it scales; as f increases, however, they become scarcer since multiple faults occurring simultaneously are less likely to be masked by data within the information matrix. That the likelihood of false positives decreases as s (and consequently area) increases is a desirable outcome. Assuming a steady fault rate, this implies that proportionally less time will be spent unnecessarily recomputing results as s scales.

The occurrence of false negatives was found to decrease across all variables tested. f and d had lower impacts upon the results than s and n, however, so the plots shown are averaged across f and d. Not shown within the plots due to this averaging is the fact that no false negatives were encountered across the range of variables tested when $f = 1$. This is a powerful result, inferring that if single permanent faults can be either bypassed or repaired before subsequent faults develop, it should be possible to operate indefinitely without allowing incorrect results to go unnoticed. Zero false negatives were observed for $s = 32$ with any n. This also held true for $s \geq 16$ with $n \geq 4$ and for $s \geq 8$ with $n = 32$. While perhaps somewhat high for the smallest s and n, it is encouraging that for more reasonable values of those variables, i.e. ones for which hardware acceleration is worthwhile, the likelihood of encountering false negatives due to multiple faults is low. In the $s = 8,\ n = 8,\ f = 2,\ d = 1$ case, for example, this was approximately 0.001%.

Finally from Figure 5.1 it can be seen that fewer masked faults were encountered as s and n increased. f and d are also not reflected here; they were again found to be of little significance to the results. While their frequencies of occurrence dropped off less sharply than those of false negatives, no masked faults were observed for $s = 32$ with any n or for $s = 16$ with $n = 32$. As noted in Section 2.7.3, masked faults are purely data-dependent: while it may be desirable to observe them from a fault detection perspective, their occurrence in none of the experiments performed led to incorrectly computed data, nor to having to recalculate any results.

Figure 5.2 shows the proportions of successfully located results for the same operator under the same fault injection conditions. Note that the data these plots represent, along with that for the remainder of located results' plots in this chapter, was scaled such that it excluded masked faults. This was done since masked faults represent results that can



(a) Detected

(b) False positive

(c) False negative

(d) Masked

Figure 5.1: Matrix-matrix multiplication permanent fault observability. Each plot shows the proportion of total accelerator execution's results that led to a particular class of output for a particular combination of s, n, f and d.

(only be) ignored; they are by definition unlocatable. The results were found to be largely independent of s and were thus averaged across the range of s tested. This is where the advantage of $d \geq 1$ becomes clear; successful fault location is more likely with multiple independent checksums. This, coupled with the opposite dependency found for false positives—where their likelihood scaled with d—implies a tradeoff: larger d allows more faults to be targetted for repair at the expense of increased unnecessary recomputation. Finally reflected here is the difficulty of locating simultaneous faults; as f increases, the proportion of faults able to be located decreases.

The results obtained for matrix-matrix multiplication under transient fault injection, presented in Figure 5.3, are rather different to those seen with permanent faults. Plots of detected transient faults, shown here averaged across the range of n tested due to being largely independent of its value, follow similar patterns to those for detected permanent faults—proportions increasing with s and f but decreasing with d—but they reach clearly defined upper limits. The reason for this is hinted at from the masked faults plots. Trivially, individual bits are likely to be masked 50% of the time with uniformly distributed

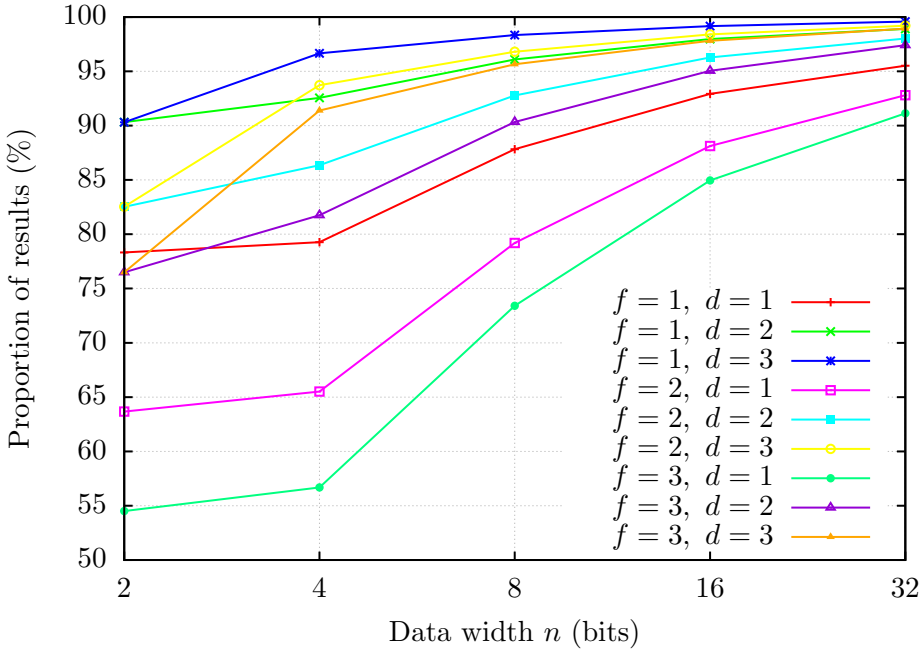
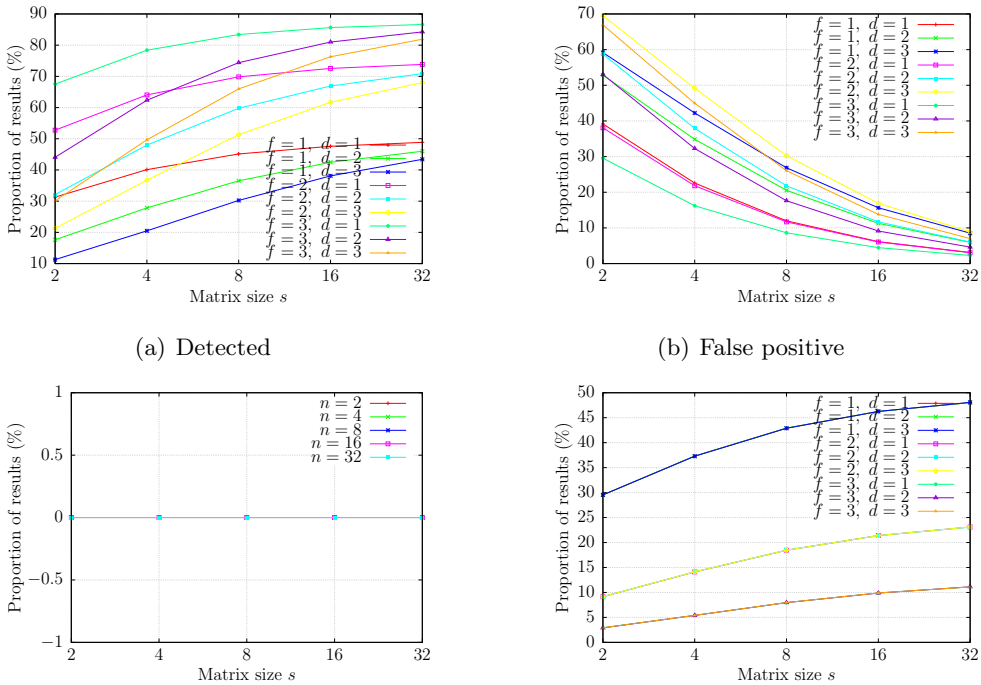


Figure 5.2: Matrix-matrix multiplication permanent fault locatability. Each plot shows the proportion of total accelerator execution's results that gave a one-to-one mapping between faulty MACs and incorrectly computed column-wise checksums for a particular combination of s, n, f and d.

data under the influence of single SA1 faults since a bit is 50% likely to already be high. Hence, for $f = 1$, masked faults approach an upper limit of 50%. For $f = 2$ they approach 25%—expected here since data and fault locations between computations were both independent—and, similarly, 12.5% for $f = 3$. The proportion of false positives was found to decrease with s and f but increase with d, as for permanent faults. Their dropoffs with f, in particular, are less pronounced. Zero false negatives were encountered for matrix-matrix multiplication testing under transient fault injection across the range of variables tested. These results are particularly encouraging, suggesting that transient fault detection for matrix-matrix multiplication is hampered only by the occurrence of false positives. In the $s = 32,\ n = 32,\ f = 1,\ d = 1$ case, however—representative of a realistically sized implementation—the likelihood of their occurrence is only around 3%.

The results of the same operator's locatability testing during transient fault injection are shown in Figure 5.4. These are very different to those obtained under permanent fault injection. In particular, localisation was successful 100% of the time when $f = 1$, however it dropped sharply for $f > 1$. The data obtained was found to be largely independent of



(a) Detected

(b) False positive

(c) False negative

(d) Masked

Figure 5.3: Matrix-matrix multiplication transient fault observability. Each plot shows the proportion of total accelerator execution's results that led to a particular class of output for a particular combination of s, n, f and d.

n, so was combined across its range of values. It can be seen that, in the transient fault case, fault location becomes a more difficult problem with increasing s as well as f, and that d has little impact upon its success, particularly for larger s.

5.4 Matrix Addition

Figure 5.5 represents the classification of results for matrix addition testing under the presence of permanent faults. Side-by-side comparison of Figure 5.5 and Figure 5.1, for matrix-matrix multiplication under the same fault conditions, shows that the results were similar. Indeed, in the detected, false positive and false negative fault cases this is particularly true; the only difference of any significance is that the lower and upper bounds seen for detected and false positive faults were fractionally lower and higher, respectively, for matrix addition. Note that all plots shown here are presented in the same forms as the corresponding plots for matrix-matrix multiplication. The primary dissimilarity between the two operators' results can be seen in the plots of masked faults: where for matrix-matrix

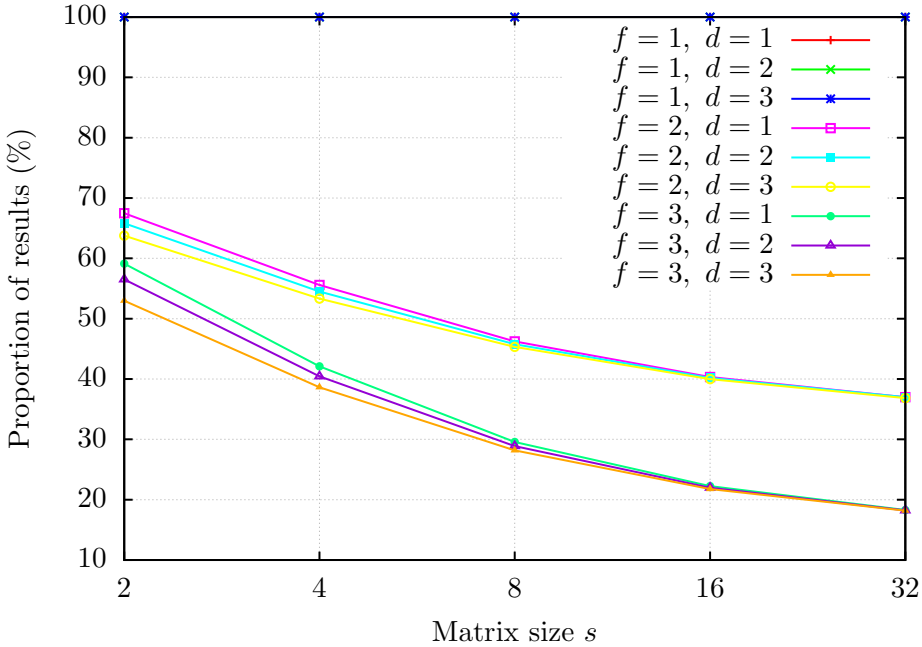
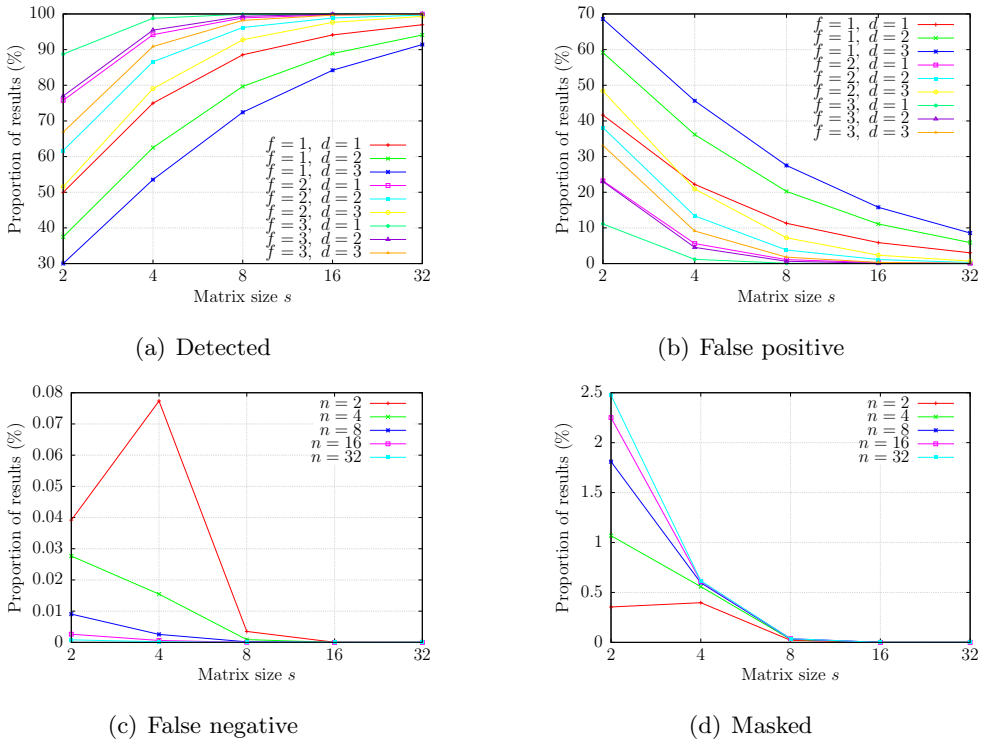


Figure 5.4: Matrix-matrix multiplication transient fault locatability. Each plot shows the proportion of total accelerator execution's results that gave a one-to-one mapping between faulty MACs and incorrectly computed column-wise checksums for a particular combination of s, n, f and d.

multiplication their likelihood dropped as n grew, here the opposite is true. Despite this, no masked faults were encountered for $s = 32$ with any tested value of n.

The results of fault locatability testing for the same operator under the same fault conditions are presented in Figure 5.6. These are also largely similar to those obtained for matrix-matrix multiplication with permanent fault injection shown in Figure 5.2; this was expected since the output matrices for the two operators are of the same form. The primary difference between the two sets of plots is that upper limits appear to be lower in the matrix addition case, indicating that faults are less likely to be successfully located for that operator.

The classification of results obtained from matrix addition simulations performed under the influence of transient faults is shown in Figure 5.7. These, as expected, are similar to those seen in Figure 5.3 for matrix-matrix multiplication under the same fault conditions. The explanation for the masked fault proportionalities observed under transient fault injection given in Section 5.3 is exemplified even more clearly in Figure 5.7, where it can be seen that the frequency of occurrence is exclusively dependent upon f. Zero false



(a) Detected

(b) False positive

(c) False negative

(d) Masked

Figure 5.5: Matrix addition permanent fault observability. Each plot shows the proportion of total accelerator execution's results that led to a particular class of output for a particular combination of s, n, f and d.

negatives were encountered during matrix addition with transient fault injection testing. The primary difference in both detected and false positive results obtained here and for matrix-matrix multiplication under the same fault conditions was that the lower bounds observed were lower in the matrix addition case.

The results of locatability testing for the same operator under transient fault injection are shown in Figure 5.8. Note that these are comparable to those presented in Figure 5.4 for matrix-matrix multiplication under the same fault conditions. As was the case for that operator, all faults were locatable when $f = 1$ and their proportions thereafter decreased as f rose. The plots shown for $f > 1$ in the matrix addition case are shallower than those in the matrix-matrix multiplication case, with lower bounds for low s but similar locatability proportions for higher s.

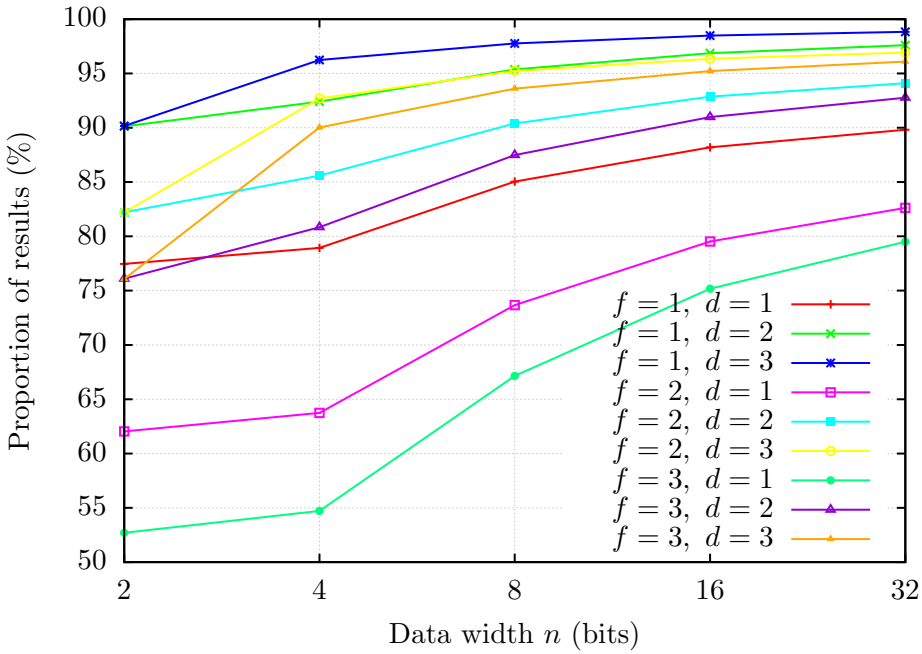
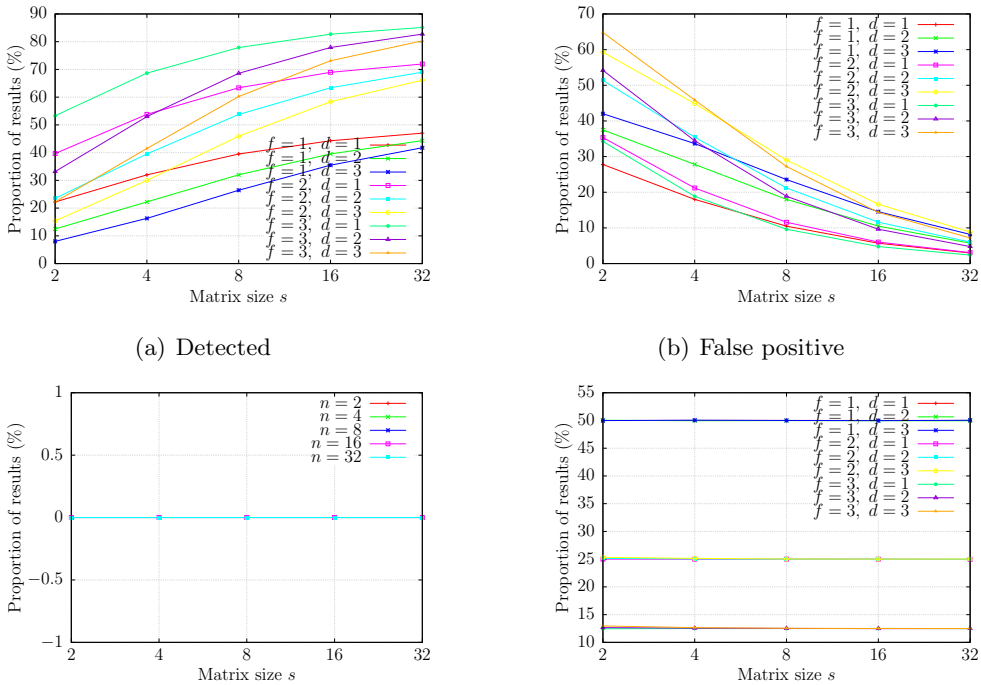


Figure 5.6: Matrix addition permanent fault locatability. Each plot shows the proportion of total accelerator execution's results that gave a one-to-one mapping between faulty adders and incorrectly computed column-wise checksums for a particular combination of s, n, f and d.



(a) Detected

(b) False positive

(c) False negative

(d) Masked

Figure 5.7: Matrix addition transient fault observability. Each plot shows the proportion of total accelerator execution's results that led to a particular class of output for a particular combination of s, n, f and d.

5.5 Matrix-vector Multiplication

Figure 5.9 summarises the results of matrix-vector multiplication simulations performed under the presence of permanent faults. Although the proportions of detected faults decreased with f and d, they were seen to be most dependent upon s and n; they are therefore displayed after being averaged across f and d. Note that detectability also increased with both s and n. The proportions of observed false positives did not vary significantly with n, hence these are averaged across the range of results for that variable. In line with previous operators, the occurrence of false positives was seen to increase with d but decrease with s and f. False negatives were not observed for $f = 1$, but are presented here averaged over both f and d since variations in s and n were more significant. Unfortunately, the proportions of false negatives encountered increased with s. They did, however, reduce with n, and increased with s less significantly as n fell in magnitude. Masked faults, displayed here averaged over the range of s tested, were found to be independent of d, decreasing with both n and f.

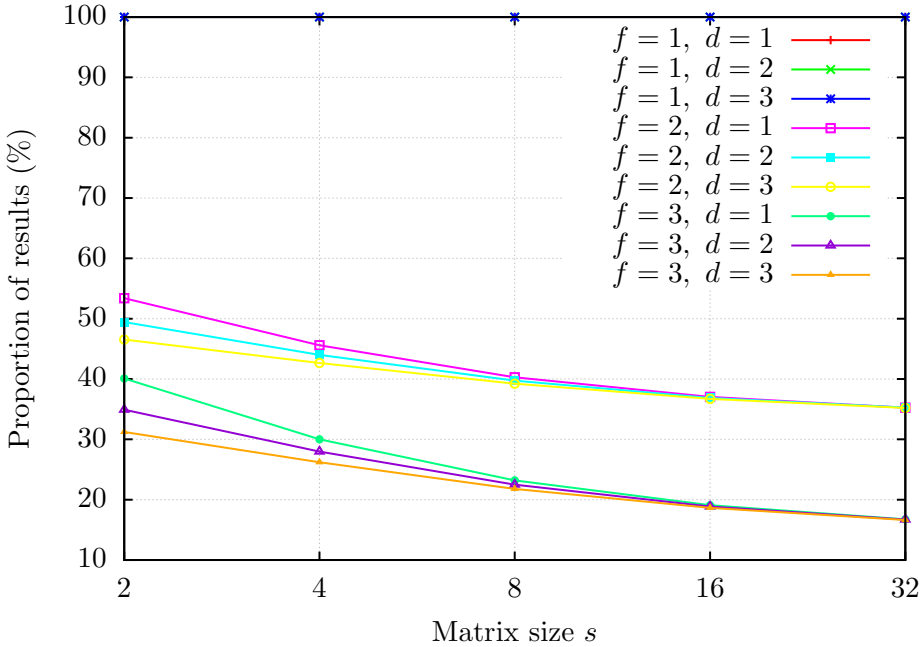
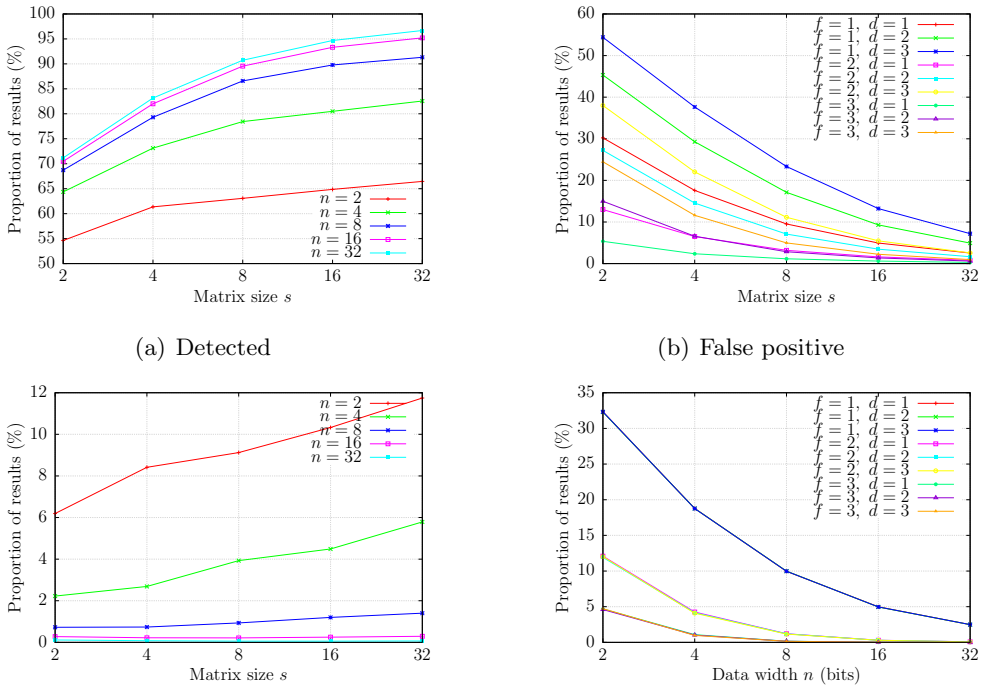


Figure 5.8: Matrix addition transient fault locatability. Each plot shows the proportion of total accelerator execution's results that gave a one-to-one mapping between faulty adders and incorrectly computed column-wise checksums for a particular combination of s, n, f and d.

Figure 5.10 presents the final fault injection simulation results; for the computation of matrix-vector multiplications in the presence of transient faults. Results shown here for detected, false positive and masked faults are combined across the range of n tested since they were found to be largely independent of that variable's value. False negative proportions are given by s and n with results combined across the range of f and d for the same reason. The asymptotic behaviour of the detected plots is largely dictated by the numbers of masked faults encountered; these are similar to those seen for transient fault injection in both the matrix-matrix multiplication and matrix addition cases. Masked fault occurrences can be seen to be independent of d, with their upper bounds dependent upon f. s can be seen to have little impact upon the likelihood of encountering false negative results, particularly for larger n.

5.6 Conclusion

In this chapter, the results of fault injection simulations performed upon a trio of ABFT-protected linear algebra operators—matrix-matrix multiplication, matrix addition and



(a) Detected

(b) False positive

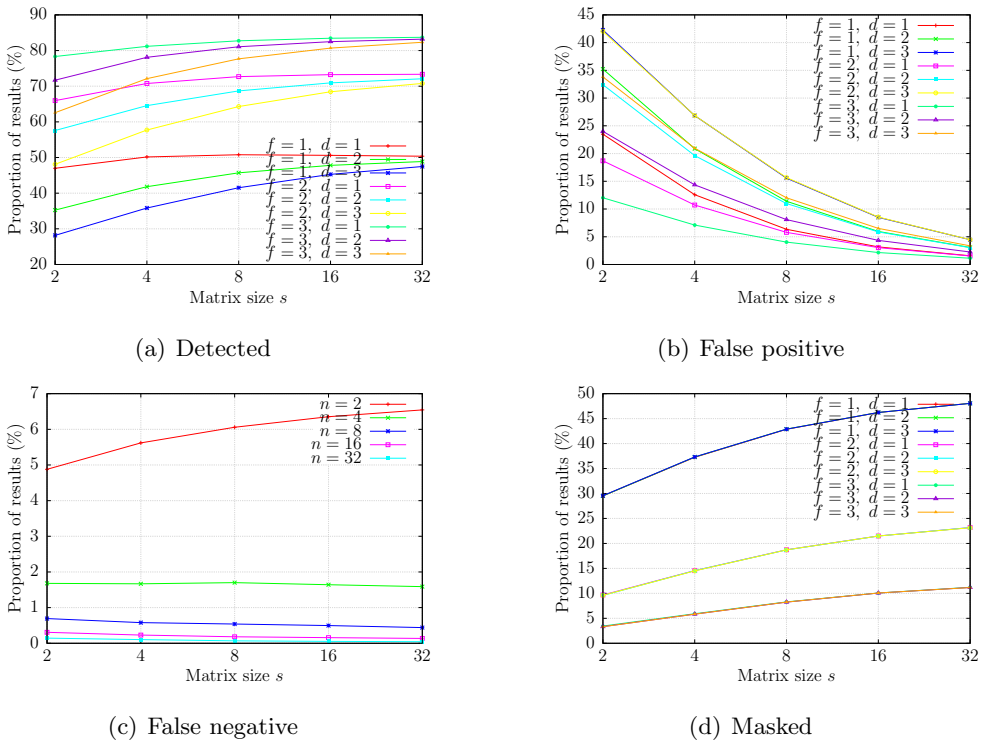
(c) False negative

(d) Masked

Figure 5.9: Matrix-vector multiplication permanent fault observability. Each plot shows the proportion of total accelerator execution's results that led to a particular class of output for a particular combination of s, n, f and d.

matrix-vector multiplication—across a range of implementational parameters and operating conditions were presented. Details of the software framework developed to simulate those operations under the influence of both permanent and transient faults were also given. Analysis of the results obtained suggested high fault tolerance across the three operators, the majority of which improve as hardware compexity grows. The results cement ABFT's status as a credible alternative to more established fault tolerance techniques despite its comparatively low overheads, particularly in terms of area.

Of the variables considered, d has proven to be somewhat disappointing: small, if any, gains were realised for distance-x, with $x > 2$, checksumming. Consequently d will remain fixed at 1 for the work presented in Chapter 6.



(a) Detected

(b) False positive

(c) False negative

(d) Masked

Figure 5.10: Matrix-vector multiplication transient fault observability. Each plot shows the proportion of total accelerator execution's results that led to a particular class of output for a particular combination of s, n, f and d.

Algorithm 7 Matrix-matrix multiplication fault injection simulation

```
1: create s × s matrix A
2: create s × s matrix B
3: A.rand_fill()
4: B.rand_fill()
5: create s × s matrix C_ref
6: C_ref ← AB
7: create (s + d) × s matrix A_c
8: create s × (s + d) matrix B_r
9: A_c ← A.add_cs('col', d)
10: B_r ← B.add_cs('row', d)
11: create (s + d) × (s + d) matrix C_f
12: if fault_type = 'perm' then
13:    bit_mask, faulty_cols ← generate_bit_mask(s, n, d, 1, f)
14:    for i = 0 to s + d − 1 do
15:       for j = 0 to s + d − 1 do
16:          C_f[i][j] ← bit_mask[j]
17:          for k = 0 to s + d − 2 do
18:             C_f[i][j] ← ((C_f[i][j] + A_c[i][k] × B_r[k][j]) mod 2^n) bitwise or bit_mask[j]
19:          end for
20:       end for
21:    end for
22: else
23:    bit_mask, faulty_cols ← generate_bit_mask(s, n, d, 3, f)
24:    for i = 0 to s + d − 1 do
25:       for j = 0 to s + d − 1 do
26:          C_f[i][j] ← bit_mask[i][j][0]
27:          for k = 0 to s + d − 2 do
28:             C_f[i][j] ← ((C_f[i][j] + A_c[i][k] × B_r[k][j]) mod 2^n) bitwise or
                bit_mask[i][j][k + 1]
29:          end for
30:       end for
31:    end for
32: end if
33: data_ok ← C_f.get_data() = C_ref
34: cs_ok ← C_f.check_cs()
35: result_type ← classify_result(data_ok, cs_ok)
36: located ← C_f.diagnose_cs() = faulty_cols
```

Algorithm 8 `generate_bit_mask()` procedure used in fault injection simulations

Require: s, n, d, $dimensions$, f

```
1:  create s + d vector faulty_cols
2:  if dimensions = 1 then
3:     create s + d vector bit_mask
4:     for i = 0 to f − 1 do
5:        repeat
6:           col = rand(0 to s + d)
7:           bit = rand(0 to n − 1)
8:        until not bit_mask[col] bitwise and 2^bit
9:        bit_mask[col] ← bit_mask[col] bitwise or 2^bit
10:       faulty_cols[col] ← true
11:    end for
12: else if dimensions = 2 then
13:    create (s + d) × (s + d) matrix bit_mask
14:    for i = 0 to f − 1 do
15:       repeat
16:          col = rand(0 to s + d)
17:          step = rand(0 to s + d)
18:          bit = rand(0 to n − 1)
19:       until not bit_mask[col][step] bitwise and 2^bit
20:       bit_mask[col][step] ← bit_mask[col][step] bitwise or 2^bit
21:       faulty_cols[col] ← true
22:    end for
23: else
24:    create (s + d) × (s + d) × (s + d) 3D matrix bit_mask
25:    for i = 0 to f − 1 do
26:       repeat
27:          row = rand(0 to s + d)
28:          col = rand(0 to s + d)
29:          step = rand(0 to s + d)
30:          bit = rand(0 to n − 1)
31:       until not bit_mask[row][col][step] bitwise and 2^bit
32:       bit_mask[row][col][step] ← bit_mask[row][col][step] bitwise or 2^bit
33:       faulty_cols[col] ← true
34:    end for
35: end if
36: return bit_mask, faulty_cols
```

Algorithm 9 `classify_result()` procedure used in fault injection simulation

Require: $data\_ok,\ cs\_ok$

```
if data_ok and cs_ok then
    result_type ← 'masked'
else if data_ok and not cs_ok then
    result_type ← 'false_pos'
else if not data_ok and cs_ok then
    result_type ← 'false_neg'
else
    result_type ← 'detected'
end if
return result_type
```

Algorithm 10 Matrix addition fault injection simulation

```
create s × s matrix A
create s × s matrix B
A.rand_fill()
B.rand_fill()
create s × s matrix C_ref
C_ref ← A + B
create (s + d) × (s + d) matrix A_f
create (s + d) × (s + d) matrix B_f
A_f ← A.add_cs('full', d)
B_f ← B.add_cs('full', d)
create (s + d) × (s + d) matrix C_f
if fault_type = 'perm' then
    bit_mask, faulty_cols ← generate_bit_mask(s, n, d, 1, f)
    for i = 0 to s + d − 1 do
        for j = 0 to s + d − 1 do
            C_f[i][j] ← ((A_f[i][j] + B_f[i][j]) mod 2^n) bitwise or bit_mask[j]
        end for
    end for
else
    bit_mask, faulty_cols ← generate_bit_mask(s, n, d, 2, f)
    for i = 0 to s + d − 1 do
        for j = 0 to s + d − 1 do
            C_f[i][j] ← ((A_f[i][j] + B_f[i][j]) mod 2^n) bitwise or bit_mask[i][j]
        end for
    end for
end if
data_ok ← C_f.get_data() = C_ref
cs_ok ← C_f.check_cs()
result_type ← classify_result(data_ok, cs_ok)
located ← C_f.diagnose_cs() = faulty_cols
```

Algorithm 11 Matrix-vector multiplication fault injection simulation

```
1: create s vector a
2: create s × s matrix B
3: a.rand_fill()
4: B.rand_fill()
5: create s vector c_ref
6: c_ref ← aB
7: create s × (s + d) matrix B_r
8: B_r ← B.add_cs('row', d)
9: create s + d vector c_r
10: if fault_type = 'perm' then
11:    bit_mask, faulty_cols ← generate_bit_mask(s, n, d, 1, f)
12:    for j = 0 to s + d − 1 do
13:       c_r[j] ← bit_mask[j]
14:       for k = 0 to s + d − 2 do
15:          c_r[j] ← ((c_r[j] + a[k] × B_r[k][j]) mod 2^n) bitwise or bit_mask[j]
16:       end for
17:    end for
18: else
19:    bit_mask, faulty_cols ← generate_bit_mask(s, n, d, 2, f)
20:    for j = 0 to s + d − 1 do
21:       c_r[j] ← bit_mask[j][0]
22:       for k = 0 to s + d − 2 do
23:          c_r[j] ← ((c_r[j] + a[k] × B_r[k][j]) mod 2^n) bitwise or bit_mask[j][k + 1]
24:       end for
25:    end for
26: end if
27: data_ok ← c_r.get_data() = c_ref
28: cs_ok ← c_r.check_cs()
29: result_type ← classify_result(data_ok, cs_ok)
```

6 Reduced-precision Algorithm-based Fault Tolerance

6.1 Introduction

In this chapter, research into the application of algorithm-based fault tolerance (ABFT) in an field-programmable gate array (FPGA)-implemented accelerator at reduced levels of precision is presented. This allows for the introduction of a previously unexplored tradeoff: sacrificing some observability, preferably 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. The implementation of two distinct truncation techniques is described, with their effects upon overheads and allowed data error compared. The methods introduced in this chapter 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, hardware-accelerated matrix multiplication is called upon once more.

While previous fixed-point ABFT-related work has assumed all data and checksums to be data width (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 degree of data error. This is achieved by effectively bounding allowed data error: in this chapter, levels of checksum truncation are used to infer maximum error propagation, however such a derivation could equally be performed in reverse.

6.1.1 Contributions

The original contributions of the work presented in this chapter are:

- The first consideration of distinct data and checksum bit-widths within ABFT-protected operations: *reduced-precision algorithm-based fault tolerance (RP-ABFT)*.
- The first implementation of circuitry incorporating RP-ABFT for resilience against hardware faults.
- Analysis of the costs and benefits of applying two forms of RP-ABFT to various precisions.
- Insight into the hardware fault tolerance of RP-ABFT.

6.1.2 Publications

The work presented in this chapter has been peer-reviewed and will appear in the 2016 proceedings of the International Workshop on Applied Reconfigurable Computing (ARC) [75].

6.1.3 Outline

The remainder of this chapter is organised as follows. Section 6.2 describes the mathematical principles behind RP-ABFT for the two proposed truncation techniques, with Sections 6.2.1 and 6.2.2 covering truncation from the most-significant bit (MSB) and least-significant bit (LSB) first, respectively. Impelementational details are given in Section 6.3, with the small modifications made to the baseline architectures used in Chapters 3 and 4 described first in Section 6.3, followed by those needed to implement the two flavours of RP-ABFT in Sections 6.3.2 and 6.3.3. In Section 6.4, the overheads associated with RP-ABFT are analysed, focussing upon area and performance in Sections 6.4.1 and 6.4.2, respectively. Section 6.5 presents analysis of the impacts the selection of the various implementational options introduced by RP-ABFT has upon fault observability within the targetted datapath, while Section 6.6 gives concluding remarks.

6.2 Principles of RP-ABFT

6.2.1 MSB-first Truncation

During checksum generation and verification, data element bits from the most-significant downwards can be sacrificed in order to reduce the size of the logic required to manipulate them into checksums. All input data elements are n-bit signed integers and the number of bits of precision removed from each during checksum generation is represented by the

truncation width (r). Output data elements are always $2n$-bit; this departure from the norm of all-n bit is elaborated upon in Section 6.3.1. In MSB-first truncation, therefore, input checksums are reduced to the least significant $n-r$ bits of precision. This also limits the precision of output checksums to the least significant $n-r$ bits.

6.2.2 LSB-first Truncation

To maintain sensitivity to faults that cause high-magnitude errors, it is possible to perform truncation from the least, rather than most, significant bits of data elements when forming and manipulating checksums. The constructions of the inputs and outputs of ABFT-protected matrix multiplication are shown in Equation 6.1. Each element is marked as either an input data element (d_{in}), output data element (d_{out}), input checksum element (cs_{in}), output checksum element (cs_{out}) or corner output checksum element ($cs_{\text{out, c}}$), as appropriate. $cs_{\text{out, c}}$ is a special form of output checksum: that which is itself formed exclusively from cs_{in} elements. The 'c' in $cs_{\text{out, c}}$ indicates *corner*.

$$\begin{pmatrix} d_{\text{in}} & \cdots & d_{\text{in}} \\ \vdots & \ddots & \vdots \\ d_{\text{in}} & \cdots & d_{\text{in}} \\ cs_{\text{in}} & \cdots & cs_{\text{in}} \end{pmatrix} \begin{pmatrix} d_{\text{in}} & \cdots & d_{\text{in}} & cs_{\text{in}} \\ \vdots & \ddots & \vdots & \vdots \\ d_{\text{in}} & \cdots & d_{\text{in}} & cs_{\text{in}} \end{pmatrix} = \begin{pmatrix} d_{\text{out}} & \cdots & d_{\text{out}} & cs_{\text{out}} \\ \vdots & \ddots & \vdots & \vdots \\ d_{\text{out}} & \cdots & d_{\text{out}} & cs_{\text{out}} \\ cs_{\text{out}} & \cdots & cs_{\text{out}} & cs_{\text{out, c}} \end{pmatrix} \tag{6.1}$$

Symbols for maximum absolute value ($\vee$) and maximum absolute error (ϵ) must be introduced next. $\vee(d_{\text{in}})$ is as defined in Equation 6.2. The r-bit LSB-first 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, as also shown in Equation 6.2, $\vee(\lfloor d_{\text{in}} \rfloor_r) = \vee(d_{\text{in}})$; the maximum negative value, for which truncation by any $0 \leq r < n$ will have no effect, also represents the maximum absolute value. $\epsilon(\lfloor d_{\text{in}} \rfloor_r)$ is as defined in Equation 6.3.

$$\vee(\lfloor d_{\text{in}} \rfloor_r) = \vee(d_{\text{in}}) = 2^{n-1} \tag{6.2}$$

$$\epsilon(\lfloor d_{\text{in}} \rfloor_r) = 2^r - 1 \tag{6.3}$$

Each cs_{in} element is formed from square matrix size (s) d_{in} elements, each independently truncated, as shown in Equation 6.4. $\vee$ and ϵ of each cs_{in} element are therefore trivial to calculate, as shown in Equations 6.5 and 6.6, respectively.

$$cs_{\text{in}} = \lfloor d_{\text{in}} \rfloor_r + \cdots + \lfloor d_{\text{in}} \rfloor_r \tag{6.4}$$

$$\vee(cs_{\text{in}}) = s\vee(\lfloor d_{\text{in}} \rfloor_r) = s2^{n-1} \tag{6.5}$$

$$\epsilon(cs_{\text{in}}) = s\epsilon(\lfloor d_{\text{in}} \rfloor_r) = s(2^r - 1) \tag{6.6}$$

cs_{out} elements are comprised of s multiplied pairs of d_{in} and cs_{in}, summed as shown in Equation 6.7. 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 $\epsilon(cs_{\text{out}})$ is found from $\vee$ of each d_{in} and ϵ of each cs_{in}, as shown in Equation 6.8.

$$cs_{\text{out}} = d_{\text{in}}cs_{\text{in}} + \cdots + d_{\text{in}}cs_{\text{in}} \tag{6.7}$$

$$\epsilon(cs_{\text{out}}) = s\vee(d_{\text{in}})\epsilon(cs_{\text{in}}) = s^2(2^r - 1)2^{n-1} \tag{6.8}$$

The $cs_{\text{out, c}}$ element is formed of s multiplied pairs of cs_{in} elements, summed as shown in Equation 6.9. Unlike for each cs_{out} element, therefore, error can be introduced by both of the multiplicands within each product. As a result, the combination of worst-case error from both cs_{in} elements, including their cross-product, within each multiplication must be taken into account when quantifying $\epsilon(cs_{\text{out, c}})$: the result is given in Equation 6.10.

$$cs_{\text{out, c}} = cs_{\text{in}}cs_{\text{in}} + \cdots + cs_{\text{in}}cs_{\text{in}} \tag{6.9}$$

$$\begin{aligned} \epsilon(cs_{\text{out, c}}) &= s\big(\vee(cs_{\text{in}})\epsilon(cs_{\text{in}}) + \epsilon(cs_{\text{in}})\vee(cs_{\text{in}}) + \epsilon(cs_{\text{in}})^2\big) \\ &= s^3(2^r - 1)(2^n + 2^r - 1) \end{aligned} \tag{6.10}$$

6.3 Implementation

6.3.1 Baseline Architecture

The baseline architecture used for comparison in this chapter is structurally identical to that shown in Figure 4.5, less the reconfigurable routing regions. It is shown in Figure 6.1. The principle difference between this architecture and those described in Chapters 3, 4 and 5 is the output data width: while in previous chapters the output data was always assumed to be n-bit, the same as input data, here the output data is expanded to $2n$-bit in order to allow the two truncation types developed in this chapter to be compared.

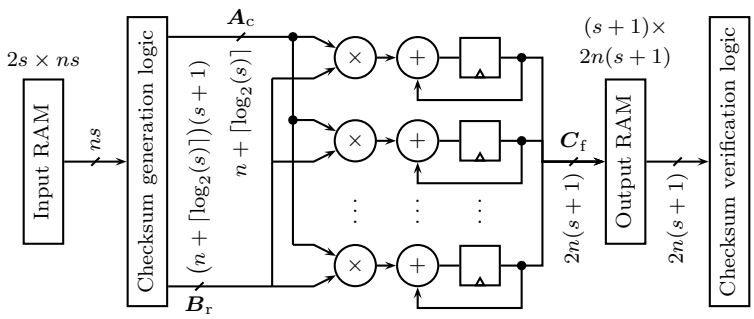


Figure 6.1: Datapath with zero truncation. While in previous chapters the output data was always assumed to be n-bit, the same as input data, here the output data is expanded to $2n$-bit in order to allow the two truncation types developed in this chapter to be compared.

Checksum generation and verification logic, shown in Figures 6.2 and 6.3, respectively, serves to perform the normal ABFT procedures described in Section 3.2.3.

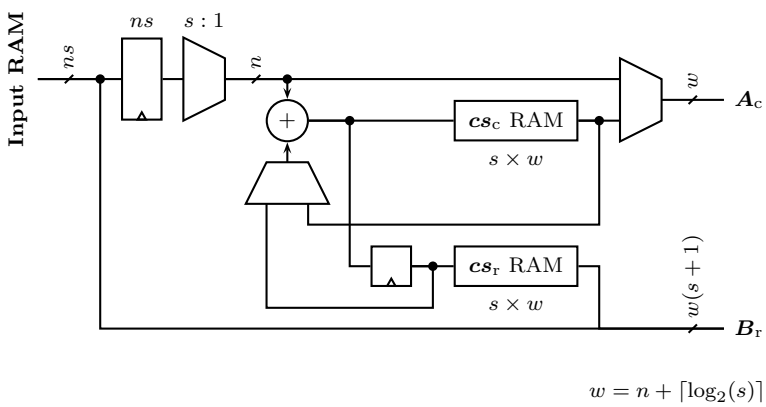


Figure 6.2: Checksum generation logic with zero truncation, used to perform the 'normal' ABFT checksum generation procedures.

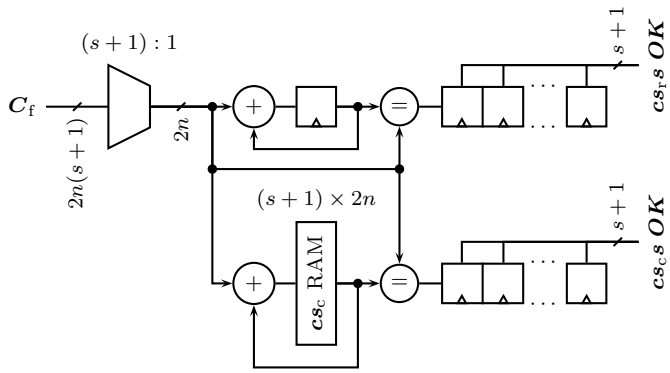


Figure 6.3: Checksum verification logic with zero truncation, used to perform the 'normal' ABFT checksum verification procedures.

6.3.2 MSB-first Truncation

Modifications to the logic shown in Figures 6.2 and 6.3 needed to achieve the MSB-first truncation explained in Section 6.2.1 are straightforward since only bit-widths change. These changes have some knock-on effects on the top-level overview shown in Figure 6.1: now, rather than $n + \log_2(s)$, the per-element paths for $\boldsymbol{A}_\mathrm{c}$ and $\boldsymbol{B}_\mathrm{r}$ only have to be n bits wide and the output taken from the output block random-access memory (BRAM) only has to be $n - r$, rather than $2n$, bits per element. These changes result in the modified datapath shown in Figure 6.4 and the checksum generation and verification logic shown in Figures 6.5 and 6.6, respectively. Note that the signs of all elements are preserved, despite the moniker 'MSB-first truncation.'

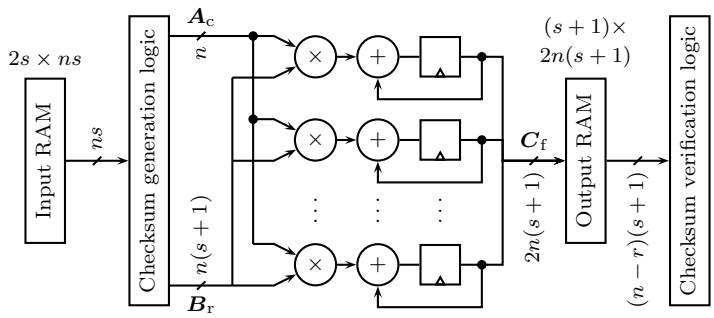


Figure 6.4: Datapath with MSB-first truncation. Rather than $n+\log_2(s)$, the per-element paths for $\boldsymbol{A}_\mathrm{c}$ and $\boldsymbol{B}_\mathrm{r}$ only have to be n bits wide and the output taken from the output BRAM only has to be $n - r$, rather than $2n$, bits per element.

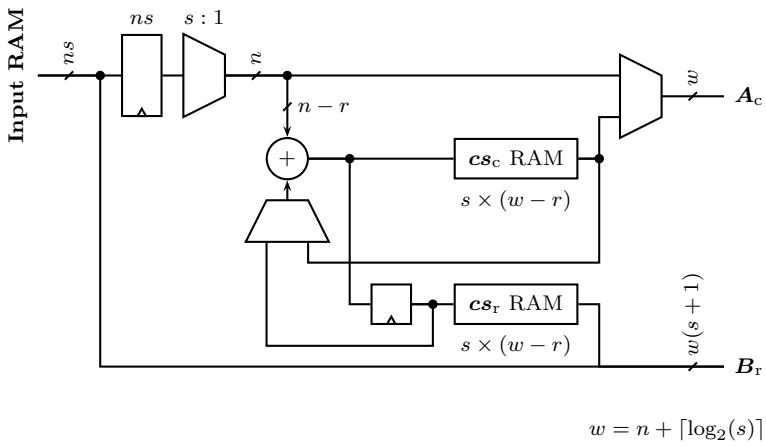


Figure 6.5: Checksum generation logic with MSB-first truncation. Compared to the zero-truncation implementation, only bit-widths are different.

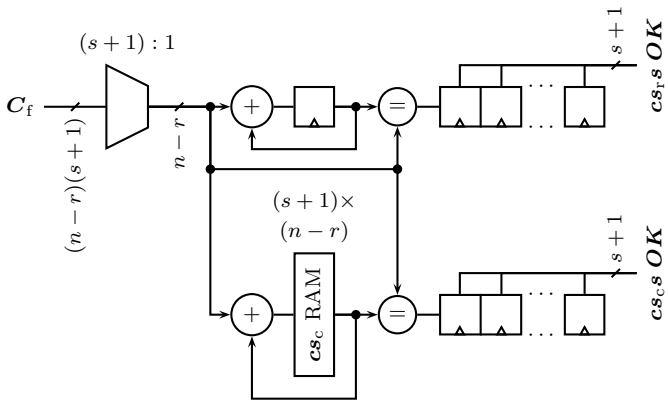


Figure 6.6: Checksum verification logic with MSB-first truncation. Compared to the zero-truncation implementation, only bit-widths are different.

6.3.3 LSB-first Truncation

Achieving the LSB-first truncation described in Section 6.2.2 requires different logic to that shown in Figures 6.2 and 6.3; the blocks shown in Figures 6.8 and 6.9 stand in their places. In terms of changes to the top-level overview, Figure 6.1, the per-element paths for $\boldsymbol{A}_{\mathrm{c}}$ and $\boldsymbol{B}_{\mathrm{r}}$ are $\Big(n+\max\big(\log_2(s)-r,\ 0\big)\Big)$-bit, rather than $\big(n+\log_2(s)\big)$-bit, to optimally fit the single largest data or checksum element. These changes are represented in Figure 6.7.

Output checksum error must be tolerated up to the levels theorised in Equations 6.8 and 6.10 as a result of the truncation performed. Clearly, there is no reason to actually perform the left-shifting shown in the explanation of the truncation procedure; for this reason, the output checksum element error threshold (θ) and corner output checksum ele-

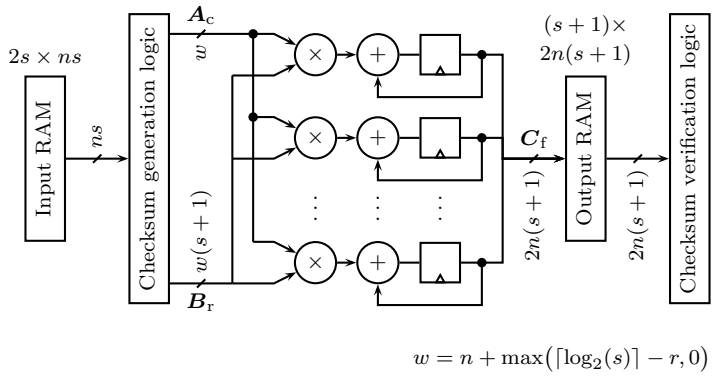


Figure 6.7: Datapath with LSB-first truncation. The per-element paths for $\boldsymbol{A}_{\mathrm{c}}$ and $\boldsymbol{B}_{\mathrm{r}}$ are $\Big(n + \max\big(\log_2(s) - r,\ 0\big)\Big)$-bit, rather than $\big(n + \log_2(s)\big)$-bit, to optimally fit the single largest data or checksum element.

ment error threshold (θ_{c}), shown in Figure 6.9 for cs_{out} and $cs_{\mathrm{out,\ c}}$ elements, respectively, need to be based upon, not equal to, $\epsilon(cs_{\mathrm{out}})$ and $\epsilon(cs_{\mathrm{out,\ c}})$. cs_{out} elements each have their widths reduced by r bits due to the right-shifter shown in Figure 6.9; as a result, θ is calculated as shown in Equation 6.11. $cs_{\mathrm{out,\ c}}$ elements, however, are constructed exclusively from cs_{in} elements: they are therefore subject to magnitude reduction by the right-shifters shown in both Figures 6.8 *and* 6.9. θ_{c} is therefore calculated as shown in Equation 6.12.

$$\theta = \frac{\epsilon(cs_{\mathrm{out}})}{2^r} = \frac{s^2(2^r - 1)2^{n-1}}{2^r} \approx s^2 2^{n-1} \tag{6.11}$$

$$\theta_{\mathrm{c}} = \frac{\epsilon(cs_{\mathrm{out,\ c}})}{2^{2r}} = \frac{s^3(2^r - 1)(2^n + 2^r - 1)}{2^{2r}} \approx s^3 2^{n-r} \tag{6.12}$$

6.4 Overheads

Designs were implemented using Xilinx Vivado 2014.4 across the following range of implementation variables:

- Target device: Xilinx Zynq-7000 XC7Z020.
- s: {2, 4, 8, 16, 32}.
- n (bits): 32 (signed, fixed-point).

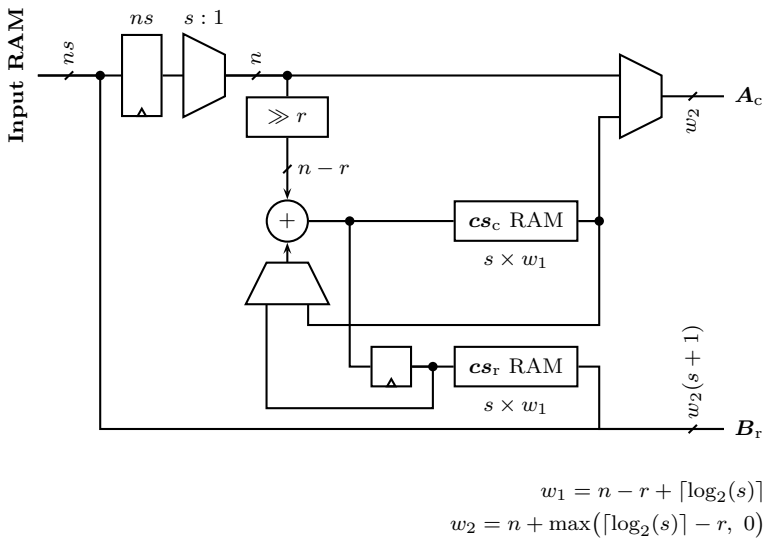


Figure 6.8: Checksum generation logic with LSB-first truncation. Truncation is performed by the r-bit right-shifter shown.

- Truncation type: {none, MSB-first, LSB-first}.
- r (bits): {0, 4, 8, 12, 16, 20, 24}.
- Data memory resource type: BRAM.
- Checksum memory resource type: distributed random-access memory (RAM).
- Multiplier resource type: digital signal processing block (DSP).
- Multiplier latency (m) (cycles): 15.
- Accumulator latency (a) (cycles): 1.

Note that the majority of parameters—s, n, resource types, m and a—were kept the same as those used in the experiments described in Chapters 3 and 4 for the sake of comparison.

6.4.1 Area

Tables 6.1, 6.2, 6.3 and 6.4 contain the raw area utilisation figures obtained for all implementations of the accelerator expressed in terms of look-up tables (LUTs), flip-flops (FFs), BRAMs, DSPs and total (combined) resources used. The latter is the mean of the four preceeding proportions, intended to give an indication as to the overall resource utilisation for a particular design. Absolute and relative (percentage) values are given, the latter indicating proportions of resources used on the target device. Figures 6.10 and 6.11 show the combined area overhead versus the equivalently sized unprotected design, i.e. that without

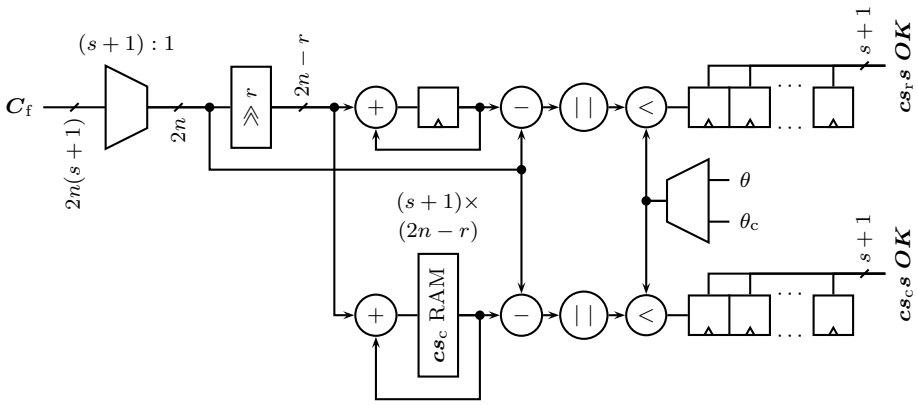


Figure 6.9: Checksum verification logic with LSB-first truncation. Truncation is performed by the r-bit right-shifter shown.

incorporated ABFT, for RP-ABFT with MSB-first truncation, while Figures 6.12 and 6.13 show the same for LSB-first truncation. Note that $r = 0$ in both MSB- and LSB-first cases refers to the same design with ABFT protection but zero truncation applied.

The MSB-first truncation results shown in Figures 6.10 and 6.11 demonstrate clear area gains for this truncation method. Thanks mostly to the severe effect of the output truncation described in Section 6.2.1 for any $r > 0$, a dramatic drop in resource usage is seen after $r = 0$ for any s. In the most extreme case tested, for $s = 32$ and $r = 28$, overhead drops from 15.2% to just 2.53%: an 83.3% reduction. The changes in overhead seen for LSB-first truncation, on the other hand, are more complex: as Figures 6.12 and 6.13 show, overheads initially increase in all cases other than $s = 32$. This is primarily due to the introduction of the subtractors shown in Figure 6.9. Gains are 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 overhead reduction, again for $s = 32$ and $r = 28$, was 23.8%: less than the equivalent MSB-first truncation's, but still not insignificant.

6.4.2 Performance

For the same set of designs, the reported timing model-inferred maximum operating frequency ($f_{\max}$) was also recorded. The raw results obtained are detailed in Table 6.5, with changes versus the equivalently sized unprotected designs for MSB- and LSB-first truncation, in the same style as those produced for area in Figures 6.10 and 6.12, shown in Figures 6.14 and 6.15, respectively. To overcome the effects of computer-aided design (CAD) noise, trendlines are included for each plot, shown as dashed lines. For the

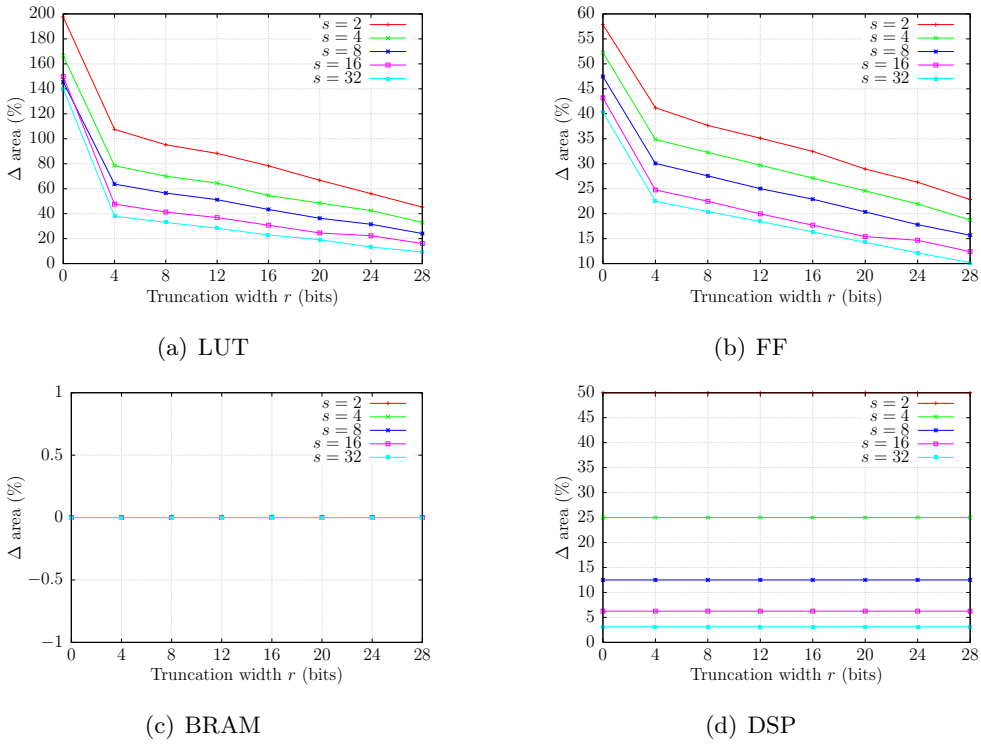


Figure 6.10: RP-ABFT with MSB-first truncation-protected accelerator resource usage overhead versus baseline, showing the change in individual resource utilisations for each design versus its equivalently sized unprotected, baseline implementation.

MSB-first truncation plots shown in Figure 6.14, these begin at $r = 4$ due to the discontinuities expected prior to that value. Note that neither the type of RP-ABFT nor r affects the (clock cycle) latency of a design versus its standard ABFT equivalent, allowing $f_{\max}$ to be used for performance comparison directly.

Due primarily to the wide (64-bit) adders needed to enable checksum verification, $f_{\max}$ reductions are significant in the zero truncation case: for $s = 32$, $f_{\max}$ drops by 40.8%. Such penalties are practically eliminated, and in a few cases become net gains, through the use of MSB-first truncation; this is shown in Figure 6.14. The large jump seen for all s to near-zero is due to the elimination of at least 50% of the output bits during checksum verification explained in Section 6.2.1. To make this point clearer, Table 6.6 lists the input and output checksum widths for each of the designs compiled. Recall that $n = 32$ in all cases. As can be seen, the verification logic's data width decreases from 64 to just 28 bits for $r = 4$, and reduces further thereafter: the regression lines demonstrate small but fairly consistent gains. LSB-first truncation designs exhibit relatively small performance

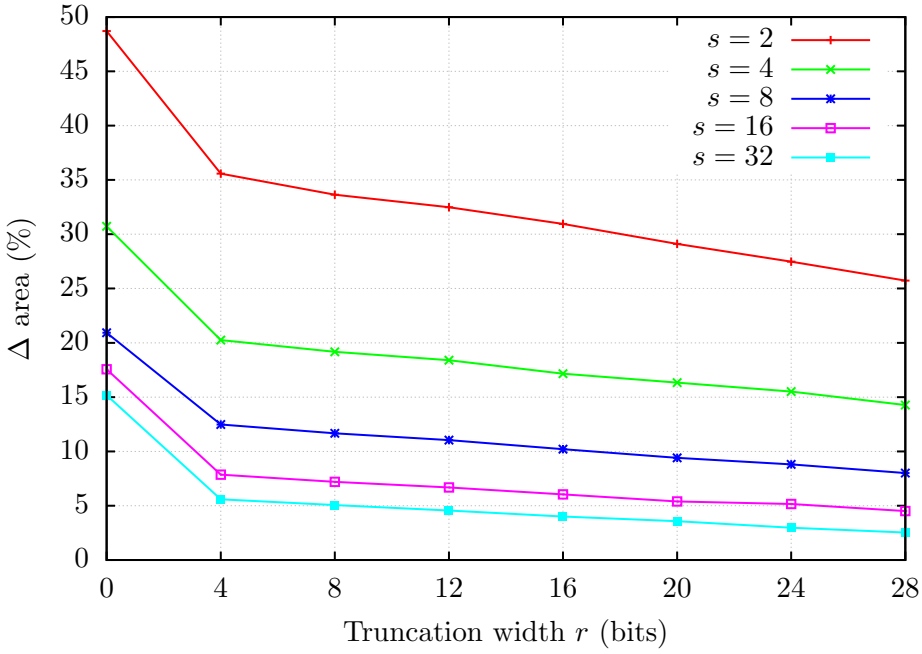


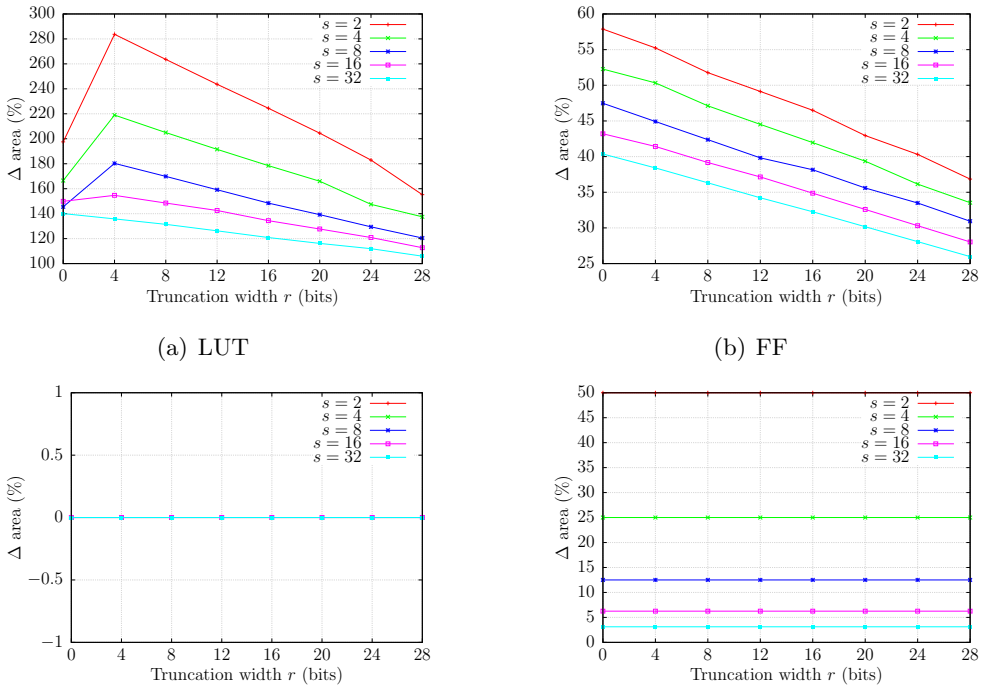
Figure 6.11: RP-ABFT with MSB-first truncation-protected accelerator combined resource usage overhead versus baseline, showing the change in combined resource utilisation for each design versus its equivalently sized unprotected, baseline implementation.

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, as shown in Table 6.6, and the introduction of additional logic as described in Section 6.4.1. Nevertheless, frequency gains were realised for larger designs, with the $s = 32$ case showing increasing gains for each value of r tested but the last.

6.5 Fault Observability

Functional simulations were performed in software to assess the fault observability of the proposed designs across the following range of variables:

- Fault type: {permanent, transient}.
- s: {2, 4, 8, 16, 32}.
- n (bits): {2, 4, 8, 16, 32}.
- Truncation type: {none, MSB-first, LSB-first}.



(a) LUT (b) FF

(c) BRAM (d) DSP

Figure 6.12: RP-ABFT with LSB-first truncation-protected accelerator resource usage overhead versus baseline, showing the change in individual resource utilisations for each design versus its equivalently sized unprotected, baseline implementation.

- r (bits): {0, 4, 8, 12, 16, 20, 24}.

For each combination of these, the steps detailed in Algorithm 12 were repeated 1,024,000 times, with results averaged thereafter.

In all cases, the fault model applied was that of individually targetted stuck-at-one (SA1) accumulator output bits. For permanent faults, such SA1s are representative of, for example, worn transistors or bridged interconnects, while they mimic effects including register and memory upsets in the case of transients.

Results gleaned from this testing were, as is the case in the all of the fault observability testing performed in this thesis, independent of fault rate, area and latency: they demonstrate the *proportions of total accelerator executions* that should be expected to result in particular classes of outputs under *fixed fault conditions*. They cannot be used to directly ascertain the expected rates of certain output classes' occurrence, although this can be achieved by scaling the proportions to take fault rate, area and/or latency, as required, into account.

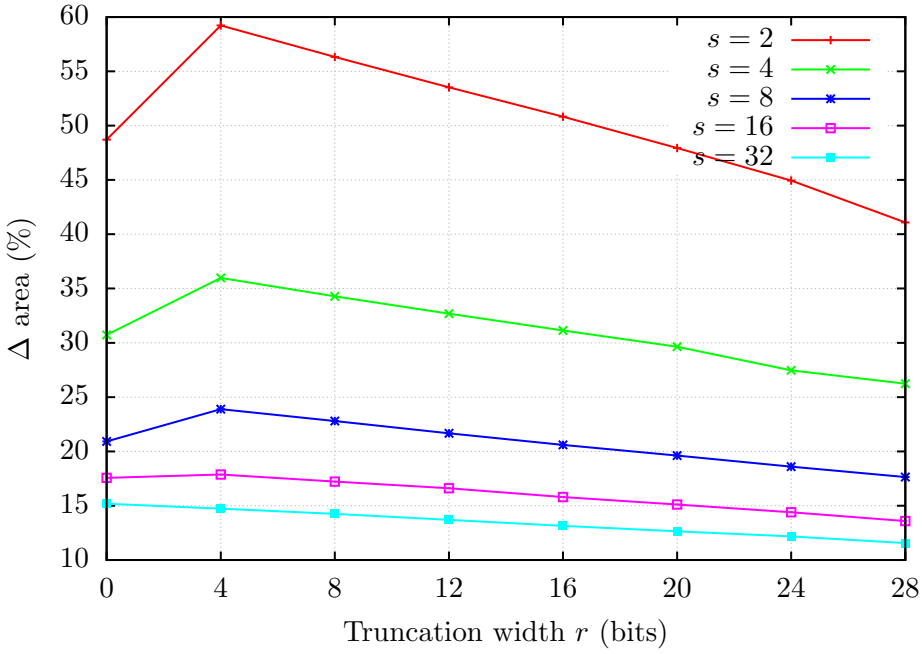


Figure 6.13: RP-ABFT with LSB-first truncation-protected accelerator combined resource usage overhead versus baseline, showing the change in combined resource utilisation for each design versus its equivalently sized unprotected, baseline implementation.

Algorithm 12 is largely similar to that used for matrix-matrix multiplication with emulated fault injection described in Section 5.2 and detailed in Algorithm 7. Changes were made to suit the introduction of r, and distance (d) was fixed at 1 since only a single row or column of checksumming was used within the hardware described in Section 6.3. Checksumming is added to $\boldsymbol{A}$ (Line 9) and $\boldsymbol{B}$ (Line 10) to form $\boldsymbol{A}_{\mathrm{c}}$ and $\boldsymbol{B}_{\mathrm{r}}$ as described in Section 6.2.1 (for MSB-first truncation) or Section 6.2.2 (for LSB-first) via procedure call `add_cs()`. When multiply-accumulation takes place (Line 18 for permanent fault injection; Line 28 for transient), each step is performed modulo-2^{2n}, rather than 2^n, as explained in Section 6.3.1. Following computation, procedures `check_cs()` (Line 34) and `diagnose_cs()` (Line 36) are called. While equivalent in purpose, these are somewhat more complex than their equivalents in Algorithm 7 since they have to be capable of operating with both truncation types.

Procedure `add_cs()` is described in Algorithm 13. For column-wise checksum generation, starting from Line 1, a new matrix is provisioned (Line 2) with an additional row for its checksums, its uppermost s rows being copied from the source matrix (Line 3). For

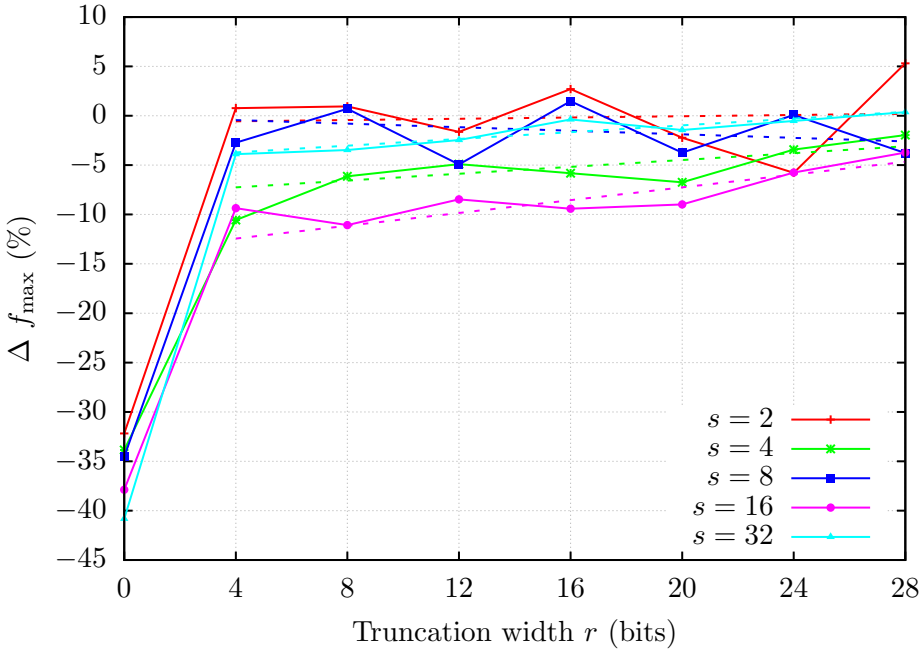


Figure 6.14: RP-ABFT with MSB-first truncation-protected accelerator $f_{\max}$ versus baseline, showing the change in execution time for each design versus its equivalently sized unprotected, baseline implementation. Trendlines are included to counter CAD noise.

each column, the appropriate checksum element is zeroed (Line 5) and then accumulated into, once per row of the source matrix, depending on the type of truncation required as specified by variable *truncation_type*. If no truncation is needed, accumulations are carried out modulo-2^{2n} (Line 8). For MSB-first truncation, they are instead performed modulo-2^{n-r}, as explained in Section 6.2.1, while for LSB-first truncation each data element of the source matrix is right-shifted by r bits prior to each accumulation (Line 12), as explained in Section 6.2.2. To prevent overflow, no modulus is used with LSB-first truncation. One of the reasons Python was chosen as the language with which to implement the framework was its default use of arbitrary-precision integers [76], thus preventing transparent overflow. For row-wise checksum generation (from Line 17), the steps are identical to those used for column-wise generation but with row and column indices reversed. Full checksum generation (from Line 33) is achieved by performing the steps for both column- and row-wise checksumming. The order this is completed in is irrelevant.

Algorithm 14 details the steps performed in procedure `check_cs()`. To verify the checksums, those present within $\boldsymbol{C}_{\mathrm{f}}$ are removed, leaving only data elements, when forming

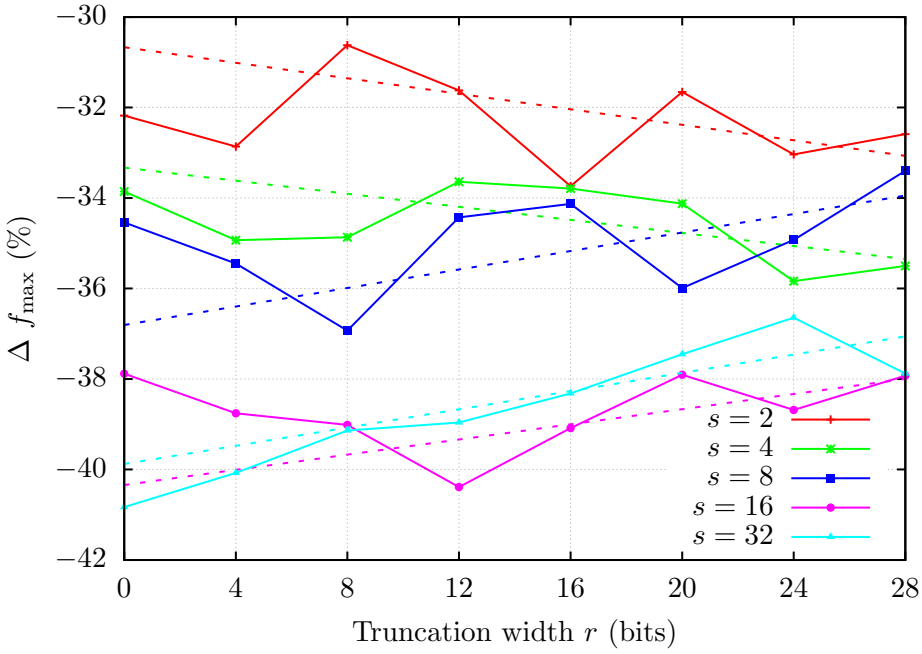
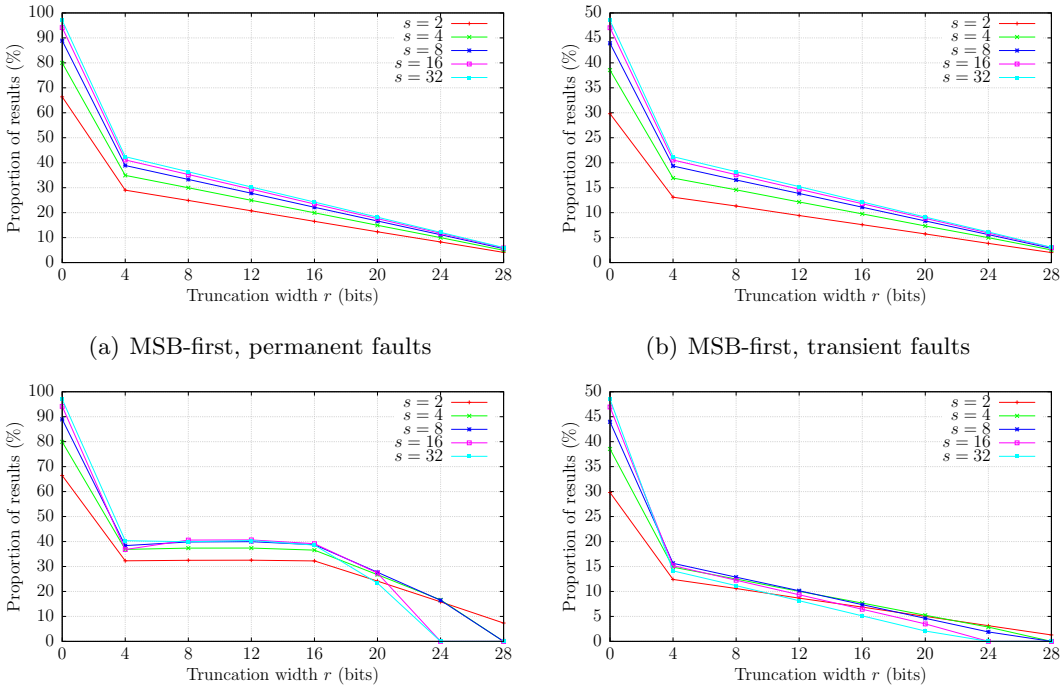


Figure 6.15: RP-ABFT with LSB-first truncation-protected accelerator $f_{\max}$ versus baseline, showing the change in execution time for each design versus its equivalently sized unprotected, baseline implementation. Trendlines are included to counter CAD noise.

$\boldsymbol{C}_{\text{copy}}$ (Line 2). $\boldsymbol{C}_{\text{copy}}$ is expanded through the addition of both column- and row-wise checksums into matrix $\boldsymbol{C}_{\text{copy, f}}$ (Line 4). Column- and row-wise checksums within $\boldsymbol{C}_{\text{f}}$ and $\boldsymbol{C}_{\text{copy, f}}$ are then compared in turn. Vectors $\boldsymbol{cs}$ and $\boldsymbol{cs}_{\text{copy}}$, containing the appropriate checksum elements from $\boldsymbol{C}_{\text{f}}$ and $\boldsymbol{C}_{\text{copy, f}}$, respectively, are created (Lines 8 and 9). If *truncation_type* indicates that no truncation was performed, $\boldsymbol{cs}$ and $\boldsymbol{cs}_{\text{copy}}$ are simply compared (Line 11). For MSB-first truncation, elements of $\boldsymbol{cs}$ are compared with those in $\boldsymbol{cs}_{\text{copy}}$ after their top $n + r$ bits have been discarded (Line 16); checksumming is blind to these bits. For LSB-first truncation, absolute values of differences between checksum elements are computed and compared to error values θ (Line 21, for non-corner elements) or θ_{c} (Line 22, for the corner element), as derived in Section 6.3.3 and shown in Equations 6.11 and 6.12, respectively, as appropriate. Element-wise comparison is performed either modulo-2^{2n-r} (Line 24, for non-corner elements) or modulo-$2^{2(n-r)}$ (Line 28, for the corner element) to account for the number of bits remaning post-right shifting. For non-corner elements this is $2n - r$ since right-shifting by r bits is performed once during their computation, while for the corner element it is $2(n - r)$ since r-bit right-shifting

happens twice.

Figures 6.16, 6.17, 6.18 and 6.19 respectively show the results of this testing for the proportions of detected, false positive, false negative and masked faults for each combination of truncation and fault type.



(a) MSB-first, permanent faults

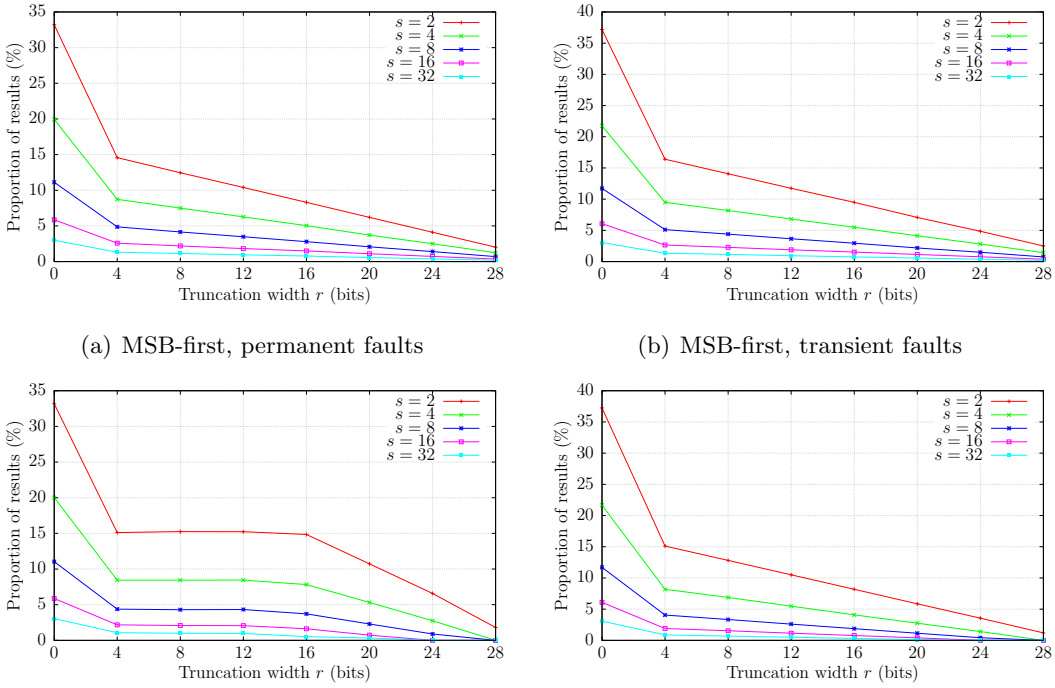
(b) MSB-first, transient faults

(c) LSB-first, permanent faults

(d) LSB-first, transient faults

Figure 6.16: Detected fault proportions for RP-ABFT-protected accelerator. Each plot shows the proportion of total accelerator execution's results that led to a successfully detected output for a particular combination of truncation type, s and r.

In the broadest sense, the results show that it becomes increasingly common for faults to be masked or missed entirely (false negative) than detected or flagged erroneously (false positive) as r grows in all cases. Shapes and proportions of results falling into each of the four result categories are largely the same for both MSB- and LSB-first truncation since the error thresholds, explained in Sections 6.2.2 and 6.3.3, used for LSB-first truncation are not considered when comparing actual and expected output data. Transients exhibit around half the likelihood of detection due to the fact that they are expected to be masked in 50% of occasions as a result of the uniformity of the input data. Results tend to be more positive for larger s since the likelihood of faults being obscured by all surrounding data decreases as the quantity of that data increases.



(a) MSB-first, permanent faults

(b) MSB-first, transient faults

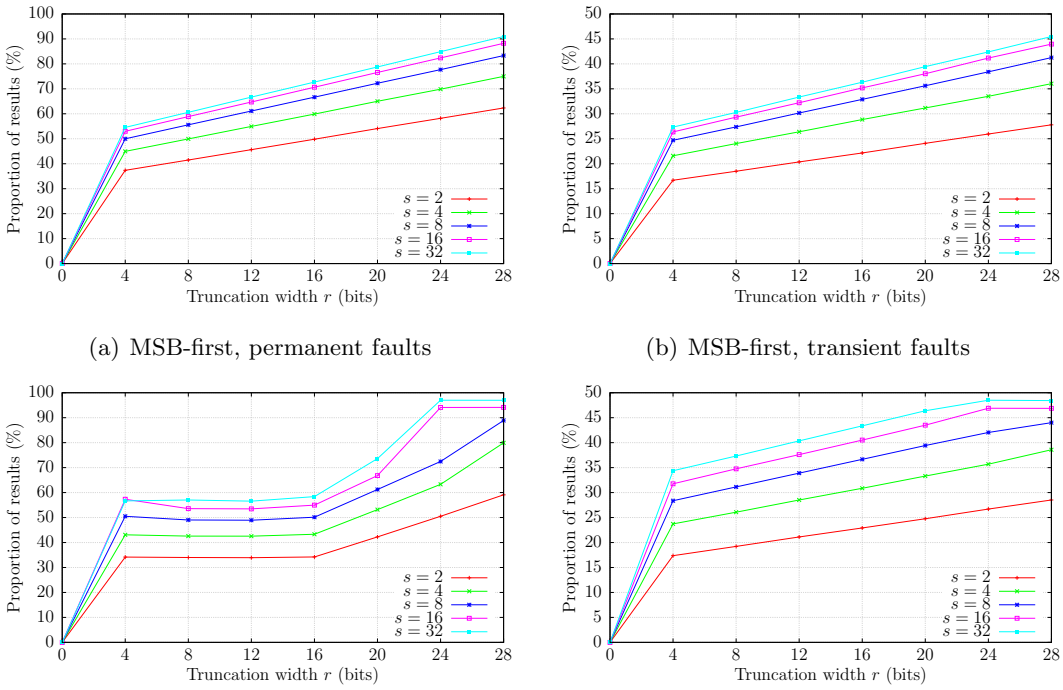
(c) LSB-first, permanent faults

(d) LSB-first, transient faults

Figure 6.17: False positive fault proportions for RP-ABFT-protected accelerator. Each plot shows the proportion of total accelerator execution's results that led to a false positive output for a particular combination of truncation type, s and r.

Perhaps the most interesting feature of these results is the shape of the plots seen for LSB-first truncation under permanent fault injection. Similarly to other plots, jumps from desirable (high for detected, low for false negative, etc.) to undesirable proportions of each result type are seen between $r = 0$ and $r > 0$, reflecting the inability to detect low-magnitude errors that begins at $r = 1$. After this, however, plots remain largely flat until $r = 16$, trending in the same directions as they did to begin with thereafter. It is around this second inflection that the detection logic starts to become ineffective. Consider $s = 32$ in Equation 6.10. Setting $\epsilon(cs_{\text{out, c}}) = 2^{63}$, i.e. $\vee(d_{\text{out}})$ for $n = 32$, reveals that at $r \approx 16$ corner checksums cease to be effective. Similarly, setting $\epsilon(cs_{\text{out}}) = 2^{63}$ in Equation 6.8 for the same s and n shows that *all* checksumming is rendered useless at $r \approx 22$.

The ability to locate faults, represented by the plots shown in Figure 6.20, is important for applications requiring correction as well as detection. The accelerator relies upon column checksum information to infer fault location since, as explained in Section 3.2.4, output matrix columns have a one-to-one mapping to the multiply-accumulators (MACs)



(a) MSB-first, permanent faults

(b) MSB-first, transient faults

(c) LSB-first, permanent faults

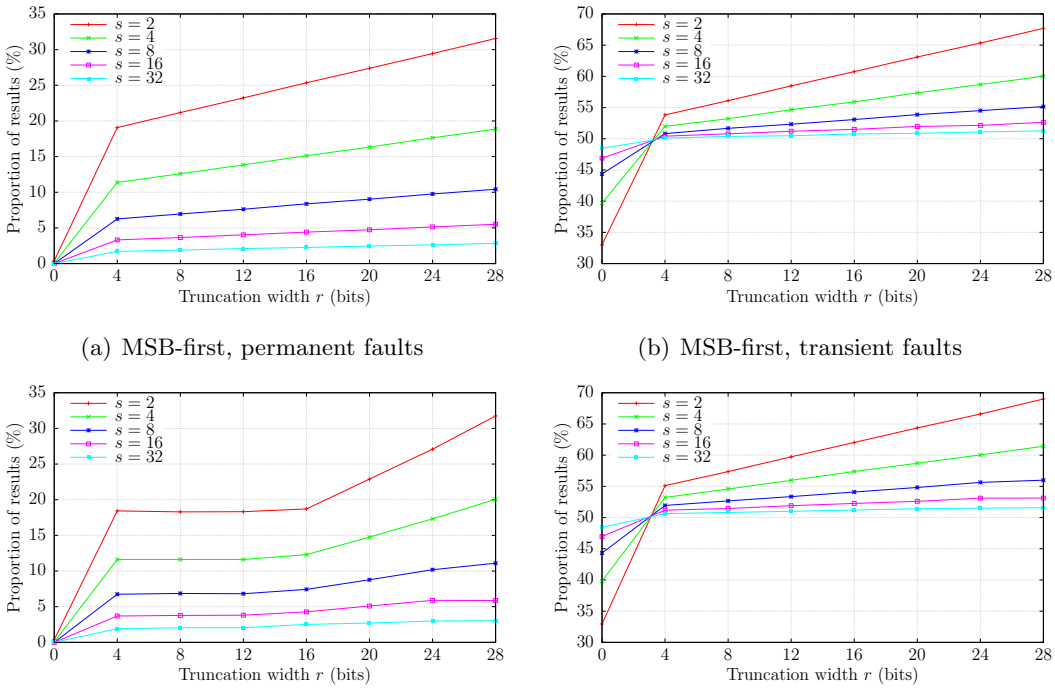
(d) LSB-first, transient faults

Figure 6.18: False negative fault proportions for RP-ABFT-protected accelerator. Each plot shows the proportion of total accelerator execution's results that led to a false negative output for a particular combination of truncation type, s and r.

used to compute their elements. As for the previously discussed result classes, the results show discontinuities—due to the harsh initial output truncation or introduction of error bounding in MSB- and LSB-first truncation, respectively—followed by steady decreases as r rises. Instances of masking were ignored when determining locatable faults since masked results are, by definition, unlocatable.

Results flagged as false negatives were analysed to determine the magnitude of allowed error in the case of faults being missed. In each of those cases, the maximum absolute error of each of the output matrix's data elements was calculated, and Figure 6.21 shows the means of those results. Assuming that unmissed results are able to be corrected, Figure 6.21's results therefore represent the average expected worst-element errors introduced by RP-ABFT.

The results indicate that MSB-first truncation offers no tradeoff between r and allowed error: the latter immediately jumps close to $\vee(d_{\text{out}})$ and remains flat thereafter; expected due to the fact that worst-case errors are introduced as soon as $r \neq 0$. While MSB-



(a) MSB-first, permanent faults

(b) MSB-first, transient faults

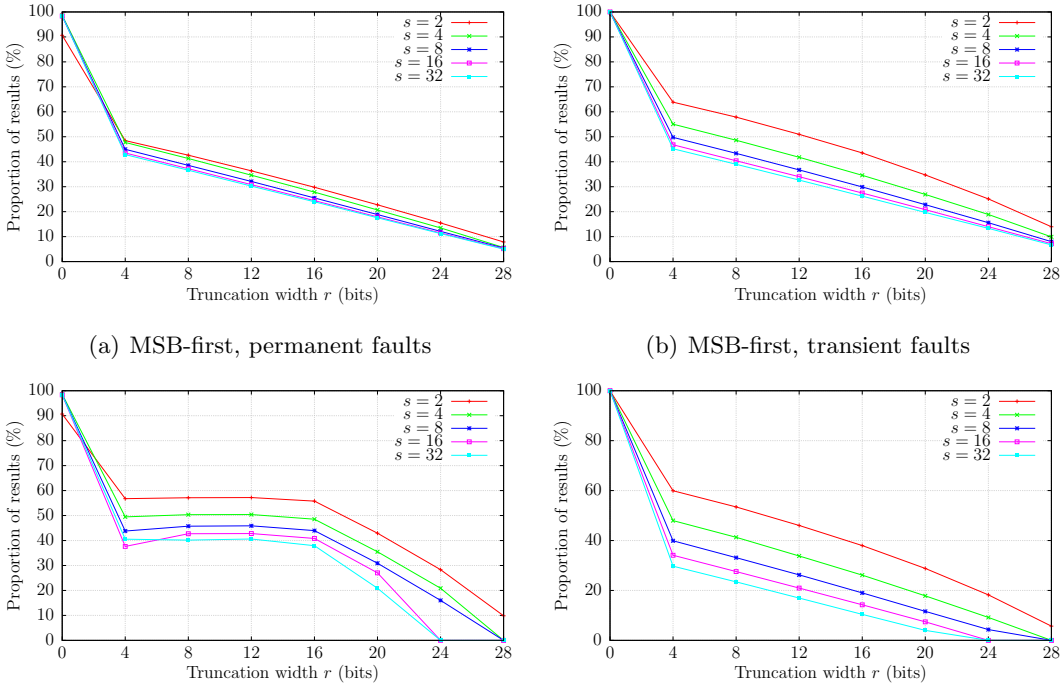
(c) LSB-first, permanent faults

(d) LSB-first, transient faults

Figure 6.19: Masked fault proportions for RP-ABFT-protected accelerator. Each plot shows the proportion of total accelerator execution's results that led to a masked output for a particular combination of truncation type, s and r.

first truncation necessitates lower overhead incursion than either its LSB-first equivalent or traditional ABFT, the results confirm that its effectiveness is severely limited when considering allowed output error; although, for the largest r, the same is also true of LSB-first truncation. RP-ABFT with LSB-first truncation does, however, allow for area (and consequently power) and performance improvements while allowing relatively small errors to pass. Although the mean results may seem high, recall that faults were emulated within every simulation iteration, and that input data was drawn from a uniform distribution.

The results shown in Figures 6.16, 6.17, 6.18, 6.19 and 6.20 do not take area into account. Additional plots, shown in Figures 6.22, 6.23, 6.24, 6.25 and 6.26, were therefore produced to attempt to capture the effect of area upon likelihood of fault manifestation, thereby scaling the previously seen fault proportions. Both s and r have an impact upon area. For both variables, total RP-ABFT-enabled resource usage figures from Table 6.4 were used to scale the results, with $s = 2$, $r = 0$ taken as the baseline. Detected and located fault proportions were scaled *down* as area increased, while false positive, false negative and masked fault proportions were scaled *up*.



(a) MSB-first, permanent faults

(b) MSB-first, transient faults

(c) LSB-first, permanent faults

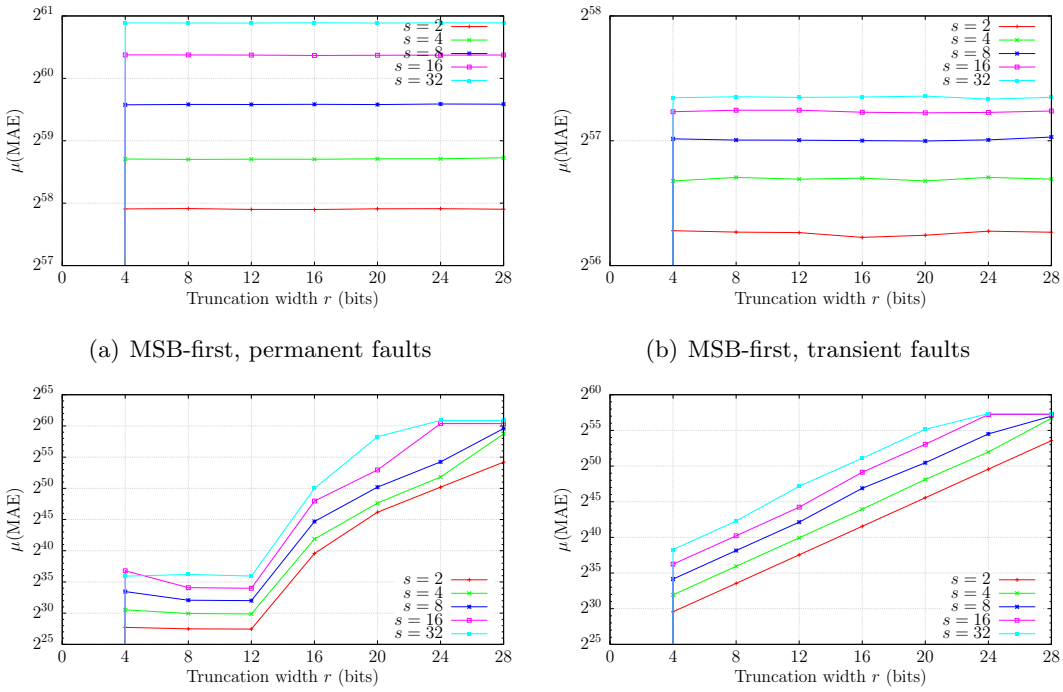
(d) LSB-first, transient faults

Figure 6.20: Located fault proportions for RP-ABFT-protected accelerator. Each plot shows the proportion of total accelerator execution's results that led to a successfully locatable output for a particular combination of truncation type, s and r.

The area-scaled plots in Figures 6.22, 6.23, 6.24, 6.25 and 6.26 unfortunately show undesirable behaviour across the truncation types, fault models and r tested for changes in s. Due to the increasing likelihood of false positive, false negative and masked faults occurring as r grows, and the increasing expected fault rate as s rises due to additional area utilisation, fault detectability and locatability are both expected to erode substantially as s increases for either MSB- or LSB-first truncation experiencing either permanent or transient faults.

6.6 Conclusion

This chapter introduced a new control within ABFT allowing for the reduction of precision within checksumming components. The resulting protection is referred to as RP-ABFT. The design, implementation and evaluation of an RP-ABFT-protected hardware accelerator were described: the mathematical principles of RP-ABFT were presented first, based



(a) MSB-first, permanent faults

(b) MSB-first, transient faults

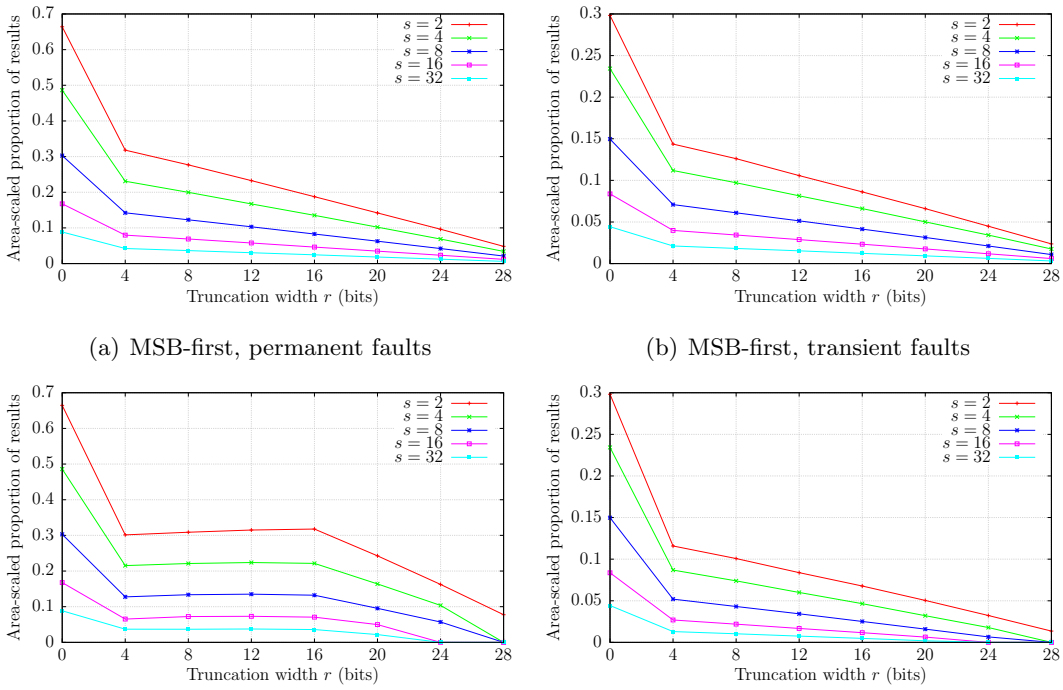
(c) LSB-first, permanent faults

(d) LSB-first, transient faults

Figure 6.21: Means of maximum absolute errors for RP-ABFT-protected accelerator. In each case of a false negative result, the maximum absolute error of each of the output matrix's data elements was calculated: this figure shows the means of those results. Assuming that unmissed results are able to be corrected, these plots' results therefore represent the average expected worst-element errors introduced by RP-ABFT.

upon which informed decisions were made regarding the truncation of parts of the standard ABFT protection circuitry needed for the chosen operator. Two distinct versions of RP-ABFT, involving MSB- and LSB-first truncation, were theorised and implemented in hardware. Experiments were conducted to ascertain the implications of RP-ABFT has upon area and performance overheads, and fault injection simulations were carried out to evaluate the fault observability of the developed protection mechanism.

The findings herein 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 $f_{\max}$ versus the equivalent 'vanilla' ABFT protection 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.



(a) MSB-first, permanent faults

(b) MSB-first, transient faults

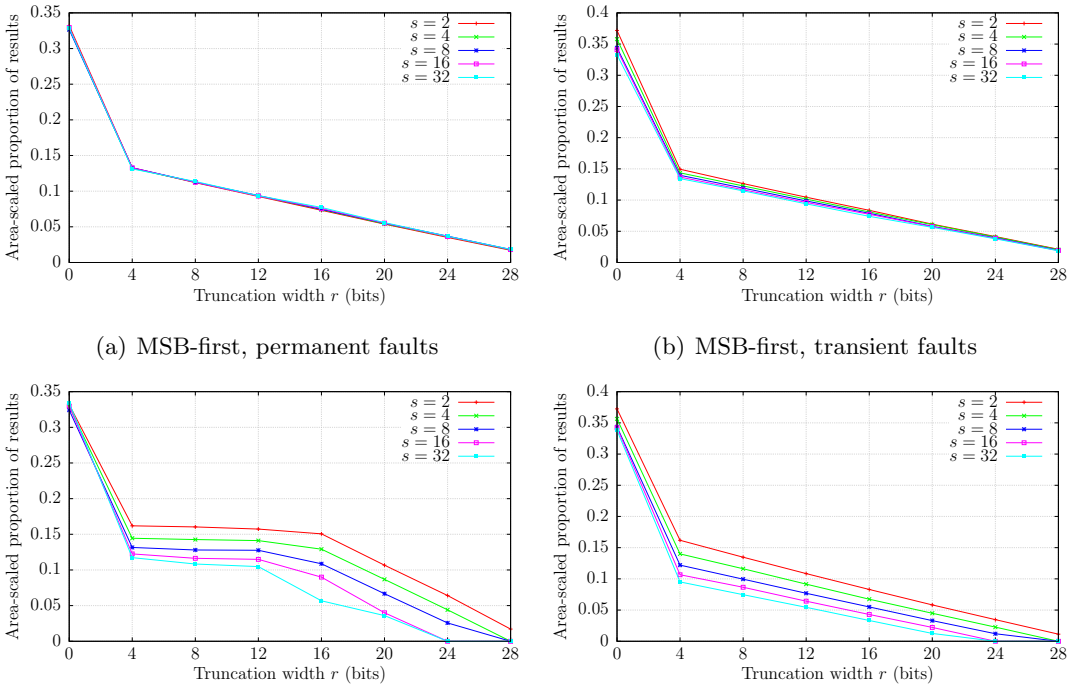
(c) LSB-first, permanent faults

(d) LSB-first, transient faults

Figure 6.22: Area-scaled detected fault proportions for RP-ABFT-protected accelerator, with total resource usage figures used to scale for s and linear scaling used for n.

6.6.1 Future Work

Several possible areas of future research regarding RP-ABFT have been identified. Through the manipulation of output error threshold values, tradeoffs between false positives and false negatives are achievable. Enhancements to the checksumming logic, particularly for performance improvement, are also possible, along with the exploration of output-only truncation to introduce additional tradeoff opportunities between overhead and maximum allowed output error. Finally, the feasability of hybrid RP-ABFT-online arithmetic implementations could be explored: by combining LSB-first truncation RP-ABFT with existing work on online arithmetic [77] [78]—operations whose results settle from MSB first—it is believed to be possible to create robust, overclocking-friendly circuitry with self-verifying properties.



(a) MSB-first, permanent faults

(b) MSB-first, transient faults

(c) LSB-first, permanent faults

(d) LSB-first, transient faults

Figure 6.23: Area-scaled false positive fault proportions for RP-ABFT-protected accelerator, with total resource usage figures used to scale for s and linear scaling used for n.

ABFT enabled	Truncation type	Truncation width r (bits)	Matrix size s 2	4	8	16	32
✗			692 (1.30%)	1064 (2.00%)	1788 (3.36%)	3336 (6.27%)	6655 (12.51%)
✓	None		2059 (3.87%)	2836 (5.33%)	4389 (8.25%)	8336 (15.67%)	15976 (30.03%)
	MSB-first	4	1436 (2.70%)	1899 (3.57%)	2926 (5.50%)	4926 (9.26%)	9182 (17.26%)
		8	1351 (2.54%)	1809 (3.40%)	2798 (5.26%)	4714 (8.86%)	8852 (16.64%)
		12	1303 (2.45%)	1750 (3.29%)	2703 (5.08%)	4565 (8.58%)	8539 (16.05%)
		16	1234 (2.32%)	1644 (3.09%)	2564 (4.82%)	4362 (8.20%)	8182 (15.38%)
		20	1154 (2.17%)	1580 (2.97%)	2437 (4.58%)	4155 (7.81%)	7927 (14.90%)
		24	1080 (2.03%)	1516 (2.85%)	2351 (4.42%)	4080 (7.67%)	7538 (14.17%)
		28	1005 (1.89%)	1415 (2.66%)	2218 (4.17%)	3873 (7.28%)	7272 (13.67%)
	LSB-first	4	2655 (4.99%)	3394 (6.38%)	5011 (9.42%)	8496 (15.97%)	15694 (29.50%)
		8	2516 (4.73%)	3245 (6.10%)	4825 (9.07%)	8289 (15.58%)	15401 (28.95%)
		12	2378 (4.47%)	3102 (5.83%)	4634 (8.71%)	8092 (15.21%)	15056 (28.30%)
		16	2245 (4.22%)	2963 (5.57%)	4442 (8.35%)	7820 (14.70%)	14699 (27.63%)
		20	2107 (3.96%)	2830 (5.32%)	4277 (8.04%)	7597 (14.28%)	14385 (27.04%)
		24	1958 (3.68%)	2633 (4.95%)	4102 (7.71%)	7368 (13.85%)	14103 (26.51%)
		28	1766 (3.32%)	2527 (4.75%)	3942 (7.41%)	7097 (13.34%)	13704 (25.76%)

Table 6.1: Baseline, MSB-first & LSB-first truncated accelerator LUT usage, containing the raw resource usage figures obtained for all implementations. Percentages of the total number of each of these resources for the target device are also included.

ABFT enabled	Truncation type	Truncation width r (bits)	Matrix size s 2	4	8	16	32
✗			1213 (1.14%)	1649 (1.55%)	2511 (2.36%)	4213 (3.96%)	7618 (7.16%)
✓	None		1915 (1.80%)	2511 (2.36%)	3703 (3.48%)	6033 (5.67%)	10693 (10.05%)
	MSB-first	4	1713 (1.61%)	2224 (2.09%)	3266 (3.07%)	5256 (4.94%)	9331 (8.77%)
		8	1670 (1.57%)	2181 (2.05%)	3203 (3.01%)	5160 (4.85%)	9172 (8.62%)
		12	1639 (1.54%)	2139 (2.01%)	3139 (2.95%)	5054 (4.75%)	9023 (8.48%)
		16	1607 (1.51%)	2096 (1.97%)	3086 (2.90%)	4958 (4.66%)	8863 (8.33%)
		20	1564 (1.47%)	2054 (1.93%)	3022 (2.84%)	4862 (4.57%)	8704 (8.18%)
		24	1532 (1.44%)	2011 (1.89%)	2958 (2.78%)	4831 (4.54%)	8544 (8.03%)
		28	1490 (1.40%)	1958 (1.84%)	2905 (2.73%)	4735 (4.45%)	8395 (7.89%)
	LSB-first	4	1883 (1.77%)	2479 (2.33%)	3639 (3.42%)	5958 (5.60%)	10544 (9.91%)
		8	1841 (1.73%)	2426 (2.28%)	3575 (3.36%)	5863 (5.51%)	10385 (9.76%)
		12	1809 (1.70%)	2383 (2.24%)	3511 (3.30%)	5778 (5.43%)	10225 (9.61%)
		16	1777 (1.67%)	2341 (2.20%)	3469 (3.26%)	5682 (5.34%)	10076 (9.47%)
		20	1734 (1.63%)	2298 (2.16%)	3405 (3.20%)	5586 (5.25%)	9916 (9.32%)
		24	1702 (1.60%)	2245 (2.11%)	3352 (3.15%)	5490 (5.16%)	9757 (9.17%)
		28	1660 (1.56%)	2202 (2.07%)	3288 (3.09%)	5394 (5.07%)	9597 (9.02%)

Table 6.2: Baseline, MSB-first & LSB-first truncated accelerator FF usage, containing the raw resource usage figures obtained for all implementations. Percentages of the total number of each of these resources for the target device are also included.

ABFT enabled	Resource	Matrix size s 2	4	8	16	32
✗	BRAM	12 (4.29%)	24 (8.57%)	48 (17.14%)	96 (34.29%)	192 (68.57%)
	DSP	8 (3.64%)	16 (7.27%)	32 (14.55%)	64 (29.09%)	128 (58.18%)
✓	BRAM	12 (4.29%)	24 (8.57%)	48 (17.14%)	96 (34.29%)	192 (68.57%)
	DSP	12 (5.45%)	20 (9.09%)	36 (16.36%)	68 (30.91%)	132 (60.00%)

Table 6.3: Baseline, MSB-first & LSB-first truncated accelerator BRAM & DSP usage, containing the raw resource usage figures obtained for all implementations. Percentages of the total number of each of these resources for the target device are also included.

ABFT enabled	Truncation type	Truncation width r (bits)	Matrix size s 2	4	8	16	32
✗			2.59%	4.85%	9.35%	18.40%	36.61%
✓	None		3.85%	6.34%	11.31%	21.64%	42.16%
	MSB-first	4	3.51%	5.83%	10.52%	19.85%	38.65%
		8	3.46%	5.78%	10.44%	19.73%	38.46%
		12	3.43%	5.74%	10.38%	19.63%	38.28%
		16	3.39%	5.68%	10.31%	19.52%	38.07%
		20	3.35%	5.64%	10.23%	19.40%	37.91%
		24	3.30%	5.60%	10.18%	19.35%	37.69%
		28	3.26%	5.54%	10.10%	19.23%	37.53%
	LSB-first	4	4.13%	6.59%	11.59%	21.69%	42.00%
		8	4.05%	6.51%	11.48%	21.57%	41.82%
		12	3.98%	6.43%	11.38%	21.46%	41.62%
		16	3.91%	6.36%	11.28%	21.31%	41.42%
		20	3.83%	6.29%	11.19%	21.18%	41.23%
		24	3.76%	6.18%	11.09%	21.05%	41.06%
		28	3.66%	6.12%	11.00%	20.90%	40.84%

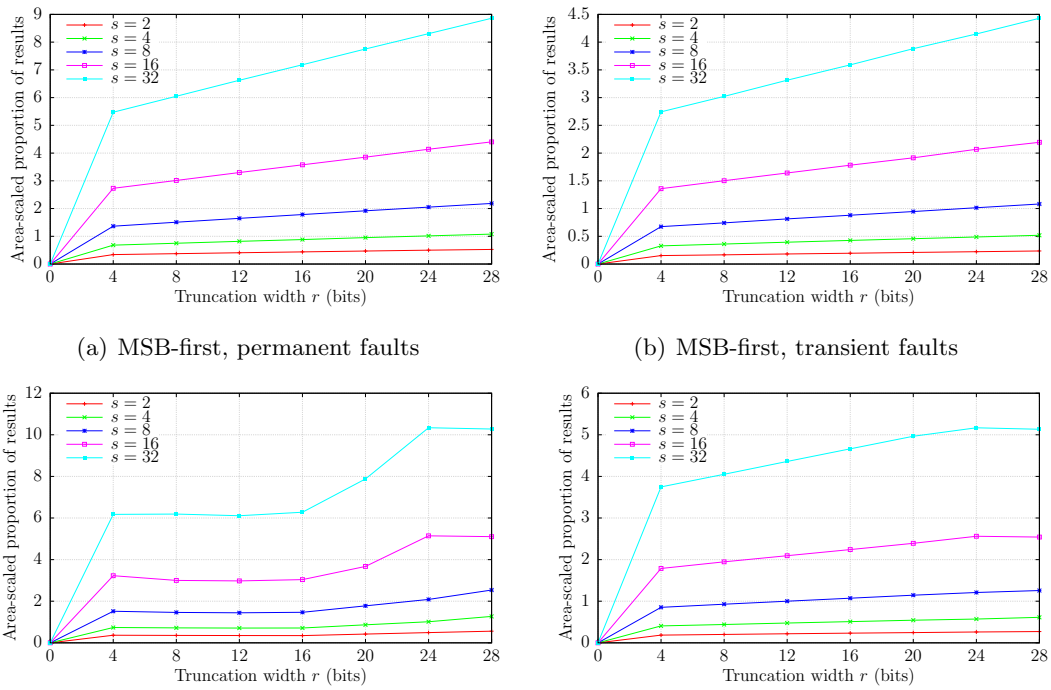
Table 6.4: Baseline, MSB-first & LSB-first truncated accelerator total resource usage, containing the means of proportional resource usage figures obtained for all implementations to give an indication of overall resource utilisations.

ABFT enabled	Truncation type	Truncation width r (bits)	Matrix size s 2	4	8	16	32
			$f_{\max}$ (MHz)				
✗			117.10	118.72	113.78	114.29	102.37
✓	None		79.42	78.52	74.48	71.00	60.57
	MSB-first	4	118.00	106.17	110.67	103.60	98.39
		8	118.20	111.45	114.58	101.63	98.81
		12	115.18	112.90	108.17	104.62	99.86
		16	120.27	111.80	115.46	103.54	101.98
		20	114.48	110.72	109.53	104.02	100.89
		24	110.33	114.64	113.87	107.72	101.84
		28	123.31	116.39	109.48	110.04	102.75
	LSB-first	4	78.62	77.25	73.45	70.00	61.34
		8	81.24	77.32	71.76	69.70	62.31
		12	80.07	78.78	74.61	68.13	62.49
		16	77.59	78.60	74.95	69.62	63.14
		20	80.03	78.20	72.83	70.97	64.03
		24	78.41	76.17	74.04	70.08	64.86
		28	78.94	76.57	75.78	70.95	63.59

Table 6.5: Baseline, MSB-first & LSB-first truncated accelerator $f_{\max}$. Averaged execution times and maximum operating frequencies achieved are shown for each design to allow side-by-side comparison of unprotected and the range of protected implementations.

Truncation type	Truncation width r (bits)	cs_{in} width (bits)	cs_{out} width (bits)
None		$32 + \log_2(s)$	64
MSB-first	4	28	28
	8	24	24
	12	20	20
	16	16	16
	20	12	12
	24	8	8
	28	4	4
LSB-first	4	$28 + \log_2(s)$	60
	8	$24 + \log_2(s)$	56
	12	$20 + \log_2(s)$	52
	16	$16 + \log_2(s)$	48
	20	$12 + \log_2(s)$	44
	24	$8 + \log_2(s)$	40
	28	$4 + \log_2(s)$	36

Table 6.6: RP-ABFT input & output checksum widths, listing the input and output checksum widths for each of the designs compiled. $s = 32$ in all cases.



(a) MSB-first, permanent faults

(b) MSB-first, transient faults

(c) LSB-first, permanent faults

(d) LSB-first, transient faults

Figure 6.24: Area-scaled false negative fault proportions for RP-ABFT-protected accelerator, with total resource usage figures used to scale for s and linear scaling used for n.

Algorithm 12 RP-ABFT fault injection simulation

```
1: create s × s matrix A
2: create s × s matrix B
3: A.rand_fill()
4: B.rand_figll()
5: create s × s matrix C_ref
6: C_ref ← AB
7: create (s + 1) × s matrix A_c
8: create s × (s + 1) matrix B_r
9: A_c ← A.add_cs('col', truncation_type, n, r)
10: B_r ← B.add_cs('row', truncation_type, n, r)
11: create (s + 1) × (s + 1) matrix C_f
12: if fault_type = 'perm' then
13:    bit_mask, faulty_cols ← generate_bit_mask(s, n, 1, 1, 1)
14:    for i = 0 to s do
15:       for j = 0 to s do
16:          C_f[i][j] ← bit_mask[j]
17:          for k = 0 to s − 1 do
18:             C_f[i][j] ← ((C_f[i][j] + A_c[i][k] × B_r[k][j]) mod 2^(2n)) bitwise or bit_mask[j]
19:          end for
20:       end for
21:    end for
22: else
23:    bit_mask, faulty_cols ← generate_bit_mask(s, n, 1, 3, 1)
24:    for i = 0 to s do
25:       for j = 0 to s do
26:          C_f[i][j] ← bit_mask[i][j][0]
27:          for k = 0 to s − 1 do
28:             C_f[i][j] ← ((C_f[i][j] + A_c[i][k] × B_r[k][j]) mod 2^(2n)) bitwise or
                bit_mask[i][j][k + 1]
29:          end for
30:       end for
31:    end for
32: end if
33: data_ok ← C_f.get_data() = C_ref
34: cs_ok ← C_f.check_cs(truncation_type, n, r)
35: result_type ← classify_result(data_ok, cs_ok)
36: located ← C_f.diagnose_cs(truncation_type, n, r) = faulty_cols
```

Algorithm 13 `add_cs()` procedure used in RP-ABFT fault injection simulation

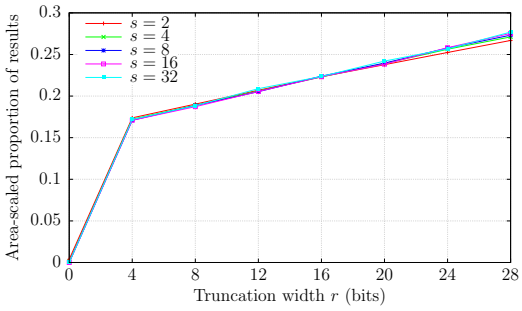
Require: *cs_type*, *trunc_type*, n, r

1: **if** *cs_type* = 'col' **then**
2: **create** $(s+1) \times s$ matrix $\boldsymbol{A}_{\mathrm{c}}$
3: $\boldsymbol{A}_{\mathrm{c}}[0 \ \cdot\cdot \ s-1][\cdot\cdot] \leftarrow \boldsymbol{A}$
4: **for** $j = 0$ **to** $s-1$ **do**
5: $\boldsymbol{A}_{\mathrm{c}}[s][j] \leftarrow 0$
6: **for** $i = 0$ **to** $s-1$ **do**
7: **if** *trunc_type* = 'none' **then**
8: $\boldsymbol{A}_{\mathrm{c}}[s][j] \leftarrow (\boldsymbol{A}_{\mathrm{c}}[s][j] + \boldsymbol{A}[i][j])$ **mod** 2^{2n}
9: **else if** *trunc_type* = 'msb_first' **then**
10: $\boldsymbol{A}_{\mathrm{c}}[s][j] \leftarrow (\boldsymbol{A}_{\mathrm{c}}[s][j] + \boldsymbol{A}[i][j])$ **mod** 2^{n-r}
11: **else**
12: $\boldsymbol{A}_{\mathrm{c}}[s][j] \leftarrow \boldsymbol{A}_{\mathrm{c}}[s][j] + (\boldsymbol{A}[i][j] \gg r)$
13: **end if**
14: **end for**
15: **end for**
16: **return** $\boldsymbol{A}_{\mathrm{c}}$
17: **else if** *cs_type* = 'row' **then**
18: **create** $s \times (s+1)$ matrix $\boldsymbol{B}_{\mathrm{r}}$
19: $\boldsymbol{B}_{\mathrm{r}}[\cdot\cdot][0 \ \cdot\cdot \ s-1] \leftarrow \boldsymbol{B}$
20: **for** $i = 0$ **to** $s-1$ **do**
21: $\boldsymbol{B}_{\mathrm{r}}[i][s] \leftarrow 0$
22: **for** $j = 0$ **to** $s-1$ **do**
23: **if** *trunc_type* = 'none' **then**
24: $\boldsymbol{B}_{\mathrm{r}}[i][s] \leftarrow (\boldsymbol{B}_{\mathrm{r}}[i][s] + \boldsymbol{B}[i][j])$ **mod** 2^{2n}
25: **else if** *trunc_type* = 'msb_first' **then**
26: $\boldsymbol{B}_{\mathrm{r}}[i][s] \leftarrow (\boldsymbol{B}_{\mathrm{r}}[i][s] + \boldsymbol{B}[i][j])$ **mod** 2^{n-r}
27: **else**
28: $\boldsymbol{B}_{\mathrm{r}}[i][s] \leftarrow \boldsymbol{B}_{\mathrm{r}}[i][s] + (\boldsymbol{B}[i][j] \gg r)$
29: **end if**
30: **end for**
31: **end for**
32: **return** $\boldsymbol{B}_{\mathrm{r}}$
33: **else**
34: {Perform steps for *cs_type* = 'col' and 'row', returning $(s+1) \times (s+1)$ matrix $\boldsymbol{C}_{\mathrm{f}}$}
35: **end if**

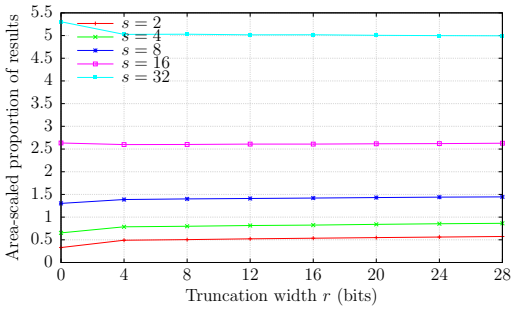
Algorithm 14 `check_cs()` procedure used in RP-ABFT fault injection simulation

Require: *truncation_type*, n, r

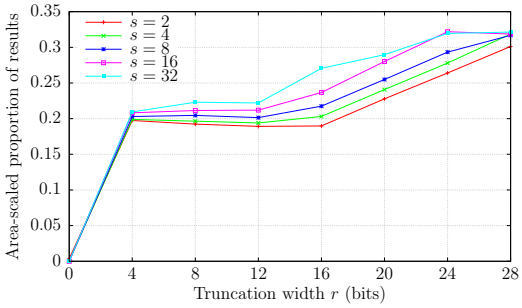
1: **create** $s \times s$ matrix $\boldsymbol{C}_{\text{copy}}$
2: $\boldsymbol{C}_{\text{copy}} \leftarrow \boldsymbol{C}_{\text{f}}$.get_data()
3: **create** $(s+1) \times (s+1)$ matrix $\boldsymbol{C}_{\text{copy, f}}$
4: $\boldsymbol{C}_{\text{copy, f}} \leftarrow \boldsymbol{C}_{\text{copy}}$.add_cs(*cs_type*, 'full', n, r)
5: **for all** *cs_type* in {'col', 'row'} **do**
6: **create** $s+1$ vector $\boldsymbol{cs}$
7: **create** $s+1$ vector $\boldsymbol{cs}$
8: $\boldsymbol{cs} \leftarrow \boldsymbol{C}_{\text{f}}$.get_cs(*cs_type*)
9: $\boldsymbol{cs}_{\text{copy}} \leftarrow \boldsymbol{C}_{\text{copy, f}}$.get_cs(*cs_type*)
10: **if** *truncation_type* = 'none' **then**
11: **if** $\boldsymbol{cs} \neq \boldsymbol{cs}_{\text{copy}}$ **then**
12: **return false**
13: **end if**
14: **else if** *truncation_type* = 'msb_first' **then**
15: **for** $x = 0$ **to** s **do**
16: **if** $\boldsymbol{cs}[x]$ **mod** $2^{n-r} \neq \boldsymbol{cs}_{\text{copy}}[x]$ **then**
17: **return false**
18: **end if**
19: **end for**
20: **else**
21: $\theta = s^2 2^{n-1}$
22: $\theta_{\text{c}} = s^3 2^{n-r}$
23: **for** $x = 0$ **to** $s-1$ **do**
24: **if** $\left|(\boldsymbol{cs}[x] - \boldsymbol{cs}_{\text{copy}}[x])\ \textbf{mod}\ 2^{2n-r}\right| > \theta$ **then**
25: **return false**
26: **end if**
27: **end for**
28: **if** $\left|(\boldsymbol{cs}[s] - \boldsymbol{cs}_{\text{copy}}[s])\ \textbf{mod}\ 2^{2(n-r)}\right| > \theta_{\text{c}}$ **then**
29: **return false**
30: **end if**
31: **end if**
32: **end for**
33: **return true**



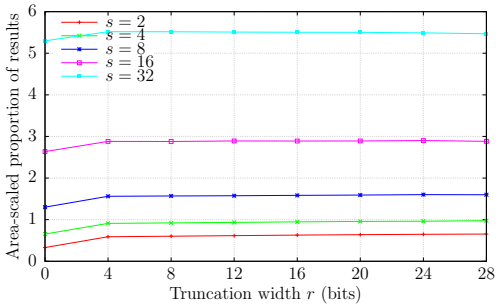
(a) MSB-first, permanent faults



(b) MSB-first, transient faults

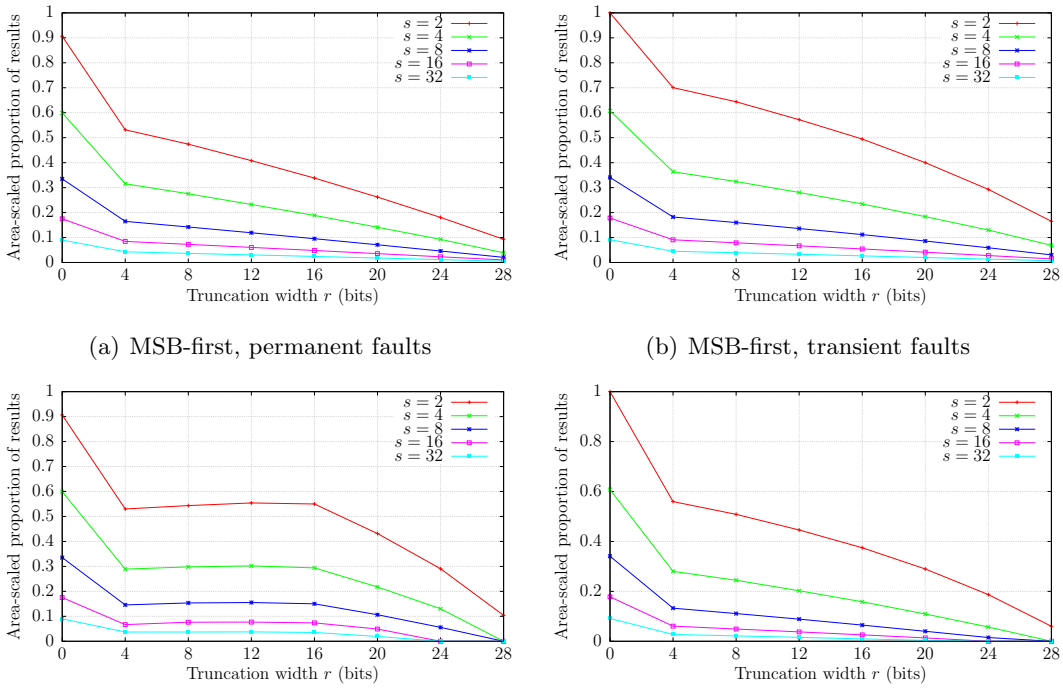


(c) LSB-first, permanent faults



(d) LSB-first, transient faults

Figure 6.25: Area-scaled masked fault proportions for RP-ABFT-protected accelerator, with total resource usage figures used to scale for s and linear scaling used for n.



(a) MSB-first, permanent faults

(b) MSB-first, transient faults

(c) LSB-first, permanent faults

(d) LSB-first, transient faults

Figure 6.26: Area-scaled located fault proportions for RP-ABFT-protected accelerator, with total resource usage figures used to scale for s and linear scaling used for n.

7 Conclusion

The work described in this thesis addresses current hardware reliability concerns. By employing low-overhead fault tolerance, as the experimental results herein have shown, engineers can gain highly robust datapaths by sacrificing some throughput and, most importantly, only limited additional area: critical for power efficiency and to limit the effect expanding physical size has upon increasing the occurrence of faults. As reliability becomes of greater concern for a wider variety of applications and settings due to increases in variability, degradation and fault susceptibility caused by continued process scaling, reliability mechanisms that necessitate the introduction of only limited overheads, such as algorithm-based fault tolerance (ABFT) and its derivatives, are likely to prove popular.

While the majority of the applicational focus within the technical content of this thesis has been on a single mathematical operation—matrix multiplication—the techniques developed are applicable to a wide range of other operators with relatively minor implementational changes. Generality has been demonstrated through the exploration of fault observability for additional operators—matrix addition and matrix-vector addition, the latter representative of the behaviour of linear filters—and the ABFT-hardening of other linear algebra-heavy applications, including those featuring lower- and upper-triangular (LU) decompositions or discrete Fourier transforms (DFTs), remains possible.

7.1 Summary

A benchmark hardware accelerator using ABFT for runtime datapath error detection was introduced as a case study for demonstrating the ability to achieve high fault detectability without incurring significant area (and consequently power) or performance overheads. Near-100% detectability was achieved while an area penalty of just 7.87% was incurred. Throughput was reduced by 31.3%. The area overhead figure, in particular, compares extremely favourably with that which would be incurred through using modular redundancy to achieve similar levels of resilience, with triple modular redundancy (TMR) necessitating

in excess of 200% additional area, for example.

Full fault detection and repair was achieved within the same accelerator with two mechanisms—data-shifting logic and dynamic partial reconfiguration (DPR)—implemented to allow operands to be routed around resources identified as faulty during operation. Area overheads for a complete, hardened application of 12.4% and 10.1% were encountered for additional logic-shifting and DPR, respectively; the latter representing a 50.7% overhead reduction over the former. In the DPR case, area overhead was reduced at the expense of partial bitstream storage; 5.10MB of memory was required. Single fault locatability of 96.7% was achieved in return for a 19.7% throughput penalty during fault-free operation, increasing to an average of 33.8% in order to correct a single fault.

A simulation framework was developed to allow the fault observabilities of several ABFT-protected operators to be established. By producing efficient simulation software, hundreds of millions of simulations could be completed within hours, allowing accurate insights into fault observability to be made fairly quickly. Object-oriented and user-friendly, the framework is extensible; more operators and fault patterns could be added with relative ease. Analysis of the results obtained demonstrated ABFT to be an effective error detection mechanism across the three operators considered.

The final technical work presented in this thesis covered the development of reduced-precision algorithm-based fault tolerance (RP-ABFT); a derivative of ABFT in which selective bit-width reduction is made in order to trade off incurred area and performance overheads for limited fault detectability. RP-ABFT using checksum logic truncation from the least-significant bit (LSB) first was found to be effective in achieving this: 16.7% of area and 8.27% of the throughput overheads caused by the introduction of ABFT protection were recovered by allowing low-magnitude errors (sub-1% of the maximum absolute output value) to propagate while retaining robustness against faults likely to cause high-magnitude errors. Findings indicate that RP-ABFT with LSB-first truncation represents a useful addition to the reliability 'toolbox.'

7.2 Future Work

While matrix multiplication represented a solid case study for the implementation of a complete hardware fault tolerance solution presented in Chapters 3 and 4, the generalisation of the techniques developed therein to other linear algebraic operators is considered to be an important avenue for further work. Refinements to the accelerator design, partic-

ularly facilitating more comprehensive pipelining of the checksum generation and verification logic to allow timing model-inferred maximum operating frequency ($f_{\max}$) recovery, also remain possible.

Several areas of future work have been determined based upon outcomes of Chapter 5's fault observability experiments. The first concerns a potential design tool capable of creating hardware for the acceleration of a range of mathematical operations, and combinations thereof, for linear algebra and signal processing applications. Basing such a tool upon an established, and preferably cross-platform, high-level development framework such as Open Computing Language (OpenCL) [79]—supported by both major field-programmable gate array (FPGA) vendors [80] [81]—would allow for verification across a wider array of usage scenarios and facilitate greater awareness of the methods developed. The data underpinning Chapter 5's results will be fundamental to the selection of design parameters made by both the tool itself and its users. Additional mathametical operations suitable for ABFT hardening, particularly the DFT, will also be explored.

Finally, regarding Chapter 6's discussion of RP-ABFT, developments are planned for its integration with both a high-level development framework for more accessible area-performance-reliability tuning and online arithmetic operators to achieve bounded overclocking error in 'reliably unreliable' circuitry.

Bibliography

[1] C. Stroud, S. Konala, P. Chen, and M. Abramovici, "Built-in Self-test of Logic Blocks in FPGAs (Finally, a Free Lunch: BIST without Overhead!)," in *IEEE VLSI Test Symposium*, 1996.

[2] J. S. J. Wong, P. Sedcole, and P. Y. K. Cheung, "A Transition Probability-based Delay Measurement Method for Arbitrary Circuits on FPGAs," in *International Conference on Field-programmable Technology (FPT)*, 2008.

[3] M. Abramovici, C. Strond, C. Hamilton, S. Wijesuriya, and V. Verma, "Using Roving STARs for On-line Testing and Diagnosis of FPGAs in Fault-tolerant Applications," in *IEEE International Test Conference (ITC)*, 1999.

[4] J. M. Levine, E. Stott, G. A. Constantinides, and P. Y. K. Cheung, "Online Measurement of Timing in Circuits: for Health Monitoring and Dynamic Voltage & Frequency Scaling," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2012.

[5] E. Stott and P. Y. K. Cheung, "Improving FPGA Reliability with Wear-levelling," in *International Conference on Field-programmable Logic and Applications (FPL)*, 2011.

[6] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "Low-overhead Fault-tolerant FPGA Systems," *IEEE Transactions on VLSI Systems*, vol. 6, no. 2, 1998.

[7] F. Hanchek and S. Dutt, "Node Covering-based Defect and Fault Tolerance Methods for Increased Yield in FPGAs," in *International Conference on VLSI Design*, 1996.

[8] J. M. Emmert and D. K. Bhatia, "A Fault-tolerant Technique for FPGAs," *Journal of Electronic Testing: Theory and Applications (JETTA)*, vol. 16, no. 6, 2000.

[9] R. F. DeMara and K. Zhang, "Autonomous FPGA Fault-handling through Competitive Runtime Reconfiguration," in *NASA/DoD Conference on Evolvable Hardware (EH)*, 2005.

[10] S. Srinivasan, P. Mangalagiri, Y. Xie, N. Viiaykrishnan, and K. Sarpatwari, "FLAW: FPGA Lifetime Awareness," in *ACM/IEEE Design Automation Conference*, 2006.

[11] S. Zafar, Y. Kim, V. Narayanan, C. Cabral, V. Paruchuri, B. Doris, J. Stathis, A. Callegari, and M. Chudzik, "A Comparative Study of NBTI and PBTI (Charge Trapping) in SiO2/HfO2 Stacks with FUSI, TiN, Re Gates," in *IEEE Symposium on VLSI Technology*, 2006.

[12] E. Stott, P. Sedcole, and P. Y. K. Cheung, "Fault Tolerance and Reliability in Field-programmable Gate Arrays," *IET Computers & Digital Techniques*, vol. 4, no. 3, 2010.

[13] S. Kiamehr, A. Amouri, and M. B. Tahoori, "Investigation of NBTI- and PBTI-induced Aging in Different LUT Implementations," in *International Conference on Field-Programmable Technology (FPT)*, 2011.

[14] E. Stott, J. S. J. Wong, and P. Y. K. Cheung, "Degradation Analysis and Mitigation in FPGAs," in *International Conference on Field-programmable Logic and Applications (FPL)*, 2010.

[15] Altera, "Cyclone III FPGAs." `http://www.altera.com/products/fpga/cyclone-series/cyclone-iii/overview.html`.

[16] C. Stroud, E. Lee, S. Konala, and M. Abramovici, "Using ILA Testing for BIST in FPGAs," in *IEEE International Test Conference (ITC)*, 1996.

[17] A. Alaghi, M. Sadoughi Yarandi, and Z. Navabi, "An Optimum ORA BIST for Multiple Fault FPGA Look-up Table Testing," in *IEEE Asian Test Symposium (ATS)*, 2006.

[18] C. Stroud, S. Wijesuriya, C. Hamilton, and M. Abramovici, "Built-in Self-test of FPGA Interconnect," in *IEEE International Test Conference (ITC)*, 1998.

[19] I. G. Harris and R. Tessier, "Testing and Diagnosis of Interconnect Faults in Cluster-based FPGA Architectures," *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, vol. 21, no. 11, 2002.

[20] C.-L. Hsu and T.-H. Chen, "Built-in Self-test Design for Fault Detection and Fault Diagnosis in SRAM-based FPGAs," *IEEE Transactions on Instrumentation and Measurement*, vol. 58, no. 7, 2009.

[21] J. Liu and S. Simmons, "BIST Diagnosis of Interconnect Fault Locations in FPGAs," in *Canadian Conference on Electrical and Computer Engineering*, 2003.

[22] P. Girard, O. Heron, S. Pravossoudovitch, and M. Renovell, "Defect Analysis for Delay-fault BIST in FPGAs," in *IEEE On-line Testing Symposium (IOLTS)*, 2003.

[23] J. S. J. Wong, P. Sedcole, and P. Y. K. Cheung, "Self-measurement of Combinatorial Circuit Delays in FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 2, no. 2, 2009.

[24] I. G. Harris, P. R. Menon, and R. Tessier, "BIST-based Delay Path Testing in FPGA Architectures," in *IEEE International Test Conference (ITC)*, 2001.

[25] C.-C. Wang, J.-J. Liou, Y.-L. Peng, C.-T. Huang, and C.-W. Wu, "A BIST Scheme for FPGA Interconnect Delay Faults," in *IEEE VLSI Test Symposium*, 2005.

[26] Y.-L. Peng, C.-W. Wu, J.-J. Liou, and C.-T. Huang, "BIST-based Diagnosis Scheme for Field-programmable Gate Array Interconnect Delay Faults," *IET Computers & Digital Techniques*, vol. 1, no. 6, 2007.

[27] M. A. Lusco, J. L. Dailey, and C. E. Stroud, "Built-in Self-test for Multipliers in Altera Cyclone II Field-programmable Gate Arrays," in *Southeastern Symposium on System Theory (SSST)*, 2011.

[28] M. D. Pulukuri and C. E. Stroud, "Built-in Self-test of Digital Signal Processors in Virtex-4 FPGAs," in *Southeastern Symposium on System Theory (SSST)*, 2009.

[29] Z. Zhang, Z. Wen, and L. Chen, "BIST Approach for Testing Embedded Memory Blocks in System-on-Chips," in *IEEE Circuits and Systems International Conference on Testing and Diagnosis (ICTD)*, 2009.

[30] P. Sedcole, J. S. J. Wong, and P. Y. K. Cheung, "Characterisation of FPGA Clock Variability," in *IEEE Computer Society Symposium on VLSI (ISVLSI)*, 2008.

[31] C. E. Stroud and N. S. Da Cunha, "Built-in Self-test of Programmable Clock Buffers in Virtex-4, Virtex-5 and Virtex-6 FPGAs," in *Southeastern Symposium on System Theory (SSST)*, 2011.

[32] S. Sunter and A. Roy, "Adaptive Parametric BIST of High-speed Parallel I/Os via Standard Boundary Scan," in *IEEE International Test Conference (ITC)*, 2011.

[33] M. Abramovici, C. Stroud, B. Skaggs, and J. Emmert, "Improving On-line BIST-based Diagnosis for Roving STARs," in *IEEE International On-line Testing Workshop*, 2000.

[34] S. Dutt, V. Verma, and V. Suthar, "Built-in Self-test of FPGAs with Provable Diagnosabilities and High Diagnostic Coverage with Application to Online Testing," *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, vol. 27, no. 2, 2008.

[35] V. Suthar and S. Dutt, "Efficient On-line Interconnect Testing in FPGAs with Provable Detectability for Multiple Faults," in *Design, Automation and Test in Europe (DATE)*, 2006.

[36] N. R. Shnidman, W. H. Mangione-Smith, and M. Potkonjak, "Fault Scanner for Reconfigurable Logic," in *Conference on Advanced Research in VLSI*, 1997.

[37] M. Abramovici and C. Stroud, "BIST-based Delay-fault Testing in FPGAs," in *IEEE International On-line Testing Workshop*, 2002.

[38] S. D'Angelo, C. Metra, S. Pastore, A. Pogutz, and G. R. Sechi, "Fault-tolerant Voting Mechanism and Recovery Scheme for TMR FPGA-based Systems," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 1998.

[39] M. A. Sullivan, H. H. Loomis, and A. A. Ross, "Employment of Reduced-precision Redundancy for Fault-tolerant FPGA Applications," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2009.

[40] B. Pratt, M. Fuller, M. Rice, and M. Wirthlin, "Reduced-precision Redundancy for Reliable FPGA Communications Systems in High-radiation Environments," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 49, no. 1, 2013.

[41] R. Glein, B. Schmidt, F. Rittner, J. Teich, and D. Ziener, "A Self-adaptive SEU Mitigation System for FPGAs with an Internal Block RAM Radiation Particle Sensor," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2014.

[42] F. G. de Lima Kastensmidt, G. Neuberger, R. F. Hentschke, L. Carro, and R. Reis, "Designing Fault-tolerant Techniques for SRAM-based FPGAs," *IEEE Design & Test of Computers*, vol. 21, no. 6, 2004.

[43] A. L. Burress and P. K. Lala, "On-line Testable Logic Design for FPGA Implementation," in *IEEE International Test Conference (ITC)*, 1997.

[44] R. Karri and N. Mukherjee, "Versatile BIST: an Integrated Approach to On-line/Off-line BIST," in *IEEE International Test Conference (ITC)*, 1998.

[45] P. Nigh and W. Maly, "A Self-testing ALU using Built-in Current Sensing," in *IEEE Custom Integrated Circuits Conference*, 1989.

[46] M. Nicolaidis, "On-line Testing for VLSI," in *IEEE International Test Conference (ITC)*, 1997.

[47] D. Mange, M. Goeke, D. Madon, A. Stauffer, G. Tempesti, S. Durand, P. Marchal, and P. Nussbaum, "Embryonics: a New Family of Coarse-grained FPGA with Self-repair and Self-reproducing Properties," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 1996.

[48] C. Ortega-Sanchez, A. Tyrrell, D. Mange, A. Stauffer, and G. Tempesti, "Reliability Analysis of a Self-repairing Embryonic Machine," in *Euromicro Conference*, 2000.

[49] V. Lakamraju and R. Tessier, "Tolerating Operational Faults in Cluster-based FPGAs," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2000.

[50] J. Narasimham, K. Nakajima, C. Rim, and A. T. Dahbura, "Yield Enhancement of Programmable ASIC Arrays by Reconfiguration of Circuit Placements," *IEEE Transactions on CAD of Integrated Circuits and Systems (TCAD)*, vol. 13, no. 8, 1994.

[51] A. Mathur and C. L. Liu, "Timing-driven Placement Reconfiguration for Fault Tolerance and Yield Enhancement in FPGAs," in *European Design and Test Conference (DATE)*, 1996.

[52] B. Girau, P. Marchal, P. Nussbaum, A. Tisserand, and H. F. Restrepo, "Evolvable Platform for Array Processing: a One-chip Approach," in *International Conference on Microelectronics for Neural, Fuzzy and Bio-Inspired Systems (MicroNeuro)*, 1999.

[53] A. Miele and P. di Torino, "A Software Framework for Dynamic Self-repair in Embedded SoCs Exploiting Reconfigurable Devices," in *IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, 2010.

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

[55] S.-J. Wang and N. K. Jha, "Algorithm-based Fault Tolerance for FFT Networks," *IEEE Transactions on Computers*, vol. 43, no. 7, 1994.

[56] A. Jacobs, G. Cieslewski, and A. D. George, "Overhead and Reliability Analysis of Algorithm-based Fault Tolerance in FPGA Systems," in *International Conference on Field-programmable Logic and Applications (FPL)*, 2012.

[57] J. Rexford and N. K. Jha, "Algorithm-based Fault Tolerance for Floating-point Operations in Massively Parallel Systems," in *International Symposium on Circuits and Systems (ISCAS)*, vol. 2, 1992.

[58] C. Braun, S. Halder, and H. J. Wunderlich, "A-ABFT: Autonomous Algorithm-based Fault Tolerance for Matrix Multiplications on Graphics Processing Units," in *International Conference on Dependable Systems and Networks (DSN)*, 2014.

[59] J.-Y. Jou and J. A. Abraham, "Fault-tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures," *Proceedings of the IEEE*, vol. 74, no. 5, 1986.

[60] J. J. Davis and P. Y. K. Cheung, "Datapath Fault Tolerance for Parallel Accelerators," in *International Conference on Field-programmable Technology (FPT)*, 2013.

[61] Xilinx, "Zynq-7000 All Programmable SoC." `http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`.

[62] Xilinx, "Zynq-7000 AP SoC (XC7Z010 and ZC7Z020) Data Sheet." `http://www.xilinx.com/support/documentation/data_sheets/ds187-XC7Z010-XC7Z020-Data-Sheet.pdf`.

[63] Xilinx, "LogiCORE IP XPS Central DMA Controller (v2.03a)." `http://www.xilinx.com/support/documentation/ip_documentation/xps_central_dma.pdf`.

[64] Xilinx, "LogiCORE IP AXI Block RAM (BRAM) Controller (v1.03a)." http://www.xilinx.com/support/documentation/ip_documentation/axi_bram_ctrl/v1_03_a/ds777_axi_bram_ctrl.pdf.

[65] Xilinx, "LogiCORE AXI Interconnect IP (v1.01.a)." http://www.xilinx.com/support/documentation/ip_documentation/ds768_axi_interconnect.pdf.

[66] Xilinx, "LogiCORE IP Interrupt Control (v2.01a)." http://www.xilinx.com/support/documentation/ip_documentation/interrupt_control.pdf.

[67] Avnet, "Introduction to the Zynq-7000 Extensible Processing Platform." http://www.em.avnet.com/en-us/design/trainingandevents/Documents/X-FEST%202012%20PRESENTATIONS/xfest12_pdf_zynq_intro_v1_1_april29.pdf.

[68] Xilinx, "7 Series DSP48E1 Slice, UG479." http://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf.

[69] Python, "16.6. multiprocessing – Process-based 'Threading' Interface." http://docs.python.org/2/library/multiprocessing.html.

[70] AMD, "AMD Server Processors." http://www.amd.com/en-us/products/server.

[71] J. J. Davis and P. Y. K. Cheung, "Reducing Overheads for Fault-tolerant Datapaths with Dynamic Partial Reconfiguration," in *IEEE International Symposium on Field-programmable Custom Computing Machines (FCCM)*, 2014.

[72] J. J. Davis and P. Y. K. Cheung, "Achieving Low-overhead Fault Tolerance for Parallel Accelerators with Dynamic Partial Reconfiguration," in *International Conference on Field-programmable Logic and Applications (FPL)*, 2014.

[73] Xilinx, "Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 All-programmable SoC Devices." http://www.xilinx.com/support/documentation/application_notes/xapp1159-partial-reconfig-hw-accelerator-zynq-7000.pdf.

[74] Avnet, "Avnet Product Brief – ZedBoard." http://zedboard.org/sites/default/files/product_briefs/PB-AES-Z7EV-7Z020_G-v12.pdf.

[75] J. J. Davis and P. Y. K. Cheung, "Reduced-precision Algorithm-based Fault Tolerance for FPGA-implemented Accelerators," in *International Workshop on Applied Reconfigurable Computing (ARC)*, 2016.

[76] Python, "5. Built-in Types." http://docs.python.org/2/library/stdtypes.html.

[77] K. Shi, D. Boland, E. Stott, S. Bayliss, and G. Constantinides, "Datapath Synthesis for Overclocking: Online Arithmetic for Latency-accuracy Tradeoffs," in *Design Automation Conference (DAC)*, 2014.

[78] K. Shi, D. Boland, and G. Constantinides, "Efficient FPGA Implementation of Digit-parallel Online Arithmetic Operators," in *International Conference on Field-Programmable Technology (FPT)*, 2014.

[79] K. Group, "OpenCL – Khronos Group." http://www.khronos.org/opencl.

[80] Altera, "Altera SDK for OpenCL." http://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.

[81] Xilinx, "Xilinx SDAccel." http://www.xilinx.com/publications/prod_mktg/sdnet/sdaccel-wp.pdf.