UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA

EDUARDO MARAÑON AGUILAR

Designing and Testing Fault-Tolerant APSoC-Based AI Accelerators for Aerospace Applications

Thesis presented in partial fulfillment of the requirements for the degree of Master of Microelectronics

Advisor: Dr. Fernanda Gusmão de Lima Kastensmidt

Porto Alegre
November 2025

CIP – CATALOGING-IN-PUBLICATION

Marañon Aguilar, Eduardo

Designing and Testing Fault-Tolerant APSoC-Based IA Accelerators for Aerospace Applications / Eduardo Marañon Aguilar. – 2025.

131 f.: il.

Thesis (Master's) – Federal University of Rio Grande do Sul. Postgraduate Program in Microelectronics. Porto Alegre, Brazil, 2025. Advisor: Fernanda Gusmão De Lima Kastensmidt.

1. SRAM-based APSoCs. 2. ZynqNet. 3. CNN. 4. Efficiency. 5. Relilability. 6. Radiation Effects. I. Kastensmidt, Fernanda Lima. II. Designing and Testing Fault-Tolerant APSoC-Based IA Accelerators for Aerospace Applications.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Profa. Márcia Cristina Bernardes Barbosa
Vice-Reitor: Prof. Pedro de Almeida Costa
Pró-Reitora de Pós-Graduação: Claudia Wasserman
Diretor do Instituto de Informática: Prof. Luciano Paschoal Gaspary
Coordenador do PGMICRO: Prof. José Rodrigo Furlanetto de Azambuja
Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro

ACKNOWLEDGMENT

I would like to express my sincere gratitude to all the people who supported me during the development of this thesis.

To my fiancée, Maryaren, for her companionship, love and wise advice throughout this journey. Her presence was crucial in overcoming the challenges that arose, and she has been an endless source of motivation and joy.

To my parents, Eliana and Domingo, for their unconditional support, and for teaching me through their example the importance of effort and perseverance.

To my advisor, Fernanda, for her academic guidance, commitment, and valuable suggestions, which were decisive for the development of this work.

To my friends, Diego and José, for their sincere friendship and for being present in pleasant and complicated moments.

To my laboratory colleague, Fabio, for his collaboration, willingness, and support during the practical stages of my research.

Finally, I want to thank the Institute of Informatics at UFRGS for providing me with a workspace and APSoC boards for conducting this research. I also extend my gratitude to the funding institutions that supported this thesis, including the Brazilian agencies Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) and Fundação de Apoio da Universidade Federal do Rio Grande do Sul (FAURGS), as well as the American agency Office of Naval Research (ONR). Additionally, this work has also been partially funded by the project IA na Borda, supported by the Center for Embedded Devices and Research in Digital Agriculture (CEDRA) of SENAI-RS, with financial resources from the PPI IoT/Manufatura 4.0 / PPI HardwareBR of the MCTI, grant number 056/2023, signed with EMBRAPII.

ABSTRACT

The rapid advancement of deep learning in both complexity and adoption has increased the demand for efficient hardware capable of handling the computational requirements of Convolutional Neural Networks (CNNs). Static Random Access Memory (SRAM)-based All Programmable System on Chips (APSoCs) have become a promising platform for deploying Artificial Intelligence (AI) inference engines due to their flexibility, low latency, and power efficiency. However, when commercial SRAM-based APSoCs are used in safety-critical missions operating in radiation-prone environments, they become vulnerable to radiation-induced Single Event Effects (SEEs), which can compromise system reliability and adversely affect mission success. As a result, design optimizations aimed at improving efficiency and performance, together with fault-tolerant strategies to enhance reliability, become essential to meet the stringent requirements of APSoC-based AI accelerators in safety-critical applications. In this context, this thesis presents a case-study implementation of the open-source ZynqNet AI inference engine, written in C++, optimized with directive pragmas and generated via High Level Synthesis (HLS). A four-layer CNN trained on the SAT-6 dataset was deployed on the Zynq-7000 APSoC (xc7z020clg484-1), with the Programmable Logic (PL) implementing ZynqNet and the Processing System (PS) managing execution via the Cortex-A9 processor. Quantization techniques were applied to reduce area and power, moving from floating-point to fixed-point and eventually 8-bit integer arithmetic, with different numbers of processing elements to exploit task parallelism. Performance was improved by packing eight 8-bit integer values to enable parallel 64-bit read and write operations, which led to the design and implementation of small caches, reducing the number of clock cycles required to access the data in the external Double Data Rate (DDR) memory. Communication architectures were also designed to optimize data exchange between the accelerator and the external DDR memory, with optional Level 1 Data (L1D) and Level 2 (L2) cache usage. A comprehensive analysis of CNN performance and Field Programmable Gate Array (FPGA) resource usage across configurations was conducted. Then, a fault injection methodology enabled rapid, cost-efficient evaluation of ZynqNet's reliability, identifying components most vulnerable to radiation faults. Coarse Grained, Modular and Distributed Triple Modular Redundancy (TMR) techniques were applied at the design level to mitigate Single and Multiple Bit Upsets in the configuration memory, alongside Error Detection and Correction (EDAC) mechanisms were integrated into the Block RAMs (BRAMs) of the ZynqNet's Caches blocks. To address persistent bit flips in the FPGA's Configuration Memory (CRAM), the memory scrubbing mechanism was enabled

for detection and correction of radiation-induced faults during the CNN computing. Radiation testing using heavy ions was conducted on both unmitigated and hardened designs to validate fault tolerance. Reported results include reliability curves, and the following metrics: Mean Faults to Failure (MFTF), Mean Fluence to Failure (MΦTF), Dynamic Cross Section (σ), and hardware design evaluation metrics such as area, power and performance. The study offers valuable insights into the trade-offs between resource usage, power, performance and reliability for the case-study ZynqNet accelerator in radiation-prone environments.

Keywords: ZynqNet AI accelerator, Reliability, Fault Tolerance, Fault Injection, Radiation Experiment.

Projetando e Testando Aceleradores de IA Baseados em APSoC Tolerantes a Falhas para Aplicações Aeroespaciais

RESUMO

O rápido avanço do aprendizado profundo, tanto em complexidade quanto em adoção, tem aumentado a demanda por hardware eficiente capaz de lidar com os requisitos computacionais de Redes Neurais Convolucionais (CNNs). Sistemas em Chip Totalmente Programáveis (APSoC) baseados em Memória Estática de Acesso Aleatório (SRAM), tornaram-se uma plataforma promissora para a implementação de motores de inferência de Inteligência Artificial (IA) devido à sua flexibilidade, baixa latência e eficiência energética. No entanto, quando APSoCs comerciais baseados em SRAM são utilizados em missões críticas de segurança operando em ambientes propensos à radiação, eles se tornam vulneráveis a Efeitos de Evento Único (SEEs) induzidos por radiação, o que pode comprometer a confiabilidade do sistema e afetar adversamente o sucesso da missão. Como resultado, otimizações de projeto voltadas à melhoria da eficiência e do desempenho, juntamente com estratégias tolerantes a falhas para aumentar a confiabilidade, tornam-se essenciais para atender aos requisitos rigorosos de aceleradores de AI baseados em APSoCs em aplicações críticas de segurança. Nesse contexto, esta dissertação apresenta um estudo de caso da implementação do motor de inferência de AI de código aberto ZynqNet, escrito em C++, otimizado com diretivas pragmas e gerado por meio de Síntese de Alto Nível (HLS). Uma CNN de quatro camadas, treinada com o conjunto de dados SAT-6, foi implementada no Zynq-7000 APSoC (xc7z020clg484-1), com a Lógica Programável (PL) implementando o ZynqNet e o Sistema de Processamento (PS) gerenciando a execução por meio do processador Cortex-A9. Técnicas de quantização foram aplicadas para reduzir área e consumo de energia, passando de ponto flutuante para ponto fixo e, posteriormente, para aritmética de inteiros de 8 bits, utilizando diferente números de elementos de processamento para explorar o paralelismo de tarefas. O desempenho foi aprimorado por meio do empacotamento de oito valores inteiros de 8 bits, permitindo operações paralelas de leitura e escrita de 64 bits, o que levou ao projeto e à implementação de caches pequenos, reduzindo o número de ciclos de clock necessários para acessar os dados na memória externa de Taxa de Dados Dupla (DDR). Arquiteturas de comunicação também foram projetadas para otimizar a troca de dados entre o acelerador e a memória externa DDR, com uso opcional de caches de Dados de Nível 1 (L1D) e Nível 2 (L2). Foi realizada uma análise abrangente do desempenho da CNN e do uso de recursos do Arranjo de Portas Programáveis em Campo

(FPGA) em diferentes configurações. Em seguida, uma metodologia de injeção de falhas permitiu uma avaliação rápida e econômica da confiabilidade do ZynqNet, identificando os componentes mais vulneráveis a falhas induzidas por radiação. Técnicas de Redundância Tripla Modular (TMR) do tipo Granularidade Grossa, Modular e Distribuída foram aplicadas em nível de projeto para mitigar falhas de um e múltiplos bits na memória de configuração, enquanto mecanismos de Detecção e Correção de Erros (EDAC) foram integrados às memórias RAM em bloco (BRAMs) dos blocos de cache do ZynqNet. Para lidar com inversões persistentes de bits na memória de configuração do FPGA (CRAM), o mecanismo de scrubbing de memória foi habilitado para detecção e correção de falhas induzidas por radiação durante a execução da CNN. Ensaios de radiação utilizando íons pesados foram conduzidos tanto em projetos sem mitigação quanto em projetos com mitigação, a fim de validar a tolerância a falhas. Os resultados reportados incluem curvas de confiabilidade e as seguintes métricas: Média de Falhas até a Falha (MFTF), Fluência Média até a Falha (MΦTF), Seção de Choque Dinâmica (σ), além de métricas de avaliação do projeto de hardware, como área, consumo de energia e desempenho. O estudo oferece insights valiosos sobre as compensações entre uso de recursos, consumo de energia, desempenho e confiabilidade para o estudo de caso do acelerador ZynqNet em ambientes propensos à radiação.

Palavras-chave: Acelerador de IA ZynqNet, Confiabilidade, Tolerância a Falhas, Injeção de Falhas, Experimento de Radiação.

LIST OF ABBREVIATIONS AND ACRONYMS

ACAP	Adaptive Compute Acceleration Platform
ACP	Accelerator Coherency Port
AHB	AMBA High-performance Bus
AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
AMD	Advanced Micro Devices
ANN	Artificial Neural Network
APB	AMBA Advanced Peripheral Bus
APSoC	All Programmable System on Chip
APU	Application Processor Unit
ARM	Advanced RISC Machines
AVF	Architectural Vulnerability Factor
AXI HP	Advanced eXtensible Interface High-Performance
AXI	Advanced eXtensible Interface
BRAM	Block Random Access Memory
CLB	Configurable Logic Block
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CRAM	Configuration Memory
DDR	Double Data Rate
DFF	D-type Flip-Flop
DSP	Digital Signal Processing
DTMR	Distributed Triple Modular Redundancy
EDAC	Error Detection and Correction
FC	Fully Connected
FF	Flip-Flop
FGPU	FPGA-based Graphics Processing Unit
FIFO	First-In First-Out
FPGA	Field Programmable Gate Array
FSM	Finite-State Machine
GP	Global Pooling
GPIO	General Purpose Input Output

HDL	Hardware Description Language
HLS	High-Level Synthesis
HSV	Hue, Saturation and Value
IFB	Internal Functional Block
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
IOP	I/O peripherals
IP	Intellectual Property
LLVM	Low Level Virtual Machine
LUT	Look-Up-Table
L1	Level 1
L1D	Level 1 Data
L2	Level 2
MAC	Multiply-Accumulate
MFTF	Mean Faults to Failure
ML	Machine Learning
MPSoC	Multiprocessor System-On-Chip
MUX	Multiplexer
MΦTF	Mean Fluence To Failure
NIR	Near Infrared
NN	Neural Network
OCM	On-chip Memory
PE	Processing Element
PL	Programmable Logic
PS	Processing System
RAM	Random-Access Memory
ReLU	Rectified Linear Unit
RTL	Register Transfer Level
SEE	Single-Event Effect
SEFI	Single-Event Functional Interrupt
SET	Single-Event Transient
SEU	Single-Event Upset
SMC	Static Memory Controller
SoC	System on Chip
SPI	Serial peripheral interface

SRAM	Static Random Access Memory
TMR	Triple Modular Redundancy
UART	Universal Asynchronous Receiver Transmitter
VLSI	Very Large Scale Integration

LIST OF FIGURES

Figure 2.1 – Neural cells and NN model. 23
Figure 2.2 – Perceptron model 24
Figure 2.3 – Example of a three layer network 25
Figure 2.4 – Convolution process 27
Figure 2.5 – Spatial hierarchy levels of features from a picture of a cat 28
Figure 2.6 – Max and average pooling 29
Figure 2.7 – Global sum pooling 29
Figure 2.8 – ReLU activation function 30
Figure 2.9 – Sigmoid activation function 31
Figure 2.10 – Zynq 7000 SoC architecture 32
Figure 2.11 – Instantiation of a CLB per column in the FPGA floorplanning. 33
Figure 2.12 – Instantiation of DSPs and BRAMs in the FPGA floorplanning. 34
Figure 2.13 – High Performance Blocks used for the connection of PL to PS AXI Channels. 36
Figure 2.14 – High Performance Blocks used for the connection of PL to PS AXI Channels. 36
Figure 2.15 – ACP Connectivity Diagram. 37
Figure 2.16 – Data flow received by FGPU running an image classification application. 39
Figure 2.17 – Block diagram interconnection of FINN in a Zynq-7000 APSoC to run an image classification application. 40
Figure 2.18 – Systolic Array architecture for CNN. 41
Figure 2.19 – ZynqNet accelerator internal components. 42
Figure 2.20 – Example of scheduling and binding process. 44
Figure 2.21 – Example of control logic extraction. 45
Figure 2.22 – Typical design flow of HLS 46
Figure 2.23 – IP-level and System-level HLS graphic differences. 47
Figure 2.24 – Pragmas directives applied in the processAllCHout function of the Zynqnet accelerator. 49
Figure 3.1 – Integration of ZynqNet and Xilinx IPs in the Zynq-7000 APSoC. 56
Figure 3.2 – Pseudocodes of processor and ZynqNet accelerator. 58
Figure 3.3 – Samples from the SAT-6 dataset and class proportions for the training and testing sets. 59
Figure 3.4 – CNN model architecture. 59
Figure 3.5 – Fault injection setup 61

Figure 3.6 – Bit cluster geometry list for fault injection based on SBU and MBU. 62
Figure 3.7 – Location of pblock_safe and pblock_dut in the FPGA floorplanning. 63
Figure 3.8 – Frame ranges for initial columns of pblock_dut. 63
Figure 3.9 – Inference error failure type. 64
Figure 3.10 – Timeout failure type. 64
Figure 3.11 – Heavy-Ions Radiation Setup. 66
Figure 4.1 – Interconnection of the ZynqNet IP with the other Xilinx IPs and the PS. 70
Figure 4.2 – Number of LUTs of each ZynqNet System design. 73
Figure 4.3 – Number of DFFs of each ZynqNet System design. 73
Figure 4.4 – Number of BRAMs of each ZynqNet System design. 74
Figure 4.5 – Number of DSPs of each ZynqNet System design. 74
Figure 4.6 – Number of essential bits of each ZynqNet System design. 74
Figure 4.7 – Dynamic Power consumption of each ZynqNet System design. 75
Figure 4.8 – Frames per second (FPS) processed by each ZynqNet System design. 77
Figure 4.9 – Communication architecture ZynqNet-PS, case 1. 78
Figure 4.10 – Communication architecture ZynqNet-PS, case 2. 78
Figure 4.11 – Communication architecture ZynqNet-PS, case 3. 79
Figure 4.12 – Communication architecture ZynqNet-PS, case 4. 79
Figure 4.13 – Communication architecture ZynqNet-PS, case 5. 79
Figure 4.14 – Communication architecture ZynqNet-PS, case 6. 80
Figure 4.15 – Number of LUTs of each communication architecture design. 81
Figure 4.16 – Number of DFFs of each communication architecture design. 81
Figure 4.17 – Number of BRAMs of each communication architecture design. 82
Figure 4.18 – Number of DSPs of each communication architecture design. 82
Figure 4.19 – Number of essential bits of each communication architecture design. 82
Figure 4.20 – Total power consumption of each communication architecture design. 83
Figure 4.21 – Frames per second (FPS) processed by each communication architecture design. 85
Figure 4.22 – Number of LUTs of the ZynqNet System design (arch 5, integer) with PE=1, 2, and 4. 86
Figure 4.23 – Number of DFFs of the ZynqNet System design (arch 5, integer) with PE=1, 2, and 4. 86
Figure 4.24 – Number of BRAMs of the ZynqNet System design (arch 5, integer) with PE=1, 2, and 4. 87

Figure 4.25 – Number of DSPs of the ZynqNet System design (arch 5, integer) with PE=1, 2, and 4. 87
Figure 4.26 – Number of essential bits of the ZynqNet System design (arch 5, integer) with PE=1, 2, and 4. 87
Figure 4.27 – Power consumption of the ZynqNet System (arch 5, integer) with PE=1, 2, and 4. 88
Figure 4.28 – Frames per second (FPS) processed by the ZynqNet System (arch 5, integer) with PE=1, 2, and 4. 90
Figure 5.1 – FPGA's floorplanning of ZynqNet's IFB designs to be tested. 93
Figure 5.2 – Reliability curves of ZynqNet's IFBs. 94
Figure 5.3 – Coarse-Grained TMR design architecture. 95
Figure 5.4 – MTMR-1 design architecture. 96
Figure 5.5 – MTMR-2 design architecture. 96
Figure 5.6 – MTMR-3 design architecture. 97
Figure 5.7 – MTMR-4 design architecture. 97
Figure 5.8 – MTMR-5 design architecture. 98
Figure 5.9 – DTMR design architecture. 99
Figure 5.10 – CRAM scrubbing mechanism in the FPGA. 100
Figure 5.11 – Hamming-based EDAC for 8-bit data. 101
Figure 5.12 – Hamming-based EDAC for 16-bit data. 101
Figure 5.13 – Number of LUTs of each Fault-Tolerant ZynqNet IP design. 102
Figure 5.14 – Number of DFFs of each Fault-Tolerant ZynqNet IP design. 103
Figure 5.15 – Number of BRAMs of each Fault-Tolerant ZynqNet IP design. 103
Figure 5.16 – Number of DSPs of each Fault-Tolerant ZynqNet IP design. 103
Figure 5.17 – Number of essential bits of each Fault-Tolerant ZynqNet IP design. 104
Figure 5.18 – Dynamic power consumption of each Fault-Tolerant ZynqNet IP design. 104
Figure 5.19 – Reliability curves of Modular TMR versions. 105
Figure 5.20 – Emulated Fault Injection reliability curves. 106
Figure 5.21 – Number of event types observed in each fault-tolerant ZynqNet System design. 108
Figure 5.22 – Number of total faults observed in each fault-tolerant ZynqNet System design. 108
Figure 5.23 – MFTF of each fault-tolerant ZynqNet System design. 109
Figure 5.24 – MF (R≥90%) of each fault-tolerant ZynqNet System design. 109

Figure 5.25 – Heavy-ions experiment reliability curves. 110
Figure 5.26 – Number of event types observed in each fault-tolerant ZynqNet System design. 112
Figure 5.27 – Total fluence of each fault-tolerant ZynqNet System design..... 112
Figure 5.28 – Mean fluence to failure (MΦTF) of each fault-tolerant ZynqNet System design. 113
Figure 5.29 – Dynamic Cross Section of each fault-tolerant ZynqNet System design. 113
Figure 5.30 – MΦ (R≥90%) of each fault-tolerant ZynqNet System design. 114

LIST OF TABLES

Table 2.1 – Comparison of the state-of-the-art with this work. 54
Table 3.1 – Comparison of the two experimental methodologies. 68
Table 4.1 – Area and dynamic power results of each ZynqNet IP with 1, 2 and 4 processing elements computing in floating-point, 8-bit fixed point and 8-bit integer. 71
Table 4.2 – Area and power consumption of the Xilinx IPs blocks within the ZynqNet System. 72
Table 4.3 – Area and dynamic power results of each ZynqNet System with 1, 2 and 4 processing elements computing in floating-point, 8-bit fixed point and 8-bit integer. 72
Table 4.4 – Power consumption of the PS elements in the APSoC. 75
Table 4.5 – Performance of the ZynqNet System with 1, 2 and 4 processing elements computing in floating-point, 8-bit fixed point and 8-bit integer. 76
Table 4.6 – Area results comparison between different types of communication architecture designs with PE=2, 8-bit integer. 80
Table 4.7 – Power results comparison between different types of communication architecture designs with PE=2, 8-bit integer. 83
Table 4.8 – Performance results comparison between different types of communication architecture designs with PE=2, 8-bit integer. 84
Table 4.9 – Area results of the ZynqNet System (arch 5, integer) with PE=1, 2, and 4. 86
Table 4.10 – Power results of the ZynqNet System (arch 5, integer) with PE=1, 2, and 4. 88
Table 4.11 – Performance results of the ZynqNet System (arch 5, integer) with PE=1, 2, and 4. 89
Table 5.1 – Area and power consumption of ZynqNet's IFBs. 92
Table 5.2 – Emulated Fault Injection results: ZynqNet's IFBs. 94
Table 5.3 – Area and dynamic power consumption of each Fault-Tolerant ZynqNet IP design. 102
Table 5.4 – Emulated Fault Injection results: fault-tolerant designs. 107
Table 5.5 – Heavy-ions experiment results: fault-tolerant designs. 111
Table 5.6 – Comparison of the state-of-the-art results with this work. 115

CONTENTS

1 INTRODUCTION 18
1.1 Motivation and Problem Statement 18
1.2 Objectives 21
1.3 Contributions 21
1.4 Thesis structure 22
2 BACKGROUND AND STATE OF THE ART 23
2.1 Neural Networks 23
2.1.1 The Perceptron 24
2.1.2 Multiple Layers of Neurons 25
2.2 Convolutional Neural Networks 26
2.2.1 Historical context 26
2.2.2 Architecture of CNNs 26
2.3 All Programmable System on Chip (APSoCs) 31
2.3.1 Programmable Logic (PL) 32
2.3.2 Processing System (PS) 34
2.4 AI accelerators implemented on APSoCs 37
2.4.1 FPGA-based Graphics Processing Unit (FGPU) 38
2.4.2 FINN 39
2.4.3 Systolic Arrays 40
2.4.4 ZynqNet 41
2.5 High-Level Synthesis 43
2.5.1 HLS work-flow 43
2.5.2 Example cases 44
2.5.3 HLS Design Flow 45
2.5.4 Levels of Synthesis 46
2.5.5 Optimization directives 47
2.6 Radiation Effects on APSoCs 49
2.7 Related works 51
3 METHODOLOGY 55
3.1 Design Tools 55
3.2 Case-study 55
3.2.1 APSoC 55

3.2.2 AI Inference Engine 55
3.2.3 Application 58
3.2.4 CNN model 59
3.3 Design evaluation metrics 60
3.4 Reliability Testing 60
3.4.1 Emulated Fault Injection 61
3.4.2 Radiation Experiment 65
4 DESIGN SPACE EXPLORATION: EFFICIENCY AND PERFORMANCE 69
4.1 Application of quantization techniques 69
4.2 Data packing and communication architectures 77
4.3 Summary of results 90
5 DESIGN SPACE EXPLORATION: RELIABILITY ENHANCEMENTS 92
5.1 Unmitigated ZynqNet 92
5.1.1 Investigating the susceptibility of ZynqNet under SEUs 93
5.2 Fault-Tolerant ZynqNet 94
5.2.1 Reliability testing via Fault Injection 104
5.2.2 Reliability testing via Radiation Experiment 109
5.3 Discussion and results comparison with state-of-the-art 114
5.4 Summary of results 115
6 CONCLUSIONS 117
REFERENCES 119
APPENDIX A — PUBLICATIONS 127
APPENDIX B — RESUMO EXPANDIDO EM PORTUGUÊS 128

1 INTRODUCTION

1.1 Motivation and Problem Statement

Real-time image classification and object detection are important capabilities in applications such as autonomous vehicles (HU et al., 2019), defense systems (D'ACREMONT et al., 2019), aerospace platforms (DOĞRU et al., 2020), and satellite-based remote sensing (EL GHAZOUALI et al., 2024). Artificial Intelligence (AI) methods based on Deep Learning (DL), particularly Convolutional Neural Networks (CNNs), are powerful tools for performing these tasks, as they can extract hierarchical features from high-dimensional data (LECUN et al., 2015; KRIZHEVSKY et al., 2017).

However, deploying CNNs on embedded platforms presents significant challenges for such applications. These devices are inherently resource-limited, with strict constraints on memory, computation, and power consumption. Techniques such as quantization, which reduce the numerical precision of network weights and activations, are widely used to adapt CNNs to embedded hardware. Post-training quantization is simple and fast but may reduce accuracy, while quantization-aware training allows models to adapt to low-precision hardware during training, improving performance at the cost of higher computational and training resource requirements (DAI et al., 2024). These limitations highlight the trade-offs between model accuracy, computational efficiency, and hardware constraints, which must be carefully balanced for successful deployment on embedded systems.

An All Programmable System-on-Chip (APSoC) is an integrated device that combines embedded processors with reconfigurable logic, providing a flexible platform for implementing Artificial Intelligence (AI) inference engines on resource-constrained embedded systems. It supports high parallelism, low latency, and efficient on-chip memory usage. Key APSoC families include Zynq-7000, Zynq UltraScale+, and Versal Adaptive Compute Acceleration Platform (ACAP), which differ in architecture, process node, processor configuration, memory, and AI acceleration capabilities. Zynq-7000 is a SoC with dual-core Advanced RISC Machines (ARM) Cortex-A9 and 28 nm Field Programmable Gate Array (FPGA) fabric, targeting general embedded applications (XILINX, 2023b). Zynq UltraScale+ is a Multiprocessor System on Chip (MPSoC) with quad-core Cortex-A53, dual-core Cortex-R5, and 16 nm logic, designed for high-performance embedded workloads (XILINX, 2025b). Versal ACAP integrates scalar processors (Cortex-A72 and Cortex-R5), adaptable logic, high-bandwidth memory, and dedicated AI/Machine Learning (ML) engines in a heterogeneous ACAP architecture at 7 nm,

supporting AI, data center, and 5G applications (XILINX, 2025a). APSoCs can be classified based on their configuration memory technology, with the most important types being Static Random Access Memory (SRAM)-based APSoCs and Flash-based APSoCs. SRAM-based APSoCs use volatile memory that must be loaded from external storage at boot, providing fast reconfiguration, high logic density, and high-performance operation. In contrast, Flashed-based APSoCs retain their configuration on-chip (non-volatile), allowing instant-on operation, and lower static power, but generally offering lower logic density and slower maximum frequency.

In Zynq APSoCs, AI inference engines are typically deployed in the FPGA at the Programmable Logic (PL) part of the device, while the ARM Central Processing Unit (CPU) cores at the Processing System (PS) are responsible for coordinating tasks and interfacing with peripherals. ZynqNet (GSCHWEND, 2016) is one of these inference engines developed for CNNs. It is designed with a lightweight custom state machine for control at the FPGA, enabling standalone inference execution without CPU intervention. Compared to engines such as FINN (UMUROGLU et al., 2017; BLOTT et al., 2018), which generate dedicated hardware blocks for each layer (e.g., convolution, fully connected, pooling), ZynqNet achieves a smaller hardware footprint by reusing the same processing structure across all CNN layers. This reduced area makes it significantly more feasible to apply TMR-based fault-tolerance techniques on devices with limited FPGA resources. Compared to conventional systolic array (SA)-based accelerators (MAFFAZIOLI et al., 2025), ZynqNet can integrate the non-linear activation function ReLU and global pooling operations directly into its processing pipeline, avoiding the need for separate hardware or additional data movement.

Designing an efficient AI accelerator on an FPGA involves several challenges (ZHANG et al., 2015). Loop tiling is crucial to ensure manageable data subsets fit on-chip, as improper division can hinder data reuse and parallelism. The arrangement of processing elements (PEs), buffer banks, and their interconnections must be carefully planned for efficient data processing. Additionally, the throughput of the PEs must align with the FPGA's off-chip bandwidth to avoid bottlenecks, as CNNs inherently challenge both throughput and energy efficiency due to significant data movement between on-chip and off-chip memory, which consumes more energy than computation (CHEN et al., 2017). To optimize performance, minimizing read requests to main memory and maximizing data bus width for efficient transfer is essential. To efficiently implement AI inference engines on APSoCs platforms, recent advances in High-Level Synthesis (HLS) have simplified the process of translating CNN architectures into FPGA logic, reducing the development time while maintaining hardware-level optimization (MARAÑON et al., 2025; ZHANG et al., 2015).

In radiation-prone environments, system reliability can be compromised by external factors. SRAM-based FPGAs, which store configuration data in SRAM cells, are particularly vulnerable to radiation-induced faults both in space and on Earth. Energetic particles can trigger Single Event Upsets (SEUs) in the configuration memory, including Single Bit Upsets (SBUs) and Multiple Bit Upsets (MBUs), potentially altering the circuit's functionality. A Single Event Upset (SEU) occurs when a charged particle deposits enough energy into a memory element, causing an unintended change in its state, which can compromise data integrity, especially in FPGAs that use SRAM (SOUVATZOGLOU et al., 2024a). MBUs may occur more frequently in environments with elevated radiation exposure, potentially disturbing clusters of adjacent memory cells or logic units. Modern devices implement mitigation techniques such as bit-interleaving and Error Correcting Codes (ECC) to reduce the likelihood that MBUs lead to system failures (AGUIAR et al., 2025). Nevertheless, the possibility of MBUs causing failures remains, particularly in high-radiation conditions (XIONG et al., 2022; GONCALVES et al., 2020), and must be considered in the design of reliable AI accelerators.

The problem addressed in this thesis is the need to design high-performance and reliable AI accelerators on SRAM-based APSoCs. These devices are selected over conventional microcontrollers or embedded GPUs because they allow fine-grained hardware customization for CNN acceleration while maintaining tight integration with the processing system, thereby enabling high performance and energy efficiency. However, this architectural advantage is challenged in mission-critical environments, where the inherent sensitivity of SRAM-based FPGA to radiation-induced faults can compromise system integrity. Although fault-tolerance strategies such as Triple Modular Redundancy (TMR), Error Detection and Correction (EDAC), and memory scrubbing, have proven effective in general FPGA designs (SOUVATZOGLOU et al., 2023; BENEVENUTI et al., 2021b; BRAGA et al., 2021; LIU et al., 2024; STERPONE e VIOLANTE, 2005), its specific application to AI accelerators remains underexplored. Furthermore, there are limited comprehensive comparative analyses that evaluate the trade-offs between these strategies and their impact on resource utilization and power. Consequently, a systematic methodology is required to assess, integrate, and optimize multiple fault-tolerance mechanisms while maintaining the high-performance benefits of APSoC-based AI accelerators in high-radiation environments.

This thesis proposes a design flow that addresses these challenges by optimizing ZynqNet for arithmetic precision and parallelism, implementing fault-tolerant mechanisms, and evaluating reliability through emulated fault injection and heavy-ion radiation testing. The methodology is generalizable to other AI accelerators deployed on SRAM-based APSoCs,

providing a pathway toward high-performance, energy-efficient, and robust AI computation in real-world applications exposed to radiation.

1.2 Objectives

The main objective of this thesis is to design, optimize, and validate an implementation flow for AI accelerators on SRAM-based APSoCs, combining performance optimization techniques with fault-tolerance mechanisms to enable efficient and reliable deployment in radiation-prone environments. Two essential aspects are addressed: improving computational efficiency by reducing latency, power, and resource utilization, and enhancing reliability by applying fault mitigation strategies tailored to APSoC-based AI accelerators.

To achieve the main objective, the following specific objectives were performed:

- Performance Optimization: Implement and evaluate the ZynqNet AI inference engine on a Zynq-7000 APSoC by applying quantization techniques, parallelism strategies, and data packing mechanisms to improve inference latency and to reduce FPGA resource utilization and power consumption.
- Fault-Tolerant Architecture Design: Integrate and compare different redundancy schemes, including Coarse-Grained, Modular, and Distributed TMR at design level, Error Detection and Correction (EDAC) mechanisms in FPGA's block memories and memory scrubbing in the configuration memory in order to mitigate radiation-induced faults.
- Reliability Evaluation: Develop a fault-injection methodology to identify the most vulnerable internal components, compare the different fault-tolerant accelerators, and validate the hardened designs through a heavy-ion radiation test campaign.
- Design Trade-off Analysis: Compare the proposed design configurations in terms of reliability, resource utilization, and power.

1.3 Contributions

This thesis advances the state of the art in fault-tolerant AI accelerator design on SRAM-based APSoCs by providing a validated methodology that combines performance optimization and reliability enhancement. The work presents a case-study deployment using the ZynqNet AI inference engine with improved computational efficiency through quantization, parallelism,

and data transfer optimizations, while also enhancing reliability through multiple fault-tolerance strategies to mitigate radiation-induced faults in radiation-prone environments. This work provides design guidelines that can be generalized to other AI inference engines intended for mission-critical applications. Finally, the methodology and hardened ZynqNet design offer general applicability, allowing other CNN applications to be deployed on the SRAM-based APSoC, provided that the input image and feature map dimensions, as well as the number of weights across all layers, are compatible with the available cache capacities.

1.4 Thesis structure

This dissertation is organized as follows. Chapter 2 provides the theoretical background, it covers the fundamentals of Neural Networks (NNs) and Convolutional Neural Networks (CNNs), describes the commonly adopted APSoC platforms and the implementation of AI accelerators on them, presents the High-Level Synthesis (HLS) process to generate Register Transfer Level (RTL) designs from C++ code, discusses the radiation effects on APSoCs and finally, presents a description of related works. Chapter 3 explains the methodology adopted in this research, including the design tools used, the case-study based on the ZynqNet accelerator implemented on the APSoC, the dataset employed, the architecture of the CNN model, the design metrics considered for evaluation such as area, power, and performance, and the description of the reliability testing methodology through both emulated fault injection and radiation experiments. Chapter 4 explores the design space for performance and efficiency optimization, analyzing the impact of quantization techniques, data packing and data transfer schemes, and presenting the corresponding results. Chapter 5 explores the design space to generate designs with higher reliability. It analyzes the susceptibility of internal components of ZynqNet under emulated radiation-induced faults, implements fault-tolerant mitigation strategies, validates all designs under fault injection and heavy-ion radiation experiments, and presents the results. Finally, Chapter 6 presents the conclusions of this thesis and outlines potential directions for future research. Appendix A details the publications generated during the development of this thesis.

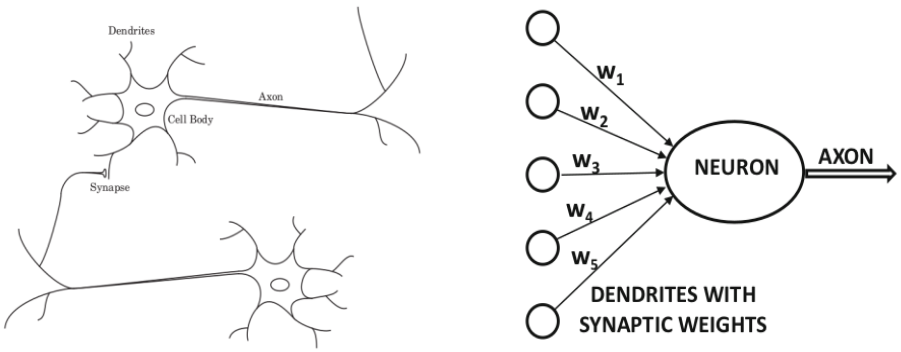
2 BACKGROUND AND STATE OF THE ART

This chapter presents the theoretical foundations relevant to the research, including Neural Networks, Convolutional Neural Networks, All Programmable System on Chip (APSoC) platforms, AI accelerators implemented on APSoCs, High-Level Synthesis, Radiation Effects on APSoCs, and related works.

2.1 Neural Networks

A Neural Network (NN) is a computational model designed to emulate the information processing mechanisms of neural cells in the human brain. There are two principal similarities between neural cells and NN. First, both are composed of basic units that perform simple functions and are highly interconnected. Second, the way in which they are connected determines the overall functionality (Figure 2.1).

Figure 2.1 – Neural cells and NN model.



(a) Drawing of neural cells. (b) Basic model of NN.

Source: (a) extracted from (HAGAN et al., 2014); (b) extracted from (AGGARWAL, 2018).

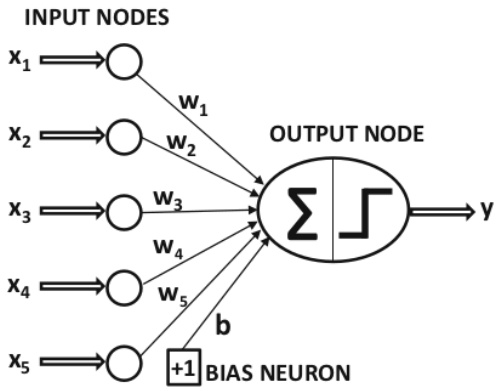
The effectiveness of NN lies in its ability to integrate a large number of basic computational units and iteratively adjust the connection weights to minimize prediction errors. This learning process requires sufficiently large datasets, which allow the model to generalize effectively and properly tune the parameters involved (HAGAN et al., 2014).

2.1.1 The Perceptron

A perceptron is a type of artificial neuron developed in the 50's and 60's by Frank Rosenblatt, who was inspired by a previous work of Warren McCulloch and Walter Pitts (NIELSEN, 2019).

It consists of a single input layer and an output node. It is a binary classification model, and it is represented in (Figure 2.2). A sign function is applied to the real output value of the sum linear function and bias to obtain the binary prediction. The bias value is necessary to overcome imbalance problems with class distribution data, for example, a case where the mean of the characteristic variables is zero, but the binary class is not.

Figure 2.2 – Perceptron model



Source: Extracted from (AGGARWAL, 2018).

Eq. (2.1) represents the behavior of the model:

$$\hat{y} = sign\{\underline{W}.\underline{X} + b\} = sign\left\{\sum_{j=1}^{d} w_j x_j + b\right\} \quad (2.1)$$

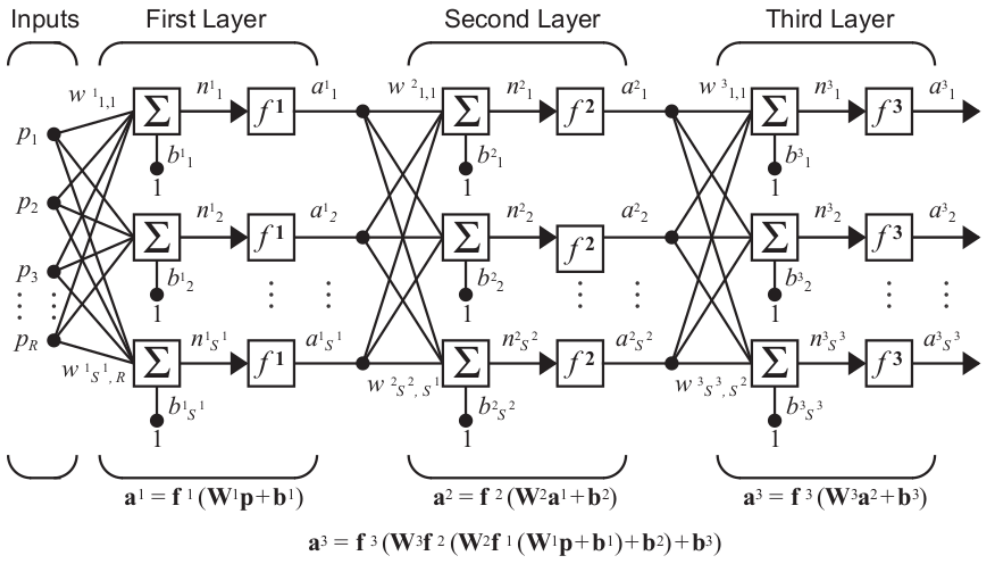
The sign function acts as an activation function and generates an output of -1 or +1, the variable $\hat{y}$ represents the predicted value, b is the bias, $\underline{W}$ and $\underline{X}$ are the matrix values of weights and features respectively.

2.1.2 Multiple Layers of Neurons

Multilayer networks offer greater capability than single-layer networks because they can capture more complex and nonlinear relationships of the input features and output. The NN receives inputs in the first layer, and it is composed of a number of hidden layers and an output layer. The hidden layers, also known as intermediate layers, perform various computations to recognize and interpret intricate patterns and representations. The output layer generates the prediction results. The number of inputs and outputs are determined by external specifications of the problem to be solved.

Figure 2.3 depicts a NN with three layers and it is composed of two hidden layers and an output layer. Since the two hidden layers are formed exclusively by neurons that are fully connected to the next layer, they are referred to as dense or fully connected layers. In each layer, the variables w, b and a represent the weights, bias and outputs respectively. The number of inputs and neurons of the NN is R and S. There are superscripts in some variables that indicate the respective layer.

Figure 2.3 – Example of a three layer network



Source: Extracted from (HAGAN et al., 2014).

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a type of NN that has a convolutional operation in at least one layer. What distinguishes CNNs from traditional neural networks is their unique layers designed to automatically detect and learn hierarchical patterns and features in the input data. They are designed to handle grid structured inputs where pixels in a local area have a significant type of relationship.

2.2.1 Historical context

The starting point of convolutional neural networks was sparked by Hubel and Wiesel's experiments on cats' visual cortex (HUBEL et al., 1959). They found that specific cells in this cortex respond to particular regions in the visual field, with their activation pattern influenced by the shape and orientation of visual stimuli such edges. This discovery hinted that mammals utilize layered cortical structures to process different levels of visual information. From a machine learning viewpoint, this reflects hierarchical feature extraction. CNNs mimic this process by capturing simple patterns in initial layers and progressively more intricate ones in subsequent layers.

A neural model known as neocognitron (FUKUSHIMA, 1980) developed by Kunihiko Fukushima in the 80s was one of the first models inspired by the functioning of the biological visual system. Despite not explicitly employing the modern concept of weight sharing in convolutional layers like current CNNs do, the Neocognitron established the foundational understanding of organizing neural layers hierarchically to process visual information effectively.

LeNet-5 (LECUN et al., 1998) surged as an early fully convolutional architecture. It was very used by banks for handwritten numbers recognition on checks, giving a practical application in optical recognition tasks. This network significantly advanced the field of deep learning, particularly in computer vision.

2.2.2 Architecture of CNNs

The architecture of CNNs is specifically designed to process effectively structured data, such as images or sequences. Principal components of a CNN architecture are detailed in the following lines:

2.2.2.1 Convolution layer

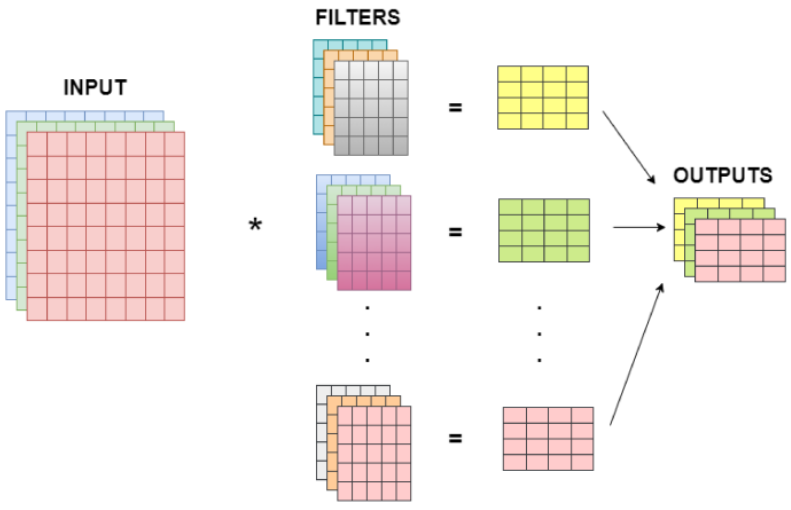
The main difference between a convolutional and a dense layer is that the first learns local patterns, whereas the second learns global patterns. The Convolutional Layer serves as the fundamental unit in CNN architecture. Its parameters include a group of trainable filters, also referred to as kernels, illustrated in Figure 2.4. Important hyperparameters mostly used within this layer are number of filters, filter size, stride and zero padding (BOUAAFIA et al., 2022).

Eq. (2.2) represents the convolution operation:

$$x_j^l = f\left(\sum_{i \epsilon M_j} x_i^{l-1} \, x \, k_{ij}^l + b_j^l\right) \tag{2.2}$$

Variables such as xj, xi, kij, bj, Mj and f represent the output of the current layer, the output of the previous layer, the current layer kernel, the current layer bias, the input feature map and the activation function, respectively.

Figure 2.4 – Convolution process



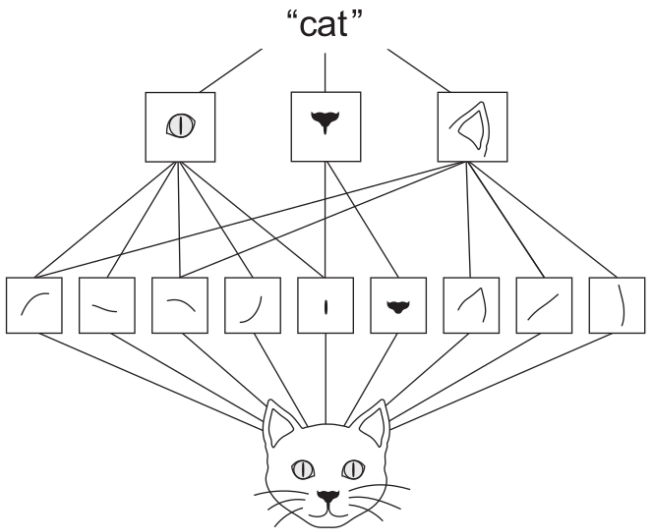
Source: Extracted from (BOUAAFIA et al., 2022).

Convolutional layers are effective because they can recognize learned patterns regardless of their position in an image, thanks to the sharing of features across the entire input. Moreover, they learn representations in a hierarchical way: lower layers capture simple features

such as edges, while higher layers progressively combine them to identify more complex patterns and concepts.

The next image (Figure 2.5), shows how the different simple features combine with each other to increase the hierarchy level of the characteristic. In this example, tiny visual details combine to form local features such as eyes or ears, which in turn come together to create higher-level concepts like "cat."

Figure 2.5 – Spatial hierarchy levels of features from a picture of a cat



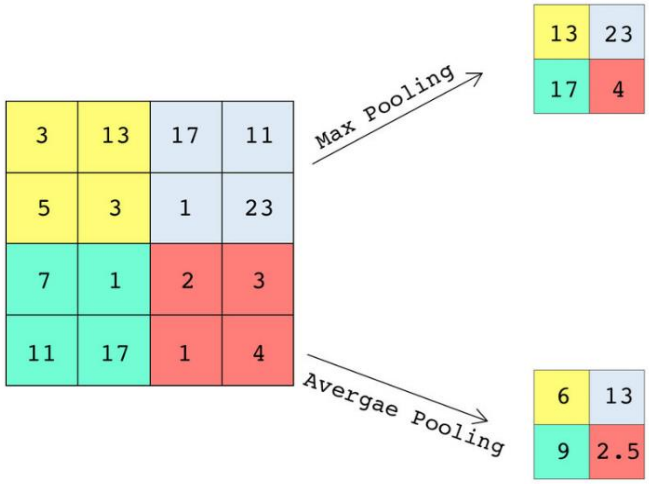
Source: Extracted from (CHOLLET, 2018).

2.2.2.2 Pooling operation

In CNNs, a pooling operation comes after the convolution and activation function layers to decrease the spatial dimensions of the input, which in turn reduces the number of parameters in the network. Pooling is a subsampling method commonly applied following a convolution operation.

Max and average pooling, as shown in Figure 2.6, are the most used types of pooling.

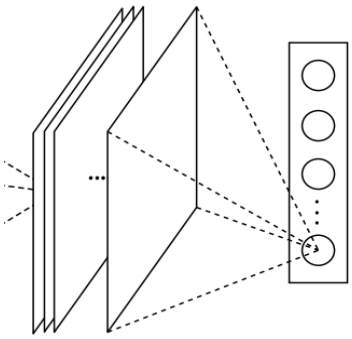
Figure 2.6 – Max and average pooling



Source: Extracted from (BOUAAFIA et al., 2022).

Another type of pooling layer is known as global sum pooling, where each feature map is condensed into a single value by summing all its elements (Figure 2.7).

Figure 2.7 – Global sum pooling



Source: Extracted from (LIN et al., 2014).

2.2.2.3 Fully Connected layer

The classification phase can consist of multiple Fully Connected (FC) layers activated by a function. The extracted features, from the CNN layers, are flattened and fed into the FC layers, where each input connects to every neuron. FC layers, usually placed at the end of the CNN architecture, perform tasks such as class ranking. Their connections determine how well each value corresponds to a specific class (BOUAAFIA et al., 2022).

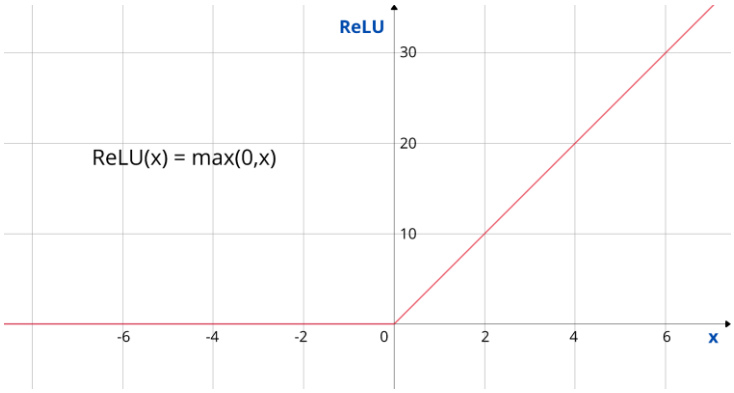
2.2.2.4 Activation Functions

Activation functions are essential in neural networks, such as Convolutional Neural Networks (CNNs), because they introduce non-linearity, enabling the network to learn complex data patterns. Common activation functions are described below:

A) ReLU (Rectified Linear Unit)

Outputs the input directly if positive; otherwise, it outputs zero. ReLU is popular for its simplicity and effectiveness in deep learning (Figure 2.8).

Figure 2.8 – ReLU activation function

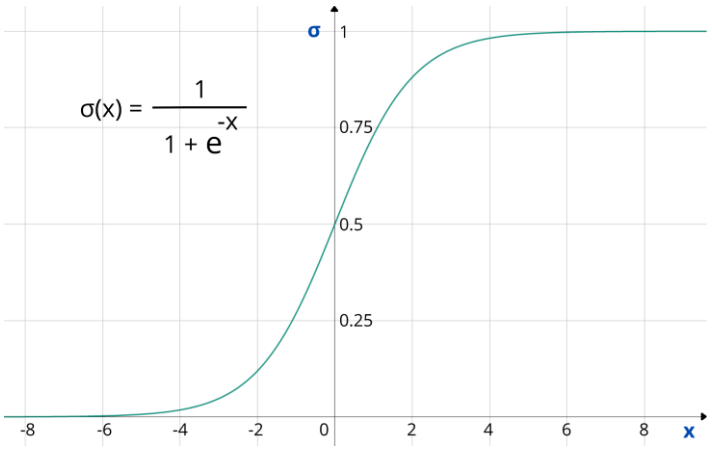


Source: The author.

B) Sigmoid

Maps the output to a range of 0 to 1, which makes it ideal for binary classification tasks (Figure 2.9).

Figure 2.9 – Sigmoid activation function



Source: The author.

C) Softmax

Transforms a vector of values into a probability distribution commonly used in the final layer of classification networks to determine class probabilities, as described in Eq. (2.3).

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_{k=1}^{n} e^{x_k}} \tag{2.3}$$

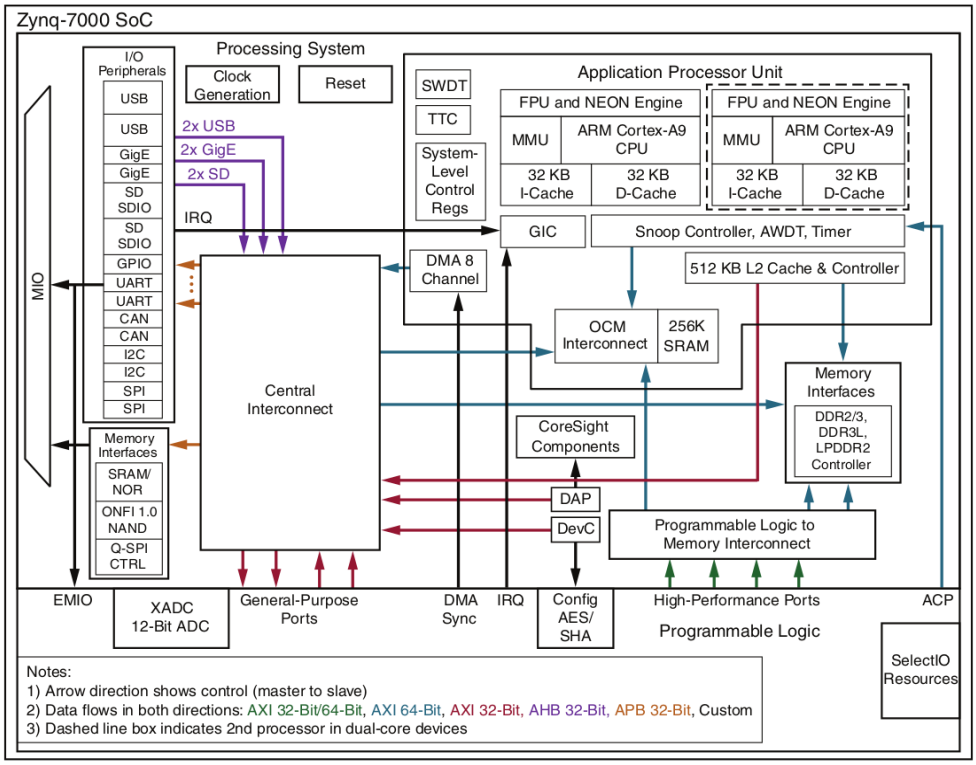
2.3 All Programmable System on Chip (APSoCs)

All Programmable System on Chip (APSoC) are integrated platforms that combine a processing system with reconfigurable logic on a single chip. This combination allows the designers to implement both software and hardware design on the same platform, offering flexibility, high-performance, and energy efficiency in the applications. Among the most widely used APSoCs are those from AMD Xilinx, including the Zynq-7000, Zynq UltraScale+ MPSoC and Versal ACAP; from Altera, such as the Arria 10 and Cyclone V SoC FPGA, and from Microchip, including the PolarFire and SmartFusion SoC FPGAs.

This subsection focuses on the AMD Xilinx Zynq-7000 ARM/FPGA APSoC, describing the main components and the architecture of the processing system and the programmable logic parts.

The Zynq-7000 series (Figure 2.10) integrates a powerful dual core Arm Cortex-A9 MPCore Processing System (PS) with AMD Programmable Logic (PL) in one chip, leveraging cutting-edge, high-performance, low-power, with a 28 nm technology.

Figure 2.10 – Zynq 7000 SoC architecture



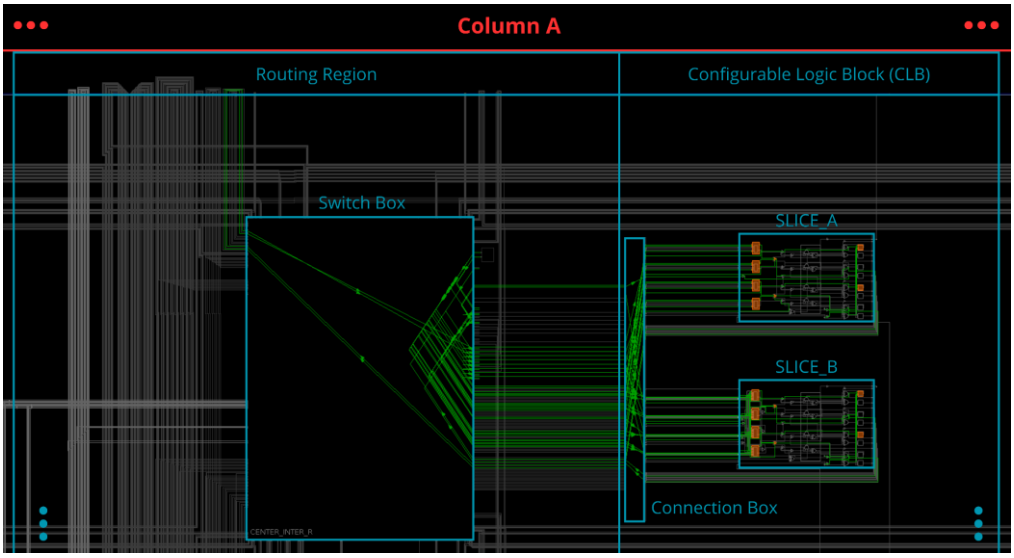
Source: (XILINX, 2018b)

2.3.1 Programmable Logic (PL)

In the SRAM-based Zynq-7000 ARM/FPGA APSoC, the PL offers a diverse set of Configurable Logic Blocks (CLBs), Digital Signal Processing blocks (DSPs), and Block RAMs (BRAMs), allowing for customized hardware acceleration, real-time operations, and intricate data handling.

Each CLB contains two slices, each one equipped with Look-Up Tables (LUTs) which implement combinational logic through truth tables; D-type Flip-Flops (DFFs) which provide storage elements for sequential operations; and multiplexers (MUXs), which control data flow selection. These components interact with the Switch Box through the Connection Box (Figure 2.11) and enable the communication with other functional elements.

Figure 2.11 – Instantiation of a CLB per column in the FPGA floorplanning.



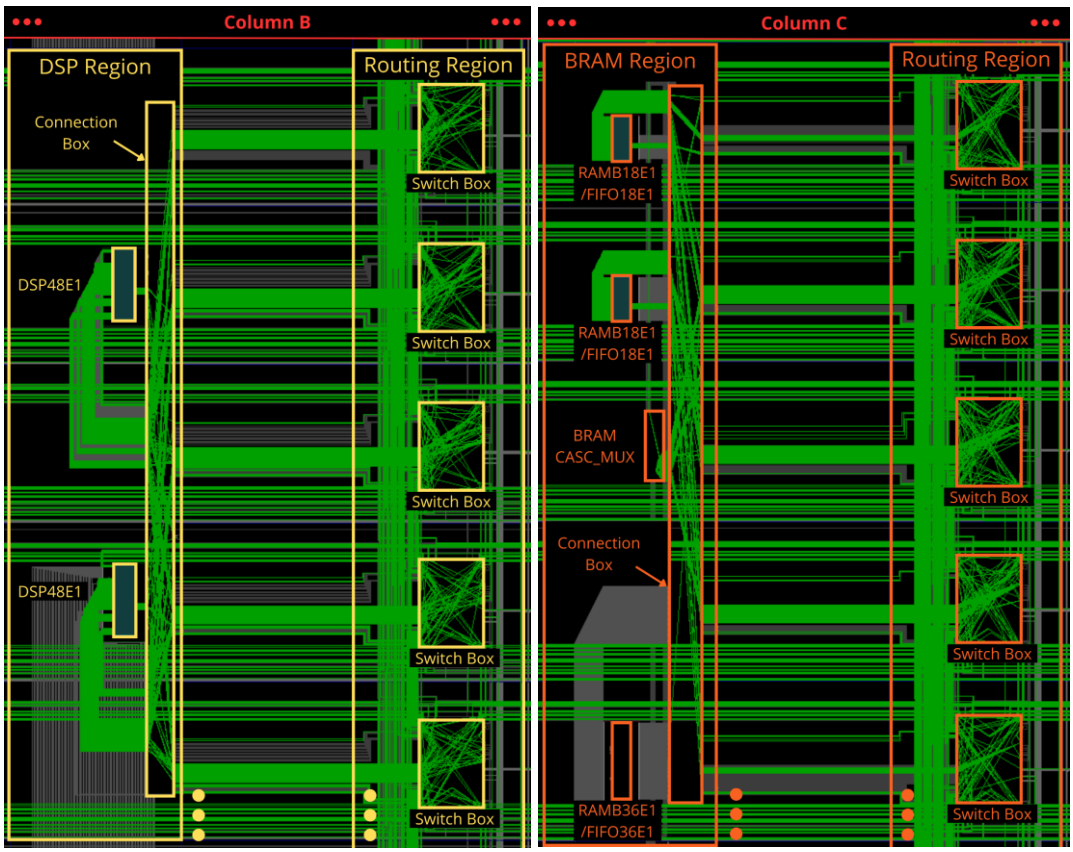
Source: The author

Specialized components like DSP blocks are capable of performing complex mathematical operations, such as multiply-accumulate (MAC), which are crucial tasks in the CNN processing. In FPGA floorplanning, there are columns dedicated to placing DSP blocks. Within each column, we can identify multiple DSP Regions and a Routing Region. Each DSP Region contains two DSP48E1 components which support functions such as shifting, logical operations, subtraction, addition, multiplication, accumulation, and pattern detection. It also includes a Connection Box, which enables communication with external components. The Routing Region spans the column and includes several Switch Boxes, which facilitate communication between DSP Regions and with other columns.

Block-RAMs are used in applications that require fast memory access and intermediate data storage. They enable quick read and write operations for the logic elements. Similar to DSPs columns, FPGA floorplanning includes columns dedicated to Block-RAMs. Each BRAM Region contains two configurable memory blocks that support 18 Kb configurations and can be instantiated as either RAMB18E1 or FIFO18E1, depending on the intended functionality. It also includes a configurable memory block that supports a 36 Kb configuration, which can be instantiated as either a RAMB36E1 or a FIFO36E1. To expand memory capacity, the BRAM_CASC_MUX can be used to enable cascading between BRAM blocks. Communication with other components outside each BRAM Region is handled by a Connection Box. As with DSP columns, the Routing Region spans the column and includes

several Switch Boxes to facilitate communication between BRAM Regions and with other columns.

Figure 2.12 – Instantiation of DSPs and BRAMs in the FPGA floorplanning.



(a) DSP column (b) Block-RAM column

Source: The author.

In the following lines, the interconnection components of the Zynq-7000 PS are described to understand how it is structured internally and to appropriately select its Advanced eXtensible Interface (AXI) communication interfaces and buses for our application.

2.3.2 Processing System (PS)

The processing system, driven by Arm Cortex-A9 MPCore CPUs, also features On-Chip Memory (OCM), external memory interfaces, and a comprehensive set of I/O peripherals. It consists of the following functional blocks:

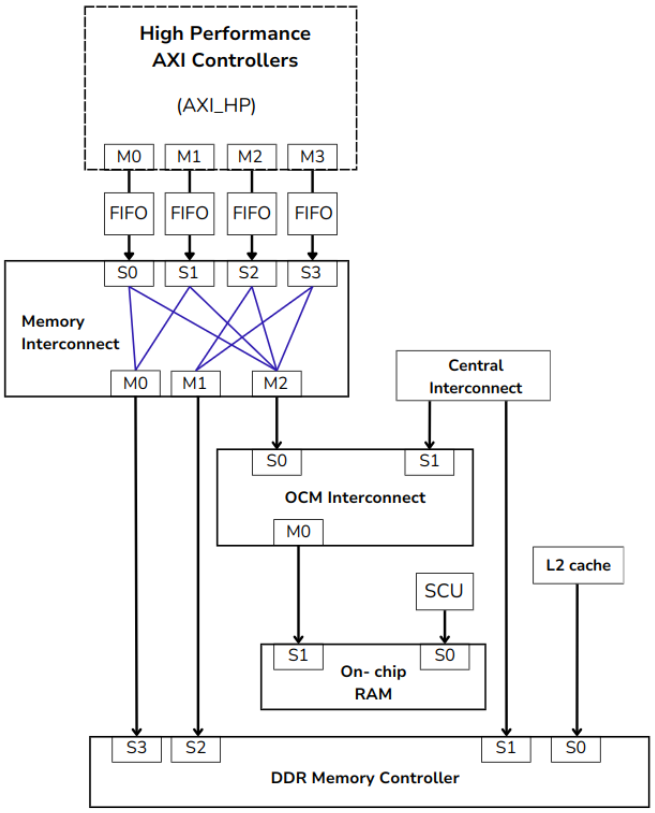
- Application Processor Unit (APU): It is a key component designed to handle general-purpose computing tasks. It can be configured as either a single-core or dual-core processor.
- Memory interfaces: They are designed to ensure effective access to both internal and external memory resources, which is crucial for achieving high-performance in computing and data management. Examples: DDR Controller, Quad-SPI Controller and Static Memory Controller (SMC).
- I/O peripherals (IOP): The I/O Peripherals consist of various standard interfaces used for communicating data with external devices. Examples: General Purpose Input Output (GPIO) and serial communication controllers.
- Interconnect: Zynq-7000 SoC devices employ various interconnect technologies tailored to meet the specific communication requirements of their functional blocks.

The interconnect within the PS features multiple switches that link system resources using AXI point-to-point channels. These channels handle the communication of addresses, data, and response transactions between master and slave clients. Some interconnect blocks communicate the Processing System with the Programmable Logic through AXI interfaces, and these are the 32/64-bit High-Performance (HP) ports, 64-bit Accelerator Coherency (ACP) port, and the 32-bit General Purpose (GP) ports.

A) AXI HP interfaces

It consists of four AXI_HP interfaces that provide high-bandwidth data paths from PL bus masters to the DDR and OCM memories. Each interface includes two FIFO buffers to handle read and write operations. The PL to Memory Interconnect routes these high-speed AXI_HP interfaces to either two DDR memory ports or the OCM (Figure 2.13).

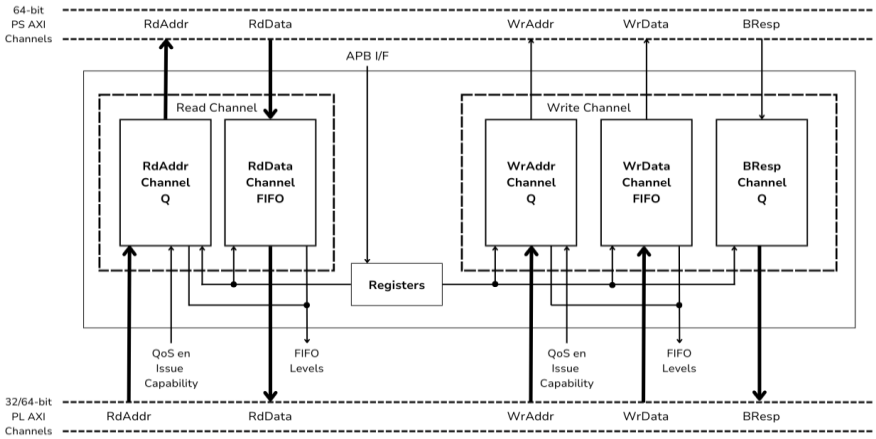
Figure 2.13 – High Performance Blocks used for the connection of PL to PS AXI Channels.



Source: Adapted from (XILINX, 2023b)

Figure 2.14 shows the blocks of connection between 32/64-bit channels from the PL side to the 64-bit channels from the PS side, it includes the transmission of addresses and data to read and write, and the BResp values of the write operation.

Figure 2.14 – High Performance Blocks used for the connection of PL to PS AXI Channels.

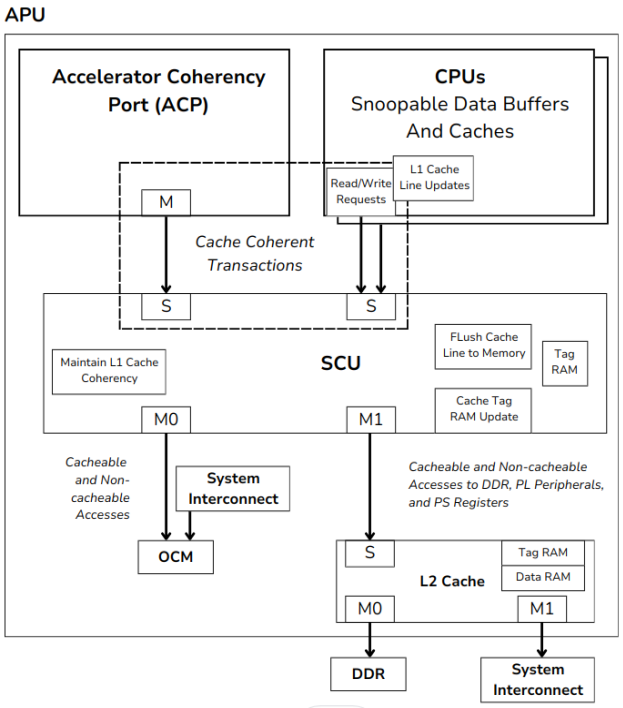


Source: Adapted from (XILINX, 2023b).

B) AXI ACP interface

The Accelerator Coherency Port (ACP) provides fast access for PL masters, with the option for cache coherence with Level 1 (L1) and Level 2 (L2) caches (Figure 2.15). From a system viewpoint, the ACP offers connectivity similar to that of the APU CPUs. This close integration means the ACP competes directly with the CPUs for accessing resources outside the APU block.

Figure 2.15 – ACP Connectivity Diagram.



Source: Adapted from (XILINX, 2023b).

C) AXI GP interfaces

These interfaces are linked directly to the master and slave interconnect ports without extra FIFO buffering, in contrast to the AXI_HP interfaces that have extensive FIFO buffering to enhance performance and throughput. Consequently, their performance is limited by the capabilities of the master and slave interconnect ports. These interfaces are intended for general-purpose applications and are not designed for high-performance use.

2.4 AI accelerators implemented on APSoCs

Accelerating CNNs through specialized hardware has become a common strategy in recent years, driven by the need for higher efficiency and computational speed. Among the available platforms, FPGAs in APSoCs offer a compelling solution due to their adaptability and capacity for parallel processing.

This subsection explores four APSoC-based AI accelerators tailored for CNN workloads: FPGA-based Graphics Processing Unit (FGPU), FINN, Systolic Arrays and ZynqNet. Each design employs a unique architectural approach to enhance inference performance on programmable hardware.

2.4.1 FPGA-based Graphics Processing Unit (FGPU)

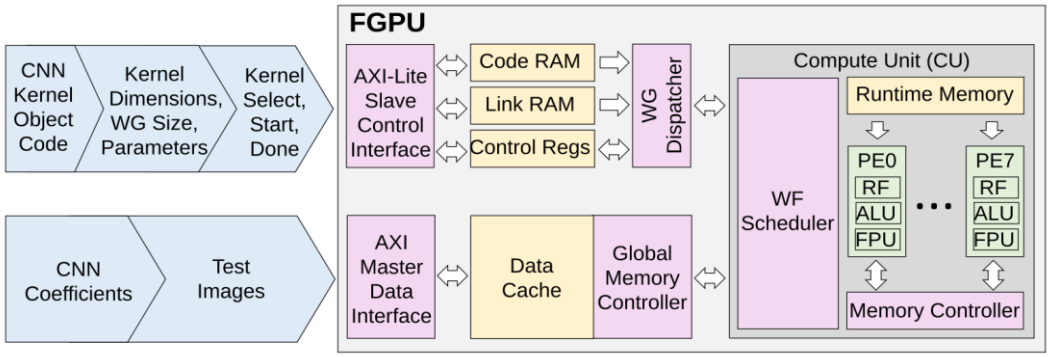
The FPGA-based Graphics Processing Unit (FGPU) is a softcore accelerator designed according to the SIMT (Single Instruction, Multiple Threads) paradigm and tailored for efficient FPGA implementation. It integrates a deep pipeline and a scalable architecture composed of multiple replicated Processing Elements (PEs). Each PE includes its own register file, an integer arithmetic unit, and optionally, dedicated hardware for floating-point operations. These PEs are organized into Compute Units (CUs), which follow the SIMD (Single Instruction, Multiple Data) model, enabling the simultaneous execution of up to eight data-parallel operations under a shared instruction stream. To support memory-intensive applications, the FGPU provides access to external global memory via AXI interfaces and includes an on-chip data cache to improve memory throughput.

Applications targeting the FGPU are written in OpenCL™ and executed under a hierarchical model. A computational kernel, such as a vector subtraction, is decomposed into multiple Work Items (WIs), each representing a single thread that processes a portion of the data. These WIs are grouped into Wave Fronts (WFs), which are independent thread batches that can run concurrently on a CU. Multiple WFs form a Work Group (WG), ensuring that all threads within a WG are assigned to the same CU. During execution, the compiled OpenCL kernel is stored in Code RAM, while the kernel's starting address, NDRange configuration, and memory pointers to external DDR data are placed in Link RAM (LRAM). A host processor is responsible for launching and configuring the accelerator by transmitting instructions and execution parameters through an AXI4-Lite interface (Figure 2.16) (BRAGA et al., 2021; BENEVENUTI et al., 2022b).

The software toolchain supporting the FGPU includes a compiler framework based on Low Level Virtual Machine (LLVM). It employs Clang as the frontend to process OpenCL

code and uses a custom backend that translates the program into the FGPU's proprietary instruction set. Although only the essential features of OpenCL 1.1 are supported by default, the framework is designed to be extensible, allowing future enhancements without altering the core architecture. Thanks to this compatibility, OpenCL code developed for standard GPUs can be compiled and executed on the FGPU with minimal modifications (GONCALVES et al., 2020).

Figure 2.16 – Data flow received by FGPU running an image classification application.



Source: Extracted from (BENEVENUTI et al., 2022b).

2.4.2 FINN

FINN is a flexible framework designed to facilitate the deployment of Quantized Neural Networks (QNNs) on reconfigurable hardware platforms such as APSoCs. QNNs are simplified forms of deep neural networks that use reduced-precision numerical formats. By representing elements like weights, activations, and biases with fewer bits, QNNs significantly decrease memory usage, power consumption, and logic resource demands. This efficiency makes them particularly well-suited for embedded systems with limited computational capacity and strict energy constraints (UMUROGLU et al., 2017).

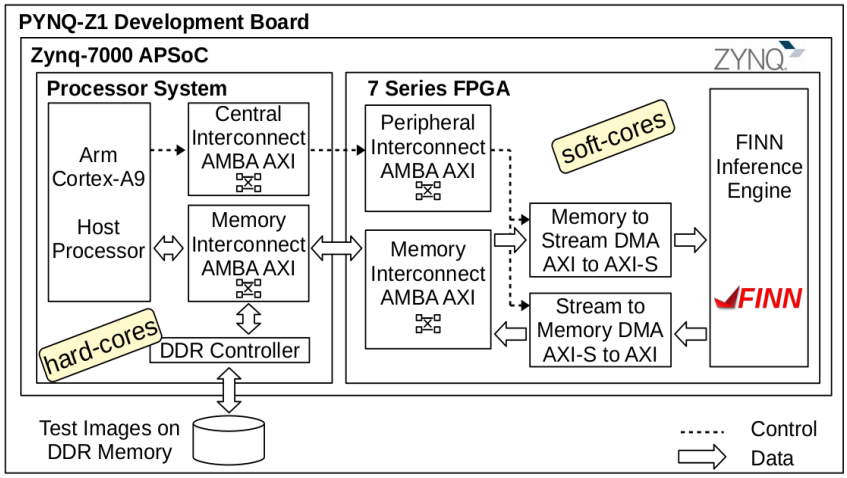
The FINN design flow begins with training and quantizing a neural network using, for instance, PyTorch, along with the Brevitas library, which enables quantization-aware training. Once trained, the model is exported in the ONNX (Open Neural Network Exchange) format, which serves as the standardized input for FINN's build system.

At this stage, users can define two configuration parameters related to folding called the number of PEs of each layer and the Single Instruction Multiple Data (SIMD) of each PE. The quantity of processing elements (PEs) per layer represents how many neurons are activated and execute computations simultaneously, while the number of SIMD lanes reflects how many

operations each PE can perform in parallel. A smaller folding increases parallelism and improves performance, but requires more FPGA resources (e.g., LUTs, DFFs, BRAMs). Conversely, a higher folding reduces hardware usage at the expense of throughput. This setting can be manually configured via a JSON file, or it can be automatically optimized by specifying a target performance level.

In the next phase of the process, the model is transformed into synthesizable hardware using Vivado or Vitis HLS tools. The generated C++ code is converted into Verilog and packaged as a reusable IP block. This IP can then be integrated into a Vivado project and implemented on the desired FPGA target (Figure 2.17) (ELY et al., 2024; SOUVATZOGLOU et al., 2023).

Figure 2.17 – Block diagram interconnection of FINN in a Zynq-7000 APSoC to run an image classification application.



Source: Extracted from (ELY et al., 2024).

2.4.3 Systolic Arrays

A systolic array is a parallel hardware architecture composed of a grid of integrated PEs that handle data in a fully synchronized and pipelined way. Each PE performs a simple operation, typically a multiply-accumulate, and exchanges data exclusively with neighboring elements. Data propagates through the array according to a predefined dataflow, enabling continuous computation as inputs and intermediate results are forwarded at each clock cycle (KUNG, H. T.; LEISERSON, C. E., 1978).

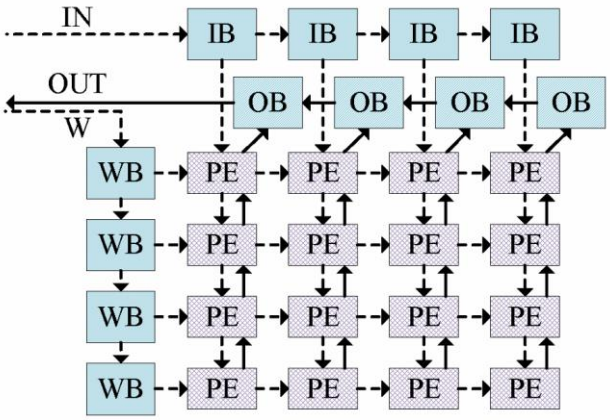
This execution model is highly efficient due to strong spatial and temporal data reuse. Once data is loaded from memory, it is reused by multiple PEs as it traverses the array,

significantly reducing the number of accesses to external memory and lowering bandwidth requirements. Moreover, the regular structure and predictable communication patterns simplify timing closure and support scalable implementations.

When implemented on SRAM-based APSoCs, systolic arrays leverage the programmable logic and on-chip memory resources available in the device. The PEs are typically mapped onto the DSP slices to efficiently implement arithmetic operations, while on-chip BRAMs are employed to buffer input data, weights and partial results. The close coupling between computation and data movement enables high throughput and low latency while maintaining energy efficiency.

Despite these advantages, systolic arrays rely on fixed dataflow schedules and regular computation patterns, which can limit their flexibility when supporting irregular workloads or dynamic control flows. Consequently, they are most effective for structured operations such as matrix multiplication and convolution.

Figure 2.18 – Systolic Array architecture for CNN.



Source: Extracted from (WEI et al., 2017).

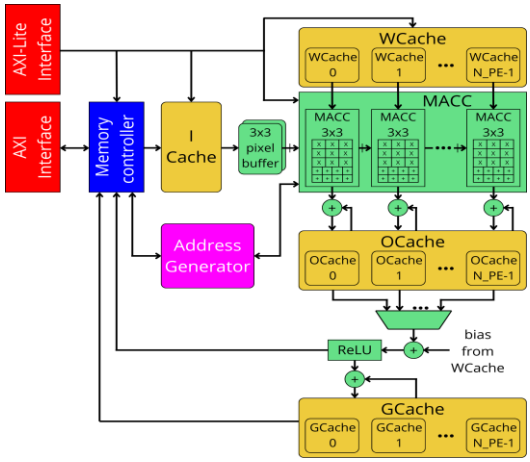
2.4.4 ZynqNet

ZynqNet is an inference engine for CNNs developed by David Gschwend (GSCHWEND, 2016) and implemented on the AMD/Xilinx Zynq-7000 APSoC for AI algorithm computation. It is described in C++ language and is synthesized into hardware description language (HDL) using AMD Xilinx Vivado High Level Sythesis (HLS) (XILINX, 2018a). ZynqNet source code in C++ is parameterizable and it is usually configured to an specific CNN model. However, once synthesized by HLS and implemented in hardware, the ZynqNet can still execute other CNN models if the model complies with the hardware

configuration. It is possible because the CNN weights are dynamically loaded from memory by ZynqNet, instead of synthesized as constants. It was originally developed to process data in 32-bit floating point, but it can be modified to process data in fixed-point and integer numeric representation. In the APSoC, the hardcore Arm Cortex-A9 processor is responsible for run-time configuration of the ZynqNet, which includes the number of input and output channels for the CNN layer, convolution stride and kernel size, and memory addresses for weights of the CNN layer, input image or feature map and output feature map. Once configured to a CNN layer, ZynqNet runs independently of the CPU reading the weight parameters and inputs from external memory. The configuration and execution steps are repeated for each CNN layer, reusing the same ZynqNet hardware.

The ZynqNet internal schematic is depicted in Figure 2.19: the Processing Elements (PEs) (green) contains the multiply-accumulate (MACC) operations of pixels by weights, the sum of bias and the application of activation function (ReLU); the Memory Controller (blue), implements data exchange with the external memory; the AMBA AXI interfaces (red), includes an AXI Initiator interface to the external memory and an AXI-Lite Target used by the CPU to configure and control the ZynqNet accelerator; the Caches (yellow), which include the input cache (ICache) storing pixels and feature maps, the weight cache (WCache) split per PE and each containing the weights for an specific output channel, the output cache (OCache) storing PE's resulting feature map, and the global cache (GCache), which accumulates the global pooling (GP) layer results; and the Address Generator (magenta), which computes memory addresses for operations in both the Caches and the external memory. These blocks can be parameterized in C++ to improve performance, for instance, by varying the number of PEs or using Vivado HLS pragma directives.

Figure 2.19 – ZynqNet accelerator internal components.



Source: The author.

2.5 High-Level Synthesis

High-Level Synthesis (HLS) plays an increasingly important role in FPGA design by enabling hardware specification at higher levels of abstraction, typically using C, C++ or SystemC languages. By raising the design abstraction, HLS shifts the focus from low-level hardware implementation details to algorithmic development, which reduces design complexity, shortens development time, and allows functional verification at the algorithmic level before hardware generation. Over the past years, several HLS tools have been developed, each incorporating distinct optimization strategies, supported features, and target architectures to address diverse application requirements. Among the most widely used solutions are Xilinx Vivado/Vitis HLS (XILINX, 2018a; XILINX, 2023a), SmartHLS from Microchip (MICROCHIP, 2023), and Intel HLS Compiler (INTEL, 2023). This thesis focuses on the use of the Xilinx Vivado HLS tool (v. 18.3) to work with the ZynqNet AI accelerator C++ source-code.

The Xilinx Vivado High-Level Synthesis (HLS) tool translates C++ specifications into register transfer level (RTL) designs, which can then be synthesized into Field Programmable Gate Arrays (FPGAs). FPGAs offer a highly parallel architecture that delivers significant benefits in power, cost and performance.

2.5.1 HLS work-flow

Scheduling

- Identify the dependencies among operations in the software code.
- Try to establish a parallel schedule that honors these dependencies.
- Typical scheduling methods include As-Soon-As-Possible (ASAP), As-Late-As-Possible (ALAP), and System-of-Difference Constraints (SDC).

Binding

- Decides which hardware resource will execute each scheduled operation.
- Uses information about the target device to achieve the best implementation.

Control logic extraction

- Extracts the control logic necessary for the FSM.
- Creates a finite state machine (FSM) to sequence operations in the RTL design.

2.5.2 Example cases

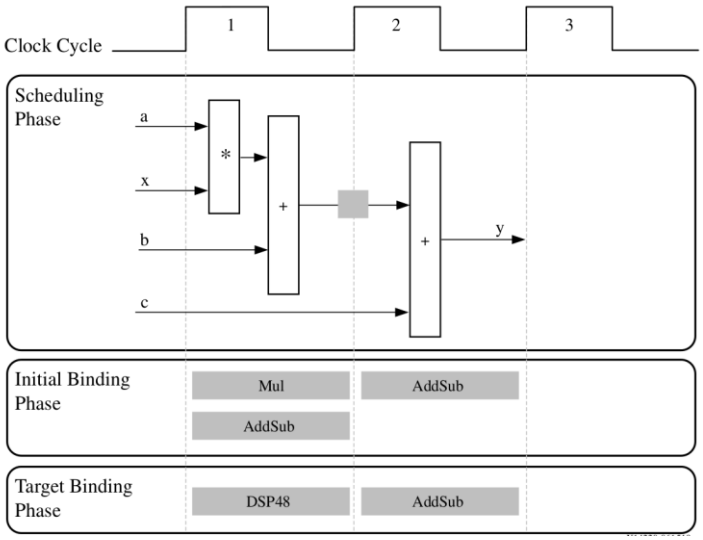
During the scheduling phase shown in Figure 2.20, High-Level Synthesis allocates the following operations to specific clock cycles:

- First clock cycle: Executes multiplication and the initial addition.
- Second clock cycle: Performs the second addition and generates the output.

Figure 2.20 – Example of scheduling and binding process.

```
int foo(char x, char a, char b, char c) {
 char y;
 y = x*a+b+c;
 return y;
}
```

(a) Code example



(b) Scheduling and binding phases.

Source: (XILINX, 2018a)

In the previous diagram, the square between the first and the second clock cycle shows when an internal register stores a variable. The first cycle reads data from ports a, x and b,

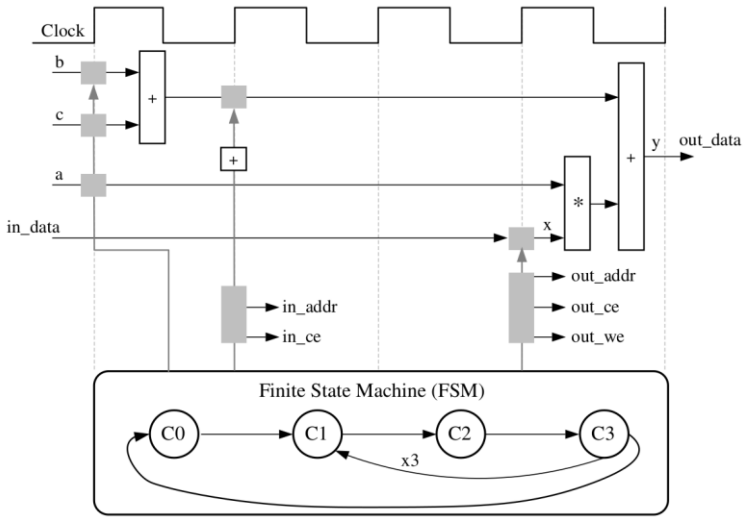
whereas the second cycle reads from the stored value of the register and port c, and produces output y.

Figure 2.21 shows a code example with its control logic extraction and input/output ports implementation. The FSM starts in state C0, progresses through states C1, C2 and C3, and repeats C1 to C3 three times before returning to C0. The addition of b and c is done once in state C0 and it is reused in state C3. During C1, the FSM generates the address to read the data in[i] from the array/block RAM and the adder is incremented to count the number of iterations along C1, C2 and C3. After that, in C2 the in[i] value is stored as variable x. Finally, the output y is generated in C3. This cycle continues until all outputs are obtained, then returns to state C0 to start again with new values of b and c.

Figure 2.21 – Example of control logic extraction.

```
void foo(int in[3], char a, char b, char c, int out[3]) {
  int x,y;
  for(int i = 0; i < 3; i++) {
    x = in[i];
    y = a*x + b + c;
    out[i] = y;
  }
}
```

(a) Code example



(b) Control Logic Extraction with I/O ports implementation.

Source: (XILINX, 2018a)

2.5.3 HLS Design Flow

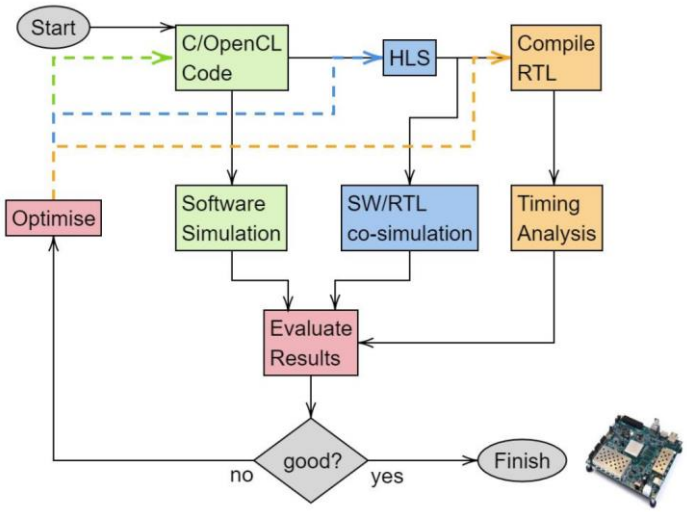
The Xilinx Vivado HLS tool translates a C function into an IP core that can be incorporated into a hardware system. It seamlessly integrates with Xilinx's other design tools

and provides robust language support and features to optimize the implementation of the C algorithm.

The following are the steps in the design flow (Figure 2.22):

1) Compile, run, and debug the C code.
2) Synthesize the C algorithm into an RTL design using optional custom optimization directives.
3) Generation of detailed reports and assess the design.
4) Verify the RTL design through an automated verification process.
5) Package the RTL design into different IP formats.

Figure 2.22 – Typical design flow of HLS



Source: (PERINA et al., 2021).

2.5.4 Levels of Synthesis

IP-Level HLS

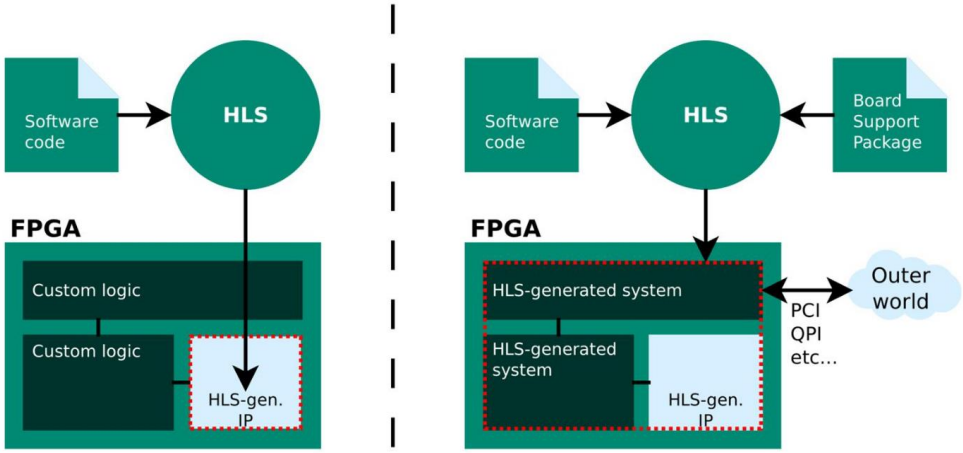
IP-level HLS focus on creating standalone hardware modules or IP cores. These modules are designed to execute specific tasks and are built to be modular and reusable in various projects (Figure 2.23).

System-Level HLS

System-level HLS is a design approach where synthesis is used to develop and optimize entire systems or subsystems. This involves combining various hardware components, such as

multiple IP cores, memory, and communication interfaces, and managing how they interact with each other. It focuses on designing and integrating complex systems (Figure 2.23).

Figure 2.23 – IP-level and System-level HLS graphic differences.



a) IP-level HLS b) System-level HLS.

Source: (PERINA et al., 2021).

2.5.5 Optimization directives

In digital hardware design, particularly when dealing with FPGAs, efficiency and performance are critical factors. A design is more efficient than others if it uses less resources while still achieving an acceptable throughput. As applications become more sophisticated, the importance of optimizing resource usage and enhancing processing speed grows significantly.

To improve the performance and to have an efficient usage of resources, special commands known as pragmas directives are inserted into the C source code to guide the synthesis and optimization process. Alternatively, a TCL script that contains all applicable directives can be created as an additional file. In the following lines, the most common pragmas directives are detailed and as an example, parts of the C code of the Zynqnet hardware accelerator are shown where these are used (Figure 2.24).

A) #pragma HLS PIPELINE

It is used to insert the pipeline technique in a loop or function, allowing the execution of multiple iterations in parallel, instead of executing an entire operation before moving on to the next. This results in an enhancement in performance, reduction in latency and an increment in resource usage.

B) #pragma HLS UNROLL

This pragma expands the loop iterations into parallel operations, executing it concurrently. The HLS tool generates multiple copies of the loop body, each computing a different iteration.

C) #pragma HLS ARRAY_PARTITION

With this directive, data access speeds are improved by splitting the array into smaller segments. This allows different parts of the array to be accessed at the same time, reducing delays.

D) #pragma HLS INLINE

It can be utilized to force inline expansion of a function, eliminating function call overhead and allowing better optimization.

E) #pragma HLS DATAFLOW

It allows concurrent execution of multiple functions and loops, creating an efficient data flow.

F) #pragma HLS RESOURCE

It controls the type and amount of hardware resources used, such as LUTs, FFs, DSPs, and BRAMs.

G) #pragma HLS INTERFACE

It defines the interface protocol of a function port, for instance, AXI4, AXI4-Lite, ap_ctrl_hs, etc.

H) #pragma HLS DEPENDENCE

It removes or reduces dependencies on loops to allow better parallelization and optimization.

Figure 2.24 – Pragmas directives applied in the processAllCHout function of the Zynqnet accelerator.

```
void ProcessingElement::processAllCHout(const channel_t ch_out,
                                        const channel_t ci,
                                        const weightaddr_t ci_offset,
                                        const image_math_t pixels[9]) {
#pragma HLS INLINE off
  LOG("PE: processAllCHout\n");
  LOG_LEVEL_INCR;
L_CH_OUT:
  for (channel_t co = 0; co < ch_out; co++) {
#pragma HLS LOOP_TRIPCOUNT min = 16 max = 1024 avg = 258
#pragma HLS unroll factor = N_PE
#pragma HLS PIPELINE II = 1

    macc2d_math_t result;
    weight_math_t weights_local[9];
#pragma HLS ARRAY_PARTITION variable = weights_local complete dim = 0

    LOG(" - process output channel %d\n", (int)co);
    LOG_LEVEL_INCR;

    // fetch weights
    WeightsCache::getNineWeights(co, ci_offset, weights_local);

    // multiply-accumulate
    macc2d(pixels, weights_local, result);

    // save result to Output Buffer
    if (ci == 0) {
      OutputCache::setChannel(co, result);
    } else {
      OutputCache::accumulateChannel(co, result);
    }
    LOG_LEVEL_DECR;
  };
  LOG_LEVEL_DECR;
}
```

Source: The author

In the Figure 2.24, INLINE off prevents the inlining during synthesis, LOOP_TRIPCOUNT specifies that the loop is expected to have a minimum of 16 iterations, a maximum of 1024 and an average of 258, UNROLL with a factor of N_PE allows multiple iteration of the loop to be executed in parallel, PIPELINE indicates that a new iteration can start every clock cycle (II: Initiation Interval = 1), and ARRAY_PARTITION allows a complete partition of the weights array, leading to a simultaneous access.

2.6 Radiation Effects on APSoCs

Although modern APSoCs built with cutting-edge technologies offer high system performance and programmable resources with low power consumption, their operation in radiation-prone environments remains mainly vulnerable to Single-Event Effects (SEEs) that may compromise system reliability. Among soft (non-destructive) SEEs, the most relevant

effects in APSoCs are Single-Event Upsets (SEUs), Single-Event Transients (SETs), and Single-Event Functional Interrupts (SEFIs) (TAMBARA, 2017).

A SEU is a transient fault defined as a radiation-induced bit flip in memory elements, such as configuration SRAMs, BRAMs, or DFFs, caused by an ionizing particle depositing sufficient energy to exceed a critical charge threshold.

A SET is a radiation-induced current or voltage spike generated in a combinational circuit by a charged particle. If the transient pulse is sufficiently wide, it may propagate through the logic and be captured by a sequential element, appearing as an error. Otherwise, the transient may be masked by logical, electrical, or latching-window masking mechanisms.

A SEFI refers to a severe disruption of a device's functional behavior caused by a radiation-induced event, extending beyond simple user data corruption. SEFIs typically manifest as abnormal operation, system hangs, or loss of functionality, and often require a reset or partial or full reconfiguration to restore normal operation.

In the processing system (PS), SEUs may affect memory and state-holding elements such as registers, caches, and on-chip memory. The execution of an application can inherently mask some of these faults, depending on the Architectural Vulnerability Factor (AVF) (MUKHERJEE et al., 2003). Error Detection and Correction (EDAC) codes are commonly employed to mitigate memory-related SEUs, although their effectiveness depends on the memory structure and protection scheme (TAMBARA et al., 2016). SETs may occur in combinational logic blocks, including Arithmetic Logic Units (ALUs) and pipeline logic. SEFIs are typically associated with disturbances in control or system-related resources, potentially leading to loss of normal processor operation.

In the programmable logic (PL), the situation is more critical due to the high susceptibility of the SRAM-based FPGA CRAM to SEUs, as it defines the functionality of configurable resources such as LUTs, DFFs, routing, BRAMs, and DSP blocks. A bit flip in CRAM can alter the structure and behavior of the digital circuit, leading to a functional error that persists until the bitstream is repaired via configuration scrubbing (readback, verification, and reload). SEUs can manifest as Single Bit Upsets (SBUs), where a single SRAM cell flips its state, or as Multiple Bit Upsets (MBUs), where two or more adjacent cells are corrupted by a single particle strike, which may affect one or multiple configuration frames. These frames are the smallest configuration data blocks in an FPGA, where each one defines a specific portion of the device's functionality, and in the Zynq-7000 family consists of 101 words of 32 bits. While SBUs may be easily masked by voting and error correcting codes, MBUs pose a greater threat by corrupting redundant or spatially adjacent critical resources simultaneously,

potentially bypassing basic error masking or correction. Beyond configuration memory, SEUs affecting user data elements (e.g., flip-flops and BRAMs) are transient and may be overwritten during normal operation. However, if left uncorrected and those corrupted values are read or passed, these faults can propagate through the system. In AI accelerators, such transient errors may affect stored weights, feature maps, or intermediate data during CNN processing, potentially leading to incorrect results. SETs may occur in combinational logic within the PL and, if captured by DFFs, may manifest as errors in user registers or BRAMs. SEFIs are typically associated with disturbances in configuration or control-related circuitry potentially requiring partial or full reconfiguration to restore normal operation.

2.7 Related works

Thanks to the flexibility, efficiency, and low power consumption offered by SRAM-based FPGAs, along with the continuous progress of deep learning in AI applications, many researchers have opted to implement CNN models on these platforms. Several studies have explored diverse FPGA-based accelerator architectures, focusing on improving performance, and reducing area, and energy consumption for different CNN workloads.

Cho et al. (2020) proposed a HLS accelerator for the LeNet-5 CNN, where memory usage was reduced by 40% and latency by 90% through different optimization techniques. The design employed fixed-point representations (Q6.14 for accumulation, Q2.14 for inter-layer transfers, and 11-bit precision for weights and inputs) and incorporated optimization directives such as loop unrolling and pipelining to enhance performance.

Kim et al. (2021) implemented a hardware accelerator for the MobileNetV2 architecture on an XC7Z045 SoC at 100 MHz. Using Imperas for architectural exploration and Vivado HLS for synthesis, the design incorporated optimizations such as reducing memory operations, clustering multiplications and additions, applying caching to maximize BRAM usage, adopting fixed-point formats, and employing HLS directives. The FPGA implementation achieved a 5.66× speedup over a single-core ARM Cortex-A72 at 1.5 GHz on a Raspberry Pi 4.

Ely et al. (2024) trained two CNN models, one of two and the other of four layers, with the SAT-6 dataset and deployed them on the PYNQ-Z1 Zynq-7000 APSoC using the FINN accelerator. The study showed that accuracy was influenced by quantization (2, 4, and 8 bits) and architecture choice, while the folding level did not affect results as it was automatically set by the expected frame rate. The findings emphasized the flexibility of the accelerator to balance area, performance, and power.

Several works have investigated the resilience of FPGA-based AI accelerators against radiation-induced faults through both emulated fault injection and radiation testing, employing reliability metrics for comparison.

Benevenuti et al. (2025) analyzed the impact of different neural network topologies, quantization levels, and architectural parallelism of FINN-based AI accelerator designs on performance, hardware resource utilization, and reliability under emulated and heavy-ion-induced faults. Results showed an improvement of 2.7x in the mean fluence to failure by selecting an adequate CNN topology and modifying the level of quantization and parallelism.

Souvatzoglou et al. (2024a) implemented a four-layer DNN trained on MNIST using the FINN framework on the PYNQ SRAM-based FPGA. Fault injection tests with Single Bit Upsets assessed SEU vulnerability across three folding-level designs, and reliability was evaluated using AVF, MTBF, and MEBF metrics. Results showed that medium folding optimized MTBF, while minimal folding, which used more area, achieved the highest MEBF.

Souvatzoglou et al. (2024b) performed a detailed neutron-irradiation-based reliability analysis of FPGA-based quantized neural networks generated with FINN. The work evaluated the influence of quantization, parallelism, and weight storage memory on reliability metrics, highlighting trade-offs between area, performance, and reliability.

Benevenuti et al. (2022b) conducted a detailed reliability analysis of an FGPU on an SRAM-based FPGA for aerial image classification using a two-layer CNN trained on the SAT-6 dataset. Both fixed-point and floating-point operations were evaluated, and fault injection tests for Single and Multiple Bit Upsets were injected on the configuration memory, considering neutron energy profiles from radiation facilities. The study generated Reliability vs. Accumulated Faults curves from emulated fault injection campaigns and Reliability vs. Neutron Fluence plots from irradiation experiments, providing insights into the impact of soft errors.

Related works addressing fault tolerance in FPGA-based AI accelerators include (LIBANO et al., 2019), where the authors first performed an extensive fault injection on two NN models to identify the layers most likely to cause failures. Based on this vulnerability characterization, they proposed a selective TMR strategy applied only to the most critical layer, and compared it against coarse grained TMR in terms of area and reliability. A neutron-based experiment validated their results achieving up to 76% and 50% reductions in critical error probability across the two networks.

Bertoa et al. (2023) proposed a fault-tolerant NN accelerator generated using the FINN framework, in which full TMR was applied to the output channels of both convolutional and fully connected layers, while selective TMR was applied to the most critical output channels of

the convolutional layers. All experiments were conducted using fault injection campaigns. The selection and number of channels to triplicate in the selective TMR implementation were based on results previously obtained by Gambardella et al. (2019), which showed that convolutional layers are more susceptible to accuracy degradation than fully connected layers and identified the required number of channels to be triplicated to achieve a minimum level of accuracy.

Wang et al. (2021) evaluated three versions of ZynqNet: a standard one, a quantized version, and a TMR implementation under fault injection. They evaluated both the standard and quantized versions under neutron irradiation.

Benevenuti et al. (2021b) implemented the ZynqNet inference engine on an SRAM-based FPGA for traffic sign recognition, using both floating-point and fixed-point (Q15:16, Q9:10) arithmetic. System resilience was evaluated via Single Bit Upset (SBU) fault injection and Heavy Ion irradiation. Fault-tolerant techniques included Hamming Code-based ECC methods, redundancy of signal and 2 MSBs, as well as CRAM scrubbing to correct configuration memory bit flips.

These studies provide valuable insights into the fault tolerance of FPGA-based AI accelerators and guide the development of resilient systems for use in aerospace electronics, autonomous vehicles, and other safety-critical applications operating in both terrestrial and high-radiation environments. Table 2.1 summarizes these works, comparing network topologies, data representations, hardware strategies, fault-tolerance techniques, and key contributions, and highlights how this thesis extends prior research by combining multiple fault-tolerance methods with performance optimizations for SRAM-based FPGAs.

Table 2.1 – Comparison of the state-of-the-art with this work.

AI Inference Engine	NN Topology	Data type representation	HW strategy / parallelism	FT Strategy	Contributions
Cho et al. (2020)	LeNet-5 CNN	Fixed-point	Loop unrolling, pipelining	-	Latency and memory usage reduction in the HLS accelerator.
Kim et al. (2021)	MobileNetV2	Fixed-point	Loop unrolling, caching techniques, cluster MACC operations	-	Performance enhancement by applying HLS directives, implementing on-chip cache memories to minimize memory operations, and clustering MACC units.
FINN: Ely et al. (2024)	2-CNN + 1-FC 4-CNN + 1-FC	Integer: 2-, 4-, 8- bits	Folding level	-	Analysis of the impact of different quantization levels, architectural parallelism, and network topologies on area, performance and power.
FINN: Benevenuti et al. (2025)	2-CNN + 1-FC 4-CNN + 1-FC	Integer: 4-, 8- bits	BRAM / LUT storage, folding level	-	Evaluate the impact of different quantization levels, network topologies, and architectural parallelism on area, performance and reliability in the presence of soft errors.
FINN: Souvatzoglou et al. (2024a)	5-FC	Binary: 1-bit	-	-	A detailed SEU vulnerability analysis of a FPGA-based Binarized NN accelerator using emulated fault injection campaigns and simulation methods.
FINN: Souvatzoglou et al. (2024b)	3-CNN + 2-FC	Binary: 1-bit, Integer: 2-bit	Folding level, BRAM / LUTRAM storage	-	Reliability analysis of quantized and binarized FPGA-based neural networks under neutron irradiation, exploring design trade-offs using FINN.
FGPU: Benevenuti et al. (2022b)	2-CNN + GP	Fixed-point	-	-	A detailed reliability analysis of an FPGA-based FGPU under neutron-based SBU and MBU injections in the CRAM.
Libano et al. (2019)	2-FCC 2-CNN + 2-FC	-	-	Selective TMR, CGTMR	Identified the most vulnerable layers via fault injection. Applied selective TMR to these layers, reducing critical errors compared to coarse-grained TMR.
FINN: Bertoa et al. (2023)	5-CNN + 3-FC	Binary: 1-bit, Integer: 2-bits	Folding level	Selective TMR	Evaluation of a fault-injection-guided selective TMR strategy for FINN-based neural networks, where only critical convolutional output channels are triplicated to maintain a minimum level of accuracy while avoiding excessive overhead.
ZynqNet: Wang et al. (2021)	Adapted SqueezeNet	Dynamic fixed-point: 8-bit	Loop unrolling, pipelining, memory partition	Selective TMR	Comparative reliability evaluation of standard, quantized, and TMR-based ZynqNet implementations under fault injection and neutron irradiation, showing that quantization significantly reduces soft error rate.
ZynqNet: Benevenuti et al. (2021b)	7-CNN + GP	Fixed-point	Loop unrolling, pipelining, memory partition	CRAM scrubbing, EDAC, signal redundancy	Reliability evaluation of floating- and fixed-point ZynqNet on SRAM-based FPGAs with ECC, MSB redundancy, and CRAM scrubbing under SBU fault injection and heavy-ion irradiation.
ZynqNet: This work	4-CNN + GP	Fixed-point, Integer: 8-bits	Loop unrolling, pipelining, memory partition	CRAM scrubbing, EDAC, CGTMR, DTMR, MTMR	Design and testing methodology for fault-tolerant AI accelerators on SRAM-based APSoCs, combining performance optimizations with reliability enhancements.

Source: The author

3 METHODOLOGY

This chapter describes the methodology followed in this work, presenting a structured approach that encompasses the design tools employed, the case-study AI inference engine running a neural network application on a commercial APSoC, the design evaluation metrics, and the experimental procedures used to assess the system's reliability under radiation effects.

3.1 Design Tools

The Xilinx Vivado HLS tool (v. 18.3) was used to modify the ZynqNet C++ source code, apply directive pragmas for hardware optimization, synthesize and generate the RTL code, and export it as an IP block. The Xilinx Vivado Design Suite tool (v. 18.3) was then used to integrate the ZynqNet IP with other Xilinx IPs in the PL, enabling data exchange with the PS, and to generate the bitstream required to program the FPGA. Finally, the Xilinx Software Development Kit (SDK) (v. 18.3) was employed to develop the embedded C application running on the ARM Cortex-A9 processor.

3.2 Case-study

3.2.1 APSoC

In this thesis, the commercial-off-the-shelf Zynq-7000 ARM/FPGA APSoC, described in Section 2.3, was used to implement the case-study AI accelerator. It was designed by Xilinx at 28 nm technology (part number xc7z020clg484-1), fabricated in the Taiwan Semiconductor Manufacturing Company (TSMC) and embedded in the Zedboard Development Board.

This platform was selected for implementation because it integrates processors in the PS to manage and configure the accelerator, and because it was available in the laboratory with a decapsulated silicon chip, enabling the generation of radiation-induced faults using heavy-ion irradiation at the available radiation test facility. The proposed design and test methodology in this work is generic and can be applied to other commercial APSoCs to compare results.

3.2.2 AI Inference Engine

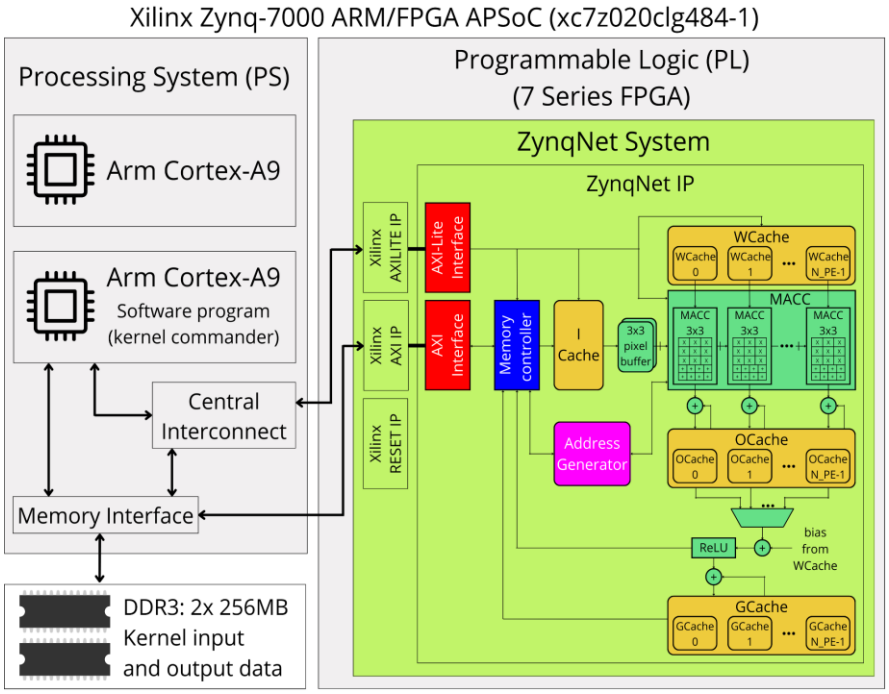
The ZynqNet AI inference engine, described in Section 2.4.4, was used to run the case-study CNN model.

ZynqNet was selected due to its efficient reuse of a processing structure across all CNN layers, which results in a smaller hardware footprint compared to layer-specialized accelerators such as FINN. This reduced area makes the application of TMR-based fault-tolerance techniques feasible on resource-constrained FPGA devices. In addition, unlike conventional systolic array-based accelerators, ZynqNet integrates ReLU activation and global pooling directly into the processing pipeline, avoiding additional hardware and data movement.

In the PL of the APSoC, the Zynqnet IP is integrated with the other IPs such as the Xilinx AXI IP, which enables communication with the external DDR memory (ZynqNet's data bus); the Xilinx AXILITE IP, which facilitates communication with the ARM processor (ZynqNet's Configuration Bus); and the Xilinx Reset IP, which manages the reset operation of the ZynqNet System, which includes all the integrated IP blocks (Figure 3.1).

The external DDR memory was used to store ZynqNet input data, including images and CNN parameters (weights and biases), as well as intermediate outputs, specifically the feature maps generated after each processed convolutional layer. Communication with external memory can become a performance bottleneck.

Figure 3.1 – Integration of ZynqNet and Xilinx IPs in the Zynq-7000 APSoC.



Source: The author.

3.2.2.1 Source-code files

A brief description of the C++ source files used in the Xilinx Vivado HLS tool is provided below, including those used to generate the accelerator and those running on the ARM processor in bare-metal mode.

A) ZynqNet accelerator

-fpga_top.(cpp/hpp): Serves as the main function to generate the ZynqNet hardware accelerator. It calls all necessary functions to execute each convolution layer. This file also defines the AXI interface protocols and transmission characteristics using pragma directives.

-memory_controller.(cpp/hpp): Manages the transmission and reception of data between the external DDR memory and the internal caches of the accelerator, including image, weight, output, and global pooling caches.

-weights_cache.(cpp/hpp), image_cache.(cpp/hpp), output_cache.(cpp/hpp), gpool_cache.(cpp/hpp): Responsible for temporarily storing data to accelerate CNN processing.

-processing_element.(cpp/hpp): Executes multiplication and accumulation operations (weights and pixels) for CNN computation. It reads input data from the image and weight caches and stores intermediate results in the output memories. For the final layer with global sum pooling, the result is stored in the global pooling cache.

-datel_common.cpp, datel.hpp, datel_config.hpp: Define the data types and operations used during CNN processing.

-network.(cpp/hpp), net_config.(cpp/hpp): Define the cache's initial memory addresses for the current layer and the type of CNN architecture.

-weights.(cpp/hpp), eval_images_8_8_1.cpp, eval_images.hpp: These files contain the CNN weights and image pixels.

B) Processor

-cpu_top.(cpp/hpp): Manages the execution of the accelerator, sends configurations through each CNN layer, and reads the results from a DDR memory space.

3.2.2.2 Workflow of ZynqNet accelerator

The pseudocode for the CPU is detailed in Figure 3.2 (a) (Algorithm 1) and outlines the steps to be performed during the processing of each image and before the execution of each CNN layer. External communication variables allow the processor to communicate with the

ZynqNet hardware accelerator and to work with functions defined in the board support package (drivers) of the ZynqNet IP block. The processor can read signals such as IsDone, IsIdle, IsReady to know about the current state of the ZynqNet. Figure 3.2 (b) (Algorithm 2) illustrates the complete procedure the ZynqNet performs in each layer.

Figure 3.2 – Pseudocodes of processor and ZynqNet accelerator.

Algorithm 1 ARM A9 CPU pseudocode

```
1: Processor initialization (optional caches and UART interface)
2: Zynqnet initialization: definition of external communication
   variables and DDR memory addresses (weights, pixels and output
   results).
3: Defintion of each CNN layer's features: input/output channel
   (size, number), kernel type, pad, stride, ReLu, global pooling.
4: while 1 do
5:     -Reset the Programmable Logic
6:     for each image do
7:         -Assign the image to the DDR.
8:         for each layer do
9:             -Setup layer configuration.
10:            -Process the layer: start the accelerator and wait
11:            for the activation of signals.
12:            -Output channel results are stored in the DDR.
13:        end for
14:        -Global pooling results are stored in the DDR.
15:    end for
16: end while
```

(a) Processor main program.

Algorithm 2 ZynqNet Layer Processing pseudocode

```
1: Receive layer configuration and DDR memory addresses.
2: Set-up layer configuration for image and weight caches, mem-
   ory controller, and processing element modules.
3: Load weights from DDR memory into the weight cache.
4: Preload one row and one pixel values to the ICache.
5: for each Y (height) do
6:     for each X (width) do
7:         -Preload one pixel (all channels) to the ICache.
8:         -Verify stride value.
9:         for each input channel (ci) do
10:            -Preload image pixel buffer (fetch 9 pixels in
11:            the ICache).
12:            for each output channel (co) do
13:                -Fetch 9 weights in the WCache.
14:                -Multiply and accumulate: pixel x weight.
15:                -Store value in the OCache.
16:            end for
17:        end for
18:        for each output channel (co) do
19:            -Addition of bias to the output channels.
20:            -Apply ReLU and store in DDR.
21:            if it is last layer and has global pooling then
22:                -Store in GCache and accumulate.
23:            end if
24:        end for
25:    end for
26: end for
27: Send the global output channel values to DDR.
```

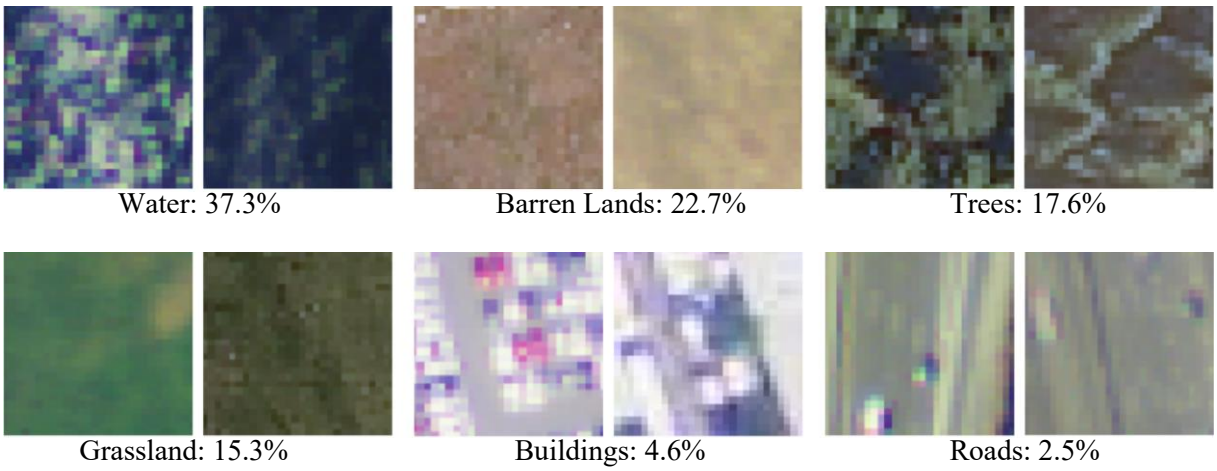
b) ZynqNet main function execution.

Source: The author (MARAÑON et al., 2025).

3.2.3 Application

The SAT-6 dataset (BASU et al., 2015) was used as a case-study for aerial image classification tasks. It contains land-cover images from the United States, covering six distinct classes: water, barren land, trees, grassland, buildings, and roads. The dataset was provided by the United States National Agriculture Imagery Program. In its original form, it consists of 28×28 pixels with 4 channels, including color (RGB) and near-infrared (NIR). In this work the RGB channels were converted to HSV to improve robustness to illumination. To meet ZynqNet's architectural constraints, images were resized from 28x28 to 32x32 pixels using OpenCV's bicubic interpolation, which estimates new pixel values from neighboring pixels to ensure smooth spatial transitions. The training set comprises 324,000 images, and the test set comprises 81,000 images. In both sets, the number of images in each class follows the proportion shown in Figure 3.3.

Figure 3.3 – Samples from the SAT-6 dataset and class proportions for the training and testing sets.



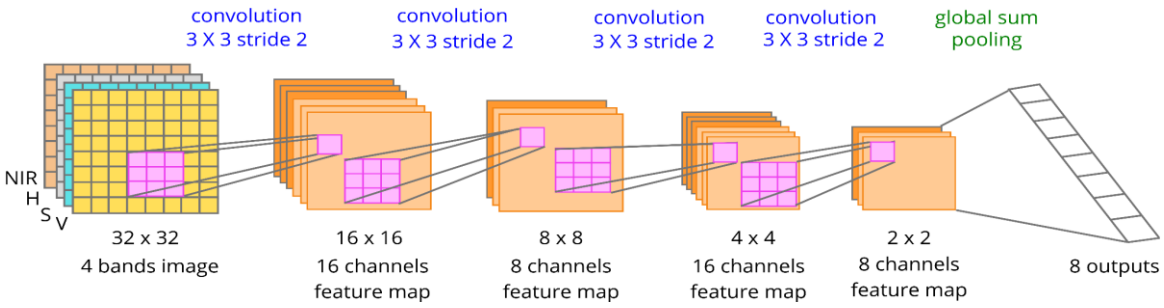
Source: Extracted from Basu et al. (2015).

3.2.4 CNN model

The case-study CNN model was adopted from prior research (BENEVENUTI et al., 2021a), where the authors used genetic algorithms to analyze the architectural design space of suboptimal CNNs. This exploration included adjustments to the number of layers, dimensions, and kernel sizes, all while respecting the hardware constraints defined by ZynqNet and the capacity of the target FPGA device. The network was trained using the deep learning framework Caffe (JIA et al., 2014) and the Ristretto extension (GYSEL et al., 2018) to use limited numerical precision, achieving an approximate accuracy of 98%.

This streamlined model consists of four convolutional layers followed by a global sum pooling layer, without any fully connected layers, which reduces computational complexity and resource usage. It contains a set of 4080 weights with bias coefficients and 2.44 x 10^5 MACC operations (Figure 3.4). This setup provides a representative and practical scenario for real-time, on-device land classification using a compact and customized CNN model.

Figure 3.4 – CNN model architecture.



Source: The author

3.3 Design evaluation metrics

To compare and evaluate the different designs implemented on the APSoC, three main metrics were considered: area, power, and performance.

The area metric refers to the hardware resource utilization on the FPGA, expressed in terms of lookup tables (LUTs), D-type flip-flops (DFFs), block memories (BRAMs), and digital signal processing blocks (DSPs). Additionally, the essential bits metric was analyzed, defining the total CRAM bits required to implement each design and responsible for preserving the intended functionality of the circuit. Any accidental alteration or corruption of these bits can compromise the design, potentially leading to functional errors or system malfunction.

Power consumption is expressed in watts (W) and two components were analyzed: static power, corresponding to the baseline consumption once the device is configured, even without activity, and dynamic power, representing the additional consumption caused by the switching of clocks, datapaths, and logic resources. The static component mainly depends on junction temperature, making thermal conditions critical for accurate estimation, while the dynamic component is influenced by resource utilization, operating frequency, and signal activity. The total power corresponds to the sum of both contributions, and as with the area results, they were obtained using the Xilinx Vivado Design Suite tool.

Performance was evaluated by reading, through the serial interface, the execution time reported by ARM Cortex-A9 processor when running the application on the ZynqNet accelerator. Performance results are expressed in terms of the number of images processed per second.

3.4 Reliability Testing

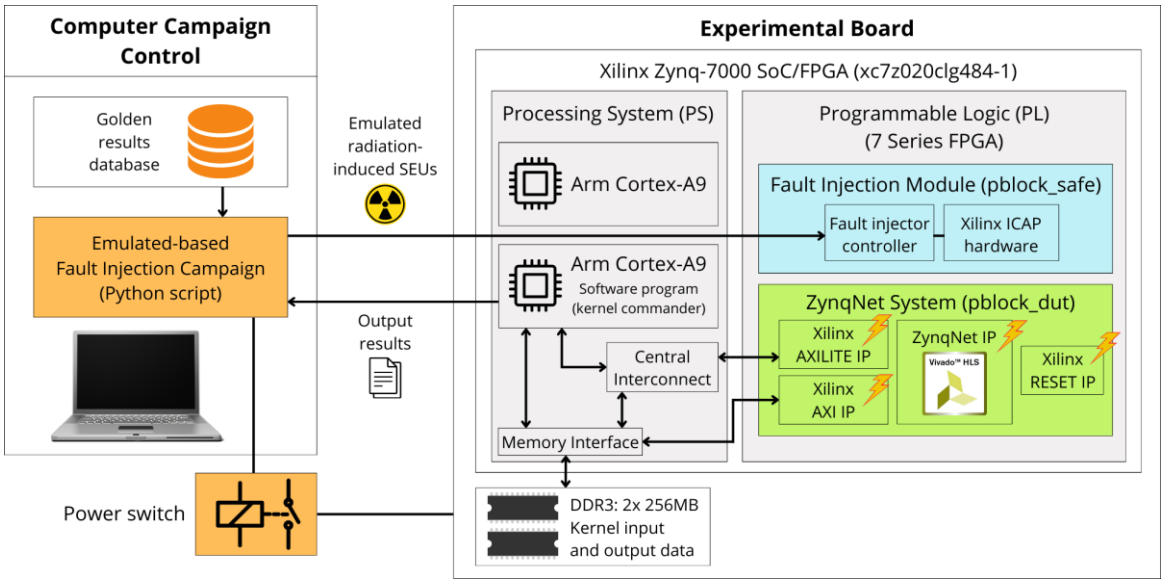
This research work employs two experimental methodologies, emulated fault injection and radiation testing, to evaluate the radiation effects on the ZynqNet design implemented on the SRAM-based FPGA of the Xilinx Zynq-7000 ARM/FPGA platform. Both experimental methods are crucial for testing, validating and optimizing ZynqNet designs that incorporate several fault-tolerance techniques such as Triple Modular Redundancy (TMR), which triplicates hardware modules and uses majority voters to mask faults, CRAM Scrubbing, an ECC technique that continuously scans FPGA configuration frames to detect and correct SBUs in the configuration memory, and Hamming-based EDAC, which implements error detection and correction codes in hardware to correct SBUs in the BRAMs of the ZynqNet’s memory

caches. By combining these two experimental approaches, it is possible to better understand the impact of these fault-tolerance techniques on system reliability under realistic radiation conditions, leading to more robust and fault-tolerant designs.

3.4.1 Emulated Fault Injection

A Fault Injection methodology is employed to emulate radiation-induced SEUs in the CRAM of the FPGA. The experimental setup, illustrated in Figure 3.5, includes two main components: a computer that runs a Python script controlling the fault injection campaign and storing the golden results database, and the experimental board. The Python script on the computer transmits command data through the serial port, specifying the random location of the SEU to be injected into the FPGA configuration memory. The receiver, known as the Fault Injection Module, is implemented in the pblock_safe region of the FPGA floorplan (Figure 3.5), isolated from the Design Under Test (DUT). This module includes the fault injection controller and the Internal Configuration Access Port (ICAP) hardware block, where the latter enables modification of the CRAM content. In the Zynq-7000 FPGA, a configuration frame, consisting of 101 words of 32 bits each, represents the smallest block accessible by the ICAP hardware for read or write operations.

Figure 3.5 – Fault injection setup

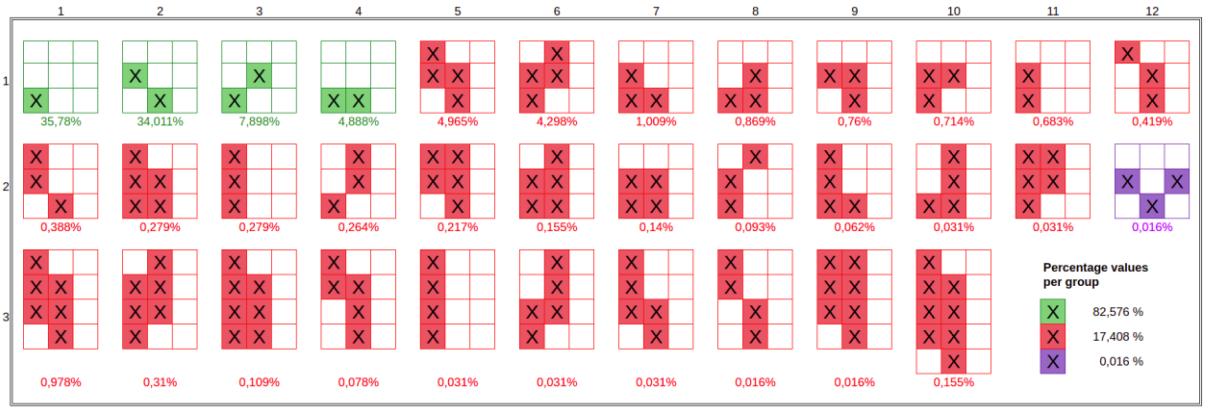


Source: The author.

To ensure realistic fault injection scenarios, the characteristics of the injected faults are derived from experimental radiation data. The type of injected faults is determined through

probabilistic selection from a distribution of bit cluster geometries obtained during a prior static irradiation test with oxygen-16 ions at 40.0 MeV and 6.69 MeV/mg/cm^2 LET, conducted under near-vacuum on the 0° beamline at the SAFIIRA facility (AGUIAR et al., 2020) of the Pelletron 8UD tandem accelerator, at the University of São Paulo (USP), Brazil. Figure 3.6 shows the most relevant SEU bit cluster geometry observed during the static tests, with percentage occurrence. Bit clusters shown in green indicate single-bit flips within a configuration frame correctable through CRAM scrubbing, those in red represent multiple-bit flips in the same frame that cannot be corrected, and the purple indicates single-bit flips not injected due to the lack of geometry cluster functionality.

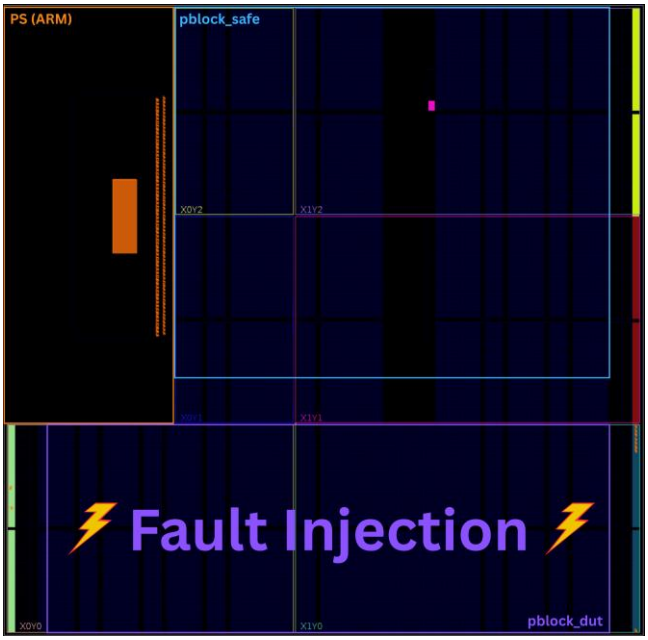
Figure 3.6 – Bit cluster geometry list for fault injection based on SBU and MBU.



Source: The author.

The faults only affect the configuration of Configurable Logic Blocks (CLBs), Digital Signal Processing blocks (DSPs), Block RAMs (BRAMs), and interconnection switch boxes (routing paths); therefore, they do not affect the user data stored in DFFs and BRAMs. They were randomly injected into a specified physical region of the bottom row of the FPGA floorplan (from column 3 to 71), referred to as the Design Under Test (DUT). This region, defined within the pblock_dut (Figure 3.7), comprises 7,705,088 CRAM bits and was selected due to its proximity to the output pins (Bank 13: light green), which were used to transmit the final classification results, as well as the availability of hardware resources to implement all the evaluated designs. In contrast, the remaining area of the FPGA floorplanning was used by the Fault Injection Module (pblock_safe).

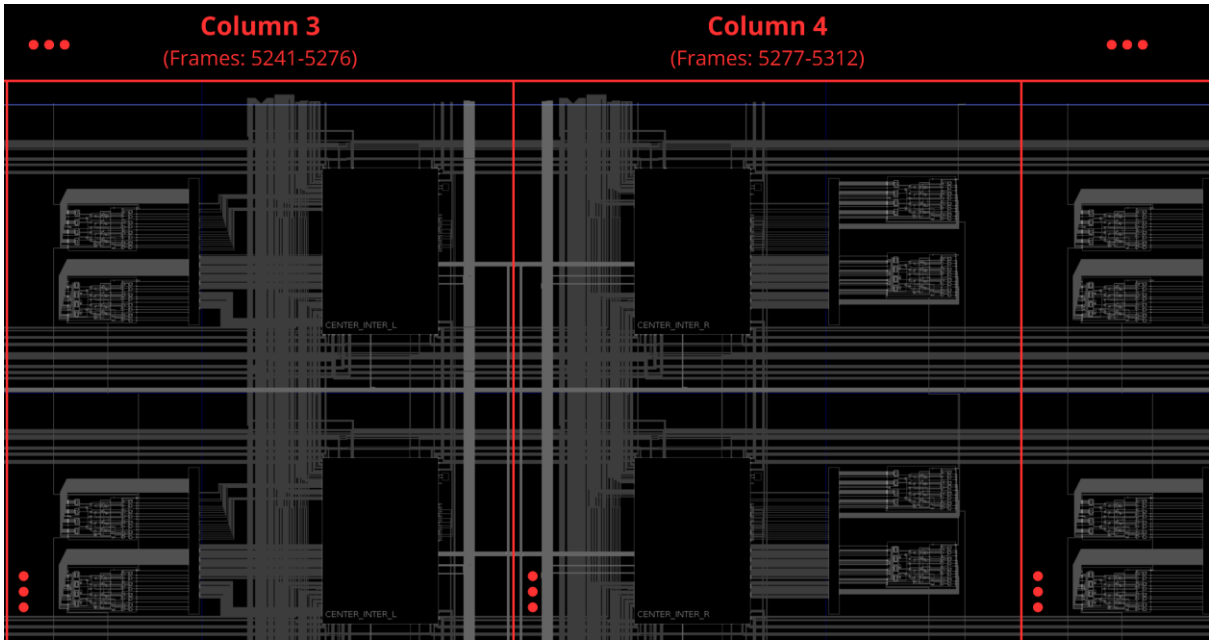
Figure 3.7 – Location of pblock_safe and pblock_dut in the FPGA floorplanning.



Source: The author.

Figure 3.8, shows the first frame ranges of the initial columns belonging to the pblock_dut.

Figure 3.8 – Frame ranges for initial columns of pblock_dut.

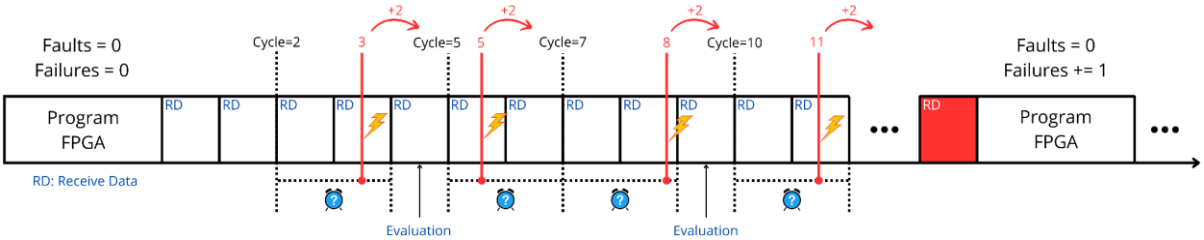


Source: The author.

Each tested design sequentially classifies a set of three different aerial images correctly, requiring a total time of 18ms under fault-free conditions. At most one fault was injected per image set, ensuring that the time between consecutive injections was randomly selected within

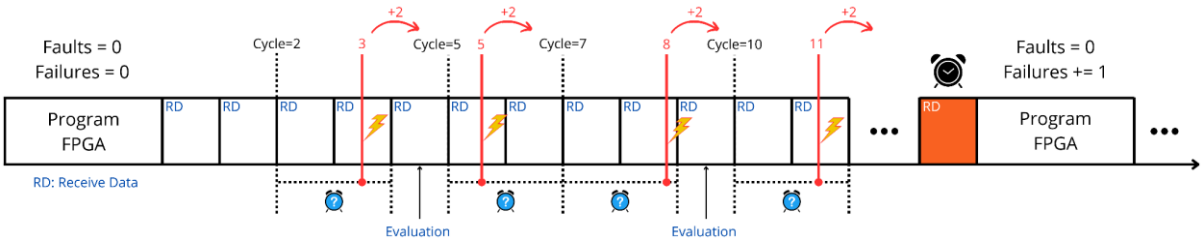
a 18-36 ms interval. Faults are continuously injected until a functional failure occurs, then the type of failure is registered along with the total number of injected faults up to that moment. Once a failure is detected, the CRAM and BRAM content are reset by power cycling and reconfiguring the FPGA to remove any remaining faults. This procedure is repeated until the required number of failure events has been collected. Failures were classified into two types: (1) Inference error, when the classification output is incorrect (Figure 3.9); and (2) Time-out, defined as missing or invalid input via the serial interface within the allowed time frame (case-study: 5x processing time), as described in Figure 3.10.

Figure 3.9 – Inference error failure type.



Source: The author

Figure 3.10 – Timeout failure type.



Source: The author

A total of 15 designs, including the unmitigated and fault-tolerant versions, were tested with 600 events registered for each, resulting in a total testing time of 19 h 40 min.

In fault injection experiments, the reliability of different designs can be compared through reliability curves R(f) and key metrics such as the Mission Faults (MF) at a given reliability threshold and the Mean Faults to Failure (MFTF).

R(f) curves are calculated using Eq. (3.1), where F(f) represents the cumulative distribution function of faults, normalized between 0 and 1.

$$R(f) = 1 - F(f) \tag{3.1}$$

From the calculated reliability curves, one can determine the maximum time the DUT can be exposed to faults while maintaining a desired reliability threshold. For this type of experiment, this metric is expressed as the number of accumulated faults referred to as Mission Faults (MF). A threshold of 90% was adopted to compare MF values among the designs.

The MFTF is estimated as the ratio of the total count of accumulated injected faults (f) to the number of failure occurrences (N_{events}), as described in Eq. (3.2). However, it is important to note that characterizing the reliability of a design solely with this metric is not sufficient because it is not possible to observe the effects of implementing the CRAM Scrubbing technique.

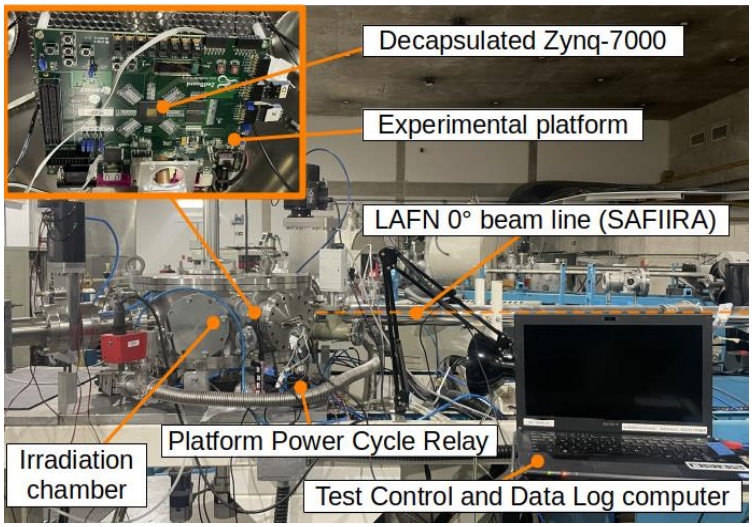
$$MFTF = \frac{f}{N_{events}} \tag{3.2}$$

3.4.2 Radiation Experiment

The radiation experiment was conducted at the SAFIIRA facility (AGUIAR et al., 2020), which was specifically built for the characterization of electronic components under ionizing radiation, and is equipped with beamline quadrupole magnets to facilitate precise beam focusing and defocusing. Additionally, two scattering chambers containing gold foil targets are employed to enable low-flux irradiation with beam energy uniformity exceeding 90%.

The experimental setup included the Zedboard programmable board placed inside the vacuum chamber, a host computer located outside the chamber and running a Python script to control the radiation campaign, and serial communication cables linking both components (Figure 3.11). Irradiation was performed under vacuum conditions using 49 MeV oxygen-16 ions at normal incidence (0°). After traversal through the gold foil targets within the scattering chambers, the effective energy of the beam was estimated to be 46.6 MeV, as determined by simulations conducted using the SRIM/TRIM software package (ZIEGLER et al., 2010). Under these conditions, the beam penetration depth in silicon was estimated to be approximately 34.7 um, with a corresponding linear energy transfer (LET) of 4.9 MeV/mg/cm² at the device surface.

Figure 3.11 – Heavy-Ions Radiation Setup.



Source: The author.

Beam flux measurements were calibrated at least once per day and periodically verified through static tests on the FPGA, by reading back the contents of both the CRAM and the BRAM. During the experiments, the average particle flux incident on the devices under test ranged from 200 to 520 p/cm²/s. For each tested design, the total fluence was calculated by summing the products of the measured flux and the experiment time. To minimize variations in beam intensity within individual tests, the irradiation was performed in small batches.

The radiation tests considered the same number of processed images and the same failure conditions as the emulated fault injection. A total of 7 designs, including the unmitigated and fault-tolerant versions, were tested with approximately 120 events registered for each, resulting in a total testing time of 6 h 57 min.

In radiation experiments, the reliability of different designs can be compared through reliability curves R(Φ) and key metrics such as the Mission Fluence (MΦ) at a given reliability threshold, the mean fluence to failure (MΦTF), and the dynamic cross-section ($\sigma_{dynamic}$).

R(Φ) curves are calculated using Eq. (3.3), where F(Φ) represents the cumulative distribution function of particle fluence (cm^{-2}), normalized between 0 and 1.

$$R(\Phi) = 1 - F(\Phi) \tag{3.3}$$

From the calculated reliability curves, one can estimate the maximum particle fluence (cm^{-2}) the DUT can be exposed to, maintaining a desired reliability threshold. For this type of

experiment, this metric is expressed as a total particle fluence (cm^{-2}) referred to as Mission Fluence (MΦ). A threshold of 90% was adopted to compare MΦ values among the designs.

The MΦTF is estimated as the ratio of the particle fluence ($\Phi_{particles}$) to the number of failure occurrences (N_{events}), as described in Eq. (3.4).

$$M\Phi TF = \frac{\Phi_{particles}}{N_{events}} \tag{3.4}$$

The dynamic cross-section refers to the probability that an energetic particle will generate a functional failure in the implemented design and can be calculated as the inverse of the MΦTF, as described in Eq. (3.5).

$$\sigma_{dynamic} = \frac{N_{events}}{\Phi_{particles}} = \frac{1}{M\Phi TF}\ [cm^2] \tag{3.5}$$

To provide a clear overview of the experimental methodologies employed in this work, Table 3.1 summarizes the main characteristics of the two approaches used to evaluate the reliability of the ZynqNet designs: Emulated Fault Injection and Heavy-Ions Radiation Experiment. The table highlights the objectives, types of soft SEEs occurred, affected hardware resources, type of observed failures, number of design tested, number of events registered per design, total time, and reliability assessment mode. This comparative summary allows the reader to quickly understand the scope and focus of each experimental methodology, complementing the detailed descriptions provided in this section.

Table 3.1 – Comparison of the two experimental methodologies.

Test	Emulated Fault Injection	Heavy-Ions Radiation Experiment
Objective	Emulate faults in the FPGA CRAM and evaluate the design reliability	Evaluate the design reliability under heavy-ion irradiation
Type of soft SEEs occurred	-SEU: PL part -SEFI	-SEU and SET: PL and PS parts -SEFI
Device scope and affected hardware resources	-PL: CRAM cells (configuration memory)	-PL: CRAM cells (configuration memory), DFFs and BRAM (user data) -PS: processor, caches, interconnections and OCM (program and user data)
Type of observed failures	Inference error and Time-out	Inference error and Time-out
Number of designs tested	15	7
Number of events registered per design	600	~120
Total time	19 h 40 min	6 h 57 min
Reliability assessment mode	-Chart: Reliability vs Accumulated injected faults. -Metrics: MF (R≥90%), MFTF.	-Chart: Reliability vs Fluence (particles/cm^2). -Metrics: MΦ (R≥90%), MΦTF, $\sigma_{dynamic}$.

Source: The author

4 DESIGN SPACE EXPLORATION: EFFICIENCY AND PERFORMANCE

This chapter presents a design space exploration of the ZynqNet accelerator to produce more efficient designs with higher performance. The objectives are to investigate the impact of quantization techniques, different types of arithmetic operations, data packing and new communication architectures between the accelerator and the external DDR memory.

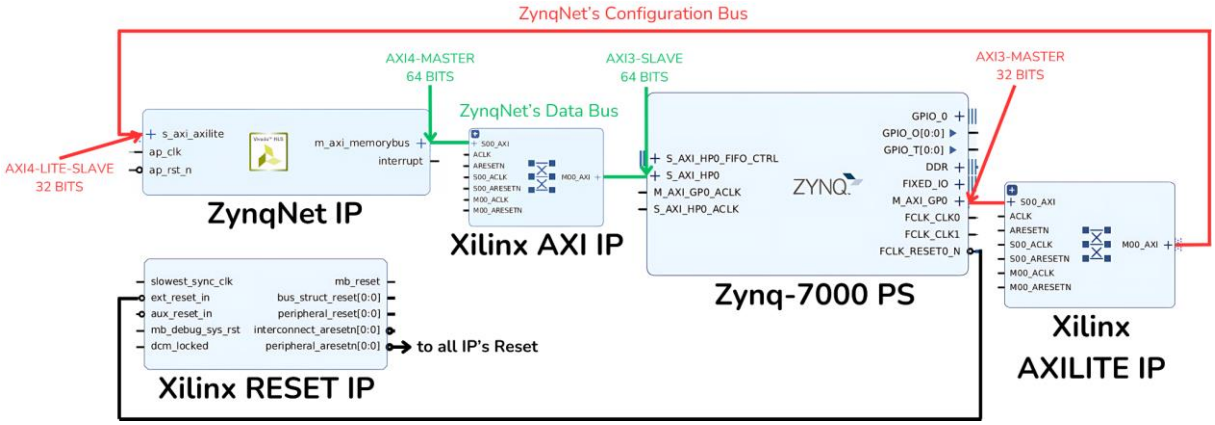
4.1 Application of quantization techniques

Quantization plays a key role in optimizing area and power, aspects that are critical in aerospace applications. To investigate this, three implementations of the case-study CNN model with different numerical representations were analyzed. The floating-point design uses 32-bit precision for image pixels, feature map pixels, weights, bias coefficients and intermediate MACC results; the fixed-point design, uses fractional representation (Qm.n), signal and integer part (Q7.0) for image pixels and feature map pixels, signal and fractional part (Q0.7) for weight and bias coefficients, and signal, integer and fractional part (Q7.8) for intermediate MACC results; and the integer design, uses 8-bit integer representation for image pixels, feature map pixels, weights, and bias coefficients, and 16-bit integer for intermediate MACC results.

The Vivado-HLS tool was used to modify the ZynqNet C++ source code and the Block Design feature within the Vivado Design Suite tool to instantiate all the necessary components to communicate the accelerator with the PS. Figure 4.1 depicts the simplest architecture to interconnect the ZynqNet accelerator with the PS where this latter contains the processor and a memory interface to communicate with an external DDR memory (described in Chapter 2).

Communication for configuring the accelerator for each CNN layer is handled using the AXI4-Lite protocol with a 32-bit data bus. Since the Processing System operates with AXI3, an AXI3 to AXI4-Lite protocol converter must be used (Xilinx AXILITE IP). For read and write operations, the accelerator uses the AXI4 protocol with a 64-bit data bus, which also requires an AXI4 to AXI3 protocol converter to interface with the Processing System (Xilinx AXI IP). The Xilinx RESET IP was used to restart the ZynqNet System and can be configured to be performed by the processor.

Figure 4.1 – Interconnection of the ZynqNet IP with the other Xilinx IPs and the PS.



Source: The author.

Using the architecture described in Figure 4.1, it is analyzed the impact of varying the number of processing elements (PEs) and the type of arithmetic operation. In the first set of results, it is described the area and power consumption after implementing each IP block and the ZynqNet System on the FPGA, as well as the power consumption of the PS elements in the APSoC.

Table 4.1 summarizes the FPGA resource utilization, essential bit count, and dynamic power consumption of the ZynqNet IP block implemented with 1, 2, and 4 PEs using floating-point, 8-bit fixed-point, and 8-bit integer arithmetic. It shows that reducing arithmetic precision significantly decreases FPGA resource utilization, essential bits, and dynamic power consumption for all processing element (PE) configurations.

LUT utilization is reduced by 49.1–71.2% when moving from floating-point to fixed-point arithmetic, and by 53.8–74.2% when using integer arithmetic. The reduction becomes larger as the number of PEs increases, indicating better scalability with lower precision.

For DFFs, fixed-point arithmetic achieves reductions between 52.7% and 67.3%, while integer arithmetic further increases the reduction to 60.1–80.4%, with the largest savings observed in the 4-PE configuration.

BRAM usage decreases consistently by approximately 42.3–46.1% for both fixed-point and integer implementations across configurations with PE=1, 2, and 4, reflecting the reduced storage requirements of lower-precision data representations.

The strongest impact is observed in DSP utilization. Fixed-point arithmetic reduces DSP usage by about 79.2–80.4%, while integer arithmetic achieves reductions between 86.8% and 88.7%, confirming that floating-point operations require more DSP utilization.

The number of essential bits is reduced by 51.9–71.9% with fixed-point arithmetic and by 56.3–75.3% with integer arithmetic, limiting the growth of these bits as the design scales.

Finally, dynamic power consumption is reduced by 62.8–75.6% when using fixed-point arithmetic and by 71.7–87.2% when using integer arithmetic, with the largest reduction observed in the 4-PE integer configuration.

Table 4.1 – Area and dynamic power results of each ZynqNet IP with 1, 2 and 4 processing elements computing in floating-point, 8-bit fixed point and 8-bit integer.

Target	**ZynqNet IP**								
PE	**1**			**2**			**4**		
Type of Arithmetic	**Floating point**	**Fixed point**	**Integer**	**Floating point**	**Fixed point**	**Integer**	**Floating point**	**Fixed point**	**Integer**
LUT	**6214**	**3162** ↓ 49.1%	**2868** ↓ 53.8%	**9679**	**3601** ↓ 62.8%	**3372** ↓ 65.2%	**16263**	**4690** ↓ 71.2%	**4197** ↓ 74.2%
DFF	**8229**	**3891** ↓ 52.7%	**3283** ↓ 60.1%	**12426**	**4851** ↓ 61.0%	**3518** ↓ 71.7%	**20054**	**6555** ↓ 67.3%	**3933** ↓ 80.4%
BRAM	**13**	**7.5** ↓ 42.3%	**7.5** ↓ 42.3%	**23.5**	**13** ↓ 44.7%	**13** ↓ 44.7%	**44.5**	**24** ↓ 46.1%	**24** ↓ 46.1%
DSP	**53**	**11** ↓ 79.2%	**7** ↓ 86.8%	**100**	**20** ↓ 80.0%	**12** ↓ 88.0%	**194**	**38** ↓ 80.4%	**22** ↓ 88.7%
Essential Bits (10^6)	**1.58**	**0.76** ↓ 51.9%	**0.69** ↓ 56.3%	**2.48**	**0.91** ↓ 63.3%	**0.83** ↓ 66.5%	**4.42**	**1.24** ↓ 71.9%	**1.09** ↓ 75.3%
Dynamic Power (W)	**0.145**	**0.054** ↓ 62.8%	**0.041** ↓ 71.7%	**0.224**	**0.075** ↓ 66.5%	**0.056** ↓ 0.75%	**0.533**	**0.130** ↓ 75.6%	**0.068** ↓ 87.2%

Source: The author.

Table 4.2 reports the number of hardware resources, essential bit count, and dynamic power consumption of the Xilinx AXI, AXI-Lite, and Reset IPs integrated into the ZynqNet System. These IPs are used in all ZynqNet configurations and therefore introduce a fixed overhead across all versions. Their resource utilization and dynamic power consumption are very low when compared to the ZynqNet accelerator, indicating that the system area, essential bits, and power consumption are dominated by the accelerator IP rather than by the corresponding Xilinx IPs.

Table 4.2 – Area and power consumption of the Xilinx IPs blocks within the ZynqNet System.

Target	Xilinx AXI IP	Xilinx AXILITE IP	Xilinx RESET IP
LUT	175	368	16
DFF	262	428	33
Essential Bits (10^6)	0.038	0.075	0.003
Dynamic Power (W)	0.001	0.005	< 0.001

Source: The author.

Table 4.3 summarizes the FPGA resource utilization, essential bit count, and dynamic power consumption of the ZynqNet System, which integrates the ZynqNet IP block with the additional Xilinx IPs in the PL, as illustrated in Figure 4.1. Results are reported for configuration with 1-, 2-, and 4- PEs using floating-point, 8-bit fixed-point, and 8-bit integer arithmetic. The results show that these metrics do not vary significantly with respect to the standalone ZynqNet IP, as the accelerator core remains the dominant contributor to area, essential bits, and dynamic power consumption.

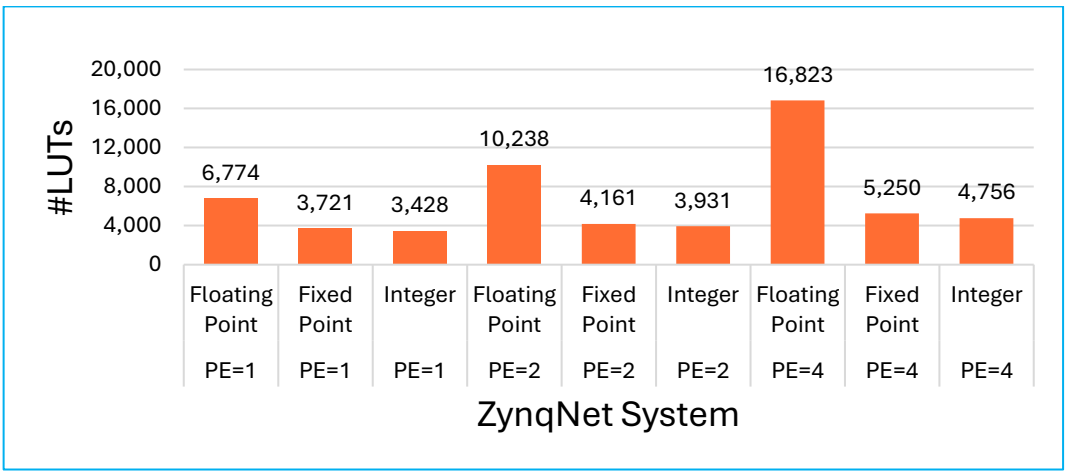
Table 4.3 – Area and dynamic power results of each ZynqNet System with 1, 2 and 4 processing elements computing in floating-point, 8-bit fixed point and 8-bit integer.

Target	**ZynqNet System**								
PE	**1**			**2**			**4**		
Type of Arithmetic	**Floating point**	**Fixed point**	**Integer**	**Floating point**	**Fixed point**	**Integer**	**Floating point**	**Fixed point**	**Integer**
LUT	**6774**	**3721** ↓ 45.1%	**3428** ↓ 49.4%	**10238**	**4161** ↓ 59.4%	**3931** ↓ 61.6%	**16823**	**5250** ↓ 68.8%	**4756** ↓ 71.7%
DFF	**8952**	**4614** ↓ 48.5%	**4006** ↓ 55.3%	**13149**	**5574** ↓ 57.6%	**4241** ↓ 67.7%	**20777**	**7278** ↓ 65.0%	**4656** ↓ 77.6%
BRAM	**13**	**7.5** ↓ 42.3%	**7.5** ↓ 42.3%	**23.5**	**13** ↓ 44.7%	**13** ↓ 44.7%	**44.5**	**24** ↓ 46.1%	**24** ↓ 46.1%
DSP	**53**	**11** ↓ 79.2%	**7** ↓ 86.8%	**100**	**20** ↓ 80.0%	**12** ↓ 88.0%	**194**	**38** ↓ 80.4%	**22** ↓ 88.7%
Essential Bits (10^6)	**1.72**	**0.90** ↓ 47.7%	**0.83** ↓ 51.7%	**2.62**	**1.06** ↓ 59.5%	**0.98** ↓ 62.6%	**4.56**	**1.38** ↓ 69.7%	**1.23** ↓ 73.0%
Dynamic Power (W)	**0.153**	**0.065** ↓ 57.5%	**0.049** ↓ 68.0%	**0.232**	**0.083** ↓ 64.2%	**0.064** ↓ 72.4%	**0.541**	**0.138** ↓ 74.5%	**0.076** ↓ 86.0%

Source: The author.

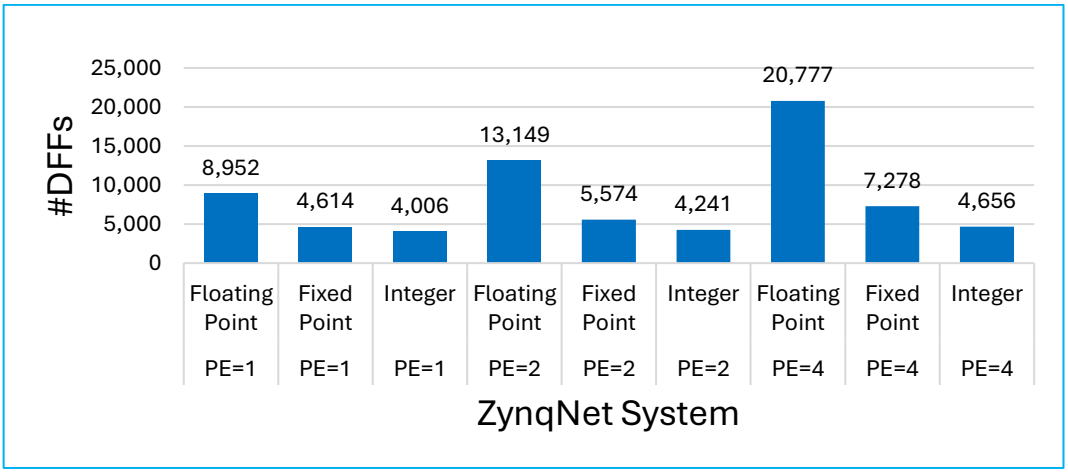
To provide a visual comparison of the results discussed previously, Figures 4.2–4.7 present bar charts comparing the LUT (Figure 4.2), DFF (Figure 4.3), BRAM (Figure 4.4), DSP (Figure 4.5), essential bit count (Figure 4.6), and dynamic power consumption (Figure 4.7) across the evaluated ZynqNet System configurations.

Figure 4.2 – Number of LUTs of each ZynqNet System design.



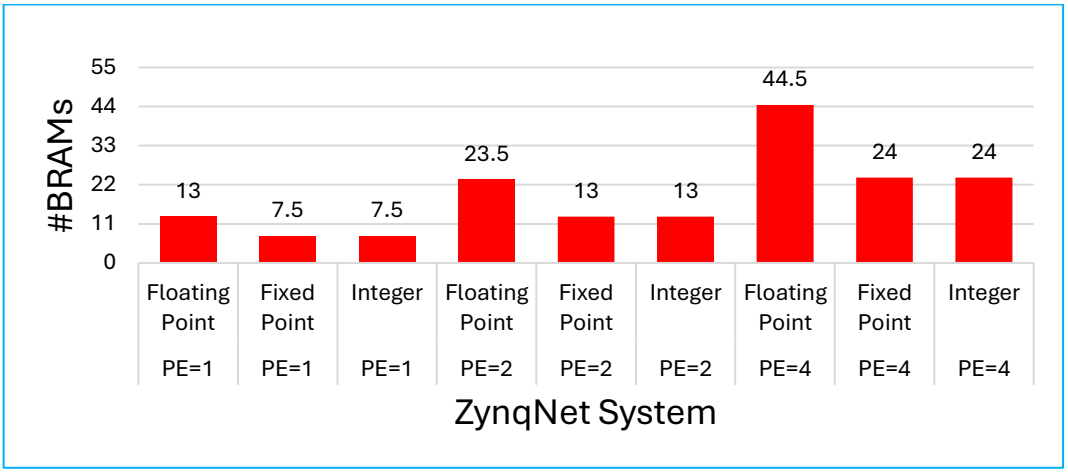
Source: The author.

Figure 4.3 – Number of DFFs of each ZynqNet System design.



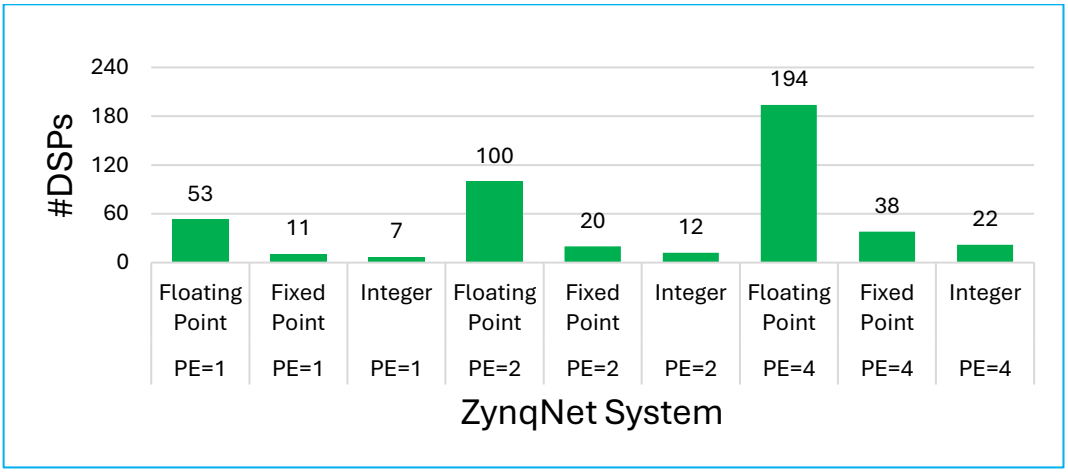
Source: The author.

Figure 4.4 – Number of BRAMs of each ZynqNet System design.



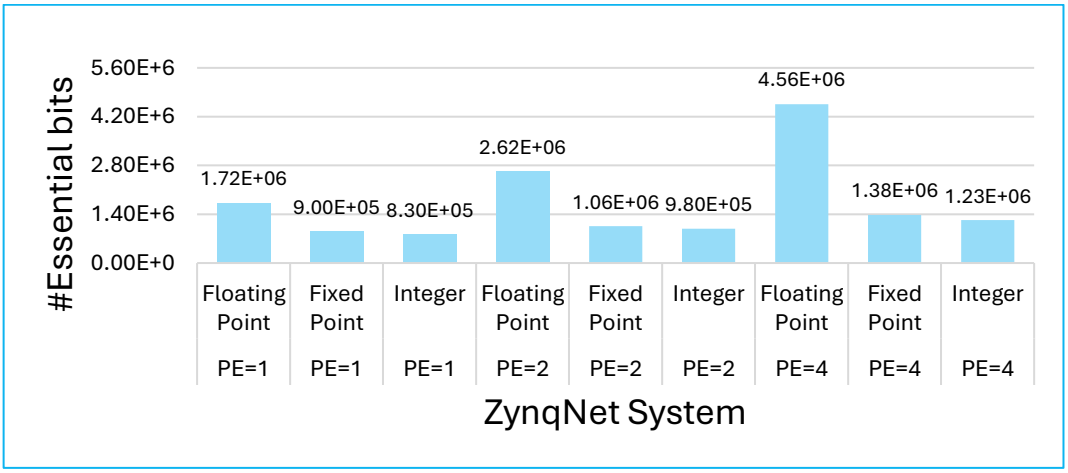
Source: The author.

Figure 4.5 – Number of DSPs of each ZynqNet System design.



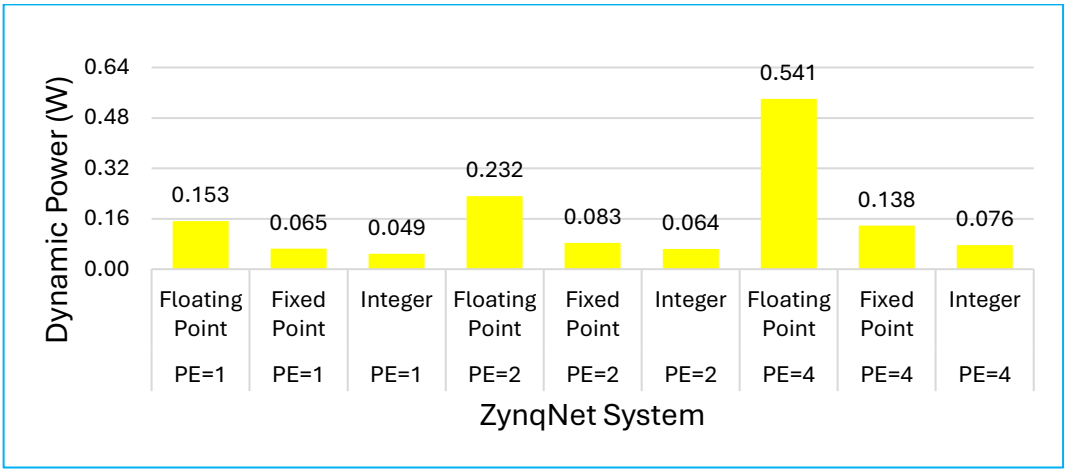
Source: The author.

Figure 4.6 – Number of essential bits of each ZynqNet System design.



Source: The author.

Figure 4.7 – Dynamic Power consumption of each ZynqNet System design.



Source: The author.

Table 4.4 presents the dynamic power consumption of the main Processing System (PS) components in the APSoC for all ZynqNet System versions analyzed in Table 4.3. Since the hardware of the PS remains unchanged across all designs, these power values are constant for all ZynqNet System implementations.

The results indicate that the memory interface is the main contributor to PS dynamic power consumption, exhibiting the highest power value among the PS components. The processor and PLLs show lower power consumption, while the I/O interface and AXI interconnect contribute only marginally to the total PS power.

Table 4.4 – Power consumption of the PS elements in the APSoC.

Target	Processor	Memory Interface	I/O Interface	PLLs	AXI
Dynamic Power (W)	0.277	0.889	0.006	0.355	0.006

Source: The author.

In the second set of results, the performance of each ZynqNet System implemented on the FPGA is evaluated while running the application software on the processor in the PS of the APSoC. Table 4.5 reports the results in terms of preprocessing time and CNN execution time, expressed in milliseconds, number of CPU cycles, and frames per second (FPS).

Table 4.5 – Performance of the ZynqNet System with 1, 2 and 4 processing elements computing in floating-point, 8-bit fixed point and 8-bit integer.

Target	**ZynqNet System**								
PE	**1**			**2**			**4**		
Type of Arithmetic	**Floating point**	**Fixed point**	**Integer**	**Floating point**	**Fixed point**	**Integer**	**Floating point**	**Fixed point**	**Integer**
Preprocessing (ms)	**3.59**	**4.13** ↑ 15.0%	**4.12** ↑ 14.8%	**3.57**	**4.10** ↑ 14.8%	**4.10** ↑ 14.8%	**3.57**	**4.12** ↑ 15.4%	**4.13** ↑ 15.7%
CNN (ms)	**3.54**	**2.72** ↓ 23.2%	**2.36** ↓ 33.3%	**3.44**	**2.61** ↓ 24.1%	**2.21** ↓ 35.8%	**3.37**	**2.54** ↓ 24.6%	**2.12** ↓ 37.1%
CNN (#cyc. x10^5)	**11.8**	**9.06** ↓ 23.2%	**7.86** ↓ 33.3%	**11.47**	**8.69** ↓ 24.1 %	**7.37** ↓ 35.8%	**11.24**	**8.45** ↓ 24.6%	**7.08** ↓ 37.1%
CNN (FPS)	**282.6**	**367.9** ↑ 30.2%	**424.1** ↓ 50.1%	**290.6**	**383.7** ↑ 32.0%	**452.1** ↑ 55.6%	**296.5**	**394.5** ↑ 33.1%	**471.0** ↑ 58.9%

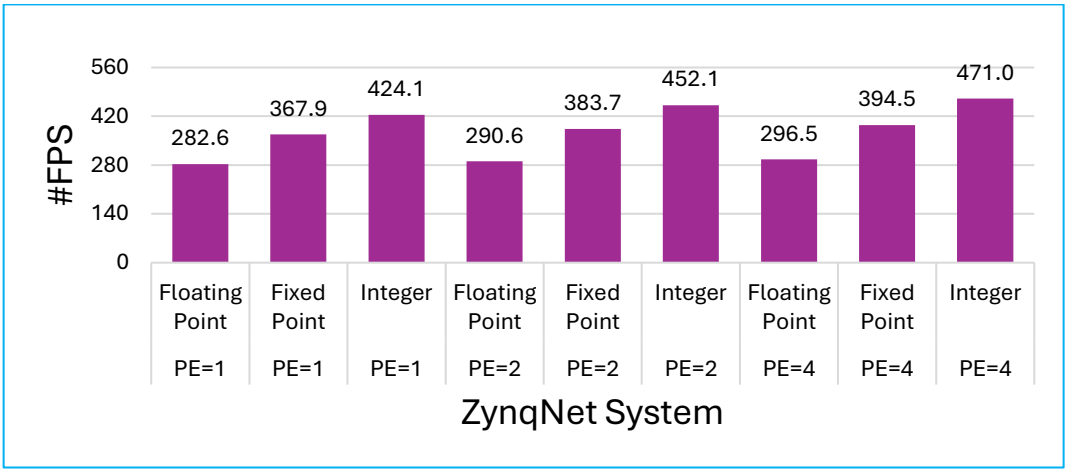
Source: The author.

The preprocessing time, which includes copying, cleaning, and scrubbing operations of weights and pixels, shows only minor variations across configurations. As this stage is executed on the PS, it is weakly affected by the arithmetic precision and is not affected by the number of processing elements.

In contrast, the CNN processing time is significantly reduced when moving from floating-point to lower precision arithmetic. For the integer implementation, the CNN execution time, expressed either in milliseconds or in CPU Cortex-A9 cycles (executing at 666.67 MHz), is reduced by 33.3%, 35.8%, and 37.1% for configurations with 1-, 2-, and 4- PEs, respectively. This reduction in execution time translates into a substantial increase in throughput. The integer configurations achieve FPS improvement of up to 50.1%, 55.6%, and 58.9% for 1-, 2-, and 4-PEs, respectively, clearly demonstrating the combined impact of lower arithmetic precision and increased parallelism on the performance of the system.

Figure 4.8 presents a bar chart comparing the frames per second (FPS) achieved by each ZynqNet System design, allowing a visual assessment of the impact of the number of processing elements and arithmetic precision on the overall throughput.

Figure 4.8 – Frames per second (FPS) processed by each ZynqNet System design.



Source: The author.

According to the results obtained, the 8-bit integer representation proved to be the most efficient in terms of area, power, and performance. In the following sections, this arithmetic representation was used in ZynqNet with two PEs, chosen as a case study.

4.2 Data packing and communication architectures

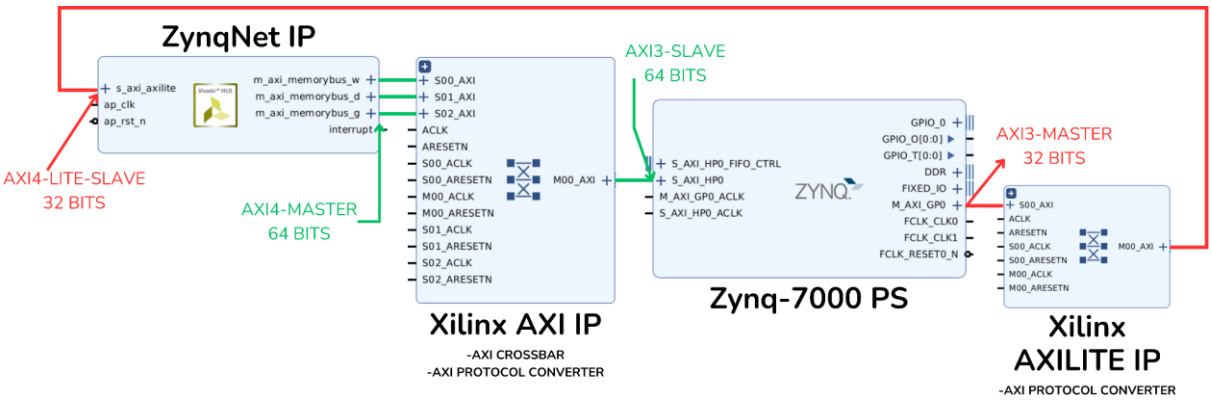
The communication between the Processing System (PS) and the Zynqnet accelerator can be a bottleneck in terms of performance. There are many ways to implement the organization of data in memory and subyacent hardware according to the Zynq-7000 architecture.

To reduce data transfer time and fully utilize the 64-bit data bus bandwidth between the ZynqNet accelerator and the external DDR memory, 8-bit integer values were packed in groups of eight into the DDR memory to optimize read and write operations. These values included image and feature map pixels, weights, and biases. In ZynqNet, three small cache memories (1×8, 64-bit) were designed and implemented with flip-flops: one for weights, one for pixels, and one for output channels, enabling parallel 64-bit input reading and output transmission.

Additionally, to study the impact of varying the number of ZynqNet AXI ports for parallel reading and writing, six communication architectures were designed (Figure 4.9 to Figure 4.14). The study consists of varying the number of AXI interfaces (w: weights, d: pixels, g: global pooling results) in ZynqNet, AXI IP blocks employed, and adopting different PS configurations. The AXILITE IP block is maintained in all evaluated arquitectures because it is only used to write data configuration before the processing of each CNN layer. The first

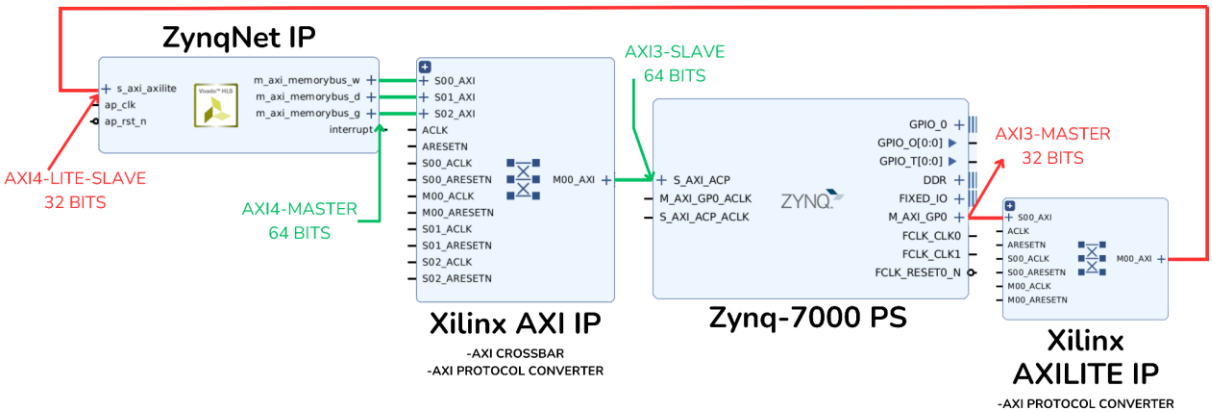
architecture (Figure 4.9) connects ZynqNet through three AXI interfaces to a single AXI Interconnect block and considers one high-performance (HP) port in the PS. The second (Figure 4.10) also uses three AXI interfaces and one AXI Interconnect block but connects to the PS through the accelerator coherency port (ACP) with both L1D and L2 caches enabled. In the third architecture (Figure 4.11), the three AXI interfaces of ZynqNet are linked to three independent AXI Interconnect blocks, each connected to a HP port in the PS. The fourth (Figure 4.12) employs three AXI interfaces and two independent AXI Interconnect blocks, mapping two HP ports in the PS. The fifth (Figure 4.13) only considers a single AXI interface in ZynqNet, connected through one AXI Interconnect block to the ACP in the PS with the L1D and L2 caches enabled. Finally, the sixth architecture (Figure 4.14) also uses a single AXI interface and one AXI Interconnect block, but connects to a single HP port in the PS.

Figure 4.9 – Communication architecture ZynqNet-PS, case 1.



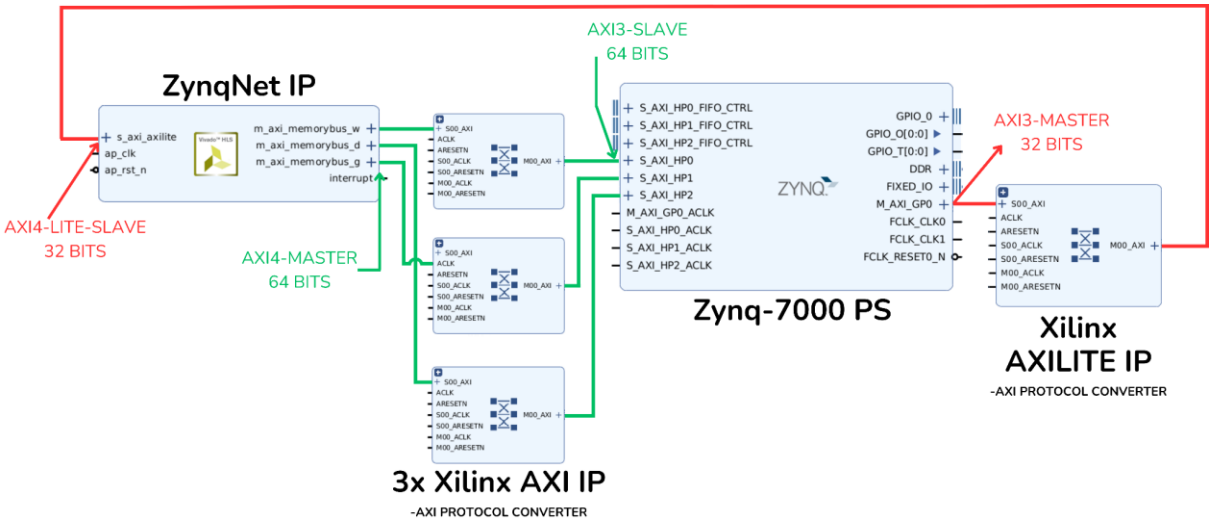
Source: The author.

Figure 4.10 – Communication architecture ZynqNet-PS, case 2.



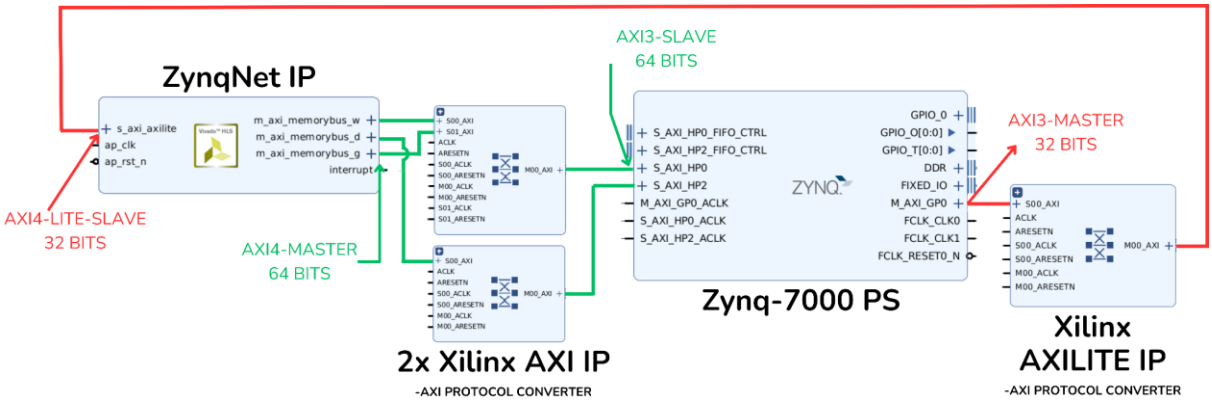
Source: The author.

Figure 4.11 – Communication architecture ZynqNet-PS, case 3.



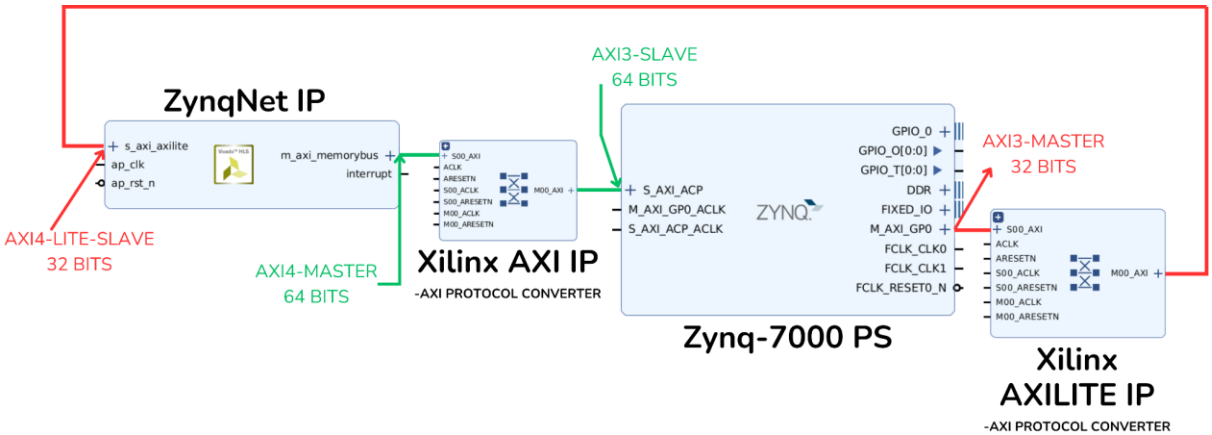
Source: The author.

Figure 4.12 – Communication architecture ZynqNet-PS, case 4.



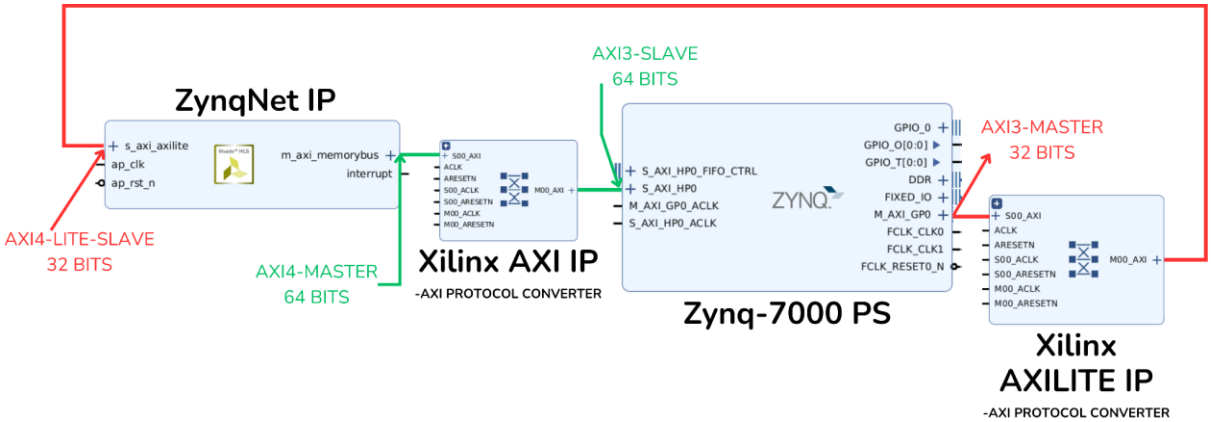
Source: The author.

Figure 4.13 – Communication architecture ZynqNet-PS, case 5.



Source: The author.

Figure 4.14 – Communication architecture ZynqNet-PS, case 6.



Source: The author.

Table 4.6 summarizes the area results obtained for the six communication architectures implemented on the case-study APSoC, considering a configuration with 2-PE and 8-bit integer arithmetic. Architecture 6 is used as the baseline for comparison. Architectures 1 to 4 exhibit a noticeable increase in LUT, DFF, BRAM usage, and essential bits with respect to the baseline, while maintaining the same number of DSPs. In contrast, architecture 5 shows resource utilization very close to architecture 6, with only marginal variations.

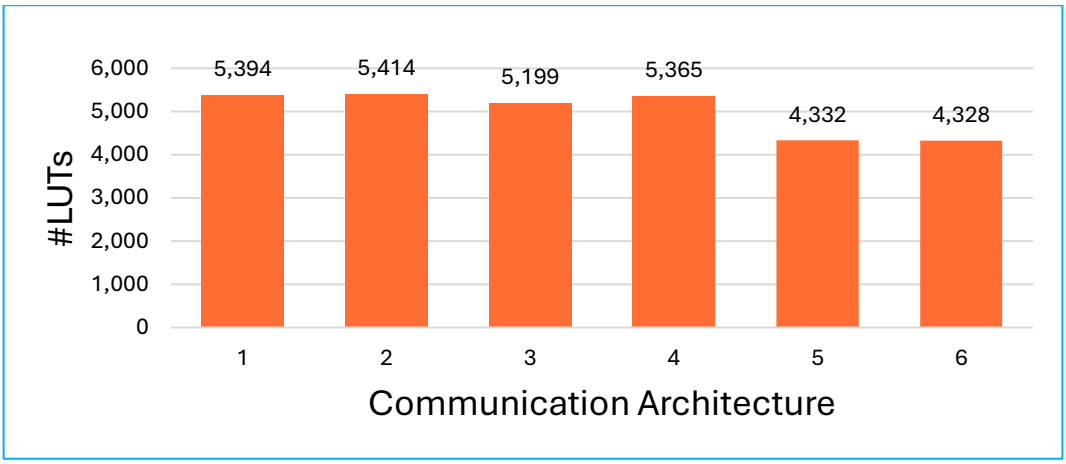
Table 4.6 – Area results comparison between different types of communication architecture designs with PE=2, 8-bit integer.

Communication architecture	**1**	**2**	**3**	**4**	**5**	**6**
LUT	**5394** ↑ 24.6%	**5414** ↑ 25.1%	**5199** ↑ 20.1%	**5365** ↑ 24.0%	**4332** ↑ 0.1%	**4328**
DFF	**6493** ↑ 27.1%	**6499** ↑ 27.3%	**6545** ↑ 28.2%	**6697** ↑ 31.1%	**5107** ↑ 0.0%	**5107**
BRAM	**15** ↑ 11.1%	**15** ↑ 11.1%	**15** ↑ 11.1%	**15** ↑ 11.1%	**13.5** ↑ 0.0%	**13.5**
DSP	**12** ↑ 0.0%	**12** ↑ 0.0%	**12** ↑ 0.0%	**12** ↑ 0.0%	**12** ↑ 0.0%	**12**
Essential Bits (10^6)	**1.27** ↑ 21.0%	**1.28** ↑ 21.9%	**1.25** ↑ 19.0%	**1.28** ↑ 21.9%	**1.04** ↓ 1.0%	**1.05**

Source: The author.

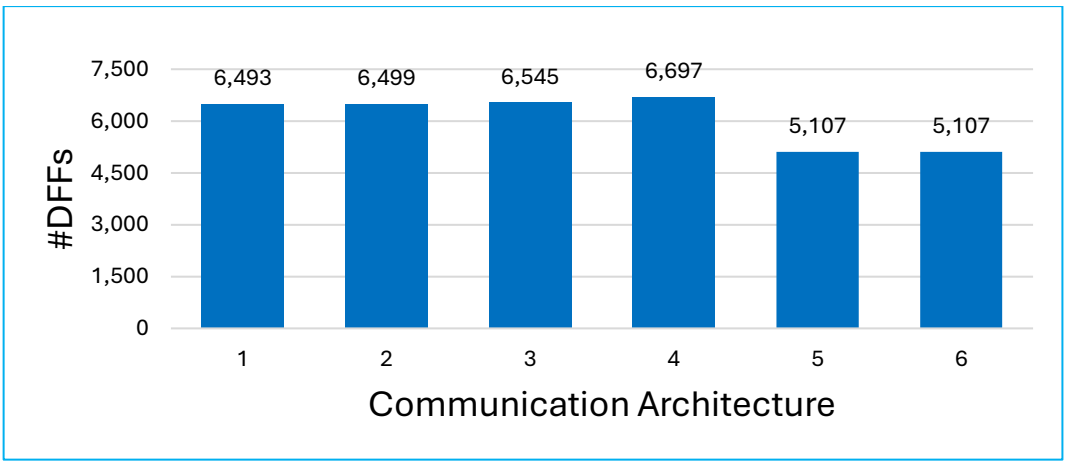
Figures 4.15 to 4.19 present bar charts comparing the resource utilization across the different communication architectures, including LUTs (Figure 4.15), DFFs (Figure 4.16), BRAMs (Figure 4.17), DSPs (Figure 4.18), and essential bits (Figure 4.19).

Figure 4.15 – Number of LUTs of each communication architecture design.



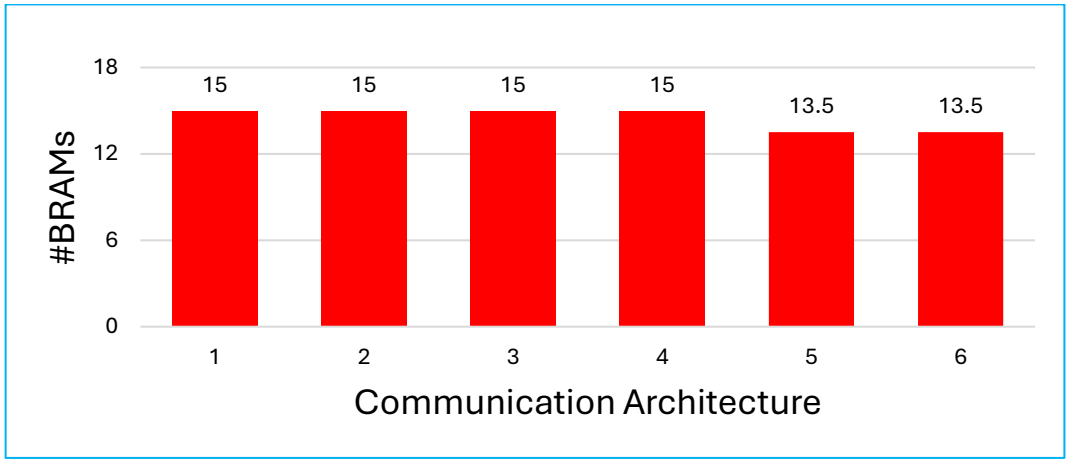
Source: The author.

Figure 4.16 – Number of DFFs of each communication architecture design.



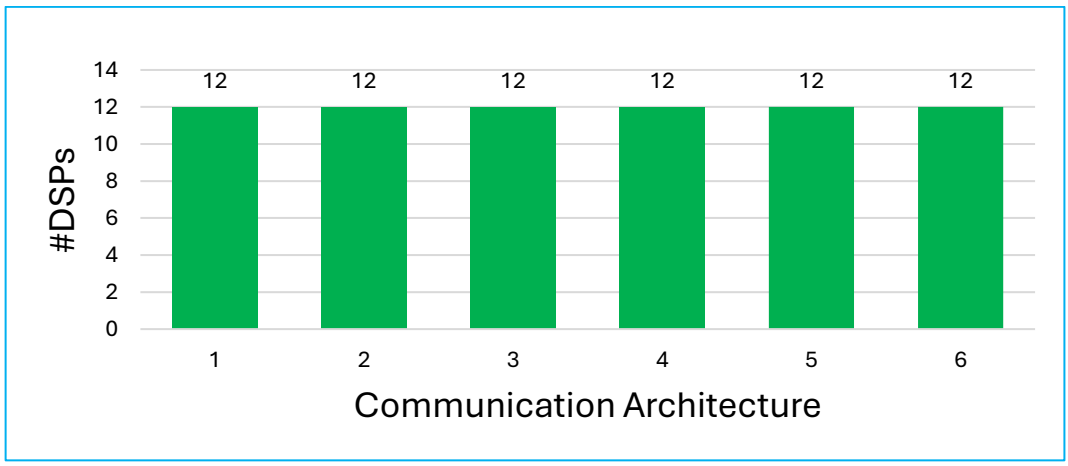
Source: The author.

Figure 4.17 – Number of BRAMs of each communication architecture design.



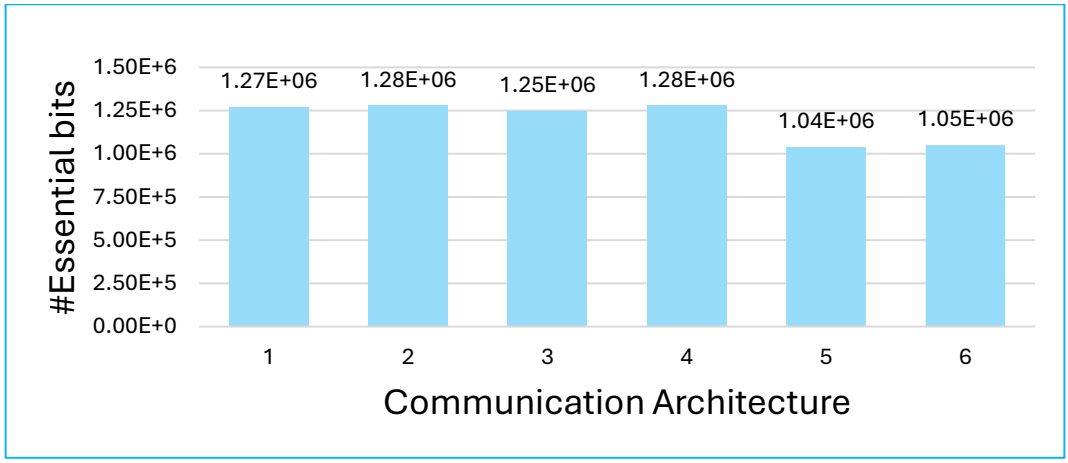
Source: The author.

Figure 4.18 – Number of DSPs of each communication architecture design.



Source: The author.

Figure 4.19 – Number of essential bits of each communication architecture design.



Source: The author.

Table 4.7 compares the dynamic, static, and total power consumption of the different communication architectures, considering PE=2 and 8-bit integer arithmetic, with architecture 6 used as the baseline. Overall, power consumption remains very similar across all architectures, with variations below 1.5% in dynamic and total power. The presence of PS cache (L1D and L2) in architectures 2 and 5 does not lead to significant power differences, indicating that communication architecture choices have a limited impact on overall power consumption in this configuration.

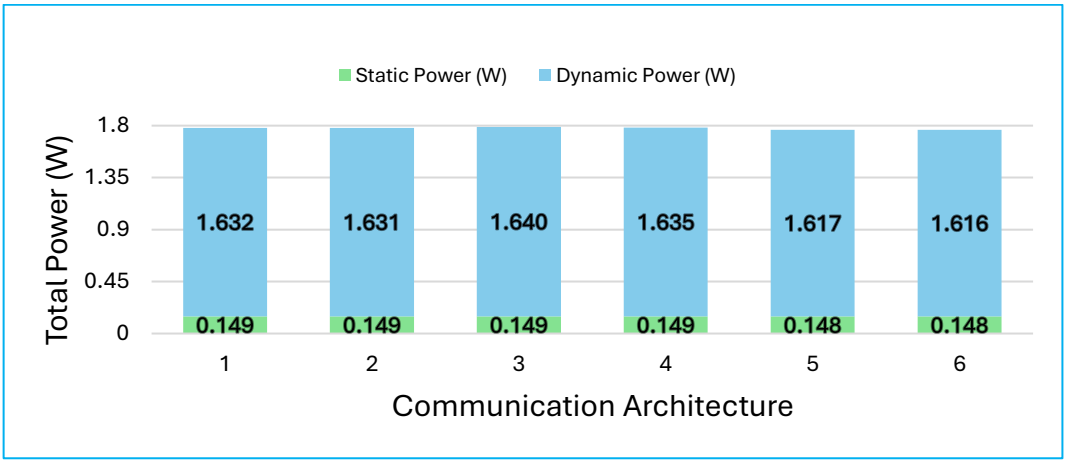
Table 4.7 – Power results comparison between different types of communication architecture designs with PE=2, 8-bit integer.

Communication architecture	**1**	**2**	**3**	**4**	**5**	**6**
PS: Cache L1D and L2	No	Yes	No	No	Yes	No
Dynamic Power (W)	**1.632** ↑ 1.0%	**1.631** ↑ 0.9%	**1.640** ↑ 1.5%	**1.635** ↑ 1.2%	**1.617** ↑ 0.1%	**1.616**
Static Power (W)	**0.149** ↑ 0.7%	**0.149** ↑ 0.7%	**0.149** ↑ 0.7%	**0.149** ↑ 0.7%	**0.148** ↑ 0.0%	**0.148**
Total Power (W)	**1.781** ↑ 1.0%	**1.780** ↑ 0.9%	**1.789** ↑ 1.4%	**1.784** ↑ 1.1%	**1.765** ↑ 0.1%	**1.764**

Source: The author.

Figure 4.20 presents a stacked bar chart comparing the static and dynamic power consumption of the evaluated communication architectures. The figure visually confirms the minor differences in power observed across architectures.

Figure 4.20 – Total power consumption of each communication architecture design.



Source: The author.

Table 4.8 compares the performance of the different communication architectures for PE=2 using 8-bit integer arithmetic. Architectures enabling L1D and L2 caches on the PS (architectures 2 and 5) significantly reduce the preprocessing time, achieving a reduction of about 40.9% compared to the baseline architecture. The highest throughput is obtained when caches are enabled, reaching up to 667.6 FPS, while architectures without cache support show only marginal variations with respect to the baseline.

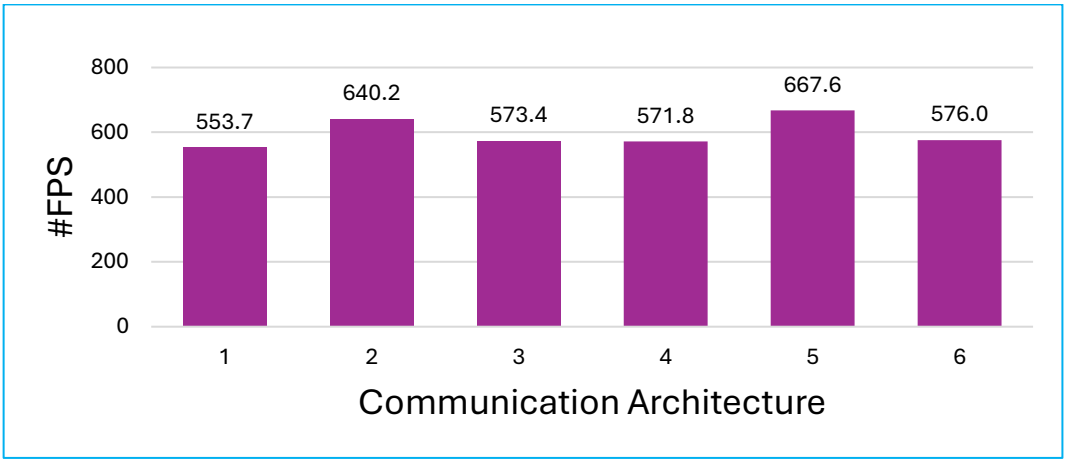
Table 4.8 – Performance results comparison between different types of communication architecture designs with PE=2, 8-bit integer.

Communication architecture	**1**	**2**	**3**	**4**	**5**	**6**
PS: Cache L1D and L2	No	Yes	No	No	Yes	No
Preprocessing (ms)	**4.07** ↑ 0.2%	**2.40** ↓ 40.9%	**4.09** ↑ 0.7%	**4.07** ↑ 0.2%	**2.40** ↓ 40.9%	**4.06**
CNN (ms)	**1.81** ↑ 4.0%	**1.56** ↓ 10.3%	**1.74** ↑ 0.0%	**1.75** ↑ 0.6%	**1.50** ↓ 13.8%	**1.74**
CNN (#cyc. x10^5)	**6.02** ↑ 4.0%	**5.21** ↓ 10.0%	**5.81** ↑ 0.3%	**5.83** ↑ 0.7%	**4.99** ↓ 13.8%	**5.79**
CNN (FPS)	**553.7** ↓ 3.9%	**640.2** ↑ 11.1%	**573.4** ↓ 0.5%	**571.8** ↓ 0.7%	**667.6** ↑ 15.9%	**576.0**

Source: The author.

Figure 4.21 presents a bar chart comparing the achieved FPS for each communication architecture, visually highlighting the performance gains obtained when cache mechanisms are enabled on the PS.

Figure 4.21 – Frames per second (FPS) processed by each communication architecture design.



Source: The author.

From these results, the architecture 5 provides the best performance, achieving the lowest CNN processing time compared to other architectures, due to the use of the L1D and L2 caches in the PS.

To further analyze the impact of the number of PEs on area, power and performance results, the ZynqNet System is evaluated using 1, 2 and 4 PEs, while keeping the 8-bit integer arithmetic and communication architecture 5 fixed.

Table 4.9 presents the results obtained from implementing these designs on the case-study APSoC. As expected, increasing the number of PEs leads to a progressive growth in hardware resources. LUT and DFF utilization increase moderately with the number of PEs, while BRAM and DSP usage show a more pronounced growth due to the use of more local memory and computing units. A similar trend is observed in the number of essential bits, which increases with system parallelism.

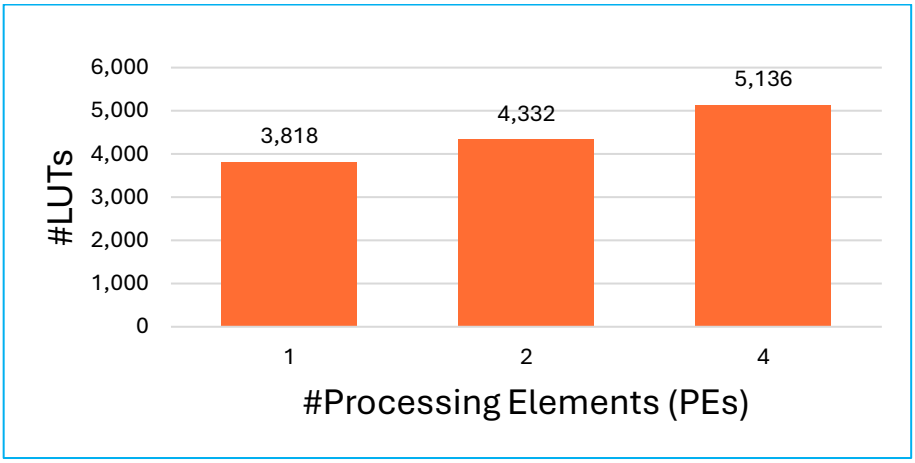
Figures 4.22 to 4.26 present bar charts comparing the FPGA resource utilization of the ZynqNet System for communication architecture 5 using 8-bit integer arithmetic with 1-, 2-, and 4- PEs. The figures show a clear and expected increase in LUT (Figure 4.22), DFF (Figure 4.23), BRAM (Figure 4.24), DSP (Figure 4.25), and essential bit (Figure 4.26) usage as the number of PEs grows.

Table 4.9 – Area results of the ZynqNet System (arch 5, integer) with PE=1, 2, and 4.

PE	1	2	4
LUT	**3818**	**4332** ↑ 13.5%	**5136** ↑ 34.5%
DFF	**4888**	**5107** ↑ 4.5%	**5539** ↑ 13.3%
BRAM	**8**	**13.5** ↑ 68.75%	**24.5** ↑ 206.3%
DSP	**7**	**12** ↑ 71.4%	**22** ↑ 214.3%
Essential Bits (10^6)	**0.91**	**1.04** ↑ 14.3%	**1.32** ↑ 45.1%

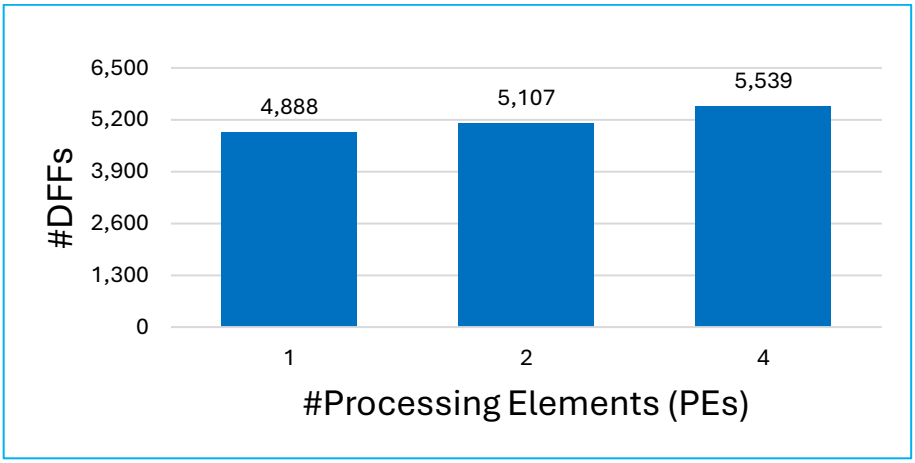
Source: The author.

Figure 4.22 – Number of LUTs of the ZynqNet System design (arch 5, integer) with PE=1, 2, and 4.



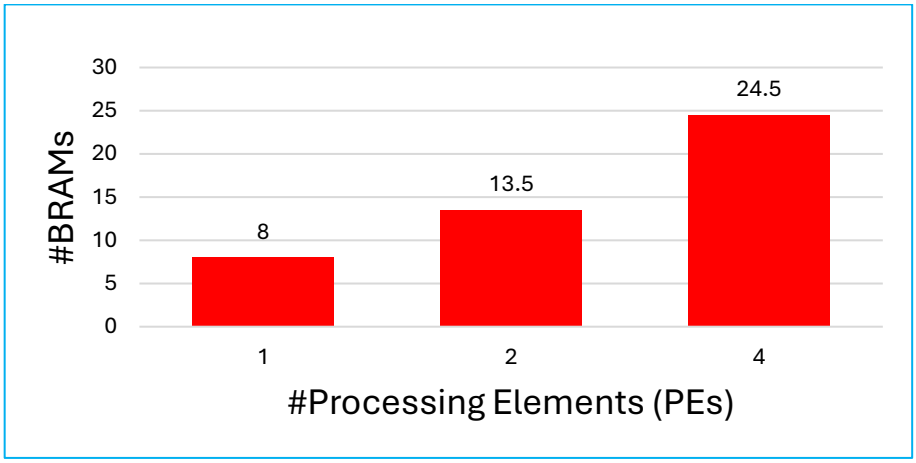
Source: The author.

Figure 4.23 – Number of DFFs of the ZynqNet System design (arch 5, integer) with PE=1, 2, and 4.



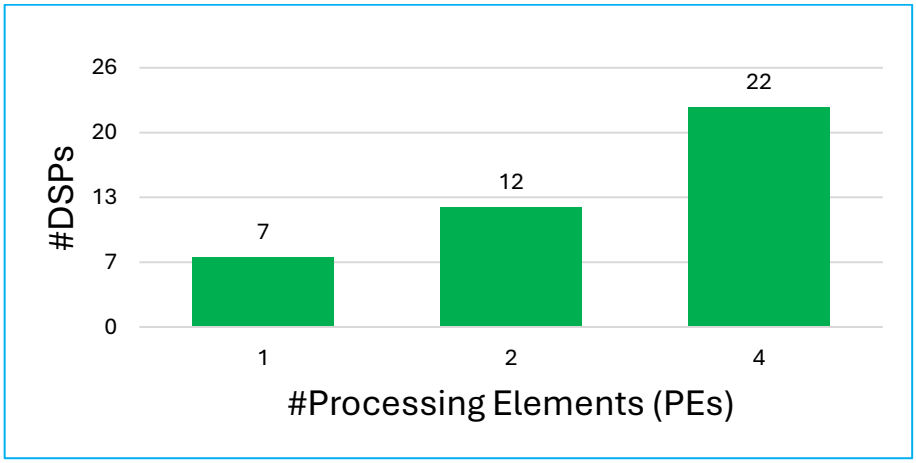
Source: The author.

Figure 4.24 – Number of BRAMs of the ZynqNet System design (arch 5, integer) with PE=1, 2, and 4.



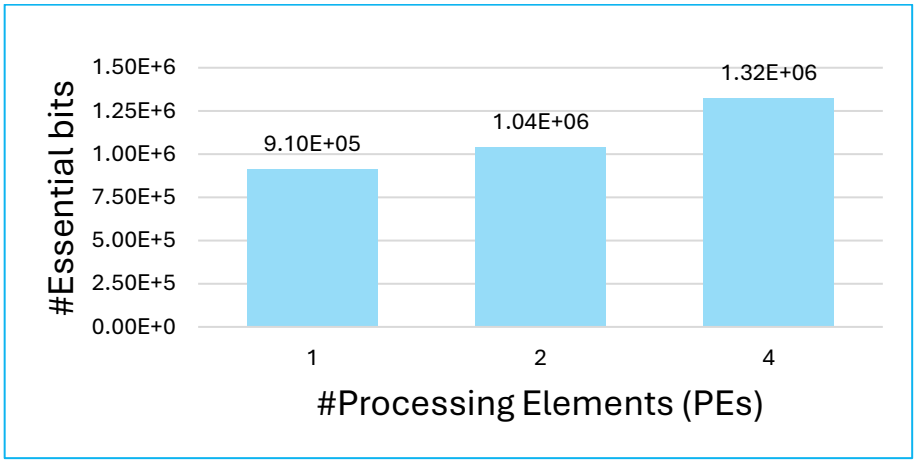
Source: The author.

Figure 4.25 – Number of DSPs of the ZynqNet System design (arch 5, integer) with PE=1, 2, and 4.



Source: The author.

Figure 4.26 – Number of essential bits of the ZynqNet System design (arch 5, integer) with PE=1, 2, and 4.



Source: The author.

Table 4.10 reports the dynamic, static, and total power consumption of the ZynqNet System using communication architecture 5 and 8-bit integer arithmetic for configurations with 1-, 2-, and 4- PEs. As the number of PEs increases, a slight rise in power consumption is observed, with total power growing by only 1.8% for 2-PEs and 3.0% for 4-PEs compared to the 1-PE configuration. This moderate increase indicates that the additional parallelism introduced by higher PE quantity has a limited impact on overall power consumption.

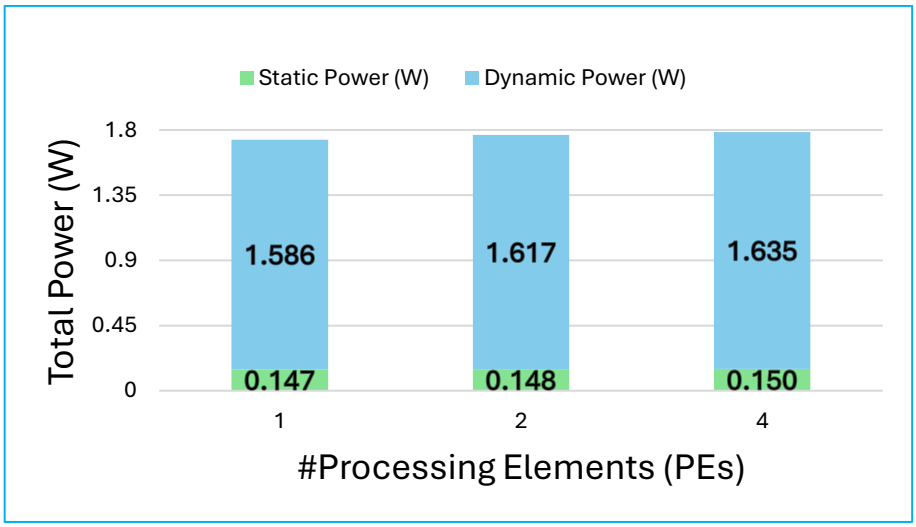
Table 4.10 – Power results of the ZynqNet System (arch 5, integer) with PE=1, 2, and 4.

PE	1	2	4
Dynamic Power (W)	**1.586**	**1.617** ↑ 2.0%	**1.635** ↑ 3.1%
Static Power (W)	**0.147**	**0.148** ↑ 0.7%	**0.150** ↑ 2.0%
Total Power (W)	**1.733**	**1.765** ↑ 1.8%	**1.785** ↑ 3.0%

Source: The author.

Figure 4.27 visually illustrates this trend, highlighting the gradual increase in total power as the number of PEs scales.

Figure 4.27 – Power consumption of the ZynqNet System (arch 5, integer) with PE=1, 2, and 4.



Source: The author.

Table 4.11 presents the performance results of the ZynqNet System using communication architecture 5 and 8-bit integer arithmetic for configurations with 1-, 2-, and 4-PEs. The preprocessing time remains constant at 2.4 ms for all configurations, as this stage is executed on the PS and is independent of the level of parallelism in the PL. The execution time of the CNN layers 1 to 3 decreases progressively as the number of PEs increases, whereas in the final layer and global pooling stage remains unchanged. Overall, the total CNN execution time is reduced by 9.4% and 13.7% when moving from 1-PE to 2- and 4- PEs, respectively. This reduction directly translates into higher throughput, with the frame rate increasing from 604.6 FPS to 667.6 FPS and 700.8 FPS for 2- and 4- PEs.

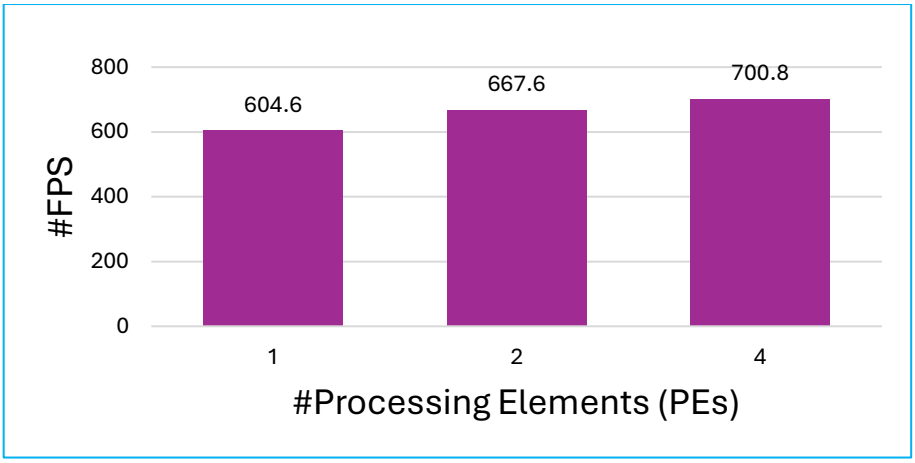
Table 4.11 – Performance results of the ZynqNet System (arch 5, integer) with PE=1, 2, and 4.

PE	**1**	**2**	**4**
Preprocessing (ms)	2.4	2.4	2.4
CNN Layer 1 (ms) **Input: 32x32x4** **Output: 16x16x16**	0.870	0.777	0.736
CNN Layer 2 (ms) **Input: 16x16x16** **Output: 8x8x8**	0.560	0.508	0.488
CNN Layer 3 (ms) **Input: 8x8x8** **Output: 4x4x16**	0.138	0.127	0.117
CNN Layer 4 + GP (ms) **Input: 4x4x16** **Output: 2x2x8/1x1x8**	0.086	0.086	0.086
CNN (ms)	**1.654**	**1.498** ↓ 9.4%	**1.427** ↓ 13.7%
CNN **(#cyc. x10^5)**	**5.51**	**4.99** ↓ 9.4%	**4.76** ↓ 13.6%
CNN (FPS)	**604.6**	**667.6** ↑ 10.4%	**700.8** ↑ 15.9%

Source: The author.

Figure 4.28 indicates that performance improves with an increased number of PEs, but communication between the accelerator and the DDR memory still reveals a bottleneck that needs to be addressed for further performance enhancement.

Figure 4.28 – Frames per second (FPS) processed by the ZynqNet System (arch 5, integer) with PE=1, 2, and 4.



Source: The author.

4.3 Summary of results

The results demonstrated that using 8-bit integer operations in the accelerator was more efficient, providing better performance while requiring fewer resources compared to floating-point or fixed-point implementations. Packing eight 8-bit values in DDR memory and integrating small caches into the accelerator for 64-bit parallel reading and writing in burst mode reduced the number of memory requests, minimized delays, and improved overall performance. In addition, exploring communication architectures between the ZynqNet accelerator and external DDR memory proved essential, as it helped prevent potential data bus bottlenecks that could have limited overall performance. Increasing the number of processing elements improved task distribution across the CNN, but it did not yield a significant performance gain, suggesting that the data transfer rate from DDR memory to the accelerator may constrain the full utilization of the available processing capacity. Activating the L1D and L2 caches in the PS further decreased CNN processing times, improving memory access efficiency. However, in its current state, the use of large caches such as the L1D and L2 in the PS may make the APSoC memories vulnerable to SEUs in radiation environments; therefore, within the scope of this work, the implementation of EDAC techniques or cache flushing methods for these memories was not considered. Considering all the results obtained in this section, the ZynqNet design with 8-bit integer operations, computing the CNN model with 2-PEs and communicating with the processor and the external memory using architecture 6, was

chosen as the most appropriate configuration, achieving the best trade-off between performance and resource utilization and making it the preferred option for further investigation.

5 DESIGN SPACE EXPLORATION: RELIABILITY ENHANCEMENTS

This chapter presents a design space exploration of the ZynqNet accelerator from a reliability perspective. It begins by analyzing the susceptibility of the unmitigated design to SEUs, identifying the modules most critical to system failures. Then, various fault-tolerant strategies, including Coarse-Grained TMR, Modular TMR, Distributed TMR, CRAM scrubbing, and Hamming-based EDAC, are applied and evaluated. Reliability testing is conducted using both emulated fault injection and radiation experiments, and the results are analyzed to quantify improvements in system resilience and fault coverage.

5.1 Unmitigated ZynqNet

The most appropriate configuration of the ZynqNet accelerator selected from the previous chapter is regarded as the unmitigated version (baseline), which means without any fault-tolerant techniques applied. To provide further information on the internal hardware, Table 5.1 summarizes the area and the dynamic power required for the ZynqNet's internal functional blocks (IFBs) implemented on the Zynq-7000 APSoC.

Table 5.1 – Area and power consumption of ZynqNet's IFBs.

Element	ZynqNet IP block, PE=2				
	MACC	**MC**	**AXI**	**Caches**	**Address Generator**
LUT	1166	774	923	245	103
FF	596	985	1490	347	59
BRAM	0	0	2	11.5	0
DSP	10	0	0	0	1
Essential Bits	281,984	186,217	232,712	115,909	24,387
Dynamic Power (W)	0.023	0.005	0.017	0.025	0.002

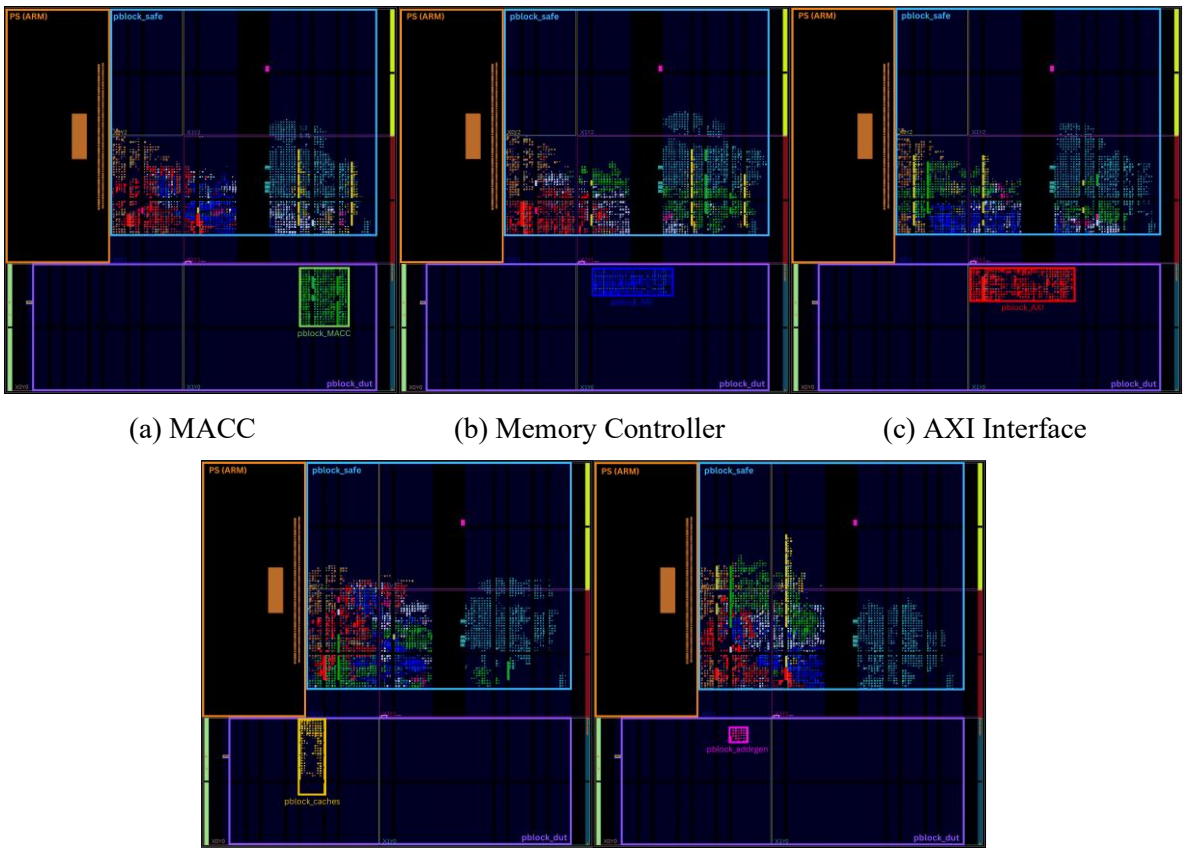
Source: The author.

5.1.1 Investigating the susceptibility of ZynqNet under SEUs

Investigating the ZynqNet's susceptibility to SEUs enables the identification of modules most likely to cause system failures. This allows prioritizing and hardening the most vulnerable components with fault mitigation techniques, such as redundancy, ensuring efficient resource usage and improved reliability. For this purpose, the fault injection methodology, described in Section 3.4.1, is employed to emulate radiation-induced SEUs in the CRAM of the FPGA.

Figure 5.1 illustrates how the ZynqNet's IFBs were placed into the pblock_dut in the FPGA floorplaning to be tested.

Figure 5.1 – FPGA's floorplanning of ZynqNet's IFB designs to be tested.



(a) MACC (b) Memory Controller (c) AXI Interface

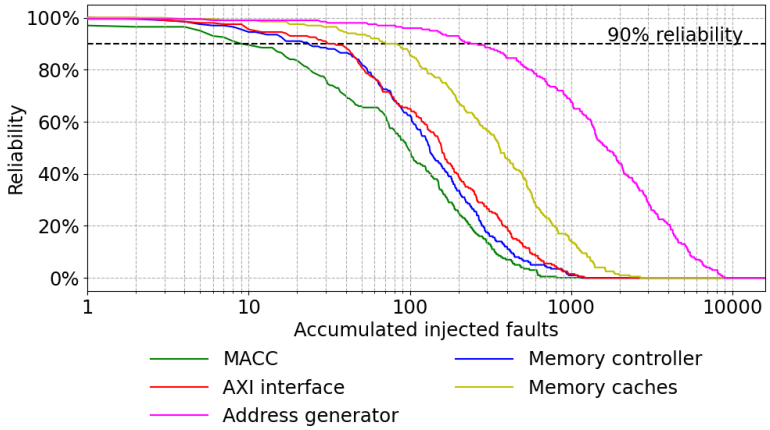
(d) Caches (e) Address Generator

Source: The author.

The results are illustrated through reliability curves in Figure 5.2 and summarized in Table 5.2 through key metrics. As a case-study, a total of 200 failure events were recorded, including inference errors and time-out events. The Mean Faults to Failure (MFTF) and the Mission Faults (MF) for a reliability greater than 90% are also reported. With these results, the

IFBs are ranked according to their susceptibility to SEUs, from the most vulnerable to the least, as follows: the MACC, the memory controller, the AXI interfaces, the cache memories, and the address generator.

Figure 5.2 – Reliability curves of ZynqNet's IFBs.



Source: The author.

Table 5.2 – Emulated Fault Injection results: ZynqNet's IFBs.

Element	**ZynqNet IP block, PE=2**				
	MACC	**MC**	**AXI**	**Caches**	**Address Generator**
Inference error	187	156	141	199	198
Time-out	13	44	59	1	2
Total events	200	200	200	200	200
Total faults	28,942	39,331	46,946	100,268	465,662
MFTF [faults]	144.71	196.66	234.73	501.34	2,328.31
MF [faults] (R≥90%)	9.00	23.49	34.00	83.00	263.00

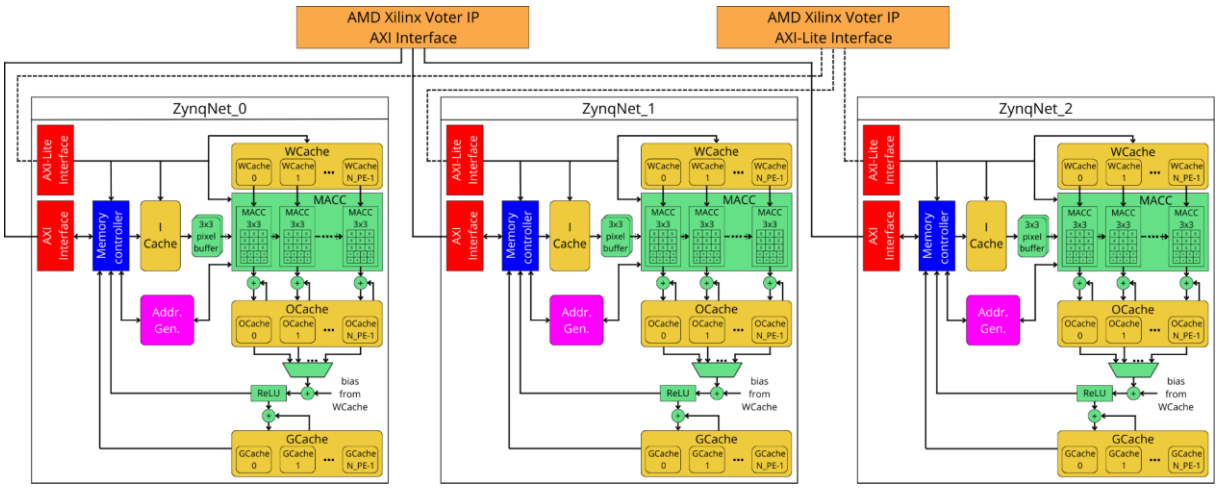
Source: The author.

5.2 Fault-Tolerant ZynqNet

To enhance the reliability of the ZynqNet accelerator and mitigate the impact of radiation-induced faults, particularly SEUs in the CRAM and BRAMs of SRAM-based APSoC, a comprehensive fault-tolerance design flow was adopted based on hardware redundancy, scrubbing and error correction codes.

Seven TMR design variants were explored for ZynqNet to enhance fault masking capabilities. The first one is based on the Coarse-Grained TMR (CGTMR) technique which triplicates the top-level module and places majority voters only at the outputs. In this implementation, the AMD Xilinx Voter IP was used to vote the output signals of ZynqNet's AXI and AXI-Lite Interfaces. Figure 5.3 illustrates the integration of these IPs within the architecture. Although this approach is simple to implement and imposes one of the lowest area overheads, the masking benefit is lost once these larger modules are damaged.

Figure 5.3 – Coarse-Grained TMR design architecture.

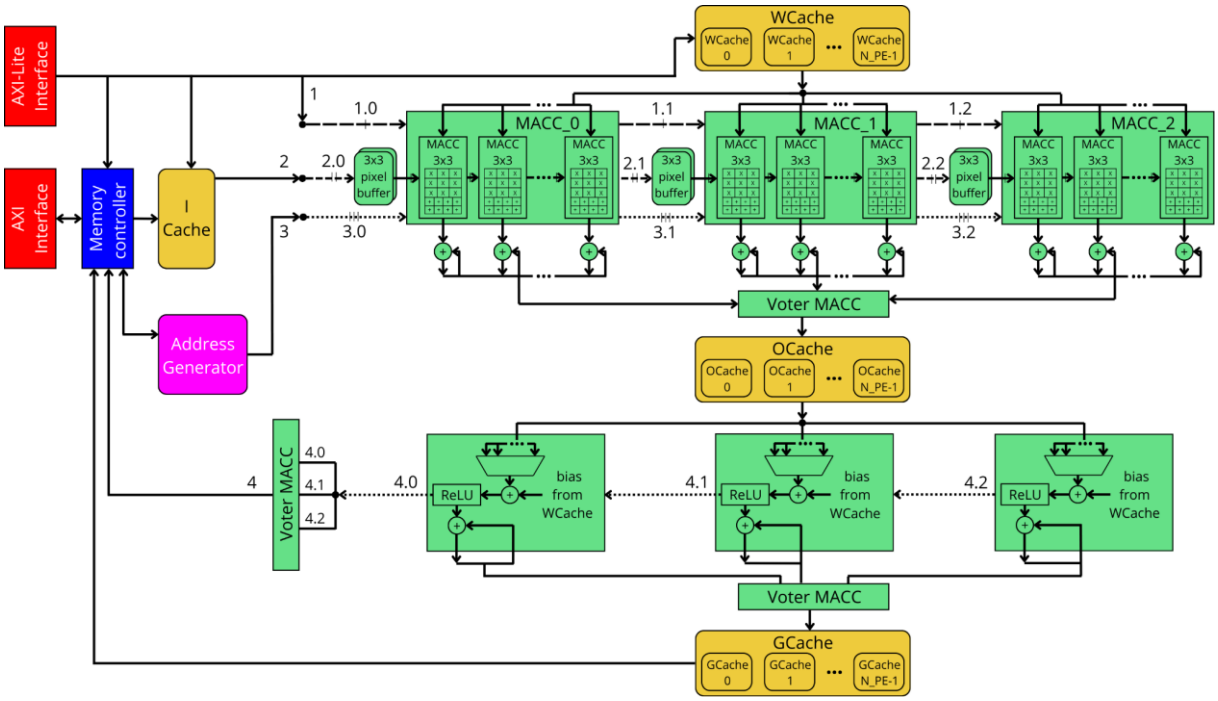


Source: The author.

Then, the next five designs are based on the Modular TMR (MTMR) mitigation strategy which increases protection granularity by triplicating individual IFBs and placing a voter on each set of triplicated outputs. The design of these variants was guided by previously obtained results regarding the susceptibility of each IFB of ZynqNet (Section 5.1.1). In each MTMR-X version, X denotes the number of modules mitigated through modular redundancy, starting from the most fault-prone IFB and proceeding to the least susceptible one. For instance, MTMR-1 hardens only the most susceptible module, the MACC, while the remaining modules are left unhardened (Figure 5.4). MTMR-2 extends modular redundancy to both the MACC and the Memory Controller (Figure 5.5), and this approach is progressively expanded in MTMR-3 (Figure 5.6), MTMR-4 (Figure 5.7), and MTMR-5 (Figure 5.8). In the case of the MTMR-5, each of the five IFBs of the baseline design is triplicated, where all replicas receive the same input and local voters produce unique outputs for each signal that feeds other blocks. A Python-based HDL level automation script was developed to apply MTMR to the ZynqNet's five IFBs: the MACC (green), the Memory Controller (blue), the AXI Interfaces (red), the Cache modules

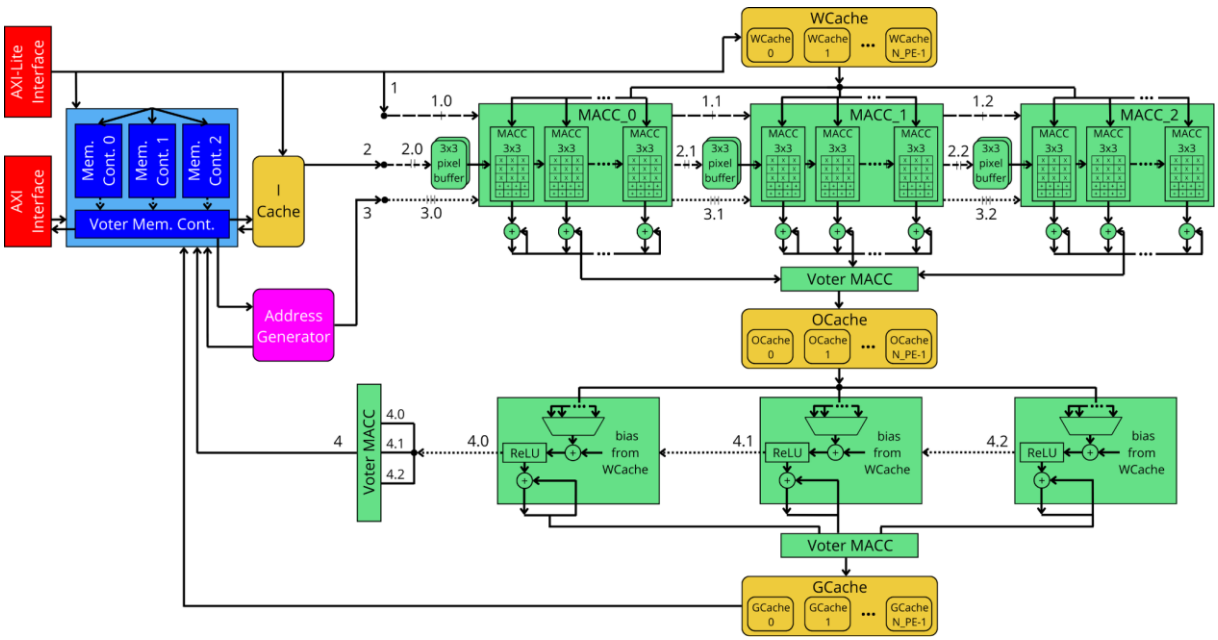
(yellow) and the Address Generator (magenta). MTMR-5 provides finer fault masking and consequently better average resilience compared to CGTMR.

Figure 5.4 – MTMR-1 design architecture.



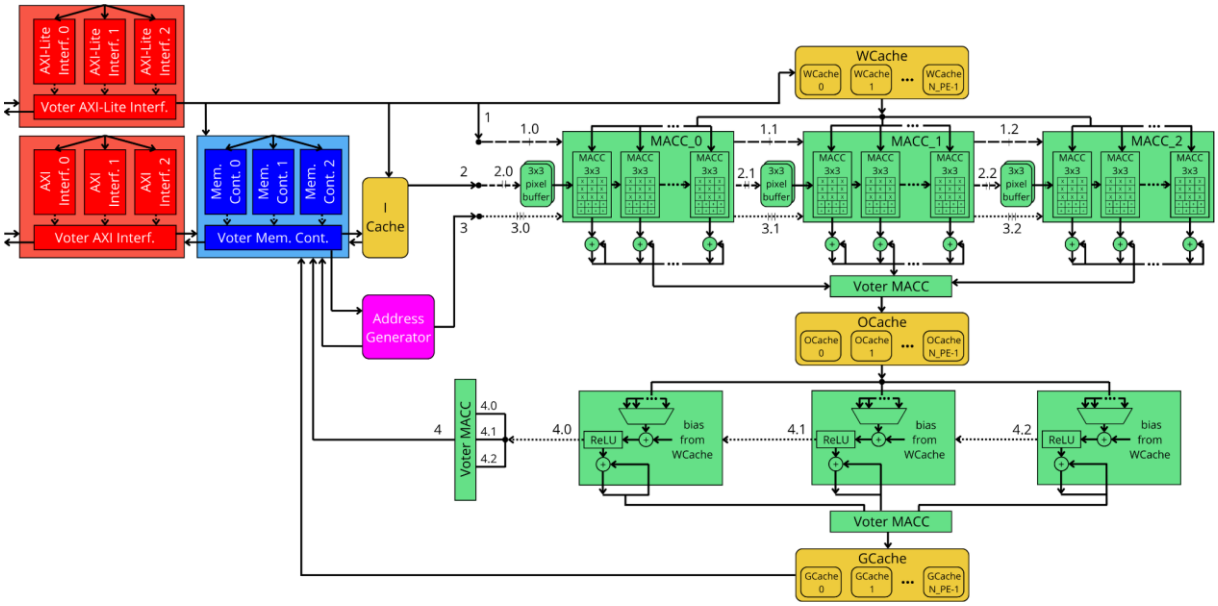
Source: The author.

Figure 5.5 – MTMR-2 design architecture.



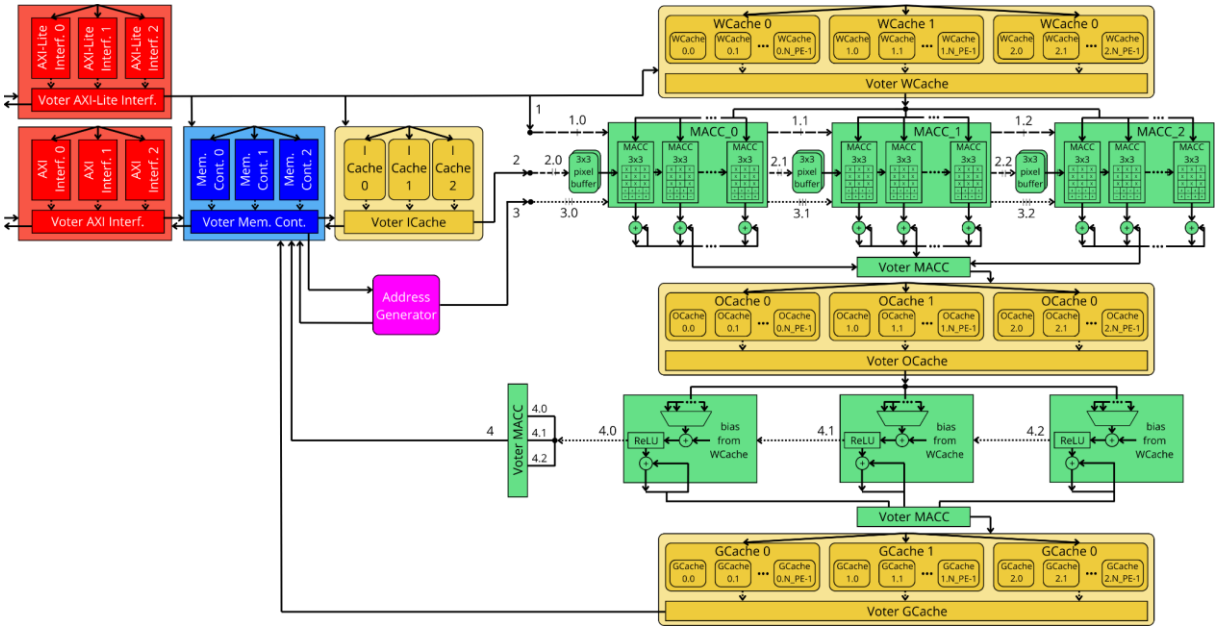
Source: The author.

Figure 5.6 – MTMR-3 design architecture.



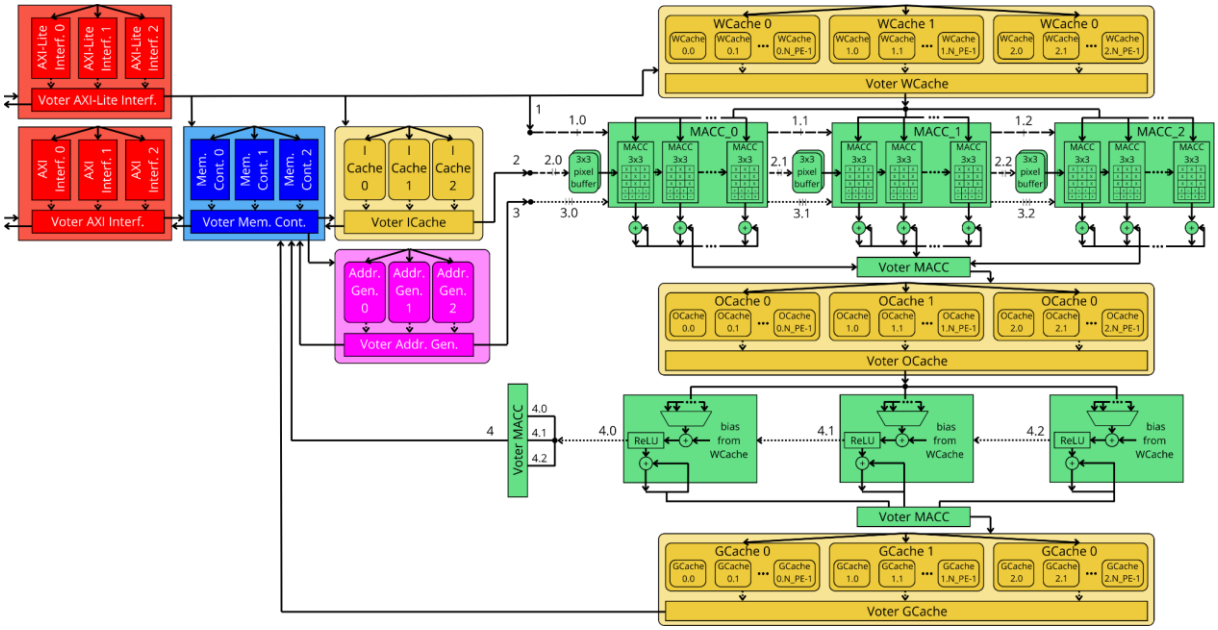
Source: The author.

Figure 5.7 – MTMR-4 design architecture.



Source: The author.

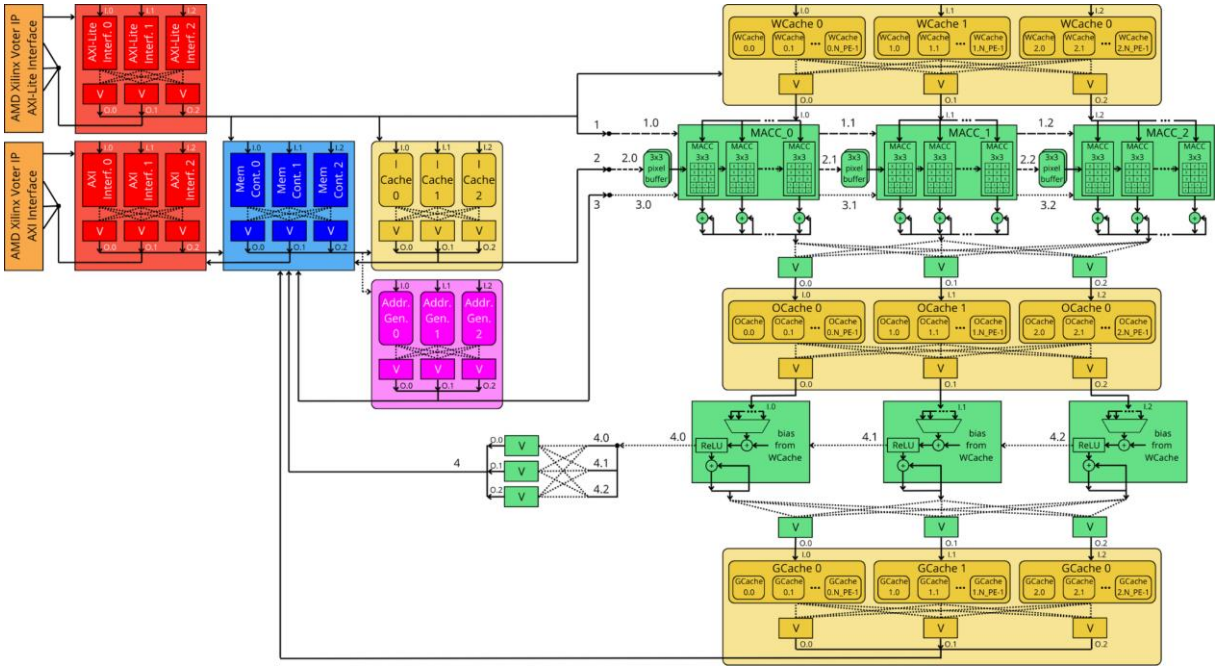
Figure 5.8 – MTMR-5 design architecture.



Source: The author.

Finally, Distributed TMR (DTMR) was implemented to achieve finer granularity as shown in Figure 5.9. It triplicates the top-level circuit, and within the entire design, each triplicated IFB has its independent inputs (I.0, I.1, or I.2) that produce its own outputs, which are locally voted to generate the corresponding output signals O.0, O.1, or O.2. These output signals within each top-level replica only connect to input signals of the same replica (e.g., O.X → I.X, with X = 0, 1, 2). For the final circuit outputs, the AMD Xilinx Voter IP was employed. To automate the design process, a custom Python script was developed to perform the HDL-level triplication of ZynqNet, generate the voters, and strategically insert them at the outputs of each triplicated IFB by modifying the corresponding connections. DTMR significantly enhances fault coverage in intermediate datapaths, where SEUs including both SBUs and MBUs may occur, but at the cost of increased area and timing overhead.

Figure 5.9 – DTMR design architecture.

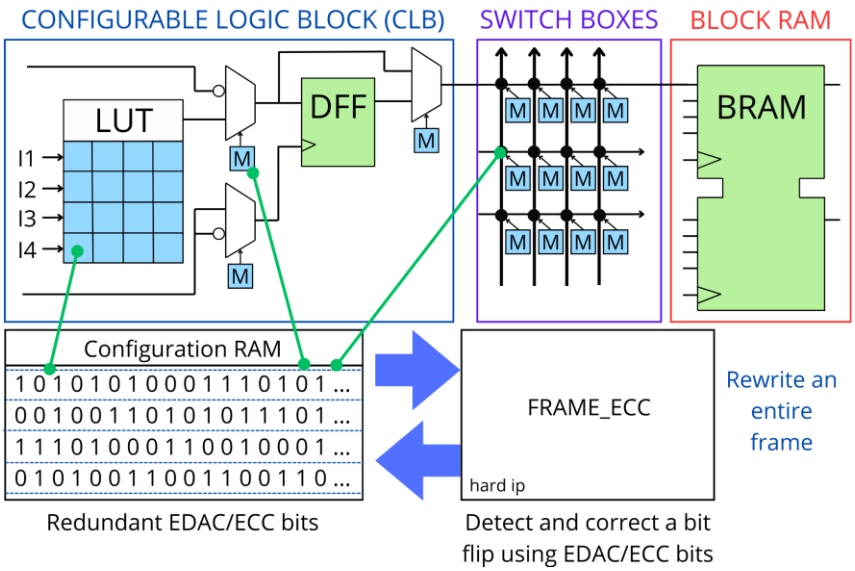


Source: The author.

These TMR strategies offer a trade-off between area overhead and SBU/MBU masking capability. As a case-study, all designs operate at 100 MHz and the number of FPS was not affected by the redundancy implementation, computing approximately 576 FPS.

To correct bit flips in CRAM, a scrubbing mechanism was added (Figure 5.10), which uses ECC code stored in the configuration frames and the bitstream. Consequently, bit flips occurring in DFFs used to implement the caches in ZynqNet are not corrected. As configured, the scrubbing process starts at the top-left frame, and the FRAME_ECC2 hardware checks each frame's ECC; if no SBU is found, it moves to the next frame; if an SBU is detected, the frame is corrected and the scan restarts from the beginning. A limitation of this scrubbing configuration arises in the presence of MBU, since successful correction requires the bit upsets to occur in different frames. Nevertheless, this scrubbing strategy effectively reduces the accumulation of SBUs per frame in the configuration memory, which are generally the dominant radiation-induced faults in SRAM-based FPGAs. Considering that the Zynq-7000 APSoC device is used at 100 MHz, the scrubber sweeps all frames in approximately ~8 ms and then returns to the initial frame to repeat the process. Therefore, in this configuration, the scrubbing takes from 0 to 8 ms to correct CRAM bit flips.

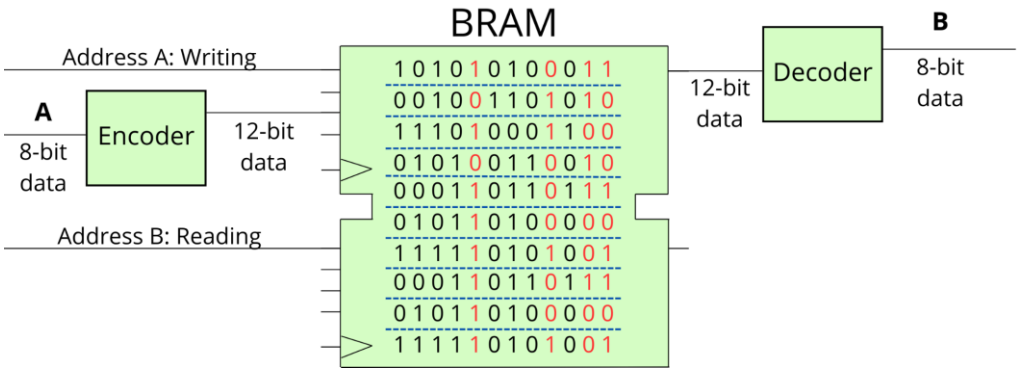
Figure 5.10 – CRAM scrubbing mechanism in the FPGA.



Source: The author.

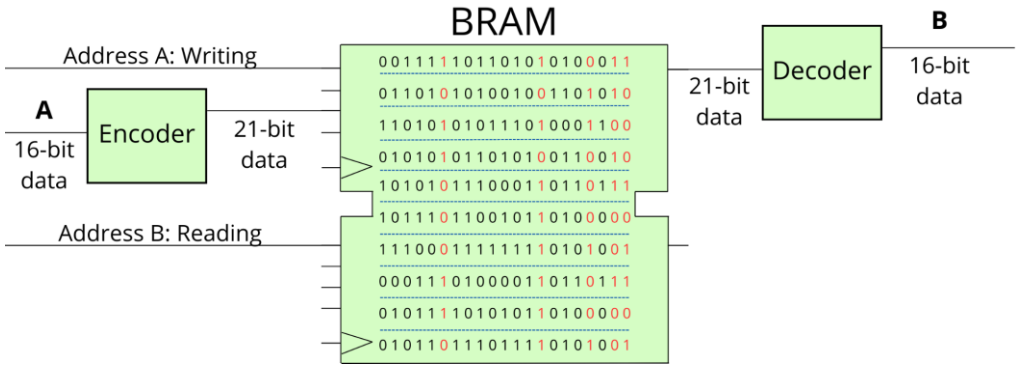
To mitigate bit flips affecting BRAM read data, Hamming-based EDAC modules were integrated into the Caches blocks within the ZynqNet accelerator. In the ICache, WCache, and GCache, 8-bit input data words were first passed through a Hamming encoder, producing 12-bit codewords. The original data bits were placed in the non-power-of-two positions, while the four parity bits were inserted at positions 1, 2, 4, and 8. Each parity bit was computed as the XOR of a specific subset of data bits according to the Hamming code scheme. The resulting codewords were stored in BRAM and subsequently decoded during read operations, allowing the detection and correction of SBU (Figure 5.11). In the OCache, where the original word width was 16 bits, the encoder inserted five parity bits at positions 1, 2, 4, 8, and 16 in each 21-bit codeword, following the same Hamming principle. The decoder then retrieves the original 16-bit data and corrects any SBU during read operations (Figure 5.12). This mechanism mitigates radiation-induced SBUs in the cache memories during read operations while maintaining low hardware overhead.

Figure 5.11 – Hamming-based EDAC for 8-bit data.



Source: The author.

Figure 5.12 – Hamming-based EDAC for 16-bit data.



Source: The author.

Table 5.3 summarizes the area and dynamic power consumption of each fault-tolerant ZynqNet IP design, reported relative to the unmitigated version. As expected, the CGTMR design presents an approximately threefold increase in area and dynamic power with respect to the unmitigated implementation. In contrast, the proposed MTMR configurations introduce a more gradual overhead, with resource usage and power consumption increasing as additional modules are hardened. This allows a clear trade-off between fault coverage and implementation cost, positioning MTMR as a scalable alternative to full redundancy. The DTMR EDAC design shows the highest overhead, reflecting the cost of combining distributed redundancy with error detection and correction mechanisms on BRAMs.

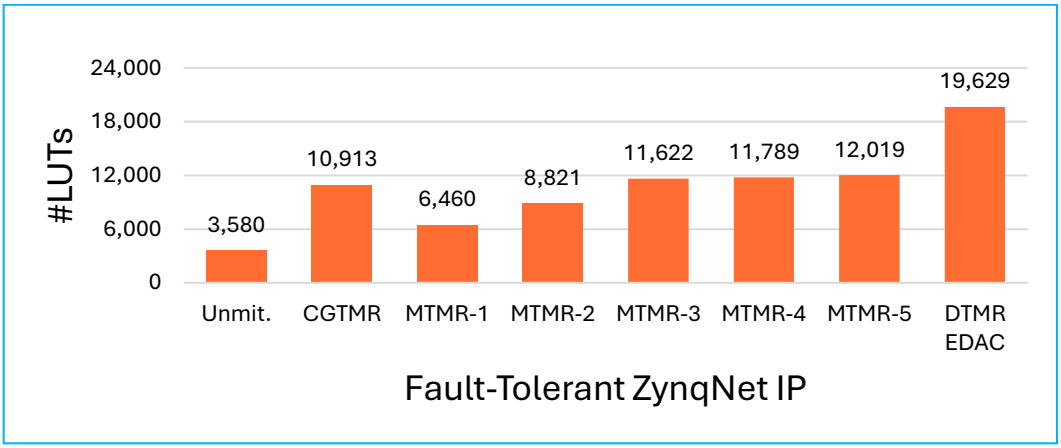
Table 5.3 – Area and dynamic power consumption of each Fault-Tolerant ZynqNet IP design.

ZynqNet IP Designs	LUT	DFF	BRAM	DSP	Essential Bits (10^6)	Dynamic Power (W)
Unmitigated	3,580	4,357	13.5	12	0.91	0.07
CGTMR	10,913 (3.0x)	13,119 (3.0x)	40.5 (3.0x)	36 (3.0x)	2.76 (3.0x)	0.23 (3.2x)
MTMR-1	6,460 (1.8x)	5,699 (1.3x)	13.5 (1.0x)	32 (2.7x)	1.54 (1.7x)	0.12 (1.7x)
MTMR-2	8,821 (2.5x)	7,977 (1.8x)	13.5 (1.0x)	32 (2.7x)	2.00 (2.2x)	0.14 (2.0x)
MTMR-3	11,622 (3.2x)	11,510 (2.6x)	17.5 (1.3x)	32 (2.7x)	2.64 (2.9x)	0.18 (2.5x)
MTMR-4	11,789 (3.3x)	12,669 (2.9x)	40.5 (3.0x)	32 (2.7x)	2.90 (3.2x)	0.25 (3.5x)
MTMR-5	12,019 (3.4x)	12,795 (2.9x)	40.5 (3.0x)	36 (3.0x)	2.94 (3.2x)	0.26 (3.6x)
DTMR EDAC	19,629 (5.5x)	15,274 (3.5x)	40.5 (3.0x)	36 (3.0x)	4.23 (4.7x)	0.34 (4.7x)

Source: The author.

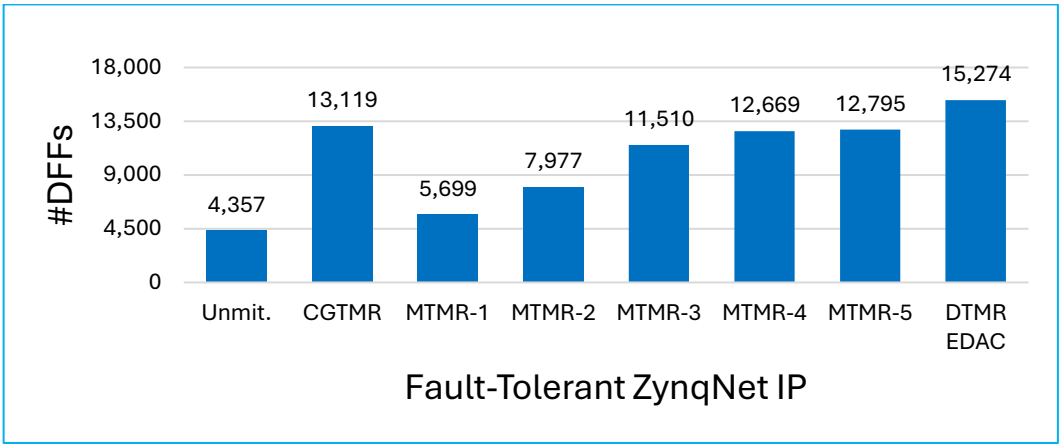
Figures 5.13 to 5.18 present bar charts comparing the FPGA resource utilization, essential bit count, and dynamic power consumption of the different fault-tolerant ZynqNet IP designs. The figures shows more clearly a progressive increase in LUTs (Figure 5.13), DFFs (Figure 5.14), BRAMs (Figure 5.15), DSPs (Figure 5.16), and essential bits (Figure 5.17) as higher levels of redundancy are applied.

Figure 5.13 – Number of LUTs of each Fault-Tolerant ZynqNet IP design.



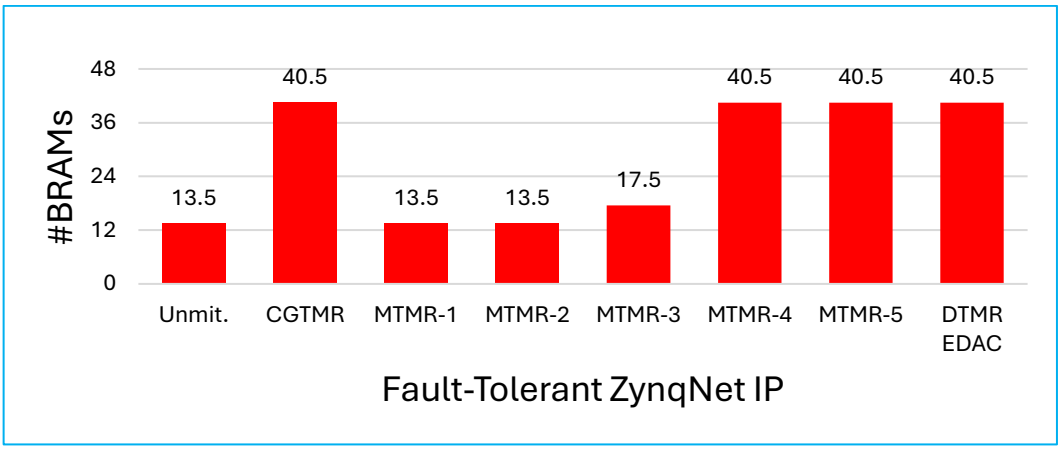
Source: The author.

Figure 5.14 – Number of DFFs of each Fault-Tolerant ZynqNet IP design.



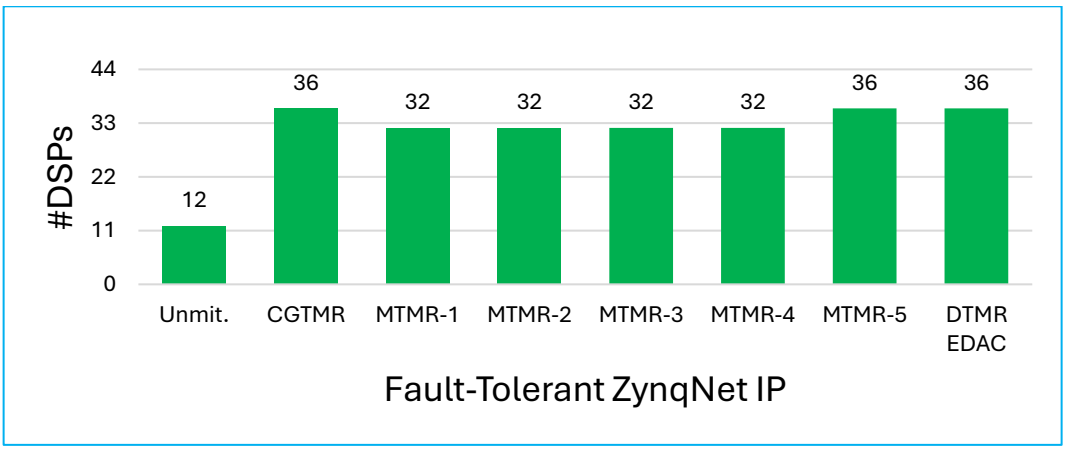
Source: The author.

Figure 5.15 – Number of BRAMs of each Fault-Tolerant ZynqNet IP design.



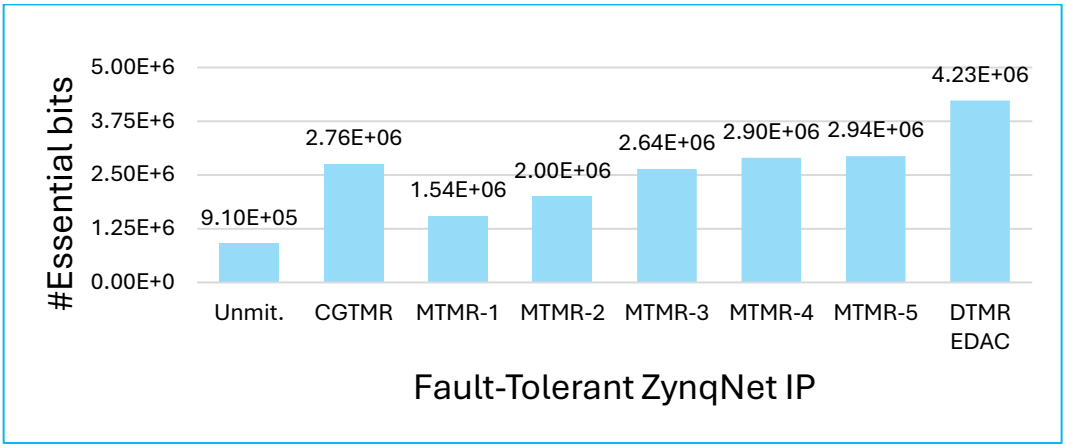
Source: The author.

Figure 5.16 – Number of DSPs of each Fault-Tolerant ZynqNet IP design.



Source: The author.

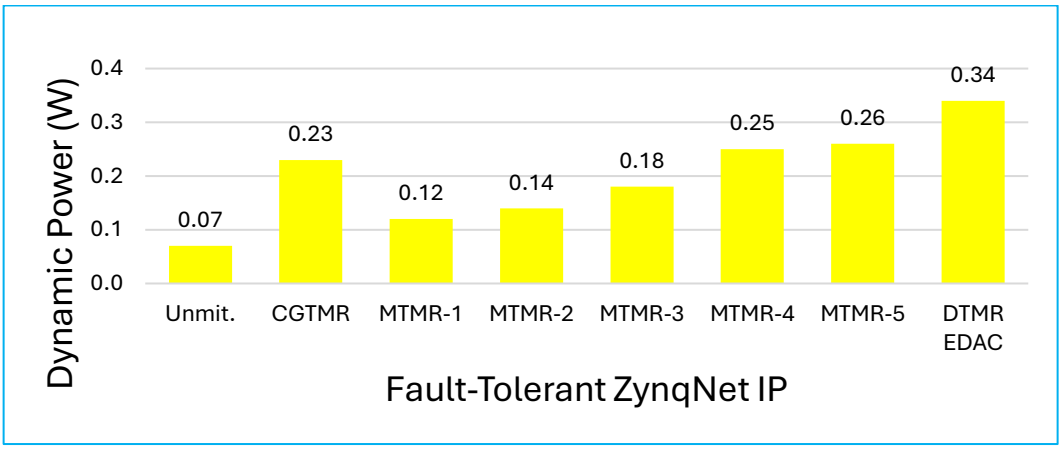
Figure 5.17 – Number of essential bits of each Fault-Tolerant ZynqNet IP design.



Source: The author.

Figure 5.18 further highlights the direct relationship between increased hardware redundancy and dynamic power consumption.

Figure 5.18 – Dynamic power consumption of each Fault-Tolerant ZynqNet IP design.



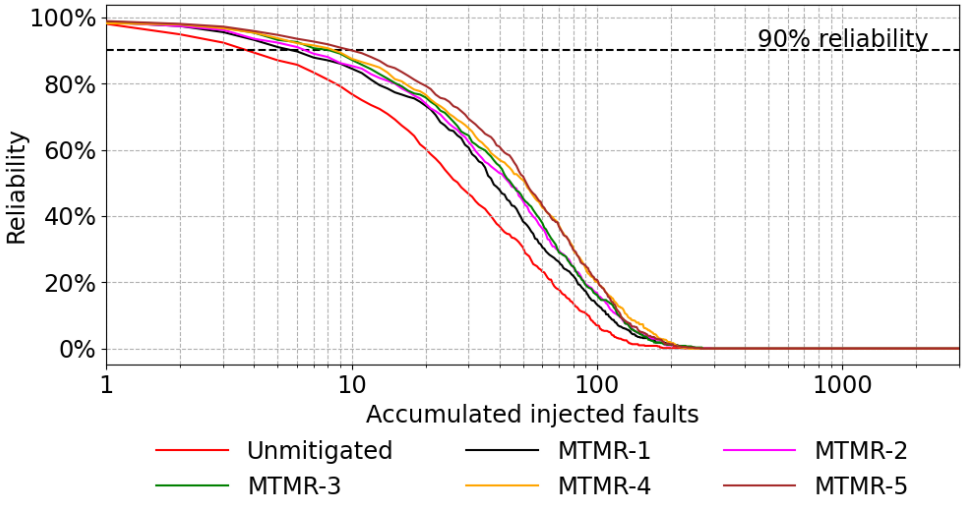
Source: The author.

5.2.1 Reliability testing via Fault Injection

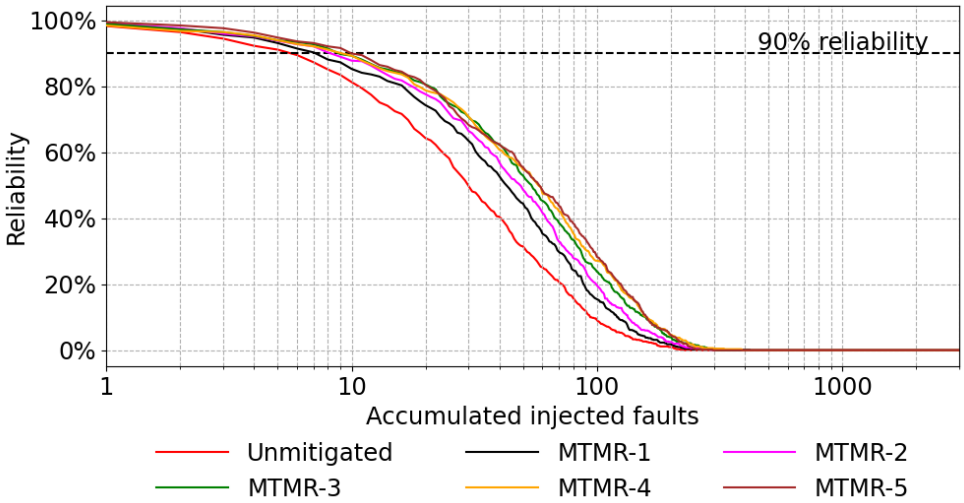
The fault injection methodology proposed in Section 3.4.1 was used to emulate radiation-induced faults in the unmitigated and fault-tolerant designs. The results are presented through reliability curves and key metrics for comparative analysis. The first set evaluates the MTMR versions with and without CRAM scrubbing. Figure 5.19 illustrates how the cumulative insertion of voters at the outputs of each IFB can shift the reliability curve to the right, indicating improved resilience to faults. Among the MTMR versions, MTMR-5 design occupies the

largest area on the FPGA due to the higher number of voter insertion, while also achieving the most reliable response in both cases: with and without CRAM Scrubbing.

Figure 5.19 – Reliability curves of Modular TMR versions.



(a) CRAM Scrubbing disabled

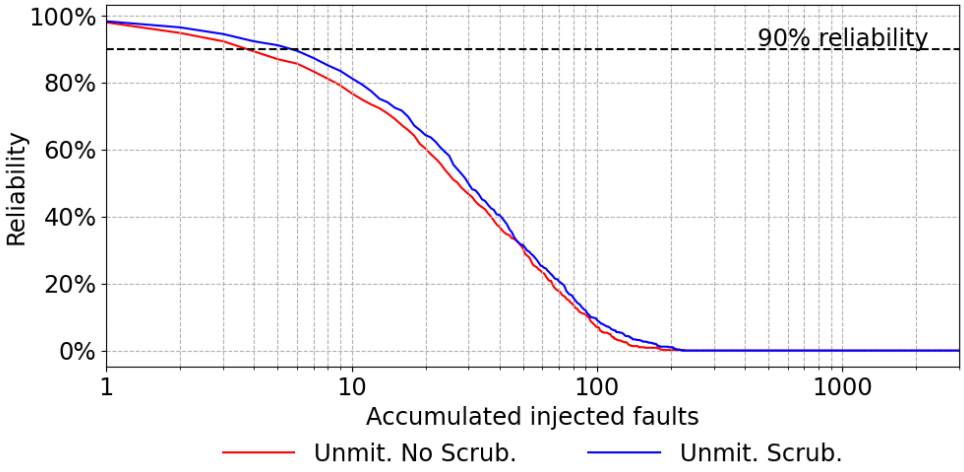


(b) CRAM Scrubbing enabled

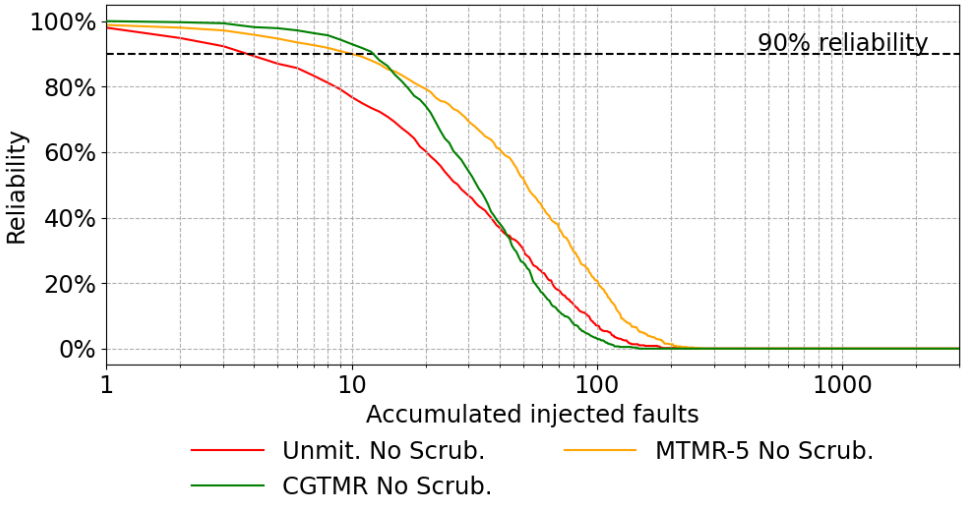
Source: The author.

Next, the results of designs incorporating CGTMR and DTMR EDAC strategies are presented. A general comparison of the most reliable version obtained after applying each TMR-based fault-tolerant strategy is illustrated by reliability curves in Figure 5.20 and key metrics in Table 5.4.

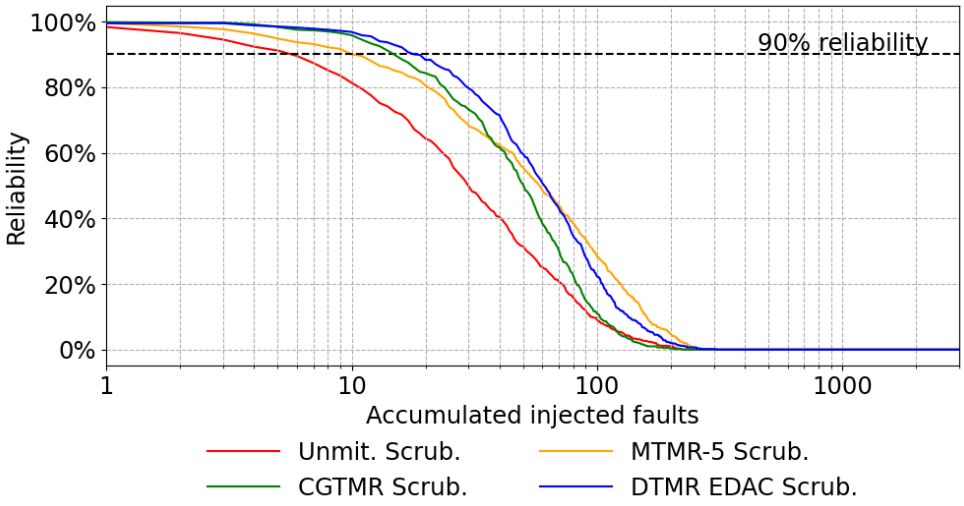
Figure 5.20 – Emulated Fault Injection reliability curves.



(a) Unmitigated designs



(b) CRAM Scrubbing disabled



(c) CRAM Scrubbing enabled

Source: The author.

Figure 5.20 depicts the reliability response under emulated fault injection for the unmitigated and fault-tolerant designs, considering scenarios without CRAM Scrubbing and with CRAM Scrubbing enabled. As expected, the unmitigated design exhibits a rapid reliability degradation, particularly when scrubbing is disabled. In contrast, all fault-tolerant configurations show a significantly slower reliability decay, with scrubbing playing a critical role in extending the operational lifetime of the system.

Table 5.4 reports the number of observed event types, the total number of injected faults, and the resulting reliability metrics. In all cases, the number of time-out events was much lower than the number of inference errors, indicating that incorrect classifications are the primary failure mode under fault injection. Among the evaluated designs, the DTMR EDAC Scrubbing configuration achieved the largest increase in the Mission Faults (MF) factor for R≥90%, with a 4.80× improvement relative to the unmitigated version. This was followed by the CGTMR Scrubbing design, which showed a 3.96× increase, and finally, the MTMR-5 design with a 2.67× gain.

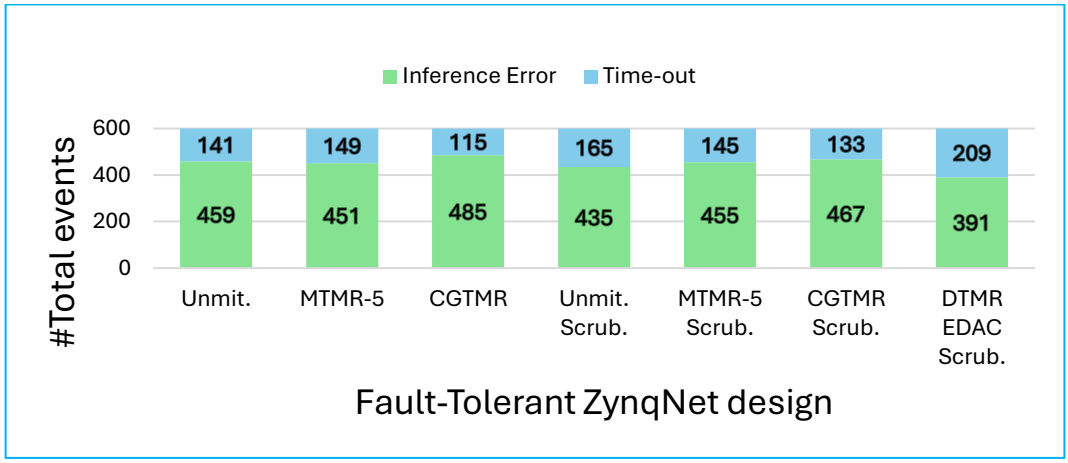
Table 5.4 – Emulated Fault Injection results: fault-tolerant designs.

Designs	**Unmitigated**	**MTMR-5**	**CGTMR**	**Unmitigated Scrubbing**	**MTMR-5 Scrubbing**	**CGTMR Scrubbing**	**DTMR EDAC Scrubbing**
Inference error	459	451	485	435	455	467	391
Time-out	141	149	115	165	145	133	209
Total events	600	600	600	600	600	600	600
Total faults	23,562	37,543	23,232	26,193	44,625	33,795	42,854
MFTF [faults]	39.27	62.57	38.72	43.66	74.38	56.33	71.42
MF [faults] (R≥90%)	3.75	10.00	12.24	5.68	10.00	14.87	18.00
MF (Norm.) (R≥90%)	1.00x	2.67x	3.26x	1.51x	2.67x	3.96x	4.80x

Source: The author

Figure 5.21 presents stacked bar charts showing the distribution of event types observed in each fault-tolerant ZynqNet System design.

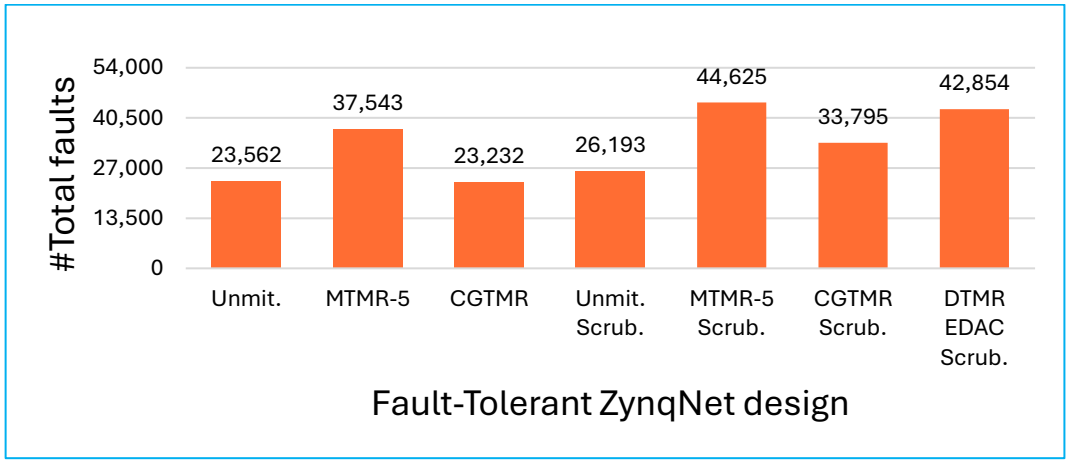
Figure 5.21 – Number of event types observed in each fault-tolerant ZynqNet System design.



Source: The author.

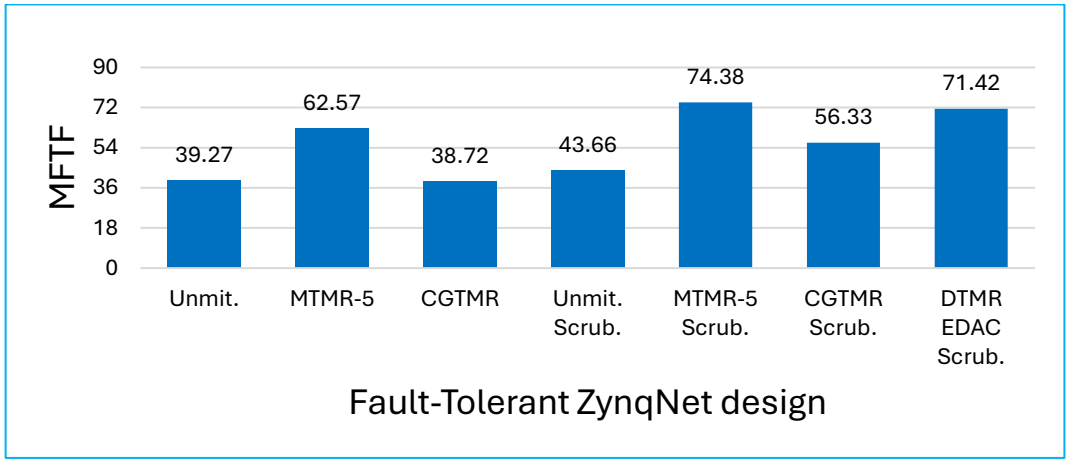
The total number of injected faults (Figure 5.22), the Mean Faults To Failure (MFTF) (Figure 5.23), and the Mean Faults for R ≥ 90% (Figure 5.24) results are compared using bar charts for all evaluated designs.

Figure 5.22 – Number of total faults observed in each fault-tolerant ZynqNet System design.



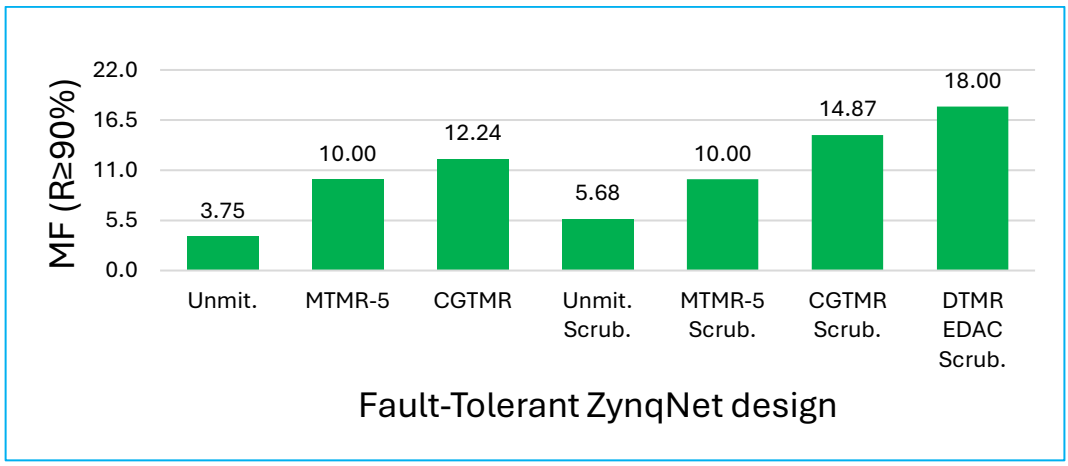
Source: The author.

Figure 5.23 – MFTF of each fault-tolerant ZynqNet System design.



Source: The author.

Figure 5.24 – MF (R≥90%) of each fault-tolerant ZynqNet System design.



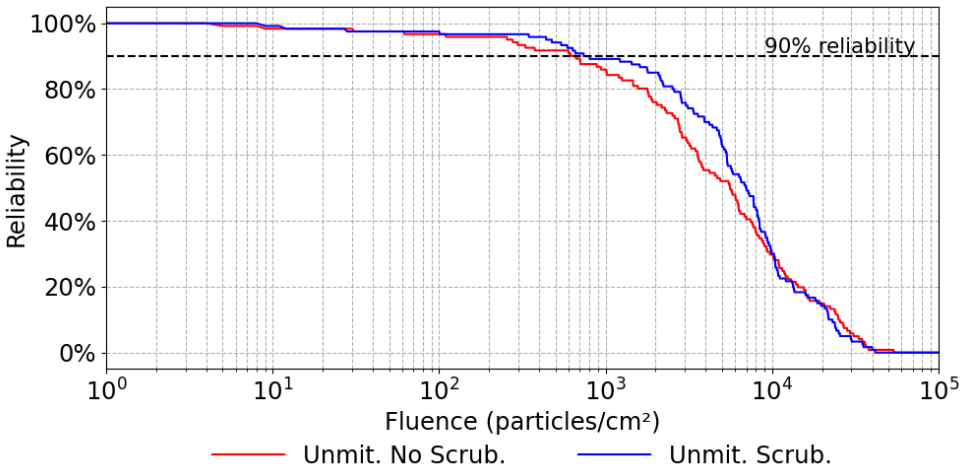
Source: The author.

5.2.2 Reliability testing via Radiation Experiment

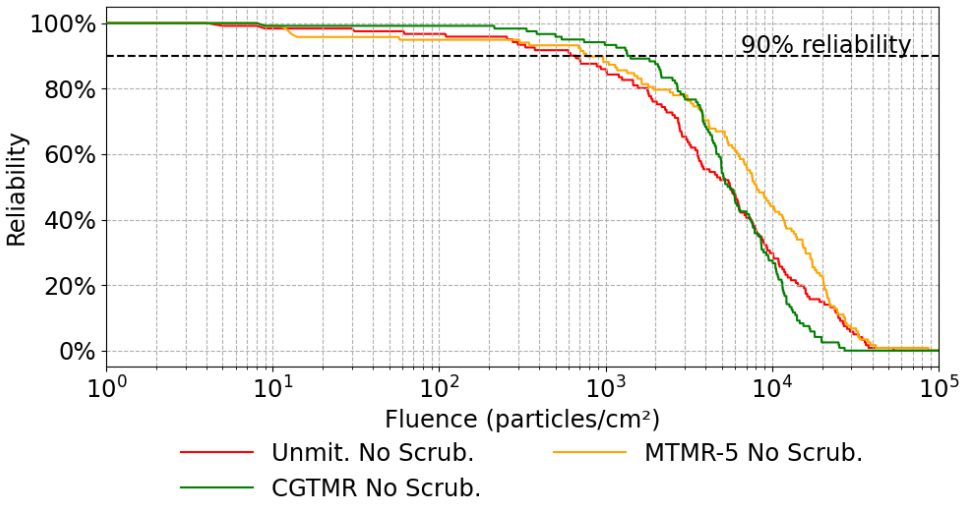
The radiation testing methodology based on heavy-ions, as detailed in Section 3.4.2, was used to test the unmitigated and fault-tolerant versions. As described at the end of Chapter 4, the caches in the PS are disabled in all evaluated designs.

The results are illustrated by the reliability curves in Figure 5.25 and summarized through reliability metrics in Table 5.5. The reliability curves in Figure 5.25 reveal a significative difference between unmitigated and fault-tolerant designs. While the unmitigated versions exhibit a rapid degradation in reliability as the fluence increases, designs incorporating redundancy and CRAM scrubbing show a slower reliability decay, indicating enhanced resilience to radiation-induced faults.

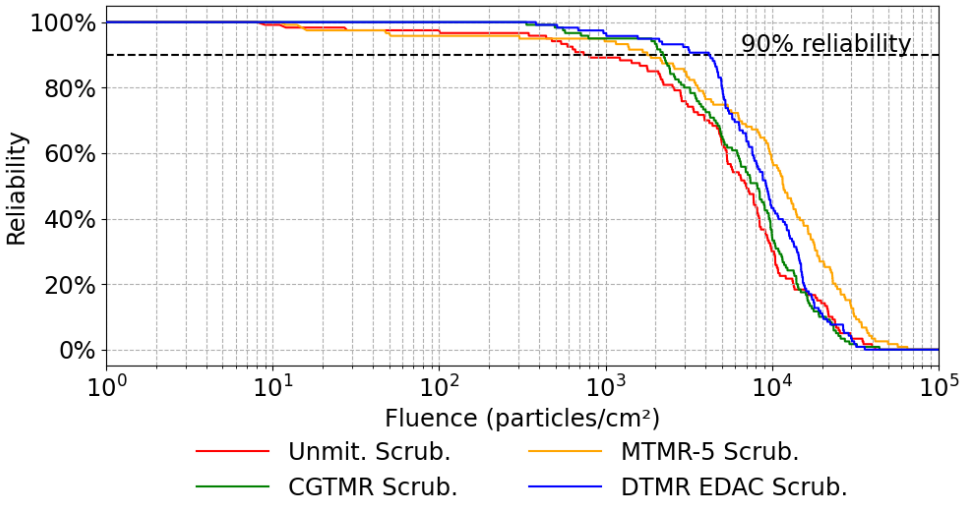
Figure 5.25 – Heavy-ions experiment reliability curves.



(a) Unmitigated designs



(b) CRAM Scrubbing disabled



(c) CRAM Scrubbing enabled

Source: The author.

Table 5.5 summarizes the heavy-ion radiation test results for the unmitigated and fault-tolerant ZynqNet designs.

Table 5.5 – Heavy-ions experiment results: fault-tolerant designs.

Designs	**Unmitigated**	**MTMR-5**	**CGTMR**	**Unmitigated Scrubbing**	**MTMR-5 Scrubbing**	**CGTMR Scrubbing**	**DTMR EDAC Scrubbing**
Inference error	69	59	71	65	61	75	49
Time-out	52	59	49	55	58	45	69
Total events	121	118	120	120	119	120	118
Total fluence [10^4 p/cm²]	109.69 ±8.08	141.70 ±9.34	87.07 ±5.44	111.38 ±7.66	177.18 ±11.41	113.41 ±6.81	130.18 ±6.97
MΦTF [10^4 p/cm²]	0.91 ±0.11	1.20 ±0.14	0.73 ±0.08	0.93 ±0.11	1.49 ±0.17	0.95 ±0.10	1.10 ±0.12
Dynamic cross section (σ) [10^{-4} cm²]	1.10 ±0.13	0.83 ±0.09	1.38 ±0.15	1.08 ±0.12	0.67 ±0.08	1.06 ±0.12	0.91 ±0.10
MΦ [p/cm²] (R≥90%)	662.1	765.8	1,412	801	1,870	2,239	4,175.8
MΦ (Norm.) (R≥90%)	1.00x	1.16x	2.13x	1.21x	2.82x	3.38x	6.31x

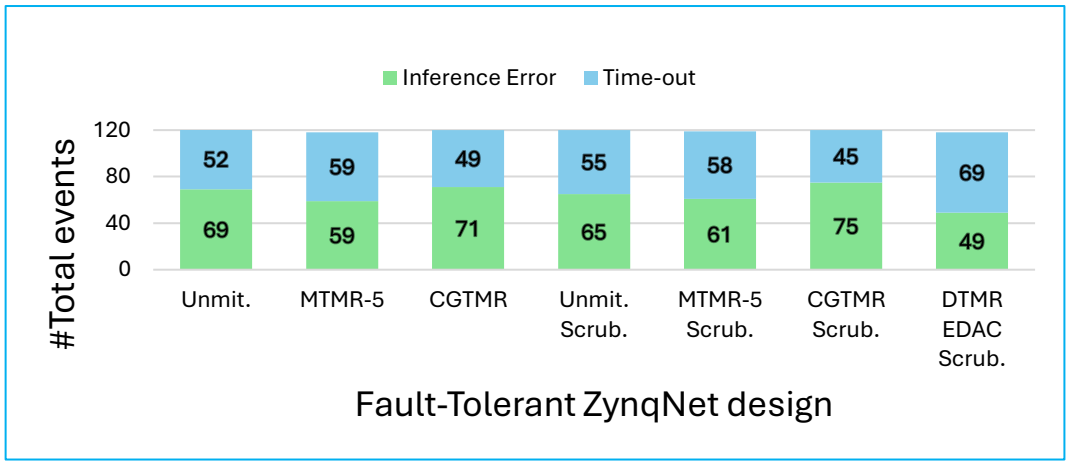
Source: The author.

Approximately 120 events were collected, resulting in both inference errors and time-out events. For each experiment, the total fluence (in p/cm^2) irradiated on the design was measured. The Mean Fluence To Failure (MΦTF) was calculated, along with the Dynamic Cross Section (σ), an inversely correlated metric that represents the probability of an energetic particle causing a functional failure in the implemented design, expressed in units of area (cm^2). Finally, the Mission Fluence (MΦ) corresponding to a reliability greater than 90% was obtained to enable a comparative analysis among all evaluated designs.

In most cases, the number of time-out events was lower than the number of inference errors. Among all evaluated architectures, the DTMR EDAC Scrubbing design achieved the highest improvement in the Mission Fluence (MΦ) metric for $R \geq 90\%$, reaching a 6.31× increase relative to the unmitigated baseline. These experimental results are consistent with those obtained from the emulated fault injection.

Figure 5.26 presents a stacked bar chart showing the distribution of event types (inference errors and time-out events) observed for each fault-tolerant ZynqNet system design.

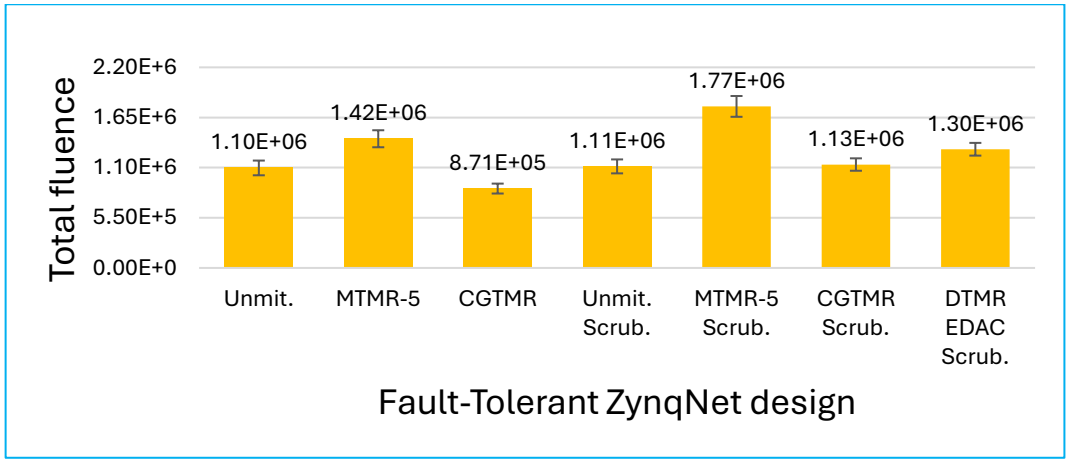
Figure 5.26 – Number of event types observed in each fault-tolerant ZynqNet System design.



Source: The author.

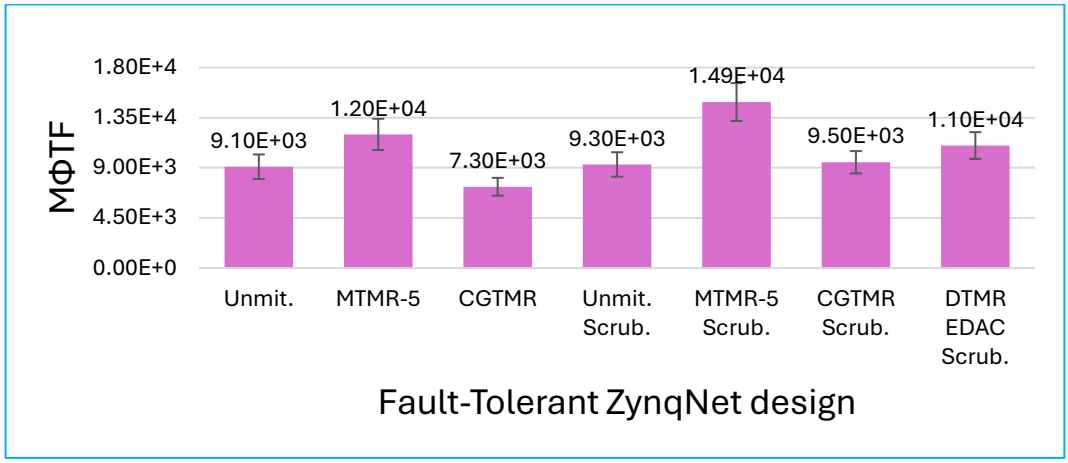
The total fluence (Figure 5.27), the Mean Fluence To Failure (MΦTF) (Figure 5.28), and the dynamic cross section (Figure 5.29) are presented as simple bar charts, including their associated experimental error margins.

Figure 5.27 – Total fluence of each fault-tolerant ZynqNet System design.



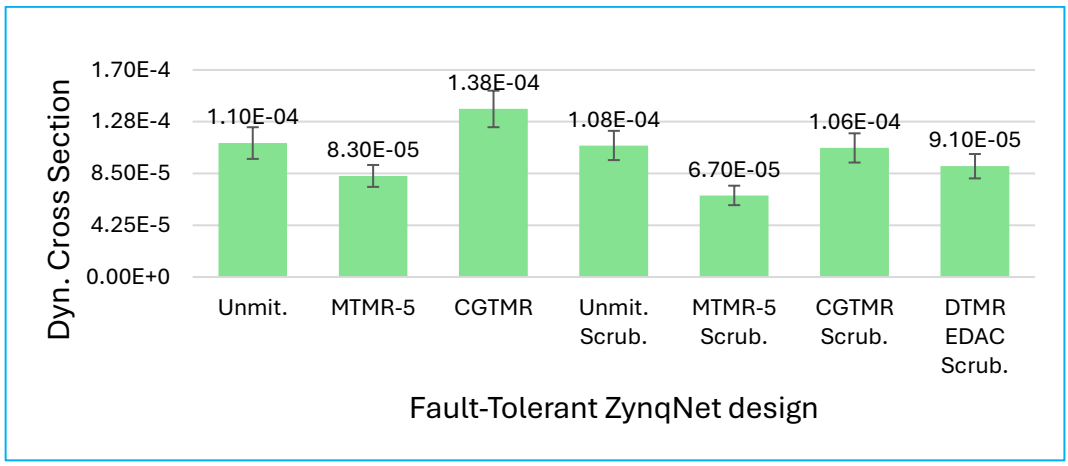
Source: The author.

Figure 5.28 – Mean fluence to failure (MΦTF) of each fault-tolerant ZynqNet System design.



Source: The author.

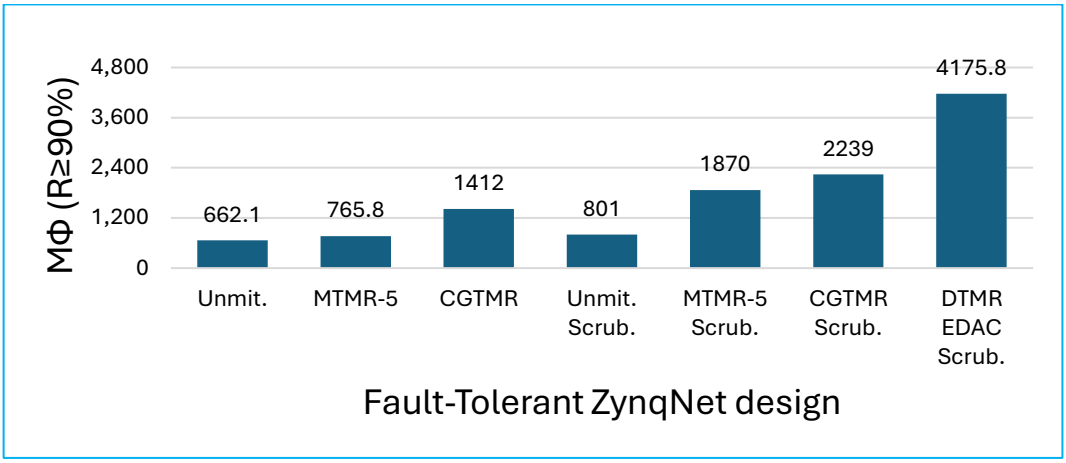
Figure 5.29 – Dynamic Cross Section of each fault-tolerant ZynqNet System design.



Source: The author.

Finally, Figure 5.30 presents a bar chart comparing the Mission Fluence for R ≥ 90% across all evaluated designs.

Figure 5.30 – MΦ (R≥90%) of each fault-tolerant ZynqNet System design.



Source: The author.

5.3 Discussion and results comparison with state-of-the-art

Table 5.6 compares state-of-the-art AI inference engines implemented on SRAM-based APSoCs with the results obtained in this work. The comparison includes the neural network (NN) topology, the adopted fault-tolerant (FT) strategy, the target APSoC device, the type of test (FI: fault injection, radiation experiment), and the corresponding area and reliability metrics. Relative values are reported as ratios with respect to each baseline case-study.

Most of the presented previous works rely on fault injection and neutron-based testing, while fewer studies report results obtained under heavy-ion irradiation. Since the dynamic cross-section is the most reported reliability metric across the compared works, it was included in the comparison table. Differences in dynamic cross-section values are influenced not only by the adopted FT technique, but also by the intrinsic sensitivity of the SRAM cells of each APSoC device, as well as by the particle type and energy used during testing.

The first five rows correspond to AI inference engines without implementing fault-tolerant mechanisms. These designs mostly exhibit lower area utilization due to the absence of hardware redundancy. It can be observed that the dynamic cross section values are lower when the type of particle used is neutron instead of heavy ion. This is because neutrons cause indirect ionization via low probability nuclear interactions, whereas heavy ions induce errors through continuous direct ionization with high energy deposition. The next seven rows include fault-tolerant approaches based on redundancy, error detection and correction, and configuration scrubbing, which result in higher area overhead. The same trend can be observed here, with dynamic cross section values for neutrons being lower than those for heavy ions.

In this context, the results obtained in this work are competitive with state-of-the-art. Specifically, the MTMR-5 Scrub. and DTMR EDAC Scrub. designs achieve dynamic cross-section slight reductions of 0.6x and 0.8x relative to the baseline, respectively. These findings validate the effectiveness of the proposed fault-tolerant strategies in mitigating SEUs in the APSoC while maintaining a trade-off between area and reliability.

Table 5.6 – Comparison of the state-of-the-art results with this work.

AI Inference Engine	**NN topology**	**FT strategy**	**APSoC device**	**Test**	**Area**				**Reliability**
					LUTs	**DFFs**	**DSPs**	**BRAMs**	**Dynamic Cross-Section**
FINN: Benevenuti et al. (2025)	4-CNN	-	Zynq-7020	FI, Heavy ion	8.6k	6.3k	-	8	0.12×10^{-3}
	4-CNN	-	Zynq-7020	FI, Heavy ion	27.4k	10.2k	-	9	0.13×10^{-3}
FINN: Souvatzoglou et al. (2024b)	4-FC	-	Zynq-7020	Neutron	11k	12.8k	-	27	0.20×10^{-9}
	3-CNN + 2-FC	-	Zynq-7045	Neutron	32.4k	39.3k	-	136	0.45×10^{-8}
ZynqNet: Wang et al. (2021)	Adapted SqueezeNet	-	Zynq-7020	FI, Neutron	7.4k (0.8x)	7.9k (0.7x)	14 (0.3x)	28.5 (0.5x)	0.28×10^{-6} (1.1x)
Libano et al. (2019)	2-FC	Selective TMR	Zynq-7020	FI. Neutron	25k (1.6x)	3.7k (1.1x)	56 (1.0x)	-	0.14×10^{-9} (0.2x)
	2-FC	CGTMR	Zynq-7020	FI. Neutron	43k (1.7x)	4.1k (1.2x)	168 (3.0x)	-	0.10×10^{-9} (0.2x)
	2-CNN + 2-FC	Selective TMR	Zynq Ultrascale+	FI, Neutron	206k (1.1x)	21k (1.0x)	-	-	0.23×10^{-6} (0.5x)
Systolic Array: Maffazioli et al. (2025)	3-CNN	CGTMR, EDAC, Scrub.	Zynq-7020	Heavy ion	20.2k (3.0x)	18.8k (3.0x)	207 (3.0x)	4.5 (3.0x)	0.8×10^{-4} (0.6x)
ZynqNet: This work	4-CNN	CGTMR, Scrub.	Zynq-7020	FI, Heavy ion	10.9k (3.0x)	13.1k (3.0x)	36 (3.0x)	40.5 (3.0x)	0.11×10^{-3} (0.9x)
	4-CNN	MTMR (MTMR-5), Scrub.	Zynq-7020	FI, Heavy ion	12.0k (3.4x)	12.8k (2.9x)	36 (3.0x)	40.5 (3.0x)	0.67×10^{-4} (0.6x)
	4-CNN	DTMR, EDAC, Scrub.	Zynq-7020	FI, Heavy ion	19.6k (5.5x)	15.3k (3.5x)	36 (3.0x)	40.5 (3.0x)	0.91×10^{-4} (0.8x)

Source: The author

5.4 Summary of results

The fault-tolerant strategies implemented in this chapter were evaluated in terms of their impact on the reliability of the ZynqNet accelerator, considering both emulated fault injection and heavy-ion radiation testing. The results demonstrate the effectiveness of applying hardware

redundancy, scrubbing mechanisms, and error correction to mitigate radiation-induced faults in SRAM-based APSoCs.

The initial fault injection experiments performed on the unmitigated ZynqNet revealed that the MACC and Memory Controller were the most vulnerable IFB to SEUs, as reflected by the lowest values of MFTF and MF in Table 5.2. These findings guided the progressive application of Modular TMR (MTMR), from MTMR-1 to MTMR-5, prioritizing the most critical IFB identified through emulation. Reliability curves in Figure 5.19 clearly show that increasing the number of hardened modules shifts the reliability curve to the right, confirming improved fault tolerance. Notably, MTMR-5 achieved the highest fault resilience among the modular approaches, especially when combined with CRAM scrubbing.

In contrast, the Coarse-Grained TMR (CGTMR) approach, featured by its simplicity and lower implementation overhead, showed limited improvements. This was expected due to its coarse protection granularity and the potential for internal module corruption, which cannot be masked when voters are placed only at the top level.

The most robust design was the DTMR EDAC Scrubbing implementation. It combines top-level redundancy with voters inserted not only at the final outputs but also at the outputs of each triplicated IFB, along with CRAM scrubbing and Hamming-based EDAC applied to the cache memories. This design delivered the best reliability in both experiments. As shown in Table 5.4, in the fault injection campaign, it achieved a 4.80x increase in MF ($R \geq 90\%$) over the unmitigated version. In the radiation experiment, presented in Table 5.5, it reached a 6.31× increase in Mission Fluence (MΦ), achieving the highest improvement factor among all designs. The higher gain observed in the radiation test compared to the emulation is due to the broader fault coverage of heavy-ion irradiation, which affects not only the configuration memory (as in fault injection), but also volatile elements such as flip-flops and BRAM. In this context, the use of Hamming-based EDAC for cache blocks proved essential, especially in mitigating SBUs in the BRAM read data, which are not addressed by CRAM scrubbing alone.

The dynamic cross-section (σ) values show a slight improvement across the evaluated designs compared to the baseline. Specifically, the MTMR-5 Scrub. and DTMR EDAC Scrub. designs achieve reductions of 0.6x and 0.8x, respectively. These results are competitive with the state-of-the-art, validating that the proposed fault-tolerant strategies effectively mitigate SEUs in the APSoC while maintaining a trade-off between area and reliability.

6 CONCLUSIONS

This thesis proposed a scalable and replicable methodology for designing an efficient, high-performance, and fault-tolerant AI accelerator on SRAM-based APSoCs. The case study for this research consisted of the ZynqNet AI inference accelerator running a four-layer CNN trained on the aerial images from the SAT-6 dataset. This work focused on addressing two key points: computational efficiency through optimized numerical representation, data packing, and communication architecture exploration, and reliability enhancements under radiation-induced SEUs through TMR-based techniques, CRAM scrubbing, and error correction and detection codes. By integrating design aspects such as area, power, performance and reliability dimensions, this research contributes to a comprehensive framework for developing AI accelerators suitable for aerospace and other radiation-prone environments.

The performance analysis demonstrated that 8-bit integer quantization provides the best trade-off between area and performance compared to floating-point or fixed-point implementation. Packing eight 8-bit values into 64-bit words in DDR memory and using small caches with 64-bit burst-mode for reading and writing, decreased memory requests and improved overall performance. Studying different communication architectures between the accelerator and the external DDR memory helped also to increase performance. However, DDR memory bandwidth posed a bottleneck, limiting the benefits of increasing the number of processing elements and preventing full utilization of the accelerator's computational capacity. Among the configurations evaluated, using two processing elements with an optimized communication architecture offered the most balanced design in terms of performance and resource utilization.

The reliability analysis highlighted that ZynqNet's IFBs differ in their susceptibility to radiation-induced faults. The MACC and Memory Controller were the most vulnerable, while other modules showed higher resilience. Applying Modular TMR effectively hardened the most critical modules, whereas Coarse-Grained TMR provided limited improvements due to its coarse protection granularity and the potential of top-level module corruption. These results emphasize the importance of module-level vulnerability analysis for the efficient allocation of fault-tolerance resources.

The most robust implementation achieved in this research was the DTMR EDAC Scrubbing design, combining distributed TMR with internal voters, configuration memory scrubbing, and Hamming-based error correction for cache memories. This approach achieved a 4.8× improvement in emulated fault injection campaigns and a 6.31× improvement in

radiation experiments compared to the unmitigated baseline, confirming the effectiveness of this fault-tolerant design.

Despite the achieved results, this work also presents inherent limitations that must be acknowledged. The reliance on external DDR memory introduces a communication bottleneck that restricts scalability when increasing the number of processing elements, preventing full exploitation of the available computational parallelism. Additionally, the use of redundancy-based fault-tolerance techniques inevitably increases area and power consumption, which constrains their applicability in extremely resource-limited platforms. However, compared to state-of-the-art AI inference engines reported in the literature, the results obtained in this work are competitive in terms of reliability improvements versus area overhead. Furthermore, within the context of SRAM-based APSoCs and safety-critical environments, both the proposed methodology and the evaluated architectures represent a significant contribution to the design of fault-tolerant systems. Specifically, the vulnerability-driven analysis enables an efficient allocation of fault-tolerance resources, as demonstrated by the MTMR configurations, while the DTMR EDAC Scrubbing design provides a highly robust solution when maximum reliability is required. This design space exploration shows that significant reliability improvements can be achieved either through modular hardening or through distributed redundancy, depending on application constraints, making the proposed approach suitable for aerospace AI inference applications.

Future research can expand this work by applying the methodology to other CNN models with higher number of layers or changing the case-study application. Storing the CNN parameters and image or feature pixels on the FPGA BRAM elements could potentially improve the performance due to their proximity to ZynqNet's processing elements. This approach avoids DDR memory bottlenecks and could allow better utilization of the accelerator's computational capacity. On the reliability side, finer-grained redundancy strategies such as Fine Grained Distributed TMR merit further investigation, along with the trade-offs between reliability, power consumption, and area. Implementing voters with a flag that indicates whether voting is active, triggered by the presence of different input values, enables tracking how far faults are masked. Additionally, extending radiation experiments by considering other types of heavy ions, varying the Linear Energy Transfer (LET), or changing the irradiated particle could provide deeper insights into the effectiveness of fault-tolerance strategies under diverse conditions for AI accelerators.

REFERENCES

AGGARWAL, C. **Neural Networks and Deep Learning**. NY, USA: Springer, 2018.

AGUIAR, V. A. P. et al. SAFIIRA: A Heavy-ion Multi-purpose Irradiation Facility in Brazil. **Rev. Scientific Instrum.**, v. 91, n. 5, Art. no. 053301, May 2020. Disponível em: <https://doi.org/10.1063/1.5138644>.

AGUIAR, Y. Q. de et al. **Single-Event Effects: From Space to Accelerator Environments**. [S.l.]: Springer, 2025. Disponível em: <https://doi.org/10.1007/978-3-031-71723-9>.

BASU, S. et al. DeepSat: A Learning framework for Satellite Imagery. In: 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Seattle Washington, USA: ACM, 2015. Disponível em: <https://doi.org/10.1145/2820783.2820816>.

BENEVENUTI, F. **Enhancements on Fault Injection for Xilinx 7 Series and Ultrascale+ SRAM-Based FPGAs.** Tese (Doutorado em Microeletrônica) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2022. Disponível em: <https://lume.ufrgs.br/handle/10183/258742>.

BENEVENUTI, F. et al. Investigating the reliability impacts of neutron-induced soft errors in aerial image classification CNNs implemented in a softcore SRAM-based FPGA GPU. **Microelectronics Reliability**, v. 138, p. 114738, 2022. Disponível em: <https://doi.org/10.1016/j.microrel.2022.114738>.

BENEVENUTI, F. et al. Neutron-induced Faults on CNN for Aerial Image Classification on SRAM-based FPGA Using Softcore GPU and HLS. In: 21st European Conference on Radiation and Its Effects on Components and Systems (RADECS). Vienna, Austria: IEEE, 2021. P. 57-60. Disponível em: <https://doi.org/10.1109/RADECS53308.2021.9954517>.

BENEVENUTI, F. et al. Reliability of FINN-Generated CNN Accelerators for Image Classification on SRAM-Based FPGAs Under Heavy-Ion-Induced Faults. **IEEE Transactions on Nuclear Science**, v. 72, n. 8, p. 2830-2838, Aug. 2025. Disponível em: <https://doi.org/10.1109/TNS.2025.3588691>.

BENEVENUTI, F. et al. Robust Convolutional Neural Networks in SRAM-based FPGAs: a case study in image classification. **Journal of Integrated Circuits and Systems**, v. 16, n. 2, Art. No. 10, 2021. Disponível em: <https://doi.org/10.29292/jics.v16i2.504>.

BERTOA, T. G. et al. Fault-Tolerant Neural Network Accelerators With Selective TMR. In: **IEEE Design & Test**, v. 40, n. 2, p. 67-74, Apr. 2023. Disponível em: <https://doi.org/10.1109/MDAT.2022.3174181>.

BLOTT, M. et al. FINN-R: An end-to-end deep-learning framework for fast exploration of Quantized Neural Networks. **ACM Transactions on Reconfigurable Technology Systems**, v. 11, n. 3, Art. No. 16, 2018. Disponível em: <https://doi.org/10.1145/3242897>.

BOUAAFIA, S.; et al. An FPGA-SoC based Hardware Acceleration of Convolutional Neural Networks. In: 2022 IEEE 9th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT). Hammamet, Tunisia: IEEE, 2022. p. 537-542. Disponível em: <https://doi.org/10.1109/SETIT54465.2022.9875430>

BRAGA, G. et al. Evaluating softcore GPU in SRAM-based FPGA under radiation-induced effects. **Microelectronics Reliability**, v. 126, p. 114348, 2021. Disponível em: <https://doi.org/10.1016/j.microrel.2021.114348>.

CHEN, Y.-H.; EMER, J. S. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. **IEEE Journal of Solid-State Circuits**, v. 52, n. 1, p. 127-138, 2017. Disponível em: <https://doi.org/10.1109/JSSC.2016.2616357>.

CHO, M.; KIM, Y. Implementation of Data-optimized FPGA-based Accelerator for Convolutional Neural Network. In: 2020 International Conference on Electronics, Information, and Communication (ICEIC). Barcelona, Spain: IEEE, 2020. P. 89-90. Disponível em: <https://doi.org/10.1109/ICEIC49074.2020.9050993>.

CHOLLET, F. **Deep Learning with Python**. NY: Manning, 2018.

D'ACREMONT, A.; et al. CNN-Based Target Recognition and Identification for Infrared Imaging in Defense Systems. **Sensors**, v. 19, n. 9, p. 2040, 30 Apr. 2019. Disponível em: <https://doi.org/10.3390/s19092040>.

DAI, D.; ZHANG, Y.; et al. Trainable fixed-point quantization for deep learning acceleration on FPGAs. 2024. Disponível em: <https://doi.org/10.48550/arXiv.2401.17544>.

DOĞRU, A.; et al. Using Convolutional Neural Networks to Automate Aircraft Maintenance Visual Inspection. **Aerospace**, v. 7, n. 12, p. 171, 7 Dec. 2020. Disponível em: <https://doi.org/10.3390/aerospace7120171>.

EL GHAZOUALI, S.; et al. FlightScope: An Experimental Comparative Review of Aircraft Detection Algorithms in Satellite Imagery. **Remote Sensing**, v. 16, n. 24, p. 4715, 17 Dec. 2024. Disponível em: <https://doi.org/10.3390/rs16244715>.

ELHEWEHY, M.; ABBASS, K. O.; NASR, O. A. Hardware-Software Co-Design Implementation of Fixed-Point GoogleNet on SoC Using Xilinx Vitis. In: 5th Novel Intelligent and Leading Emerging Sciences Conference. Giza, Egypt: IEEE, 2023. p. 274-278. Disponível em: <https://doi.org/10.1109/NILES59815.2023.10296801>.

ELY, A. F. et al. Image Classification CNNs using the FINN Engine for SRAM-based APSoC in Satellite Applications. In: 2024 37th SBC/SBMicro/IEEE Symposium on Integrated Circuits and Systems Design (SBCCI). João Pessoa, Brazil: IEEE, 2024. p. 175-179. Disponível em: <https://doi.org/10.1109/SBCCI62366.2024.10703997>.

FUKUSHIMA, K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. **Biological Cybernetics**, v. 36, p. 193-202. 1980.

GAMBARDELLA, G. et al. Efficient Error-Tolerant Quantized Neural Network Accelerators. In: 2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT). Noordwijk, Netherlands: IEEE, 2019. P. 131-136. Disponível em: <https://doi.org/10.1109/DFT.2019.8875314>.

GONCALVES, M. M. et al. Investigating floating-point implementations in a softcore GPU under radiation-induced faults. In: 2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS). Glasgow, UK: IEEE, 2020. p. 498-501. Disponível em: <https://doi.org/10.1109/ICECS49266.2020.9294939>.

GSCHWEND, D. **ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network.** Dissertação (Mestrado) – Department of Information Technology and Electrical Engineering, ETH Zurich, Zurich, 2016. Disponível em: <https://arxiv.org/abs/2005.06892>.

GYSEL, P. et al. Ristretto: A Framework for Empirical Study of Resource-Efficient Inference in Convolutional Neural Networks. **IEEE Transactions on Neural Networks and Learning Systems**, v. 29, n. 11, p. 5784-5789, 2018. Disponível em: <https://doi.org/10.1109/TNNLS.2018.2808319>.

HAGAN, M. T.; et al. **Neural Network Design**. Stillwater, Oklahoma: Martin Hagan, 2014. ISBN 978-0-9717321-1-7.

HOWARD, A. G.; et al. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. Google Inc., Mountain View, CA, 2017. Disponível em: <https://arxiv.org/abs/1704.04861>

HU, X.; et al. Sinet: A scale-insensitive convolutional neural network for fast vehicle detection. **IEEE Transactions on Intelligent Transportation Systems**, v. 20, n. 3, p. 1010-1019, Mar. 2019. Disponível em: <https://doi.org/10.1109/TITS.2018.2838132>.

HUBEL, D. et al. Receptive fields of single neurones in the cat's striate cortex. **The Journal of Physiology**, p. 574-591. 1959. Disponível em: <https://doi.org/10.1113/jphysiol.1959.sp006308>.

INTEL. **High Level Synthesis Compiler Pro Edition User Guide (UG-20037).** 2023. Disponível em: <https://www.intel.com/content/www/us/en/docs/programmable/683456/23-2>

JIA, Y. et al. Caffe: Convolutional Architecture for Fast Feature Embedding. In: 22nd ACM International Conference on Multimedia, Orlando, Florida, USA: ACM, 2014, p. 675-678. Disponível em: <https://doi.org/10.1145/2647868.2654889>.

KIM, W.; et al. Fast Prototyping of a Deep Neural Network on an FPGA. In: 2020 International SoC Design Conference (ISOCC). Seoul, South Korea: IEEE, 2021. p. 214-215. Disponível em: <https://doi.org/10.1109/ISOCC50952.2020.9333030>.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. ImageNet classification with deep convolutional neural networks. **Communications of the ACM**, v. 60, n. 6, p. 84-90, 2017. Disponível em: <https://doi.org/10.1145/3065386>.

KUNG, H. T.; LEISERSON, C. E. Systolic arrays for VLSI. Carnegie Mellon University, 1978. Disponível em: <https://www.eecs.harvard.edu/htk/static/files/1978-cmu-cs-report-kung-leiserson.pdf>.

LECUN, Y. et al. Gradient-based learning applied to document recognition. IEEE: 1998. V. 86, n. 11, p. 2278-2324. Disponível em: <https://doi.org/10.1109/5.726791>.

LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. **Nature**, v. 521, p. 436-444, 2015. Disponível em: <https://doi.org/10.1038/nature14539>.

LIBANO, F. et al. Selective Hardening for Neural Networks in FPGAs. **IEEE Transactions on Nuclear Science**, v. 66, n. 1, p. 216-222, 2019. Disponível em: <https://doi.org/10.1109/TNS.2018.2884460>.

LIN, M. et al. Network In Network. Neural and Evolutionary Computing, Computer Science, Cornell University. 2014. Disponível em: <https://doi.org/10.48550/arXiv.1312.4400>.

LIU, Z. et al. Recent advances on reliability of FPGAs in a radiation environment. **Microelectronics Journal**, v. 148, p. 106176, 2024. Disponível em: <https://doi.org/10.1016/j.mejo.2024.106176>.

MAFFAZIOLI, U. et al. Radiation Heavy-Ion Fault Analysis of URSA: a Reconfigurable Systolic Array for CNN Acceleration in SRAM-based APSoC. In: 2025 25th European Conference on Radiation and Its Effects on Components and Systems (RADECS), Antwerp, Belgium, 2025. Accepted abstract PE-6L, in press.

MARAÑON, Eduardo; BENEVENUTI, Fabio; KASTENSMIDT, Fernanda Lima. Exploring parallelism in ZynqNet AI engine to accelerate image classification in aerospace. In: 2025 IEEE 16th Latin American Symposium on Circuits and Systems (LASCAS), Bento Gonçalves, Brazil: IEEE, Feb. 2025, v. 1, p. 239-243. Disponível em: <https://doi.org/10.1109/LASCAS64004.2025.10966288>.

MICROCHIP. **SmartHLS Software User Guide.** 2023. Disponível em: <https://microchiptech.github.io/fpga-hls-docs/2023.2>

MUKHERJEE, S. S. et al. Measuring architectural vulnerability factors. **IEEE Micro**, v. 23, n. 6, p. 70-75, Nov.-Dec. 2003. Disponível em: <https://doi.org/10.1109/MM.2003.1261389>.

NIELSEN, M. **Neural Networks and Deep Learning.** 2019. Disponível em: <http://neuralnetworksanddeeplearning.com>.

PERINA, A. et al. FPGA High-Level Synthesis: Good Practices for Quality and Productivity. Moodle USP. 2021. Disponível em: <https://acesse.dev/oLoSo>.

RAY P. Convolutional Neural Network (CNN) and its Application. Analytics Vidhya, Medium. 2021. Disponível em: <http://surl.li/nzagow>.

SHARMA, P. Applications of Convolutional Neural Networks (CNN). Analytics Vidhya. 2023. Disponível em: <https://www.analyticsvidhya.com/blog/2021/10/applications-of-convolutional-neural-networkscnn>.

SOUVATZOGLOU, I. et al. Tradeoff Between Performance and Reliability in FPGA Accelerated DNNs for Space Applications. In: European Data Handling & Data Processing Conference (EDHPC), Juan Les Pins, France: IEEE, 2023, p. 390-396. Disponível em: <https://doi.org/10.23919/EDHPC59100.2023.10396101>.

SOUVATZOGLOU, I. et al. Analyzing the Single Event Upset Vulnerability of Binarized Neural Networks on SRAM FPGAs. In: 2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Athens, Greece: IEEE, 2021. P. 1-6. Disponível em: <https://doi.org/10.1109/DFT52944.2021.9568280>.

SOUVATZOGLOU, I. et al. Assessing the Reliability of FPGA-Based Quantized Neural Networks Under Neutron Irradiation. **IEEE Transactions on Nuclear Science**, v. 71, n. 12, p. 2565-2577, 2024. Disponível em: <https://doi.org/10.1109/TNS.2024.3491503>.

STERPONE, L.; VIOLANTE, M. Analysis of the robustness of the TMR architecture in SRAM-based FPGAs. **IEEE Transactions on Nuclear Science**, v. 52, n. 5, p. 1545-1550, 2005. Disponível em: <https://doi.org/10.1109/TNS.2005.856543>.

SUN, S. et al. Design of Efficient CNN Accelerator Based on Zynq Platform. In: 2020 15th International Conference on Computer Science & Education (ICCSE), Delft, Netherlands: IEEE: 2020, v. 64, p. 489-493. Disponível em: <https://doi.org/10.1109/ICCSE49874.2020.9201800>.

SZEGEDY, C.; et al. Going deeper with convolutions. In: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, USA: IEEE, 2015. P. 1-9. Disponível em: <https://doi.org/10.1109/CVPR.2015.7298594>.

TAMBARA, L. A. **Analyzing the Impact of Radiation-induced Failures in All Programmable System-on-Chip Devices.** Tese (Doutorado em Microeletrônica) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2017. Disponível em: <https://lume.ufrgs.br/handle/10183/164461>.

TAMBARA, L. A. et al. Analyzing the impact of radiation-induced failures in programmable SoCs. **IEEE Transactions on Nuclear Science**, v. 63, n. 4, p. 2217-2224, Aug. 2016. Disponível em: <https://doi.org/10.1109/TNS.2016.2522508>.

UMUROGLU, Y. et al. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In: 2017 ACM/SIGDA International Symposium on Field-Programmable Gate

Arrays, Monterey, CA, USA: IEEE, 2017. P. 65-74. Disponível em: <https://doi.org/10.1145/3020078.3021744>.

WANG, H.-B. et al. Impact of Single-Event Upsets on Convolutional Neural Networks in Xilinx Zynq FPGAs. **IEEE Transactions on Nuclear Science**, v. 68, n. 4, p. 394-401, Apr. 2021. Disponível em: <https://doi.org/10.1109/TNS.2021.3062014>.

WEI, X. et al. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In: 54th ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA: IEEE, 2017. P. 165-170. Disponível em: <https:/doi.org/ 10.1145/3061639.3062207>.

XILINX. **Versal Adaptive SoC Technical Reference Manual (AM011)**. Technical Information Portal, AMD. 2025. Disponível em: <https://docs.amd.com/r/en-US/am011-versal-acap-trm/Introduction>.

XILINX. **Vitis High-Level Synthesis User Guide (UG1399).** Technical Information Portal, AMD. 2023. Disponível em: < https://docs.amd.com/r/2023.2-English/ug1399-vitis-hls>

XILINX. **Vivado Design Suite User Guide High-Level Synthesis (UG902).** Technical Information Portal, AMD. 2018. Disponível em: <https://docs.amd.com/v/u/2018.3-English/ug902-vivado-high-level-synthesis>.

XILINX. **Zynq UltraScale+ Device Technical Reference Manual (UG1085)**. Technical Information Portal, AMD. 2025. Disponível em: <https://docs.amd.com/v/u/en-US/ug1085-zynq-ultrascale-trm>.

XILINX. **Zynq-7000 SoC Data Sheet: Overview (DS190).** Technical Information Portal, AMD. 2018. Disponível em: <https://docs.amd.com/v/u/en-US/ds190-Zynq-7000-Overview>.

XILINX. **Zynq-7000 SoC Technical Reference Manual (UG585).** Technical Information Portal, AMD. 2023. Disponível em: <https://docs.amd.com/r/en-US/ug585-zynq-7000-SoC-TRM>.

XIONG, X. et al. Soft error sensitivity analysis based on 40 nm SRAM-based FPGA. **Electronics**, v. 11, n. 23, 2022. Disponível em: <https://www.mdpi.com/2079-9292/11/23/3844>.

ZHANG, C.; et al. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In: ACM/SIGDA International Symposium on Field Programmable Gate Arrays

(FPGA), Monterey California, USA: ACM, 2015. p. 161-170. Disponível em: <https://dl.acm.org/doi/10.1145/2684746.2689060>.

ZIEGLER, J. F.; ZIEGLER, M. D.; BIERSACK, J. P. SRIM – The stopping and range of ions in matter (2010). **Nucl. Instr. and Methods in Physics Res. Section B**, v. 268, n. 11, p. 1818-1823, 2010. Disponível em: <https://doi.org/10.1016/j.nimb.2010.02.091>.

APPENDIX A — PUBLICATIONS

First author:

1. **Hardening a RISC-V Softcore for Embedded Aerospace Applications in SRAM-based FPGA.** E. M. Aguilar, F. Benevenuti and F. L. Kastensmidt. 37th SBC/SBMicro/IEEE Symposium on Integrated Circuits and Systems Design (SBCCI), Joao Pessoa, Brazil: IEEE, 2024. P. 135-139, doi: 10.1109/SBCCI62366.2024.10703996.
2. **Impact of Partial TMR on RISC-V Processor Reliability.** E. Marañon, F. Benevenuti, F. Kastensmidt, N. Medina, N. Added, V. Aguiar and M. Guazzelli. IEEE Design & Test, 2025, doi: 10.1109/MDAT.2025.3608640.
3. **Exploring Parallelism in ZynqNet AI Engine to Accelerate Image Classification in Aerospace.** E. M. Aguilar, F. Benevenuti and F. L. Kastensmidt. IEEE 16th Latin America Symposium on Circuits and Systems (LASCAS), Bento Gonçalves, Brazil: IEEE, 2025. P. 239-243, doi: 10.1109/LASCAS64004.2025.10966288.
4. **Reliability Assessment of Fault-Tolerant ZynqNet AI Accelerators in SRAM-Based APSoC.** E. Marañon, F. Benevenuti, N. H. Medina, N. Added, V. A. P. Aguiar, S. G. P. N. Alberton, J. V. Schervenin, M. S. Pereira, F. L. Kastensmidt. European Conference on Radiation and Its Effects on Components and Systems (RADECS), Antwerp, Belgium, 2025. (Accepted and to be published).
5. **Reliable AI Accelerator Design Using Multiple Fault-Tolerance Techniques on SRAM-based APSoCs.** E. Marañon, F. Benevenuti, N. H. Medina, N. Added, V. A. P. Aguiar, S. G. P. N. Alberton, J. V. Schervenin, M. S. Pereira, F. L. Kastensmidt. IEEE Transactions on Nuclear Science, 2026. (Submitted).

Co-author:

1. **Power-Performance Analysis of a Softcore RISC-V SoC with CNN Accelerators for In-Orbit Computing.** V. K. Veiga, F. Benevenuti, U. Maffazioli, A. F. Ely, E. Marañon, A. C. S. Beck, J. R. Azambuja, F. L. Kastensmidt. 2026 IEEE 17th Latin America Symposium on Circuits and Systems (LASCAS), Arequipa, Peru: IEEE, 2026. (Submitted).

APPENDIX B — RESUMO EXPANDIDO EM PORTUGUÊS

Projetando e Testando Aceleradores de IA Baseados em APSoC Tolerantes a Falhas para Aplicações Aeroespaciais

A classificação de imagens e a detecção de objetos em tempo real são capacidades essenciais em aplicações críticas, como veículos autônomos, sistemas de defesa, plataformas aeroespaciais e sensoriamento remoto por satélite. Métodos de inteligência artificial (IA), especialmente aqueles baseados em redes neurais convolucionais (CNN), têm se mostrado eficazes nessas tarefas. No entanto, a implementação desses modelos em sistemas embarcados é limitada por restrições de memória, capacidade computacional e consumo de energia.

Os sistemas em chip programáveis (APSoC) baseados em arranjos de portas programáveis em campo (FPGA) permitem a implementação eficiente de aceleradores de inferência, oferecendo paralelismo, baixa latência e uso eficiente de memória interna. Entre esses dispositivos, APSoCs baseados em memória de acesso aleatório estática (SRAM) destacam-se pelo desempenho e flexibilidade, tornando-se atrativos para aplicações de alto desempenho.

Apesar das vantagens arquiteturais, APSoCs baseados em SRAM apresentam elevada suscetibilidade a falhas induzidas por radiação, como eventos de inversão de bits únicos (SBU) e múltiplos (MBU), que podem comprometer a integridade funcional do sistema. Embora técnicas de tolerância a falhas, como redundância modular tripla (TMR), detecção e correção de erros (EDAC) e scrubbing da memória de configuração (CRAM), sejam amplamente utilizadas em projetos com FPGA, sua aplicação específica em aceleradores de inteligência artificial ainda é pouco explorada.

O objetivo principal deste trabalho é projetar, otimizar e validar um fluxo de implementação de aceleradores de inteligência artificial em APSoCs baseados em SRAM, combinando técnicas de otimização de desempenho e mecanismos de tolerância a falhas para permitir sua aplicação confiável em ambientes sujeitos à radiação.

Como objetivos específicos, destacam-se:

- Implementar e otimizar o motor de inferência ZynqNet em um APSoC Zynq-7000, aplicando técnicas de quantização, paralelismo e empacotamento de dados.
- Integrar e comparar diferentes esquemas de tolerância a falhas, incluindo TMR, mecanismos de EDAC e scrubbing da CRAM.

- Desenvolver uma metodologia de injeção de falhas para avaliar a confiabilidade dos aceleradores e identificar os componentes mais suscetíveis.
- Validar os projetos por meio de testes com radiação de íons pesados e analisar a relação entre confiabilidade, utilização de recursos e consumo de energia.

As principais contribuições deste trabalho são:

- Proposição e validação de uma metodologia integrada para o projeto de aceleradores de IA tolerantes a falhas em APSoCs baseados em SRAM, combinando otimização de desempenho e aumento de confiabilidade.
- Avaliação da melhoria da eficiência computacional no acelerador de IA por meio de técnicas de quantização, paralelismo e otimizações de transferência de dados.
- Integração e avaliação de múltiplas estratégias de tolerância a falhas para mitigar efeitos de falhas induzidas por radiação em ambientes críticos, incluindo mecanismos de redundância, correção de erros e scrubbing da CRAM.
- Validação experimental dos projetos por meio de injeção de falhas emulada e testes com radiação de íons pesados.

Devido à flexibilidade, eficiência e baixo consumo de energia dos FPGAs baseados em SRAM, aliadas ao avanço contínuo da inteligência artificial (IA), diversos trabalhos investigaram a implementação de redes neurais convolucionais (CNN) nessas plataformas. Esses estudos concentram-se principalmente em otimizações de desempenho, redução de área e consumo energético, explorando diferentes arquiteturas de aceleradores e estratégias de paralelismo.

Cho et al. (2020) e Kim et al. (2021) propuseram aceleradores baseados em síntese em alto nível (HLS), utilizando representações em ponto fixo e diretivas de otimização, como unrolling e pipelining, obtendo reduções significativas de latência e melhorias de desempenho. Outros trabalhos, como Ely et al. (2024) e Benevenuti et al. (2025), utilizaram o framework FINN para avaliar o impacto da quantização, topologia da rede e paralelismo arquitetural sobre métricas de área, desempenho e confiabilidade.

No contexto de confiabilidade, vários estudos analisaram a resiliência de aceleradores de IA em FPGA frente a falhas induzidas por radiação, empregando tanto injeção de falhas quanto experimentos com nêutrons e íons pesados. Trabalhos como Souvatzoglou et al. (2024a, 2024b) e Benevenuti et al. (2022b) investigaram o efeito da quantização, paralelismo e tipo de armazenamento sobre métricas de confiabilidade, enquanto Libano et al. (2019) e Bertoa et al.

(2023) propuseram estratégias seletivas de redundância modular tripla (TMR) para reduzir erros críticos com menor sobrecarga de área.

Estudos específicos com o acelerador ZynqNet, como Wang et al. (2021) e Benevenuti et al. (2021b), avaliaram versões padrão, quantizadas e tolerantes a falhas, incorporando técnicas como EDAC, redundância de sinais e scrubbing da CRAM, com avaliações baseadas em injeção de falhas e testes com radiação.

Em relação ao estado da arte, este trabalho diferencia-se por combinar otimizações de desempenho com múltiplas técnicas de tolerância a falhas em um acelerador de inferência de IA, adotado como estudo de caso, implementado em FPGA baseado em SRAM. Enquanto muitos trabalhos avaliam isoladamente quantização, paralelismo ou confiabilidade, este estudo integra estratégias como scrubbing da CRAM, EDAC e diferentes configurações de redundância modular, incluindo abordagens de granularidade grossa (CGTMR), modular (MTMR) e distribuída (DTMR). Além disso, a comparação experimental com trabalhos anteriores é realizada por meio de métricas de área e confiabilidade, com destaque para a seção de choque dinâmica, amplamente utilizada na literatura. Os resultados obtidos demonstram que as configurações MTMR-5 com scrubbing e DTMR com EDAC e scrubbing alcançam reduções da seção de choque dinâmica de 0,6× e 0,8× em relação ao caso base, respectivamente, mantendo um compromisso equilibrado entre aumento de área e melhoria de confiabilidade. Dessa forma, os resultados apresentados posicionam este trabalho como competitivo em relação ao estado da arte, contribuindo com uma metodologia de projeto e avaliação para aceleradores de IA tolerantes a falhas em FPGAs baseados em SRAM.

Este trabalho demonstrou que é possível projetar aceleradores de inferência de inteligência artificial em APSoCs baseados em SRAM conciliando eficiência computacional e elevada confiabilidade frente a falhas induzidas por radiação. A análise de desempenho evidenciou que a quantização inteira de 8 bits oferece o melhor compromisso entre área e desempenho, enquanto estratégias de empacotamento de dados e arquiteturas de comunicação otimizadas reduzem acessos à memória externa e aumentam a eficiência do acelerador. Ainda assim, a largura de banda da memória DDR mostrou-se um fator limitante para a escalabilidade, restringindo os ganhos obtidos com o aumento do paralelismo.

Do ponto de vista da confiabilidade, os resultados indicaram que diferentes módulos do acelerador apresentam níveis distintos de vulnerabilidade a eventos de inversão de bits. A identificação dos blocos mais sensíveis permitiu a aplicação direcionada de técnicas de tolerância a falhas, evidenciando que abordagens baseadas em redundância modular distribuída

são mais eficazes do que soluções de granularidade grossa quando se busca maximizar a relação entre confiabilidade e sobrecarga de recursos.

Entre as arquiteturas avaliadas, a configuração DTMR com EDAC e scrubbing da memória de configuração apresentou o maior nível de robustez, alcançando melhorias tanto em campanhas de injeção de falhas quanto em experimentos com radiação. Esses resultados confirmam a efetividade da combinação de redundância distribuída, correção de erros e scrubbing para mitigar eventos de falha em ambientes críticos.