Adaptive Intelligent Systems for Extreme Environments

Yufan Lu

A thesis submitted for the degree of Doctor of Philosophy in Computing and Electronic Systems

School of Computer Science and Electronic Engineering

University of Essex

April 6, 2023

Abstract

As embedded processors become powerful, a growing number of embedded systems equipped with artificial intelligence (AI) algorithms have been used in radiation environments to perform routine tasks to reduce radiation risk for human workers. On the one hand, because of the low price, commercial-off-the-shelf devices and components are becoming increasingly popular to make such tasks more affordable. Meanwhile, it also presents new challenges to improve radiation tolerance, the capability to conduct multiple AI tasks and deliver the power efficiency of the embedded systems in harsh environments. There are three aspects of research work that have been completed in this thesis: 1) a fast simulation method for analysis of single event effect (SEE) in integrated circuits, 2) a self-refresh scheme to detect and correct bit-flips in random access memory (RAM), and 3) a hardware AI system with dynamic hardware accelerators and AI models for increasing flexibility and efficiency.

The variances of the physical parameters in practical implementation, such as the nature of the particle, linear energy transfer and circuit characteristics, may have a large impact on the final simulation accuracy, which will significantly increase the complexity and cost in the workflow of the transistor level simulation for large-scale circuits. It makes it difficult to conduct SEE simulations for large-scale circuits. Therefore, in the first research work, a new SEE simulation scheme is proposed, to offer a fast and cost-efficient method to evaluate and compare the performance of large-scale circuits which subject to the effects of radiation particles. The advantages of transistor and hardware description language (HDL)

simulations are combined here to produce accurate SEE digital error models for rapid error analysis in large-scale circuits. Under the proposed scheme, time-consuming back-end steps are skipped. The SEE analysis for large-scale circuits can be completed in just few hours.

In high-radiation environments, bit-flips in RAMs can not only occur but may also be accumulated. However, the typical error mitigation methods can not handle high error rates with low hardware costs. In the second work, an adaptive scheme combined with correcting codes and refreshing techniques is proposed, to correct errors and mitigate error accumulation in extreme radiation environments. This scheme is proposed to continuously refresh the data in RAMs so that errors can not be accumulated. Furthermore, because the proposed design can share the same ports with the user module without changing the timing sequence, it thus can be easily applied to the system where the hardware modules are designed with fixed reading and writing latency.

It is a challenge to implement intelligent systems with constrained hardware resources. In the third work, an adaptive hardware resource management system for multiple AI tasks in harsh environments was designed. Inspired by the "refreshing" concept in the second work, we utilise a key feature of FPGAs, partial reconfiguration, to improve the reliability and efficiency of the AI system. More importantly, this feature provides the capability to manage the hardware resources for deep learning acceleration. In the proposed design, the on-chip hardware resources are dynamically managed to improve the flexibility, performance and power efficiency of deep learning inference systems. The deep learning units provided by Xilinx are used to perform multiple AI tasks simultaneously, and the experiments show significant improvements in power efficiency for a wide range of scenarios with different workloads. To further improve the performance of the system, the concept of reconfiguration was further extended. As a result, an adaptive DL software framework was designed. This framework can provide a significant level of adaptability support for various deep learning algorithms on an FPGA-based edge

computing platform. To meet the specific accuracy and latency requirements derived from the running applications and operating environments, the platform may dynamically update hardware and software (e.g., processing pipelines) to achieve better cost, power, and processing efficiency compared to the static system.

List of publications

- Y. Lu, X. Zhai, S. Saha, S. Ehsan, and K. McDonald-Maier. A self-scrubbing scheme for embedded systems in radiation environments. *In 2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–4. IEEE, 2020.

- Y. Lu, X. Zhai, S. Saha, S. Ehsan, and K. D. McDonald-Maier. Fpga based adaptive hardware acceleration for multiple deep learning tasks. *In 2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 204–209. IEEE, 2021.

- Y. Lu, X. Chen, X. Zhai, S. Saha, S. Ehsan, J. Su, and K. McDonald-Maier. A fast simulation method for analysis of SEE in vlsi. *Microelectronics Reliability*, 120:114110, 2021.

- Y. Lu, X. Zhai, S. Saha, S. Ehsan, J.Su and K. McDonald-Maier. A simulation and evaluation scheme for Single Event Effects in VLSI. *In 2021 8th International Workshop on Analogue and Mixed-Signal Integrated Circuits for Space Applications (AMICSA)*, ESA, 2021.

- Y. Lu, X. Zhai, S. Saha, S. Ehsan, and K. D. McDonald-Maier. A self-adaptive SEU mitigation scheme for embedded systems in extreme radiation environments. *IEEE Systems Journal*, 16(1):1436–1447, 2022.

- C. Gao, S. Saha, Y. Lu, R. Saha, K. McDonald-Maier and X. Zhai. Deep Learning on FPGAs with Multiple Service Levels for Edge Computing. *2022*

27th International Conference on Automation and Computing (ICAC), IEEE, 2022.

- Lu, Y., Gao, C., Saha, R., Saha, S., McDonald-Maier, K. D., & Zhai, X. (2022). FPGA-Based Dynamic Deep Learning Acceleration for Real-Time Video Analytics. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 13642 LNCS, 68–82. https://doi.org/10.1007/978-3-031-21867-5_5

Prizes and awards

- The prize of Xilinx University Program (XUP), Xilinx Adaptive Computing Challenge 2021 [1], *All-in-one Self-adaptive Computing Platform for Smart City*[2]. **The design is based on the third part of the work in this thesis.**

Acknowledgments

I would like to thank my supervisors – Professor Klaus D. McDonald-Maier and Dr. Xiaojun Zhai, for their invaluable supervision, support and tutelage during the course of my PhD degree. My gratitude extends to the China Scholarship Council for the funding opportunity to undertake my studies at the University of Essex. Additionally, I would like to express gratitude to Dr Sangeet Saha for his treasured help in experiments and paper writing. I also thank Ying Li for her support during Xilinx Adaptive Computing Challenge 2021. I want to thank Dr Chris Frost for his help in neutron radiation experiments. I would like to thank my friends, lab mates, and colleagues - Dr. Shichao Yu, Dr. Junpeng Shi, Dr. Zuyuan Zhu, Dr. Bingxin Cai, Dr Lili Yan, Kai Zhang, Shengnan Li, Tao Chen, Phil George, Yucheng Wang and Zeba Khanam for a cherished time spent together during the pandemic. My appreciation also goes out to my family for their encouragement and support throughout my studies.

Contents

List of Figures

List of Tables

Abbreviations

AI artificial intelligence.

ASICs application-specific integrated circuits.

AXIs Advanced eXtensible Interfaces.

BCH Bose–Chaudhuri–Hocquenghem.

BRAM block random access memory.

CAD computer-aided design.

CMOS complementary MOS.

CRs cosmic rays.

DD displacement damage.

DFX Dynamic Function eXchange.

DL deep learning.

DNNs deep neural networks.

DPR Dynamic reconfiguration.

DPU Deep-Learning Processor Unit.

DRAM dynamic random-access memory.

DSP digital signal processor.

DUT devices under test.

ECC error correction code.

ECUs electronic control units.

FPGA field-programmable gate array.

FPS frame rate per second.

FSM finite-state machine.

HDL hardware description language.

IC integrated circuit.

ICAP internal configuration access port.

IoT Internet of Things.

LUT lookup table.

MFLOPs million floating-point operations per second.

MOS metal-oxide-semiconductor.

MOSFET metal–oxide–semiconductor field-effect transistor.

NAND NOT-AND.

NLP natural language processing.

NMOS n-channel MOS.

OTP one-time programmable.

PIPB propagation-induced pulse broadening.

PL programmable logic.

PLL Phase-locked loop.

PMOS p-channel MOS.

PR partial configuration.

PROM programmable read-only memory.

PS processing system.

RAM random access memory.

RS Reed-Solomon.

RTL register transfer level.

SEC-DED single-error correctable and double-error detectable.

SEE single event effect.

SEU single event upset.

SPICE simulation program with integrated circuit emphasis.

SRAM static random-access memory.

STR space-time redundancy.

STT-RAM non-volatile spin-transfer torque RAM.

TCAD technology computer-aided design.

TID total ionising dose.

TMR triple modular redundancy.

VART Vitis AI Runtime.

VCU video codec unit.

VLSI very large-scale integration.

VVAS Vitis Video Analytics SDK.

ZB zettabytes.

Chapter 1

Introduction

1.1 Background

The recent advances in artificial intelligence (AI) namely in machine learning and deep learning (DL), have been successfully applied in a wide range of areas, including image classification [3, 4], speech recognition [5], object detection [6], semantic segmentation [7] and natural language processing (NLP) [8]. Meanwhile, thanks to the continued development of integrated circuit technologies, nowadays integrated circuits are becoming cheaper and more powerful. It makes DL applications increasing popular in mobile and embedded devices (e.g., smartphones, wearable devices, self-driving cars, robotic systems and other Internet of Things (IoT) devices [9]).

The impact of DL in embedded systems for radiation environments is also growing [10]. On the one hand, DL algorithms are very well suited to handle complex and varying scenarios. For example, DL can play an active role in the operation of a spacecraft in space environments, allowing for precise automated control and facilitating onboard tasks, such as docking or navigation [11]. In addition, the hardware platforms are becoming more affordable and accessible, thus widening the range of DL applications in radiation environments [12]. This increased accessibility also contributes to the research of DL deployment on constrained platforms themselves [13, 14], especially because the communication channel is often limited

in radiation environments.

However, there remain many challenges related to building hardware systems for radiation environments [15]. The first is the deployment of DL applications. Like with many other embedded systems, the onboard computational power is not sufficient compared to the requirements of DL applications [16]. As DL applications were introduced into an increasing number of tasks, the conflicts become more apparent than ever. For instance, a satellite may run multiple DL applications simultaneously, such as weather monitoring, vegetation and ground cover classification and object detection [17], thus requiring additional hardware capability. However, the extra hardware normally brings extra volume, weight and power consumption, which is often not practical in such systems.

In addition to challenges faced by DL for embedded systems, radiation effects impose extra requirements [18]: the need for radiation-hardened hardware, robustness and extensive verification. Radiation environments include natural radiation environments (e.g., space environment [19]) and man-made radiation environments (e.g., nuclear power systems [20]). In such radiation environments, energetic particles can hit, penetrate and interact with semiconductor materials, causing faults in hardware systems. Commonly, there are two major types of radiation effects, 1) cumulative effects and 2) single event effects (SEEs), which have been reported as the dominant effects on electronic systems [21]. On the one hand, it is challenging to design hardware circuits for radiation environments, due to highly frequent SEEs in extreme radiation environments, where the error rate could be more than 10 per device per minute [22]. On the other hand, there are increasing challenges for SEE simulation, as today's integrated circuits are becoming increasingly larger.

The motivation of this thesis is to strengthen AI systems in radiation environments. There are three aspects of research work that have been completed in this thesis: 1) a fast simulation method for analysis of SEE in integrated circuits, 2) a self-refresh scheme to detect and correct bit-flips in random access memory (RAM) and 3) an AI hardware acceleration system with hardware reconfiguration

for increasing reliability, flexibility and efficiency.

Firstly, due to the complexity of SEE and the increasingly large size of existing circuits, it has become a challenge to design a large-scale circuit suitable for the verification and simulation of SEE mitigation performance. Current tools usually require simulation and SEE injection for a specific physical netlist [23]. However, when designing digital circuits, we often encounter the problem of how to easily and quickly verify the performance of different logic designs in a radiated environment, without generating a physical netlist for each iteration of the design. Therefore, in the first research work, a new SEE simulation scheme is proposed to offer a fast and efficient method to evaluate and compare the performance of large-scale circuits subject to the effects of radiation particles.

Secondly, when an operation is conducted in a strong radiation environment such as a nuclear power system, errors can be accumulated much faster than in traditional radiation environments. Considering that the RAM is typically the most SEE-sensitive element in embedded systems, it is critical to harden RAMs. However, even the typical radiation-resistant components intended for the space environment may not be able to meet the requirements in our case, such as nuclear power systems. Therefore, in the second work, a portable scheme combined with error correction code (ECC) and refreshing techniques is proposed to correct errors and mitigate error accumulation in extreme radiation environments. Compared to other works, it can be easily applied to the existing hardware modules for extreme radiation environments.

In the third work, a combination of the requirements for DL deployment and SEE mitigation performance is used to build an adaptive intelligent system for extreme environments. The concept of "refreshing" is also extended. At the hardware level, the features of field-programmable gate array (FPGA) are utilised, partial reconfiguration, to improve the reliability of the systems. More importantly, this feature also provides the capability to manage the hardware resources for DL acceleration. The on-chip hardware resources, such as lookup table (LUT),

block random access memory (BRAM) and digital signal processor (DSP), are dynamically managed to improve the flexibility, performance and power efficiency of deep learning inference systems. At the software level, to further increase the resilience of the system, a flexible DL software framework is introduced in addition to the hardware reconfiguration. When facing various work scenarios, a range of DL networks with different model complexities can be dynamically switched in real time for adapting to different performance, power and accuracy requirements.

1.2 Simulation of radiation effects

Modelling and simulating the effects of ionising radiation have been long used for better understanding the radiation effects on the operation of devices and circuits [24, 25]. In SEE simulations, we can inject SEEs and observe the outputs of circuits to see what SEE will cause. SEE simulations can offer the possibility of reducing radiation experiments and testing the hypothetical devices or conditions, which are not feasible (or not easily measurable), by experiments. In addition, due to the smaller feature sizes in microelectronics and the higher price of manufacturing processes [26], it is becoming more important to use the simulation for the SEE resistant designs.

However, the complexity of SEEs makes it difficult to perform fast and accurate SEE simulations, especially in large-scale circuits. SEEs are fundamentally the collection of electron-hole pairs [27], which are generated in ionisation processes. The collection of electron-hole pairs is observed as transient currents or the charge of circuit cells. Depending on the energy released by the particles, semiconductor materials (e.g., linear energy transfer) and the physical designs of the circuit, the magnitude of the current and voltage of the pulses might vary. To achieve accurate SEE evaluation results, many parameters have to be taken into account. Obviously, this kind of SEE simulation requires massive calculation power for even a single metal-oxide-semiconductor (MOS) component. Furthermore, in digital circuits, the propagation of errors will also be affected by the propagation path and logic

designs, which makes the SEE simulation highly complex.

Although physically-based numerical simulation tools, such as simulation program with integrated circuit emphasis (SPICE) and technology computer-aided design (TCAD), are popular for the analysis of SEEs at the transistor level, the study of radiation effects at circuit level with logic designs is still limited. Because integrated circuits are becoming larger and contain tens of billions of transistors, it is unrealistically time and resource intensive to carry out fault injections in large-scale circuits, which causes challenges to arise between costs and accuracy in the SEE evaluation and simulation.

System level simulation, such as simulation based on hardware description language (HDL), is another method to rapidly evaluate the SEE mitigation performance of the circuits [28]. As the HDL simulation is based on the behaviour model of circuits, it features high efficiency for large-scale circuits. Normally, the HDL SEE simulation carries out the error injections in the data stream or memory units, based on the probability of bit-flips [29]. However, the behaviour of the circuit does typically not reflect the actual physical parameters of the circuit, which have a strong correlation with SEEs.

It raises a question:"is it possible to combine the advantages of semiconductor simulation tools and HDL simulation tools to analyse the SEE in large-scale circuits?" When comparing the SEE mitigation performance of different logic designs in radiation environments, we care more about the propagation and logic responses, than the changes in currents and voltages. In such cases, the SPICE simulation is too slow, and the HDL simulation based on error rates is not precise enough.

To resolve these issues, a new SEE simulation scheme is proposed, to offer a fast and cost-efficient method to evaluate and compare the performance of large-scale circuits in this thesis. The scheme consists of the following features: 1) building the SEE behaviour models based on SPICE or TCAD, 2) generating the HDL netlists and injection scripts based on the HDL designs, 3) applying the SEE behaviour

models in the HDL simulations to analyse and compare the performance of the circuit designs and 4) modifying the hardware designs according to the results and repeat the simulation processes.

1.3 Error mitigation techniques for memory system

Static random-access memory (SRAM) cells are normally based on the 6-transistors (6T) structure [30], as a high risk to catch glitches. It makes SRAM highly susceptible to single event upsets (SEUs)[31, 32]. Nowadays, with more computer systems deployed in radiation environments, SEUs in RAM components have become the primary short-term reliability concerns in space systems [31].

In order to mitigate SEU in RAMs, a series of error mitigation strategies have been considered, including: triple modular redundancy (TMR) technology, ECC and Scrubbing technology [33, 34, 35, 36, 37]. TMR is a recognised technique for improving the reliability of the circuit in a radiation environment[38]. It can be applied to a range of applications from circuit modules to top systems. ECC is widely known as an anti-interference encoding strategy to enhance the reliability of memory devices and communication systems [39]. The key concept is to use extra bits to store redundant information for error correction. Scrubbing or refreshing is an effective error mitigation technology for memory devices to resolve the accumulation of errors [40, 41, 42, 43]. To apply this strategy, data in RAMs will be read out, corrected and written back. Therefore, additional bandwidth is required to conduct refreshing operations [44].

There are still some difficulties to apply those strategies in extreme radiation environments. Firstly, unlike most computer systems designed for the natural terrestrial environments with low error rates (less than one correctable error per year [45]), the hardware systems in extreme radiation environments, such as space or nuclear power systems, face much higher radiation density and error rates. It

makes ECC designs more complex [46]. Secondly, the hardware resources are normally expensive and constrained in radiation-specific systems [47]. However, TMR also incurs a high overhead cost. It is not a good option for low-cost systems.

Scrubbing or refreshing seems to be an effective error mitigation technology for memory devices to resolve the accumulation of errors in extreme radiation environments [40, 41, 42, 43, 44]. The basic idea is to overwrite memory cells with the correct data when an error has been detected. A conventional scrubbing scheme consists of reading, detection and rewriting. Compared to the simple ECC, the systems equipped with scrubbing techniques can be applied to check each memory unit periodically. Each unit may be checked frequently, before the accumulation of multiple-bit flips, in case these errors are not correctable. Therefore, scrubbing is appropriate for data retention in memory (e.g., DRAM [48], STT-RAM [49], NAND flash [50]). Because the FPGA structure is also based on RAMs, it is applicable to FPGAs.

However, when it comes to an FPGA system with many pre-designed hardware modules using block random access memory (BRAM), the existing hardening methods are often problematic. Firstly, due to the pre-designed finite-state machine (FSM) in the hardware modules, modifications in the time sequence of the hardware modules will be difficult, which means that we cannot simply add the correction operations. Secondly. considering the usage of the dual-port BRAMs, there will not be enough RAM access ports to connect scrubbers to. Thirdly, because the TMR methods will need triple the resources to work, the available RAM space will be limited.

Therefore, considering the high error rate in extreme environments and the convenience of the application to the existing circuit modules, a hardening scheme combined with ECC and refreshing techniques is proposed, to correct errors and mitigate error accumulation in extreme radiation environments. It is designed to extend the operation time of electronic devices that are exposed to high radiation, causing errors to occur every second. It features high reliability, flexibility and low

hardware costs. In this work, FPGAs are used to build prototypes for real-world radiation experiments. The experiment results show that the proposed design can significantly mitigate errors in extreme radiation environments.

Furthermore, the configuration RAM of FPGAs is also subject to the concept of "refreshing". Reconfiguration is a capability of FPGAs that may be used to alter hardware functionality as well as hardened circuits. Considering that the AI hardware systems for radiation environments face challenges, including not only radiation effects but also constrained computational capability and energy resources, it could be a potential solution for both issues. Therefore, an adaptive intelligent system based on the concept of reconfiguration is also investigated as the fellow on work.

1.4 Adaptive intelligent systems

In addition to the effects of radiation, it is a challenge to deploy AI systems on embedded devices.

When it comes to intelligent systems, there are many challenges for hardware design including energy efficiency [51], performance [52] and limited hardware resources [53]. Compared to standard AI systems, intelligent systems in radiation environments face more challenges. The systems in radiation environments are more sensitive to power consumption, because they are expected to have longer operation times. For example, satellites often need to be powered by solar energy to operate for several years on launch missions [54]. In the radiation environment of nuclear power plants, robotic systems also require a longer operation time to reduce the risks to human operators during maintenance [55]. Furthermore, AI hardware systems for radiation environments must consider the radiation effects in the design stage. Some parts of hardware resources are used for error detection and correction, which leaves fewer resources for AI applications. Moreover, considering that most of the hardware systems for use in radiation environments are highly mission-dependent, the flexibility of the hardware is also important.

1.4.1 Adaptive hardware design

Therefore, there are sensible reasons to choose FPGAs for DL acceleration in harsh environments when the budget is limited [56] and high power efficiency is required. Although one-time programmable (OTP) FPGAs are available currently, the dominant types are SRAM based FPGAs, which can be reprogrammed as the design evolves. These FPGAs store information about the user circuit in SRAMs. On the one hand, SRAMs can be reconfigured to deploy different circuits, which brings high flexibility. On the other hand, as mentioned above, SRAMs are vulnerable to radiation effects. In radiation environments, high-energy particles can easily cause SEUs, which makes FPGAs less reliable than other hardware components. Fortunately, there are already many radiation-resistant hardening methods available for FPGAs. By using the reconfiguration feature, the circuits can be refreshed to dynamically mitigate errors.

The motivation of the third work is to fully use the reconfiguration feature of FPGAs to improve the efficiency of AI inference instead of just error mitigation. The third work considers a combination of the requirements for DL deployment and SEE mitigation performance, in order to build an adaptive intelligent system for extreme environments. By using "Dynamic Function eXchange (DFX)" [57], a reconfiguration feature of Xilinx (FPGA Vendor) latest FPGAs, the executing system is capable of dynamically allocating the hardware resources according to the exact requirements such as performance and power consumption. Through the deployment of hardware accelerators with different configurations, it is achievable to dynamically adjust the performance, power consumption and available FPGA resources for various tasks. Under the proposed scheme, the hardware accelerators are grouped into accelerator pools to accept acceleration tasks, so the reconfiguration can be conducted seamlessly, without disrupting the executed programmes.

1.4.2 Adaptive DL inference

Due to a distinct evolution of DNN architectures [58, 59], there have been more sophisticated network architectures proposed to improve the network inference performance. Despite the massive potential demonstrated by such new DL architectural concepts [60] to improve on the current DL techniques, they are likely to introduce the type of hardware and software required to deliver such capabilities efficiently in the future.

Since the concept of reconfiguration can be used in hardware design, there is a natural idea to apply 'reconfiguration' to software design. To further improve the efficiency of the hardware system proposed in the third work, an adaptive DL software framework is proposed to provide significant support for the adaptability of various DL algorithms on an FPGA-based edge computing platform. In this work, the concept of "dynamic reconfiguration" was adopted. The dynamic DNNs model was also taken into account, where multiple DNNs can be dynamically deployed for different conditions. The system is capable of configuring both the hardware and software processing pipelines dynamically to achieve better cost, power and processing efficiency for the dedicated application requirements and operating environments. More importantly, a practical FPGA-based test platform for real-time model management is designed and implemented in this work. It may help to develop subsequent optimisation algorithms for hardware and software scheduling.

1.5 Contributions

The works in this thesis can be divided into two categories: 1) hardening circuits in radiation environments and 2) improving the efficiency of AI inference on hardware systems. The road map of the works is shown in Fig. 1.1. The works in this thesis started from the SEE simulation for large-scale circuits, as it is the preparation step of SEE hardening design. Subsequently, a SEE hardening design based on the

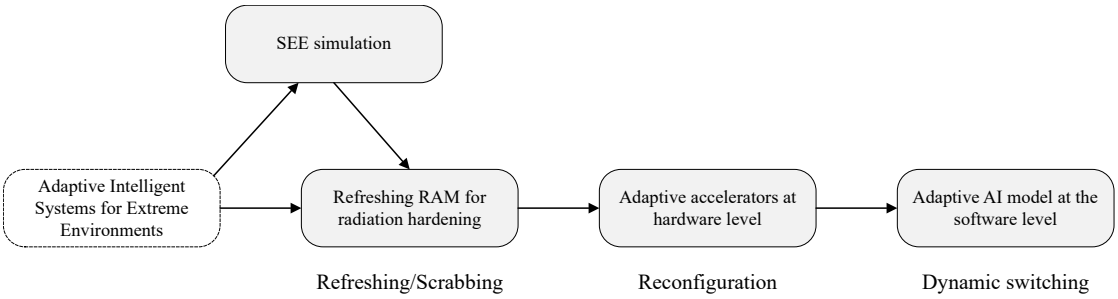


Figure 1.1: Road map of the works in this thesis.

refreshing concept was proposed to mitigate errors in RAM devices in the second work. The keyword of the second work is refreshing. The third work focused on the challenges of AI inference works for embedded platforms. By adopting the concept of refreshing, hardware reconfiguration was used to improve flexibility, performance and efficiency. Moreover, the concept of reconfiguration was further extended in the final chapter, where an adaptive DL software framework was also proposed to further improve the flexibility, performance and efficiency of embedded platforms. The contributions of each work are summarised as follows.

In the first work, a new SEE simulation scheme is proposed to offer a fast and cost-efficient method to evaluate and compare the performance of large-scale circuits. The main contributions of this work can be summarised as follows:

1. The SEE simulation scheme provides a rapid, convenient and universal comparison method with which to evaluate the designs of circuits in the context of SEEs. Due to various manufacturing processes, physical layouts and radiation environments, the simulation tools and simulation environments may also vary in different SEE research. It is difficult to repeat or compare those experiments directly. The SEE models can be easily integrated into the current circuit design workflow without significant cost. It can create a universal simulation environment to provide a quick analysis of the relative performance, which can significantly reduce the total simulation time for the time-consuming back-end simulation.

2. This work introduces a range of new SEE behaviour models. Based on the

transistor level simulations, the SEE behaviour models are firstly embedded into a range of digital functions in the HDL described circuits, the transient currents and voltages are then converted into digital pulses and bit-flips. Unlike the typical transistor level based SEE behaviour models that fully rely on low-level current and voltage simulation inputs, the SEE models use only high-level digital functions in HDL. Therefore, they can offer lightweight and fast simulations for large-scale circuits.

3. This scheme can offer a high level of flexibility in the design. All parts in this scheme including gate components, SPICE simulation and HDL simulation are decoupled. The gate components can be modified to adapt to different manufacturing processes, and the SEE SPICE model can be also modified to adapt to different radiation environments, as required. In this way, the scheme can make full use of existing models to build simulation environments and be adapted for various requirements.

In the second work, a self-refreshing scheme with ECC is proposed, to harden the RAM systems based on CPUs and customised circuits in extreme environments. The main contributions of this work are stated as follows:

1. This design is highly flexible. Compared to conventional external scrubbers [61, 62, 63], the controller module is transparent to other modules. No additional latency is introduced in the systems. There is no need to modify the designs to adapt to the hardware changes, hence, it can be easily applied in various embedded systems.

2. The design is an area-efficient design, which makes it suitable to harden low-cost computer systems. Compared to conventional internal scrubbers [64], this design requires no dedicated components (e.g., ICAP). In systems with multiple customised modules, which operate separate RAMs, this design can be deployed multiple times to protect one or more modules.

3. The SEU mitigation design can achieve high SEU correction rates in various conditions. The results of the simulation and the radiation environment test follow the same trend. In the simulation, the design can correct more than 99.97% of SEUs errors at the SEU injection rate of 6.25×10^4 bit/s. During the one-hour neutron radiation experiment, the SEU correction rate achieves 100%.

There are two parts in the third work. In the first part, an adaptive hardware system intended for DL tasks is proposed, to manage the hardware resources seamlessly according to the exact system requirements. In the second part, an adaptive DL software framework is proposed to provide a significant level of adaptability support for various DL algorithms on an FPGA-based edge computing platform. The main contributions of this work are stated as follows:

1. An improved flexible DNN hardware accelerator framework that can be applied to configure the hardware and software processing pipelines dynamically is proposed, to improve the power consumption and latency performance metrics.

2. The efficiency of the Deep-Learning Processor Unit (DPU) with different hardware configuration is evaluated, which can be useful for future optimisation. In our experiment, DPUs with different frequencies show similar power efficiency at full speed. When DPUs are not fully in use, the power efficiency decreases with increasing frequency and size.

3. A practical FPGA-based test platform for real-time software and hardware management is designed and implemented in this work. It can help to develop subsequent optimisation algorithms for hardware and software scheduling.

4. A comprehensive evaluation of DNN model sizes and inference performance is conducted, with Xilinx DPUs used in video analytic applications. This framework allows run-time reconfiguration to increase the power and computing efficiency of both the DNN model/software and hardware, to meet

the requirements of dedicated application specifications and operating environments.

1.6 Organisation of the thesis

As the introduction of the thesis, Chapter 1 introduces the motivation and contributions of the works. Chapter 2 reviews the background of the radiation effects on electronics, SEE hardening works, DL acceleration and AI hardware. Chapter 4 presents a radiation hardening design for RAMs. Chapters 5 and 6 discuss a system design on FPGAs for DL in hardware and software respectively, in which the reconfiguration features of FPGAs are used to improve the reliability, flexibility and power efficiency of the AI systems. Chapter 7 concludes all the work in this thesis and presents the future works.

Chapter 2

Literature review

This chapter contains a review of the background of the study and the related published works. Firstly, this chapter contains an overview of the radiation environments and the challenges for electronic devices. Secondly, it presents the introduction of SEE, which is the dominant effect in radiation environments. The existing works related to SEE simulation and mitigation are reviewed, and the drawbacks of the current methods are also discussed. Thirdly, the background of hardware systems for AI inference is introduced. The advantages of FPGAs are analysed. The reconfiguration feature offers FPGAs high computational performance, power efficiency, flexibility and SEE mitigation performance, thus making it suitable for radiation environments.

2.1 Overview of radiation environments

Radiation widely exists in our environments. For example, in the terrestrial environment, radiation sources could be the secondary cosmic rays in the Earth's atmosphere or radioactive contaminants inside chip materials. To a large extent, in space, radiation sources could be the trapped particles, the particles emitted by the sun, and galactic cosmic rays. Radiation can be generated in biomedical devices, nuclear power plants and artificial environments in high-energy physics experiments. This section contains a brief overview of space, atmospheric, ter-

restrial, and artificial radiation environments. The radiation effects in different environments and different electronics devices will also be compared.

2.1.1 Space environment

Natural space radiation can be classified into three categories: 1) electrons and protons trapped by planetary magnetic fields (e.g. Earth), 2) cosmic rays (very energetic atomic nuclei) produced in supernova explosions within and outside our galaxy [19] and 3) protons and a tiny fraction of heavier nuclei produced in energetic solar events.

The planet radiation belts, also known as the Van Allen belts [65, 66], were the first discovery of the space age, measured with the launch of a US satellite, Explorer 1, in 1958 [67]. It is a zone of trapped megaelectron volt (MeV) particles. Most of the particles originate from solar wind and are captured by the magnetosphere of planets [68, 69].

Cosmic rays (CRs) are high-energy particles that originate outside the solar system [70, 71]. CRs originate as primary CRs, which are those originally produced in various astrophysical processes. CRs is composed of around 98% nuclei and 2% electrons and positrons [71]. The energy of CR particles may exceed 10^{20} eV [72, 73]. Due to the high energy, CR particles can easily penetrate integrated circuit chips, posing a significant risk to the operational stability of spacecraft such as satellites and the safety of astronauts.

The sun is both a source and a modulator of space radiation [74]. Solar CRs consist of the high-energy particles emitted by the sun. They were registered for the first time in 1942 [75], 30 years after the discovery of Galactic CRs. Normally, solar protons have insufficient energy to penetrate the Earth's magnetic field. However, during unusual solar events [76], primarily in solar flares, protons can be accelerated to sufficient energies to reach the Earth's magnetosphere and ionosphere around the north pole and south pole.

It has been reported multiple times that electronic devices in satellites suffer

faults [77] in space environments. For example, the NASA/DoD Tracking and Data Relay Satellite (TDRS-1) experienced upsets in RAM chips in the control systems. The rates of 1 to 2 per day clearly showed modulation with CRs, while during the solar particle events of September to October 1989, the rates reached 20 per day [78]. Another example is a hardware failure in the instrument carried by the European Remote Sensing Spacecraft (ERS1). A latch-up failure occurred and led to a loss of the instrument [79].

2.1.2 Atmospheric and terrestrial radiation environment

Like the other planets, the Earth is continuously irradiated by CRs, including solar and galactic rays. Fortunately, with the combined effect of the Earth's magnetic field and atmosphere, most high-energy particles will not reach the Earth's surface. Earth has a thick atmosphere of oxygen and nitrogen, which can interact with CRs to mitigate radiation.

When CRs arrive at the earth's atmosphere, the collision with atoms and molecules consumes energy. It generates a cascade of lighter particles, including x-rays, protons, alpha particles, ions, muons, electrons, neutrinos, and neutrons. They are called air shower secondary radiation or cascades particles. Those particles may continuously interact with the atmosphere. When they finally hit the ground, most of them are the third to seventh generation cascade particles.

In atmospheric environments, the peak of the cosmic ray intensity occurs at about 10-25 km [80, 81], which is also the altitude of many commercial flights. In the last 40 years, it has been discovered that the electronics in aircraft systems, which are subjected to increasing levels of cosmic radiation and their secondaries as altitude increases, are also sensitive to SEEs. The soft error-rates increase with the altitude [82].

The experiments conducted by IBM and Boeing showed that the failure rate of electronics at airplane altitude is about one hundred times worse than at sea level. [83]. Some works demonstrated that ions are the primary sources at higher

altitudes than 20 km while the SEUs derived from neutron interactions are dominant at lower altitudes [84, 85]. Researchers found that SEU rates would increase during large-scale solar particle events.

2.1.3 Artificial radiation environment

In addition to radiation in the natural environment, there are some man-made radiation sources as well, including nuclear power systems, nuclear weapons experiments, particle accelerators and medical radiation. In these radiation environments, electronic devices can also be affected by high-energy particle rays and experience soft errors.

With the Trinity test of 1945, the nuclear weapon testing era began, as did the global distribution of radioactive fallout from those tests. Between 1945 and 1980, over 500 nuclear tests that injected radioactive debris into the atmosphere were conducted at various sites around the world [86]. The extremely high energy in nuclear tests has strong effects on electronics. The effects were first noticed in the Starfish nuclear weapon test conducted by the US in 1962. The tests were held on Johnston Island in the Pacific Ocean. The particles from the test were injected into the Earth's atmosphere, which formed a radiation belt at an altitude of around 400 km and caused faults in electronics. For instance, the Telstar satellite experienced a total ionising dose (TID) that was 100 times larger than expected. Within seven months, the Starfish nuclear weapon test destroyed seven satellites [87].

Nuclear power systems are also one of the most common man-made radiation environments. In 2020, nuclear power in the United Kingdom generated 16.1% of the country's electricity [88]. During the routine operation of nuclear installations, the releases of radionuclides are low. Radiation is mainly generated in the nuclear fuel cycle, including mining, milling, and reactor operation.

In nuclear power systems, electronic devices need to deal with radiation as well. Cherenkov radiation [89] is an example of radiation effects, generated by an underwater nuclear reactor. It occurs when a charged particle (such as an

electron) passes through a dielectric medium at a speed greater than the phase velocity (speed of propagation of a wavefront in a medium) of light in that medium. Although many of the systems developed for space applications are specialized for high ionizing radiation transients, the electronic devices in nuclear power systems care more about neutron radiation and TID than about the proton, electron, and heavy-ion radiation. In addition, temperature and ageing are factors to consider in the design of such systems.

In general, nuclear applications will require rad-hard circuitry to survive MGy TID and $10^{16} n/cm^2$ neutron fluences, while the commercial devices for low-earth orbit satellites are designed to survive $1 - 10$ KGy TID. Applying the existing space-application rad-hard electronics in terrestrial neutron environments requires additional research [90].

2.2 Single Event Effect

2.2.1 Background of SEE

As mentioned in Chapter 1, there are two major types of radiation effects, 1) cumulative effects and 2) SEEs. Cumulative effects are the long-term effects that can change the parameters of semiconductor materials, and these can be divided into two categories: 1) TID [91] and 2) displacement damage [92]. By contrast, SEEs arise through the action of a single ionizing particle as they penetrate sensitive nodes within electronic devices. During the penetration, random glitches are generated, which may lead to system failure in the worst scenarios. In recent years, the family of SEEs has been proved to cause the dominant effects on electronic systems in radiation environments [21].

2.2.2 Mechanisms responsible for SEEs

SEEs are induced by the ionization process during the penetration of high-energy particles in semiconductor materials. Typically, there are two steps in the physical

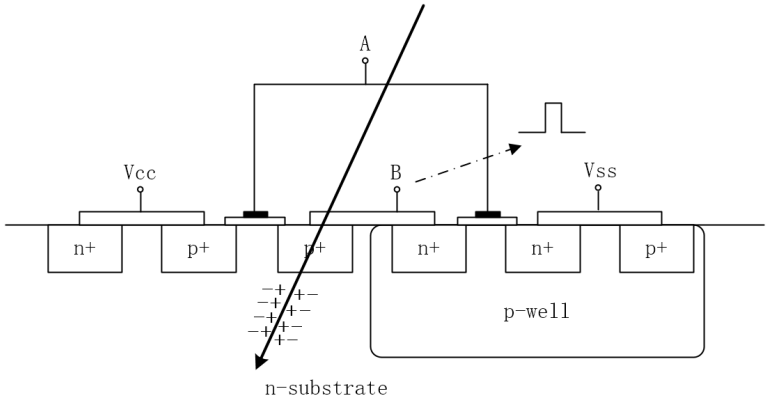


Figure 2.1: The transient pulse will be generated due to the ionization and collection process. It is a structure of an inverter. The input A is 1 and the output B is 0. The PMOS between Vcc and B is the sensitive node.

mechanisms of SEEs: 1) the charge deposition by the energetic particle strike and 2) the subsequent collection of that charge by devices in the region of particles strike [27].

The charge deposition is the process of releasing charge in a semiconductor device. Charge deposition can be caused by direct or indirect ionization. Direct ionization is a mechanism by which incident particles, mostly heavy ions, directly ionize semiconductor materials during penetration. In contrast, the indirect ionization is caused by the secondary particles created in nuclear reactions between the incident particle, mostly protons and neutrons, and the struck device. Both mechanisms will free electron–hole pairs and provide the basis for the subsequent generation of instantaneous currents, which is also the key reason for integrated circuit malfunction.

The collection is the second step in the SEE mechanism. When a particle strikes a semiconductor device, the most sensitive regions in the integrated circuits are usually reverse-biased PN junctions. The high field present in a reverse-biased junction depletion region can collect the particle-induced charge through drift processes efficiently, thus leading to a transient current at the junction contact. The strikes near a depletion region can also result in significant transient currents as carriers diffuse into the vicinity of the depletion region field where they can be

efficiently collected.

The transient currents can be observed as instantaneous voltage pulses in the logic gate devices. Fig. 2.1 shows the illustration of a SEE glitch in an inverter. At the very beginning, input A of the inverter is high, and output B is low. There is a voltage difference directly between Vcc and output B, and the PN junction is also reverse-biased. When a high-energy particle strikes the region between Vcc and B, the p-channel MOS (PMOS), many electron-hole pairs are created due to ionization. Subsequently, the electrons and holes move toward the two segments, respectively, and a transient current is generated under the influence of the voltage difference. From a macroscopic point of view, when an energetic particle bombards the inverter, a brief positive pulse is generated at the output.

If a particle hits the region between output B and Vss, there will not be a strong collection process and transient current due to the low voltage difference. Hence, in the current state, where input A is high, the PMOS is a sensitive node, while the n-channel MOS (NMOS) is not. It can also be expected that if we change the input value to low, then the sensitive node will be PMOS.

2.2.3 Classification of SEEs

All of the possible effects are grouped by the family of SEEs on electronic components. Typically, they can be divided into a number of effect categories. Table 2.1 list the family of SEEs. There are both soft (recoverable) and hard (unrecoverable) errors. In this thesis, I will mainly discuss SETs and SEUs, which are two dominant soft errors in the SEE family [93].

2.2.3.1 Single event transient

SET refer to the transient currents or voltages which affect combinational circuits. When SETs occur, there may be a transient in gate output. Sometimes, those transient pulses may propagate through subsequent gates and eventually cause a SEU when it reaches a memory element [95].

Table 2.1: The basic categories of SEEs [94].

SEU	Single event upset	Temporary change of memory or control bit
SET	Single event transient	Transient introduced by single event
SEL	Single event latch up	Device latches in high current state
SES	Single event snap back	Regenerative current mode in NMOS
SEB	Single event burn out	Device draws high current and burns out
SEGR	Single event gate rupture	Gate destroyed in power MOSFET
SEFI	Single event functional interrupt	Control path corrupted by an upset
MBU	Multi-bit upset	Several bits upset by the same event

To cause a fault in the circuit system, a SET must meet four criteria as follows [96]:

1) The generated transient current in SET must be strong enough to propagate through the circuit, which means that the charge deposition and collection have to be strong enough. In other words, the particle should carry sufficient energy to hit the sensitive node.

2) There must be an open logic path for pulses to finally reach a memory element, which means that the design of the logic path could affect the performance of radiation resistance.

3) The SET should have amplitude and duration to change the state of the memory element. It is related to the physical layout of the circuits. The duration of SET pulses is not consistent during the propagation. The pulses could be extended or narrowed in different gate chains. Hence, the circuit netlist is a parameter affecting the radiation resistance performance.

4) In synchronous sequential circuits, the SET must arrive at the latch during the latching edge of the clock. It is worth mentioning that the frequency of the circuits can also be affected by the probability that transient glitches are captured as valid data. As the frequency increases, the frequency of clock edges will also increase. According to the fourth criteria, it is easier for SETs to be captured. In addition, a higher frequency circuit typically means faster or fewer gates per pipeline stage. In this case, SETs will have a greater ability to propagate.

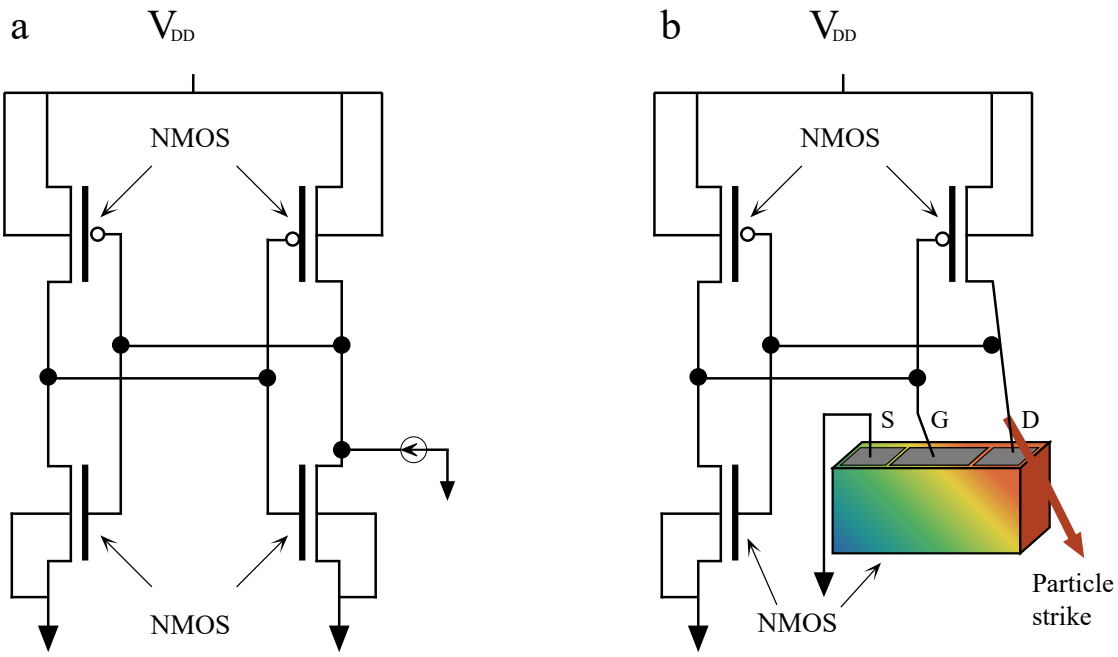


Figure 2.2: (a) Schematic diagram of two cross-coupled inverters in CMOS technology. A current pulse is injected at the drain of the off NMOS transistor. (b) Mixed Mode simulation is used to simulate SEU. The Off NMOS is studied by means of device simulation and the remaining transistors are studied with the coupled circuit simulator [30]

2.2.3.2 Single event upset

SEUs are the bit-flips in storage cells. In SRAMs, SEU could be a reversed state. In DRAMs, SEU could be wrong values due to the change in stored charge. In logic circuits, SEUs can also be the error bits when a SET is captured by a latch or a flip-flop. In this thesis, SEUs refer to the upset in SRAM devices.

SRAM is one of the most sensitive components to SEUs. Most SRAM cells are based on the 6T structure with two coupled complementary MOS (CMOS) inverters and two NMOS accesses. The two inverters form a loop circuit that feeds the output back into the input. At any given moment, these two inverters have reversed inputs with one input of high and one output of low. There will always be two sensitive nodes with one PMOS and one NMOS, because they are in the "off" state. When they are struck by the particle, there will a transient pulse fed back into the cell. When the charge is adequate enough to cause an upset, then there is a SEU. Fig. 2.2 shows the simulation of SEU injection in the SRAM cell.

A transient current is injected into the circuit to simulate the SEU.

2.2.3.3 Multi-bit upsets

In addition to single bit upset (SBU), there could be more than one upset in the circuit system, which can be referred to as multi-bit upset (MBU). Multiple errors can be generated when a particle travels through the sensitive node in different cells or when the free carriers of the ion track can be collected by different junctions of transistors of several memory cells.

The upsets could be in the same word or different words in the logic aspect. If the upsets are in different words, then the typical mitigation methods for SBU, like hamming correcting codes, are still effective. However, if the bits are in the same word, it will be difficult to detect and correct errors. In addition, as the elementary cell area is continuously decreasing for successive technology nodes, the probability that recoils have a range long enough to reach different cells increases. In this case, the probability of multiple cells upsets increases [95].

2.3 SEE simulation and evaluation

The SEE simulation can offer the possibility of reducing radiation experiments and testing the hypothetical devices or conditions which are not feasible (or not easily measurable) by experiments [24, 25]. To study the effects of SEEs, researchers need to simulate the generation of SEEs in circuits, which is normally referred to as the injection of SEEs [97]. The SEEs injection can generally be divided into two main categories. One is to inject SEEs through radiation experiments and the other is to simulate the occurrence of SEEs using simulation tools.

2.3.1 Radiation experiments

There are two primary methods of radiation experiments: 1) ground-based radiation experiments and 2) space flight experiments. The space flight experiment

is a realistic measurement performed by spacecraft. In space flight experiments, circuits are exposed to the real working radiation environment, so that the results of space flight experiments can accurately reflect the circuit response. Space flight experiments are the most effective way of verifying whether a chip can operate stably in space. Over the last century, SEE research has mainly been based on the errors detected in satellite-based systems. However, there are fewer opportunities today to conduct space flight experiments due to the long period and high price. In addition, space radiation experiments cannot be used as a universal experimental method.

Ground-based radiation experiments are also known as ground-based simulation experiments or ground-based irradiation experiments. Ground radiation experiments are conducted using several radiation sources such as heavy ion accelerators, proton accelerators [98], 252CF sources [99, 100], neutron sources [101, 102] and pulsed lasers [103]. Ground-based radiation experiments can be relied on to simulate various environments via different irradiation intensities and different particles. The ground-based radiation environment is relatively accurate as it uses a realistic radiation environment that also produces realistic SEEs. In addition, the ground environment has more controllable variables and better experimental flexibility compared to the space environment.

However, there are drawbacks as well. Firstly, the equipment and facilities for ground-based radiation sources are generally large and expensive, and the use and maintenance of the equipment are costly. For example, the ISIS pulsed neutron source [104] in the UK is expected to spend £16 million on radioactive waste disposal after it stops running. Secondly, only a few facilities can conduct ground-based radiation. It takes a long time to apply for slots and to wait in the queue. Due to the high time and economic costs, research teams tend to conduct software simulation instead of radiation experiments.

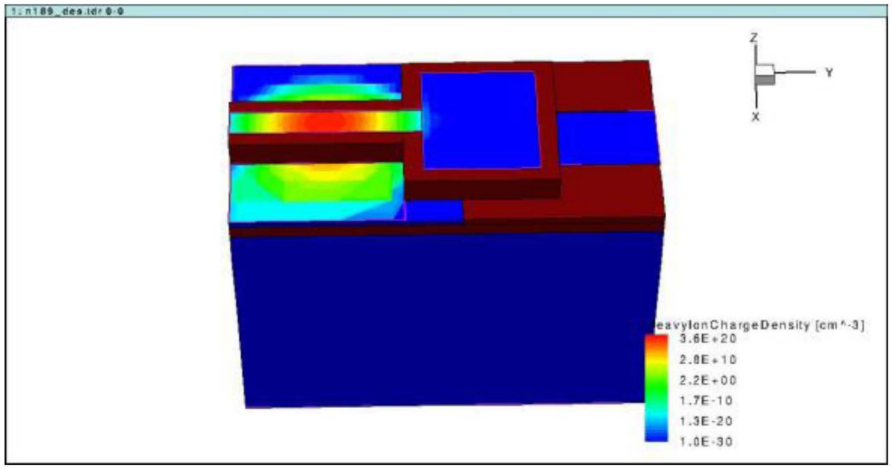


Figure 2.3: SEE injection in 3D TCAD simulation. A heavy ion hits the center of the channel [108].

2.3.2 SEE simulations at the transistor level

Over the last two decades, TCAD (technology computer aided design) has proven its value and steadily expanded its role in the advancement of technology. As CMOS scaling continues, semiconductor devices are being pushed to their physical limits, requiring such advanced physics as quantum mechanics to be included in modeling these advanced semiconductor transistors. So far, the TCAD community has responded to and answered the challenges in providing tools and novel techniques to address the increasing complexity of modeling semiconductor transistors with new architectures, transport phenomena, switching mechanisms, and materials [105].

SEE researchers have also noticed the advantages of TCAD and TCAD has been widely used for SEE as a simulation tool at the transistor level. At this level, the size of the circuit is usually limited to the circuit size of a few semiconductors. In transistor simulations, the parameters such as the density ionization and the structure of the semiconductor can be modified easily, which is helpful during the study on how to design radiation-resistant semiconductors [106, 107].

Traditionally, these tools tend to build a model to describe the currents and voltages in the SEE. Typically, these tools take three steps to conduct simulations [30]. The first one is to take a large number of physical parameters as inputs (e.g.,

particle energy, material, angle, orientation, external electric field, physical size and 3D structure). Normally, the more physical parameters there are, the more accurate the results will be. The second one is to create transport models and transfer equations to predict the ionization process. In SEEs, electron-hole pairs are created in ionization, which can be expressed as an energy-based transfer or as a power-based transfer. For example, the drift-diffusion model was taken as the standard level of solid-state device modeling for many years. The third one is to convert all output results to the changes in current and voltage. Fig. 2.3 shows an example of the SEE simulation 3D TCAD [108] with a heavy ion hitting the center of the channel. The 3D structure of the transistor and the hit point of the particle are included in the simulation.

To perform calculations with a large number of parameters limits the use of traditional modeling methods. As the number of transistors in a circuit grows, the parameters involved in the calculation will increase by orders of magnitude with unaffordable costs concerning simulation time. At present, a range of simplified methods has been proposed about how to conduct simulations to solve the problems.

2.3.3 SEE simulations at the gate level

SEE simulation can be performed using the standard simulation codes widely used in the integrated circuit (IC) industry for circuit design and optimisation, such as the popular Berkeley SPICE [109], Silvaco SmartSPICE [110], Synopsys HPSICE [111], Orcad PSPICE [112] and Mentor Graphics ELDO simulators [113]. Such circuit simulators solve systems of equations that describe the behavior of electrical circuits (e.g., Kirchoff's laws) [30].

The simulation codes in SPICE are based on the compact model. As the basic components, compact models can describe the static or dynamic electrical behaviour of the different elementary devices (e.g., transistors, diodes and resistors) constituting the circuit. Unlike the model at the transistor level, the compact mod-

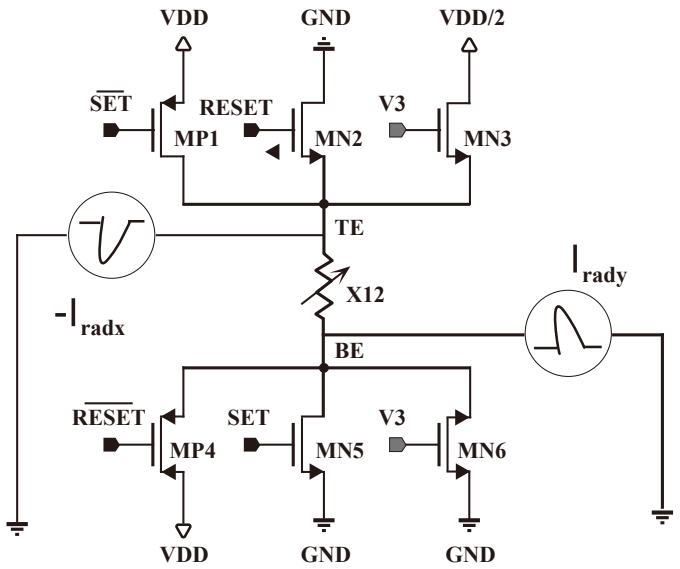


Figure 2.4: SEE injection in OxRRAM memory cell [114].

els are based on the analytical formulas that approximate the measured terminal characteristics, which significantly reduces the complexity of the calculation

By using SPICE, the simulation circuit can be scaled up to tens of transistors. In SPICE, the single-event induced transient is usually modeled as a current source connected at the struck node of the circuit. The accuracy of the transient current used as the input stimulus may have a considerable impact on the precision. Fig. 2.4 shows an example that SEEs is injected in Oxide Resistive RAM (OxRRAM) as transient sources [114], where the magnitude of injection current represents the intensity of radiation.

Because of the increased scale of simulation and the convenience of operating the circuit, SPICE can analyse the behaviours of the circuits, which is impossible in traditional transistor simulation. Fig. 2.5 shows an example that a SEE is injected into a chain of shift registers during clock rising. The timing of SEE injection is controlled so that the behaviours in different circuit states can be observed.

Although today's EDA companies all provide SPICE simulators with different algorithms for IC design, the so-called "fast simulation" on large scale circuits will take days to weeks to complete on a cluster of powerful servers. When it comes to the SEE simulation, the complexity further increases considering the time sequence and injection nodes. Hence, the scale in SEE SPICE simulation seems to be limited

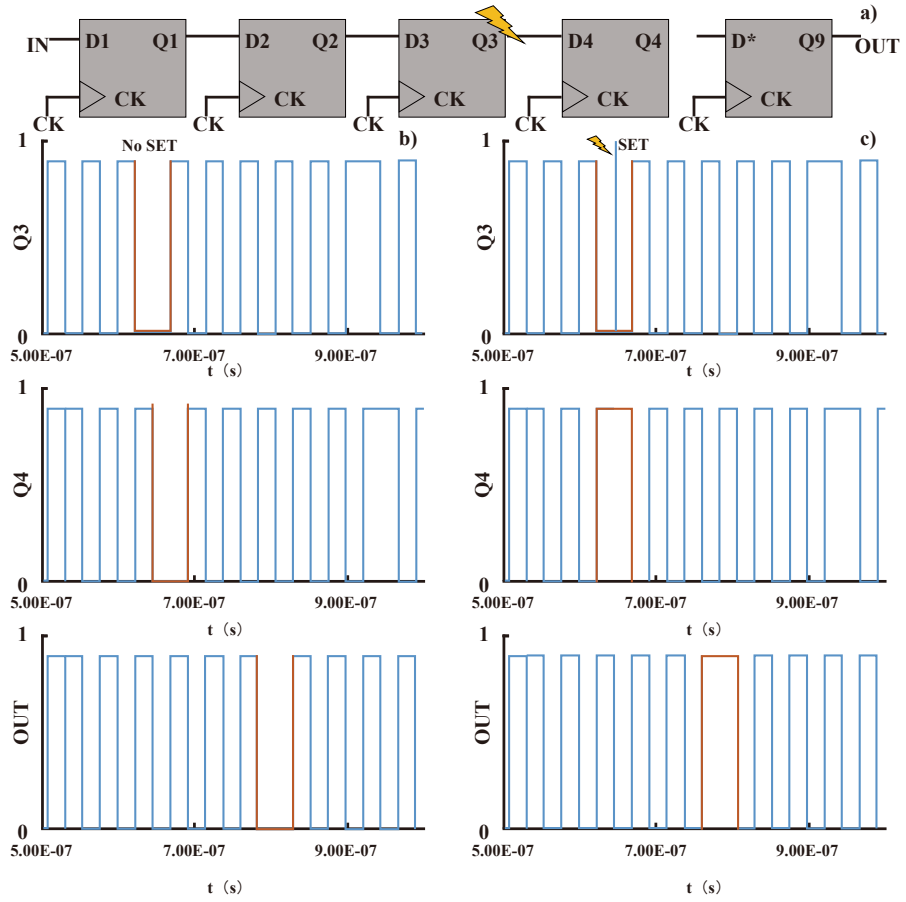


Figure 2.5: SEE injection in a chain of shift registers during clock rising [115].

at the "gate-level" (the circuits with a small number of gates).

2.3.4 SEE simulation in hardware description language

It is common today to use HDL in an IC development flow for those chips with many billions of transistors. After the design specification step in the IC design cycle, HDL is introduced to create behavioural models for representing architectural data flow. Subsequently, individual modules are coded at register transfer level (RTL). This is synthesized later into the connections of available components as offered in the technology library. The library includes commonly used primitive components such as flip-flops, logic gates, memory, and phase lock loops. The conversion output is known as gate level equivalent circuits. Before a new IC design is sent for fabrication, it is normally simulated on HDL compliant simulators to prove design viability.

Simulation semantics are time driven. At each time step, every circuit node is numerically evaluated for convergence. Then the simulator advances to the next time step and repeats the evaluation cycle. This is an intensive computation process. Although SPICE was originally designed as a general-purpose circuit simulator, the numerical details of voltage and current at every time step in SPICE are unnecessary for today's digital circuit functionality verification [116]. For the reason of higher efficiency, the dominant HDLs are higher-level languages (e.g. Verilog and VHDL) focusing on circuit behaviours.

Despite their tantalising efficiency advantages in very large-scale integration (VLSI) simulation, Verilog or VHDL seem to be unpopular in SEE simulations. The reason for this is obvious: SEE has a strong correlation with the physical level, but HDL simulation is abstracted from the behaviour of the circuit, which means that HDL circuits do not correspond to the real physical netlist, and it is not possible to find the correct sensitive node in an HDL circuit. There are some works trying to address the problem. In paper [117], the authors proposed a fault-injection method for HDL design by converting circuits into LUTs based design

for deployment on FPGAs. In paper [118], the authors proposed a SEE injection method by simulating SEE behaviours. However, it only includes three behaviours (i.e., stuck, pulse and bit-flip) without detailed parameters and propagation effects.

Most models in HDL simulations are based on error rates to inject SEEs. However, probability-based HDL models do not describe SETs very well. When these models are used, some basic parameters, such as width, delay and propagation, are not well included in the simulation. It results in the lower accuracy of HDL simulations compared to SPICE and TCAD. Therefore, the HDL simulation of SEE is yet to be investigated.

To sum up, HDL simulations are fast and convenient which make it suitable for large scale circuits. However, compared to gate level simulation, HDL simulations are less accurate. In this thesis, both simulations are unitized. In this way, the SEE simulation can be accurate and efficient for analyzing SEE mitigation designs.

2.4 SEE mitigation methods

SEE mitigation methods for IC can also be discussed at different levels ranging, including physical [119], gate [120], circuit design and systems levels [121]. At the physical level, the basic idea is to mitigate the ionisation effect by improving materials. At the gate level, the physical wiring, channel width and charge size of metal–oxide–semiconductor field-effect transistor (MOSFET) can be improved to enhance the radiation resistance of the circuit. At the circuit design level, the circuit layout and logic design can also affect the response of the circuit to SEE. The relevant sensitive nodes can be spatially separated to avoid being hit together by single particles. At the system level, there are more methods available to reinforce the circuitry. The common idea is to use redundancy to improve the reliability of the system. For example, multiple processors can be used as hot redundancy for data stream processing [122].

For embedded systems, most of the SEE mitigation methods are in circuit design and systems. For instance, TMR, Duplication with Compare (DWC) [123]

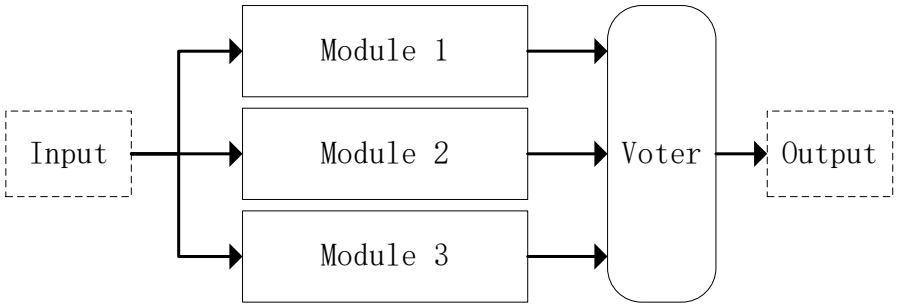


Figure 2.6: TMR fault masking.

or Reduced Precision Redundancy (RPR) [124]. Alternatively, information redundancy techniques can be used to detect and mask failures in certain types of circuits, for example ECCs or Algorithm Based Fault Tolerance (ABFT) [125]. Apart from failure masking, failure recovery techniques can also be used to mitigate errors during run-time. Failure recovery is usually done by refreshing the memory, which is often referred to as scrubbing [126].

2.4.1 Triple modular redundancy

TMR is a recognised technique for improving the reliability of the circuit in a radiation environment. Fig. 2.6 shows the traditional structure of a TMR system. The circuit is replicated three times and a simple majority voter is placed on the outputs. It can be applied to a range of applications from circuit modules to top systems. For example, TMR was used on an 8051-like micro-controller design and shown to completely address design failures due to single-bit configuration upsets [127]. However, a TMR also incurs a high overhead cost. Firstly a TMR design requires at least three times hardware resources for redundancy. Moreover, additional logic is required to implement the voting circuits. Some studies have shown that TMR can require up to six times the area of the original circuit [128]. Secondly, TMR can negatively affect timing, because the voters inserted follow combinational logic, thereby increasing the path lengths. Thirdly, the extra resources ultimately require more power [129].

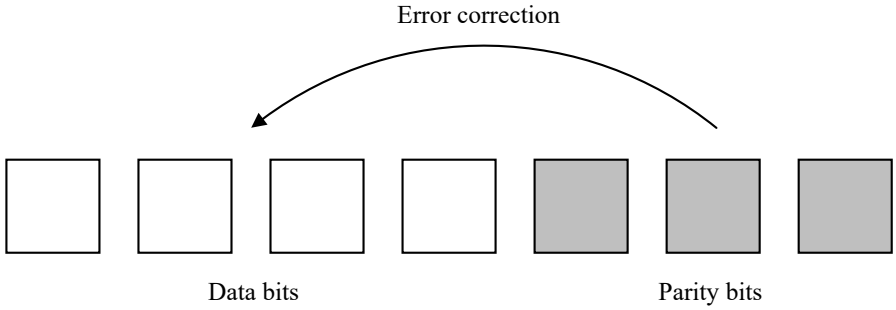


Figure 2.7: Hamming (7,4) is a linear ECC that encodes four bits of data into seven bits by adding three parity bits.

2.4.2 Error correction codes

ECC is widely known as an anti-interference encoding strategy to enhance the reliability of memory devices and communication systems. The idea is to use extra bits to store redundant information. Compared to TMR, ECC technology requires fewer hardware resources and less memory space. ECC is therefore widely used in memory devices or communication systems as a SEE mitigation technology for fault detection and correction [130, 131, 132, 133, 134, 135]. Modified Hamming codes and Hsiao codes [136, 137] are the most widely used single-error correctable and double-error detectable (SEC-DED) codes. Fig. 2.7 shows the distribution of bits in Hamming(7,4) codes. However, when circuits work for a long time in radiation environments, there might be multiple errors that exceed SEC-DED corrective capability. To deal with multiple errors, Bose–Chaudhuri–Hocquenghem (BCH) code and Reed-Solomon (RS) code [138, 139] utilise additional parity-bits to carry out correction. However, the capability of these ECC is still limited [140]. If the system is incapable of correcting the existent bit flips in a limited time, then the bit flips can eventually accumulate and may cause unrecoverable errors. Moreover, with the increasing capability of the correcting code, the complexity of the correcting code will also significantly rise concerning the hardware resources and memory space.

2.4.3 Scrubbing

Scrubbing or refreshing is an effective error mitigation technology for memory devices to resolve the accumulation of errors [40, 41, 42, 43, 44]. The basic idea is to overwrite memory cells with the correct data when an error has been detected. A conventional scrubbing scheme consists of reading, detection and rewriting. Compared to the simple ECC, the systems equipped with scrubbing techniques can be applied to check each memory unit periodically. Each unit may be checked frequently, before the accumulation of multiple bit flips, in case these errors are not correctable. Therefore, scrubbing is appropriate for data retention in memory (e.g., DRAM [48], STT-RAM [49], NAND flash [50]). Because the FPGA structure is also based on RAMs, it is applicable to FPGAs.

Many works have been done for applying refreshing to memory. Typically, the works can divided into three categories: 1) refreshing cells, external scrubber and internal scrubber. There are two aspects can be used to compare refreshing methods: 1) the number of failure occurrences that are to be expected either during a specific mission time frame or the overall mission lifetime and 2) trading-off power, area and reliability overheads [141].

2.4.3.1 Refresh memory cells

Paper [142] presents a refresh circuit for resolving the soft-error failures that occur in the memory cell. Two voltage-sense amplifiers are added to detect the errors in bit lines. When errors occur, voltage-sense amplifiers can trigger data refreshing operations.

Fig. 2.8 shows the architecture of the refreshing memory cell. The refresh circuit consists of SA1 (1-input), SA2 (2-input) and an error detection unit. SA1 and SA2 are the voltage-sense amplifiers used to detect the RAM state of the register cell. By comparing the voltage of different locations, different states of the memory device can be recognised.

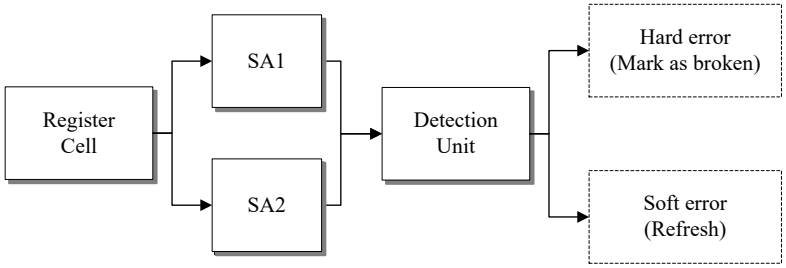


Figure 2.8: Adding a refresh circuit to the memory cell (*SA* indicates voltage-sense amplifiers) [142].

2.4.3.2 External scrubber

Typically, the external scrubber [61, 62, 63, 31] is independent of the target devices. For example, in paper [61],a separate FPGA is programmed as the external scrubber for Geostationary Mission. In this paper, a programmable read-only memory (PROM) was used to save the initial program. Overwriting the contents of configuration memory on a periodic basis can prevent system failure due to error accumulation.

It is a post-configuration write operation in the configuration memory of Xilinx FPGA without disrupting system operation. The basic block diagram of the system is shown in Fig. 2.9. There are two FPGAs in this scheme: 1) a targeted SRAM FPGA required to be hardened and 2) an external scrubber implemented in a separate FPGA.

During the power stage, the configuration parameters of Xilinx FPGA are first initialized to their default value. Configuration takes place after the proper initialization of the configuration parameters. A read-back test is then performed immediately after successful configuration to ensure there are no hard errors in the target devices. The scrubbing operation on the FPGA resources is conducted only after a successful read-back test. The contents of configuration memory are refreshed at an interval of six minutes.

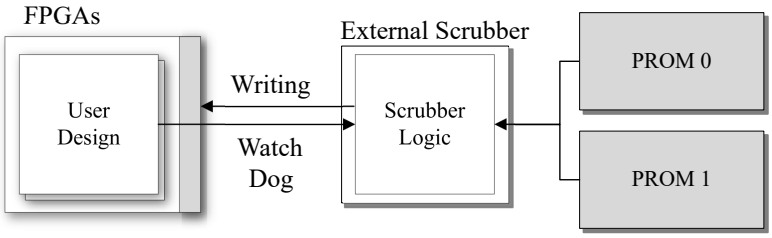


Figure 2.9: Refreshing the configuration memory in FPGAs by using external scrubber and PROM.

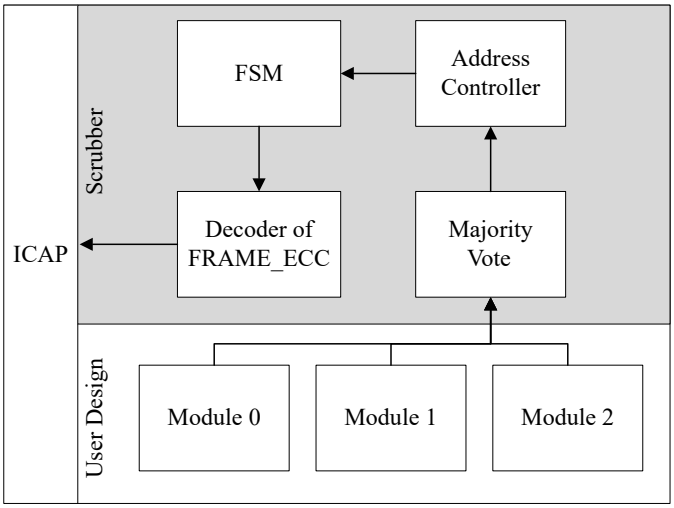


Figure 2.10: The internal refresh scheme by using ICAP to access the configuration memory [64].

2.4.3.3 Internal scrubber

Another method is to use an internal scrubber in the scrubbing scheme. In papers [64, 143], the authors proposed scrubbing methods for FPGA configuration RAM. The papers apply the internal scrubbing method through the internal configuration access port (ICAP) to read and write configuration RAMs.

The basic architecture [64] of the scrubbing platform is illustrated in Fig. 2.10. The FRAME_ECC logic calculates the syndrome value according to the bits in one frame including the ECC bits by reading frames from the configuration RAMs. The majority voter is designed to detect the unexpected outputs in the user design. Once the errors are detected, the FSM in the srcubber is triggered to refresh the configuration RAMs. This method is suitable for configuration RAMs. In the configuration RAMs, bits are static, so there are no additional read or write operations to occupy the access ports.

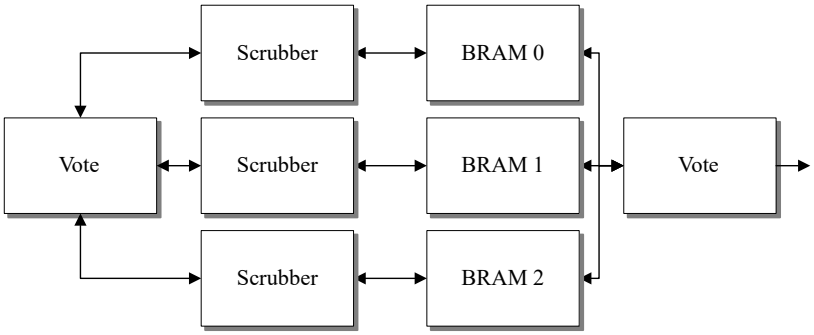


Figure 2.11: Using dual-port RAMs to implement scrubbers in BRAM systems [144].

However, scrubbing methods can be challenging in BRAMs, if all available ports are in use. In paper [144], the authors proposed a scheme combined with TMR and scrubbing methods, as shown in Fig. 2.11. In this paper, dual-port BRAM modules are used to mitigate error in the LEON3 processor. The processor uses only one port or the BRAM modules, which leaves the other port for scrubbing operations. In this scheme, BRAM and scrubber are tripled and the correct value is determined by voting between the redundant copies. However, this method will still occupy the BRAM access ports. To apply this method, single-port BRAMs are replaced by dual-port BRAMs. If the dual-port BRAMis already in use, then the method in paper [144] will be limited.

2.5 Intelligent hardware system

The radiation effect is not the only challenge for intelligent systems in extreme environments. The implementation of the hardware system for AI algorithms is another challenge.

Modern AI started from the experiment with electronic neural nets [145] in the 1950s. Compared to those traditional methods of machine learning and pattern recognition, the key advantage of DL is that it does not require the extraction of manual features. When trained for a particular task, DL systems learn multi-layer hierarchical representations of the suitable data for the task automatically. However, the idea of backpropagation requires the use of neutrons with continuous

non-linearities (e.g., sigmoids), which was not practical before the 1980s when the workstation performance was under one million floating-point multiply-accumulate operations per second [146].

Due to the rapid advancement of digital technologies in recent years, deep neural networks (DNNs) have emerged as a key technique in modern AI, which enables high accuracy for many applications [147]. However, there remain some challenges facing the practical application of DNNs, for example, intensive arithmetic operations and memory bandwidths[148] are required for resource-constrained edge devices, which hinders general CPUs from achieving the expected performance. Therefore, various hardware accelerators have been applied to improve the throughput of DNNs, such as application-specific integrated circuits (ASICs) [149], FPGAs and graphic processing units (GPUs) [150].

To sum up, the challenges of the AI hardware for embedded systems can be concluded in four aspects: 1) power and energy efficiency [51]: the biggest challenge for an embedded system is power efficiency, as most embedded systems are powered by batteries. With a given battery, the lifetime of an embedded system primarily depends on the amount of power consumption. At present, these neural networks run on powerful GPUs (e.g., Nvidia 3090) that dissipate a huge amount of power in typical application scenarios, which is not suitable for embedded systems. 2) performance [52]: With more powerful AI applications deployed on edge devices, the demand for computing power is becoming increasingly higher. For the reason of the costs, embedded systems cannot simply increase their computing power, by adding servers as in cloud environments or using the most powerful chips. 3) limited hardware resources [53]: For current multiple-task systems, edge devices are increasingly required to operate more than one AI task. However, the constrained hardware resources (e.g., DSPs and RAMs) are typically insufficient to support multiple tasks, which poses challenges for resource allocation. 4) heat dissipation [151]: unlike cloud servers, which can rely on water cooling and environment control methods to reduce temperatures, embedded systems are often

unable to use large heat sinks due to weight and size constraints. Hence, embedded systems are less efficient than others at dissipating heat and cannot use chips with high heat and power consumption, which further limits the performance of embedded devices.

To build an intelligent hardware system, it is critical to overcome those challenges in two aspects: 1) deployment of DL models and 2) deployment of hardware accelerators.

2.5.1 Deployment of DL models

With the development of unsupervised, self-supervised, weakly supervised and multi-task learning, the DL network is getting increasingly larger. Despite the improvement of semiconductor technology, current chips still face difficulties in meeting the requirements of rapidly growing computing power for neural networks. Furthermore, in embedded systems, where power and resource hardware resources are limited, it remains a challenge even nowadays to deploy high-performance accelerators and neural networks, not to mention in harsh environments.

The earlier efforts on deploying DL models can be divided into cloud and edge categories. Compared to edge and embedded devices, cloud devices have many advantages in power and computing performance, which makes it easier to deploy networks. However, cloud computing means that data must be sent to severs, which incurs extra costs. Firstly, with the increasing number of mobile and embedded devices, the amount of data generated by embedded devices has gradually exceeded the processing capacity of the cloud. By 2020, 50 billion IoT devices had been connected to the internet. On the contrary, it was also estimated that nearly 850 zettabytes (ZB) of data were generated outside the cloud [152], while global data center traffic was only 20.6 ZB. Secondly, the data transmission between cloud and edge devices will induce delay. However, mangy new types of applications like cooperative autonomous driving have strict requirements. The physical distance makes it impossible to shrink the delay. Finally, cloud technology heavily depends

on the internet connectivity. Hence, it will be inappropriate, if the edge systems have to operate in harsh environments like the radiation environments where network connection cannot be continuously maintained. Edge computing emerges as an attractive alternative. By comparison, in the offline mode, the training task is still performed in the cloud, but the trained model is sent to the mobile or edge devices to conduct inference locally (edge-side inference). However, the trained deep models may have a large number of parameters and require complex computations, which poses great challenges for the limited resources in embedded systems.

2.5.2 Hardware accelerators for DL inference

There are many hardware accelerators for AI inference. According to the papers published in recent years [153], the hardware used for DL implementation in research can be divided into a number of categories. Fig. 2.12 shows the overall distribution of the papers according to this categorization. As can be seen from the figure, the hardware accelerators used for neural network deployment are mainly based on FPGAs, GPUs and ASICs. The differences are listed in Table 2.2.

The ASICs for DL applications (e.g., AI chips) have performance and power consumption advantages. Compared to the powerful general-purpose chips (e.g., CPUs), the ASICs for DL are designed to have greater parallel computation power and memory bandwidth. Hence, many companies have developed or are developing their own AI chips. For example, IBM launched its "neuromorphic chip" TrueNorth AI in 2014 [154]. TrueNorth contains 5.4 billion transistors, 1 million neurons and 256 million synapses, so it can efficiently perform deep network inference and deliver high-quality data interpretation.

However, although it is a great option to implement DL accelerators, the unaffordable price prevents ASICs from being used in the research or scenarios (e.g., space and nuclear power plants) with small production quantities. Unlike ASICs, FPGAs are flexible for changes and updates after implementation. FPGAs are semiconductor devices based on a matrix of configurable logic blocks connected

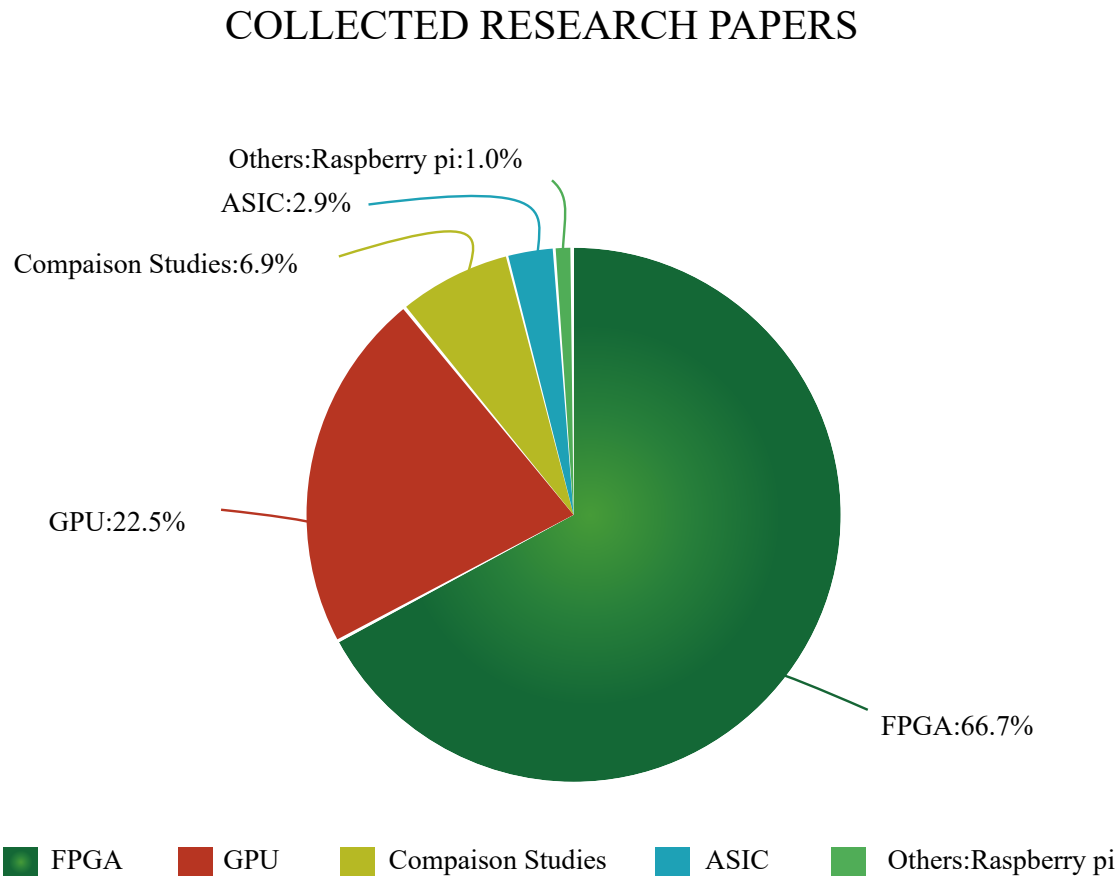


Figure 2.12: Hardware implementation platform distribution in the collected research papers [153].

via programmable interconnects. FPGAs can be reprogrammed to the desired application or functionality requirements after manufacturing. This feature distinguishes FPGAs from ASICs because ASIC design cannot be altered after fabrication. This feature makes FPGAs much more flexible for changes and updates after implementation. Furthermore, FPGAs usually take less implementation time than ASICs, making them ideal for prototyping and validation.

FPGAs used to be selected for low speeds, complexity and volume designs in the past, while current FPGAs quickly push the 500 MHz performance barrier. FPGAs are considered as to be an easy and suitable alternative solution to DL acceleration with unprecedented logic density increases and other features, such as embedded processors, DSP blocks, clocking and high-speed serial at a low price. Despite the slightly higher power consumption than ASICs, the high flexibility and rapid prototyping capabilities make FPGAs the dominant devices in research.

GPU is also known as the display core. It is a chip dedicated to the image and graphics-related processing on personal computers, workstations, game consoles and mobile devices (such as tablet computers and smartphones). One feature of GPUs is their capability to perform intensive scalar and parallel computing, which makes them a promising option for DL acceleration. Another feature of GPUs is that they are designed with a fast memory hierarchy with direct memory access to resolve memory bandwidth issues. This allows higher transfer rates while reducing time costs.

Unlike FPGAs and ASICs providing hardware level solutions, GPUs provide software level solutions. FPGAs and ASICs provide higher flexibility during the design stage and accelerators can be particularly optimized for DL applications. By contrast, GPUs are severely restricted by the existing underlying hardware. It results in the lower performance or power efficiency of GPUs in some cases.

The implementation cost for both GPU and FPGA is considered medium compared to ASIC. This is because of the high fabrication cost of ASICs. Although fabrication costs are also high for GPUs, a large volume of production lowers the

per chip price. On the other contrary, because GPU manufacturers tend to use advanced processes, the performance of GPUs is rather excellent at the same price. Therefore GPUs are popular in the cloud and server-side GPUs. Moreover, the power-efficient embedded system tends to use FPGAs.

Therefore, there are sensible reasons to choose FPGAs for DL acceleration in harsh environments when the budget is limited, and high power efficiency is required. The radiation environment is such an environment where hardware systems place a higher demand on performance, weight, volume, energy efficiency and adaptability. Considering that most of the hardware systems for radiation are highly mission-dependent, the production number is not sufficient to cover the production of ASICs. By contrast, FPGAs can take advantage of the flexibility to build affordable hardware systems. The potential of FPGAs has been noticed for a while as the co-processing hardware for image and signal processing in harsh environments (such as satellites).

Table 2.2: Hardware platforms for AI acceleration [153].

	FPGA	ASIC	GPU
Required technical skills	VHDL/Verilog, FPGA development environment	VHDL/Verilog, CAD tools for chip fabrication	GPU high-level programming skills
Implementation time/Time to market	Medium	High	Medium
Implementation level	Hardware	Hardware	Software
Implementation fexibility	High	High	Low
Flexibility for changes after implementation	High	Low	High
Cost	Medium	High	Medium to low
Energy efficiency	Medium	High	Low
Area efficiency	Low	High	Low
Performance	Medium	High	Medium to low

The power efficiency of GPUs in specific computation tasks is normally high. However, due to the architecture for general purpose, the overall power efficiency in embedded system is low.

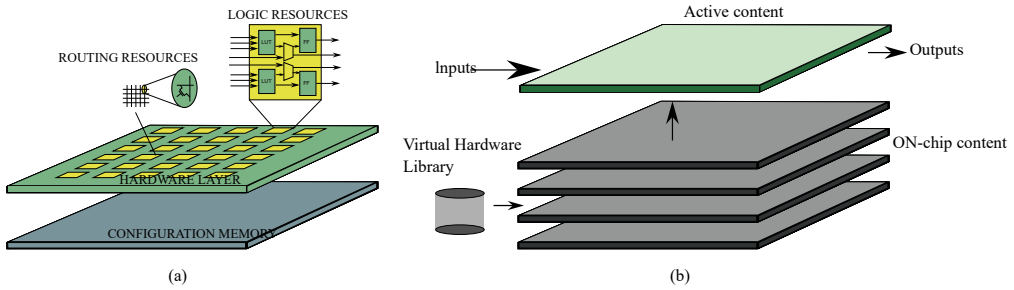


Figure 2.13: (a) Typical FPGA architecture consisting of configuration memory and hardware logic layer. (b) Multi-context FPGAs increase effective logic capacity by using more than one configuration memory plane. [155]

2.6 Partial reconfiguration of FPGAs

As their capabilities and sizes have increased, FPGAs have been used in a wide range of domains, where their reconfigurability offers a distinct advantage over the implementation of ASICs. This feature offers FPGAs unmatched hardware flexibility. Moreoever, modern FPGAs have the capability to dynamically reconfigure partial regions, which is referred to as partial configuration (PR). Through this feature, the hardware function can be changed at runtime in response to the application requirements [155].

There are many reasons for which the partial dynamical configuration on a single FPGA is advantageous. These include: 1) reducing the size of the FPGA required to implement a given function, with the consequent reductions in cost and power consumption; 2) providing flexibility in choosing the algorithms or protocols available to an application; 3) enabling new techniques in design security; 4) improving FPGA fault tolerance; 5) accelerating configurable computing; and 6) delivering updates (fixes and new features) to those deployed systems.

2.6.1 Mechanism of partial reconfiguration

Conceptually, all FPGA devices can be considered as being composed of two distinct layers: the configuration memory layer and the hardware logic layer [156], as shown in Fig. 2.13 (a). In the hardware layer, there are plenty of computational hardware resources, including LUTs, flip-flops, DSPs, memory blocks, transceivers

and so on. The hardware layer also contains some routing resources and switch boxes that allow components to be connected to form a circuit. Both the computational resources and routing resources can be controlled by the RAMs in the configuration layer. When FPGAs are powered on, the configuration memory will keep the configuration details via a binary file called configuration file or bitstream. This binary file contains all the information that determines the implemented circuit, such as the values stored in the LUTs, the initial set and reset status of the flip-flops, the initialisation values for memories, the voltage standards of the input and output pins and the routing information for the programmable interconnect to enable the resources to form the described circuit. The function implemented by the hardware logic layer is thus wholly determined by the values stored in the configuration memory [155].

PR refers to the modification to the partial configuration information while the others remain unchanged. As the configuration memory is modified, the corresponding regions in the hardware layer will also be changed. Dynamic reconfiguration (DPR) is different from PR. It refers to the modification to the hardware designs during the runtime without a reset. Today, the mainstream FPGAs (e.g. Xilinx) support PR functionality via dedicated internal interfaces (e.g. ICAP) to rewrite the configuration RAMs. Due to the lack of drivers and software support, DPR used to be applied in only a few projects. For now, the latest Xilinx FPGAs support a new feature called dynamic function exchange (DFX), packaging DPR and corresponding software.

2.6.2 Applications of partial reconfiguration

2.6.2.1 Dynamic system adaptation

The systems are applied to dynamically modify their behaviour for various environments. It is particularly useful in applications in which the high computational requirements exceed what software can provide. Due to the high flexibility of FPGAs with PR, It is popular in such applications as software defined radio [157].

With the use of PR, various hardware filters for radio processing are implemented according to the requests. In paper [158], a cognitive radio design is proposed. It can modify the functionality at runtime, making operations more effective in unknown environments. Another example is the applications with adaptive data clustering (e.g., K-means clustering and support vector machines), where kernels are selectively modified with multiple kernels hosted in the same FPGA [159, 160]. PR allows individual classifiers to be adapted to meet the need for many identical classifiers in the system.

In automotive applications, the potential of PR has been observed. Because the vehicle's life is normally much longer than the updating cycle time of algorithms, the ability to continuously update hardware is very attractive. In addition, FPGAs with PR provide the capability of rapid development in driver assistance applications. In paper [161], the authors propose a system combining CPUs and FPGAs. The CPU is used for control and management in this system, while FPGA is used for image processing.

Within space applications, PR is widely used because of resource constraints. In paper [162], the authors propose a design on Virtex-4 FPGA for a well-established network-on-chip protocol in the space community. In this works, PR is used to implement hardware functions for different purposes such as credit-based flow control, the detection of link errors, link error recovery and hot-plug ability.

FPGAs with PR are ideal for building a hardware system for changing scenarios, as hardware functions can be modified on the fly according to the requirements. Most of the PR examples are application-specific. The flexibility improves the adaptability of FPGAs across a range of domains.

2.6.2.2 Partial reconfiguration for radiation

Essentially, PR is based on a similar idea to scrubbing: using new data (hardware bitstreams) to overwrite the previous data in the memory (configured RAM). Therefore, PR can be used to mitigate errors in FPGAs, as SRAM-based FPGAs

are highly vulnerable to SEUs. PR has been proposed as a solution to mitigate SEUs in SRAM-based FPGAs because it provides an auxiliary path to the configuration memory. In paper [163], the authors partition the FPGA into a number of regions to isolate SEU errors, and then apply duplication with comparison to ensure correct computation. If the outputs of identical modules are different, there is an error. Once an error is detected, that region is reconfigured. In paper [164], the authors proposed a scrubbing method for configuration RAMs. The bitstream is stored in a radiation-hardened memory and refreshed by a configuration controller. Some other methods enable redundancy through PR. In paper [165], redundant electronic control units (ECUs) are implemented in PR regions. When there are detected errors, the corresponding region is reconfigured, and the redundant ECU performs as a backup.

2.6.2.3 System cost reduction

PR can help reduce overall system cost by enabling time multiplexing of the functionality on a smaller chip instead of a larger FPGA. Because the energy consumption of smaller chips is generally low, this also helps reduce the overall cost [155]. PR has been proved to be useful for media-related processing, as they are resource-constrained applications. In paper [166], PR is used for the implementation of an MP3 decoder. In paper [167], the authors proposed a PR-based scalable H.264/AVC deblocking filter architecture. The filter adapts to different user requirements at runtime. In paper [168], the AdaBoost algorithm for human detection is implemented by using PR. There are two computationally intensive tasks: integral image computation and feature extraction/decision. By dynamically implementing the corresponding hardware function, the required hardware resources can be significantly reduced.

To sum up, when the requirement is low cost, the use of PR can be divided into two categories. One is the deployment of different hardware functions through time-division multiplexing, thus reducing the consumption of resources. The other

one is the scalable hardware design for different scenarios.

2.6.2.4 High-performance computing

Through PR, FPGA can also provide the capability of computing acceleration within the general-purpose computing systems. In paper [169], a design of high-performance reconfigurable computing is proposed. The FPGA takes on a significant proportion of a large scientific application, with PR allowing the fabric to be used in different computational steps during runtime, when the applications are too large to fit on a single FPGA. In paper [170], the authors used PR for an autonomous computing system with placement and routing implemented on the FPGA fabric itself, which allows the FPGA to create new circuit bitstreams for self-modifying hardware. In paper [171], the authors proposed a real-time operating system for highly adaptive efficient and dependable computing on FPGAs. In this system, each task can apply for hardware resources to build accelerators. However, the accelerator is not adjustable for the same task.

Another very popular scenario of FPGA is cloud computing [172]. For example, Microsoft used FPGAs to deploy accelerators for Bing search [173]. In paper [174], the author proposed a scheme to manage multiple PR blocks as virtual FPGAs, which can make it easier to manage resources. In paper [175], the authors provide a comprehensive survey on FPGA-based hardware accelerators for cloud computing.

FPGAs have also been used in neural networks, where algorithms are constantly changing and upgraded. PR can provide adaptation to AI tasks for the inference computation . In paper [176], the authors proposed an online evolvable pattern recognition system, where the classification module dynamically evolves using PR. In this paper, a processor configures a PR region with different classification modules to evaluate the input pattern.

2.6.3 Challenges of FPGA system with partial reconfiguration

PR has shown significant advantages in such applications as autonomous driving, communications and space. This also includes AI hardware systems under nuclear radiation. However, there are still many challenges in using PR to implement such an AI system.

Firstly, it remains challenging to implement a system with multiple accelerators on the current commercial FPGAs due to the limited support in tools and drivers. To build such a system, developers need to overcome the limitations of existing flows, including hardware design, cabling planning, driver implementation, system calls and application optimisation. Even though such FPGA providers as Xilinx have provided some of the automation tools, it is still very difficult to introduce an abstract concept into a specific system.

Secondly, more research is required on how to manage multiple accelerators on FPGAs appropriately. When PR techniques are applied, there may be multiple accelerators in FPGA systems. It presents a challenge for optimising the system with multiple accelerators and applications. In addition, during PR, the switching of hardware functions should avoid any impact on running applications.

The third issue is the scheduling of PR to achieve the best performance. Obviously, it is not free to conduct reconfiguration. Although Xilinx claims that their FPGAs can configure small blocks in milliseconds, the reconfiguration of the larger accelerator and the switching of the device tree will take seconds. From an application perspective, switching too frequently is likely to outweigh the benefits.

Finally, it is still a question of how to build the autonomously self-adaptive systems that combine reconfiguration capability with intelligence and the ability to adapt to bitstream capabilities, which will be discussed in this thesis.

2.7 Conclusion

As we can conclude, the challenges fall into two categories: SEE and DL. Specifically, the challenges include SEE simulation for large circuits, SEE mitigation for extreme radiation environments, limited resources for AI inference, flexibility for multiple tasks and power efficiency for long-term operation.

Firstly, SEE simulations require more computational resources and time, if we want more accurate results. Due to the complexity of SEE and the increasingly large size of existing circuits, it has become a challenge to design a large-scale circuit suitable for the verification and simulation of anti-SEE performance. On the one hand, transistor or gate-level tools (e.g., SPICE) are accurate enough for small-scale circuits. However, The requirement for a large amount of computational power prevents them from being used for large circuits. On the other hand, although the HDL can simulate the behaviours of large-scale circuits, it lacks details in SEE to analyse the SEE propagation. To address the issue, in Chapter 3, I will present a new SEE simulation scheme combining the advantages of SPICE and HDL simulation.

Secondly, in strong radiation environments such as a nuclear power system, errors can accumulate much faster than in traditional radiation environments. In this case, the typical error mitigation methods intended for the space environment may not be able to meet specific requirements. The TMR technique incurs a high overhead cost. The ECC has limited capability to deal with accumulated errors. The scrubbing methods have the potential to mitigate SEUs. However, it requires extra hardware support (e.g. additional IO ports), limiting the range of applications. Due to the long-term effects in strong radiation environments, the devices will only work for hours and be replaced. The error mitigation performance and costs will be more important than power consumption. Therefore, in chapter 4, I will present a portable scrubbing design requiring no additional IO ports.

Thirdly, when it comes to AI systems, the biggest challenges are the high requirements for DL deployment, including weight volume, power consumption,

price, radiation resistance and complex scenarios. The reconfigurable feature makes FPGAs a perfect option. On the one hand, reconfigurable technology has proven system reliability in radiated environments. On the other hand, the reconfigurable feature provides FPGAs with adaptability for various scenarios. Some of the works have proposed to utilize reconfiguration features to improve and solve the problems in the hardware aspect. However, I think the system can be further improved by adopting the concept in both hardware and software. Therefore, an adaptive hardware resource management system based on the reconfiguration feature is designed for multiple AI tasks in harsh environments. It can improve comprehensive performance, including reliability and power efficiency. Furthermore, the idea of the dynamical switch is adapted at the software level. A series of DL networks are pre-built in different sizes. When facing various work scenarios, the networks can be dynamically switched in real-time to adapt to different performance, power and accuracy requirements. The work is presented in chapters 5 and 6.

Chapter 3

Simulation of SEEs in integrated circuits

3.1 Introduction

Today's strategies to evaluate the effects of SETs and SEUs on the integrated circuits of different designs still have certain limitations. Real world radiation experiments (e.g., outer space experiments or nuclear radiation facility experiments) demand sophisticated mechanical setup and they are expensive as well. Indeed, they can provide "high level" results like "error rates" and "sensitivity to radiation" of different chips, but the detailed radiation effects on various circuit modules are hard to find, due to the complexity of the integrated circuits.

Existing transistor simulation tools, which are transistor-based analog circuit simulation, can produce accurate results by considering detailed physical parameters (e.g., capacitance, resistance, current, voltage, and 3D structure) in simulation. However, they will take days to weeks to complete on a cluster of powerful servers. When it comes to the SEE simulation, the complexity further increases considering the time sequence and injection nodes.

HDL level simulation might be used to quickly assess the SEE mitigation performance of circuits. It carries out the error injections in the data stream or

memory units based on the probability of bit-flips [29, 28], which makes HDL simulations lacking in detail when analysing SEEs (especially SETs) in some specific circuits. Moreover, the HDL design only describes the behaviour of the circuit and does not reflect the actual size of the circuit, so the accuracy of the SEE injection based on the error rate is also questionable.

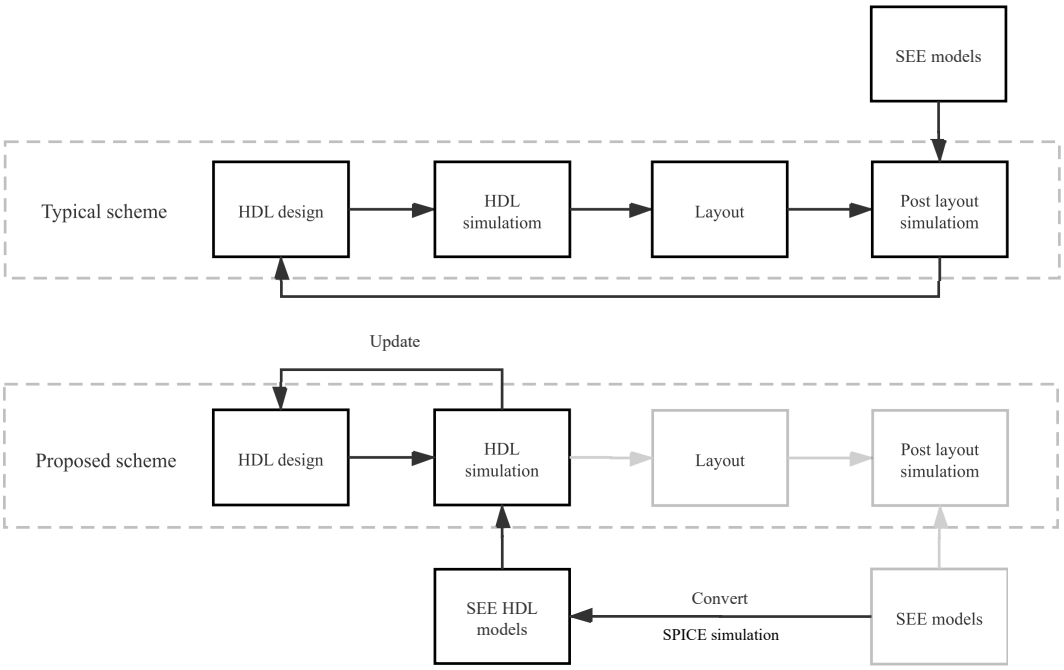


Figure 3.1: The comparison between a typical SEE evaluation scheme and the proposed scheme.

To utilise the advantages of SPICE simulation and HDL simulation tool, a new SEE simulation scheme is proposed [177, 178], to offer a fast and cost-efficient method, requiring less time and computational resources for SEE simulations, to evaluate and compare the performance of large-scale circuits. The scheme consists of the following features: 1) creating SEE behaviour models using SPICE or TCAD, 2) generating HDL netlists and injection scripts using the HDL designs, and 3) using the SEE behaviour models in HDL simulations to study and compare the performance of the circuit designs. 4) Based on the results, update the hardware designs and rerun the simulation operations.

Validation of the proposed scheme has been conducted using a 180nm logic gate library from the SMIC [179] to build a set of SEE models for the simulation. The

baseline circuits that we used in this work, are a series of commonly used circuit designs from the ISCAS89 benchmark [180]. Experimental results exhibit that the proposed scheme can handle SEE simulations for more than 40 different circuits with the sizes varying from 100 transistors to 100k transistors. Additionally, using low-level SEE behaviour models, the proposed simulation scheme is able to provide details of the error propagation and vulnerable logic design in the HDL simulation. As shown in Fig. 3.1, compared to the typical SEE evaluation scheme, the proposed scheme provides a rapid solution to analyse the vulnerable modules in the logic designs before post layout simulations.

3.2 The proposed SEE simulation scheme

The workflow of the proposed scheme is shown in Fig. 3.2. Due to the complexity of the large-scale circuits, it is very difficult to directly undertake physical simulation for VLSI. Because all digital circuits are made up of basic logic components, a set of SEE behaviour models for logic components will be generated in the proposed scheme. The SEE behaviour models are generated from SPICE simulations based on the gate libraries and SEE SPICE models. In this scheme, the SEE behaviour models can be reused for different HDL designs so that there is no need to re-conduct the generation of the SEE behaviour models. It also provides the flexibility to customise the gate libraries and SPICE models to achieve more accurate results.

3.2.1 Process of SEE simulation

The process of the SEE simulation for VLSI includes three steps: 1) generate the SEE models for basic circuit units, 2) build HDL netlists for the large-scale circuits and 3) carry out simulation and analysis.

The first step is the most important step in the proposed scheme. The accuracy of the SEE models determines the accuracy of the simulation results. On the physical level, SEEs are transient currents caused by the particles, which can be

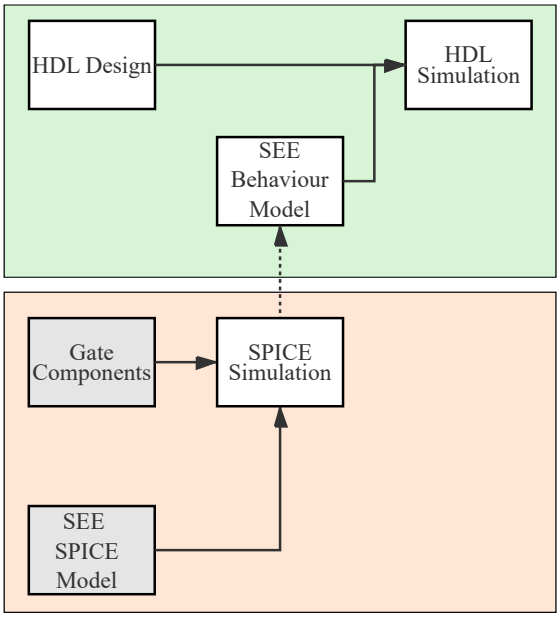


Figure 3.2: The workflow of the proposed scheme. SPICE simulation part can be skipped, if SEE behaviour models have been generated.

affected by the capacitance between the transistors, the resistance of the circuit wire, the semiconductor material of the chips and the intensity of the radiation. In this step, we use transient current sources in the SPICE to simulate the SEEs. By injecting the transient current at the sensitive circuit nodes, the changes of voltage will be observed on the output ports.

The second step is to build HDL netlists of large-scale circuits. Compared to the small circuits, the large-scale circuits are much more complicated. In a circuit with thousands of transistors, there might be millions of possibilities which need to be simulated. It is not viable to do that in SPICE. We replace the basic logic components with the corresponding SEE models. Therefore, HDL netlists should correspond with the structure of the physical circuits. However, normal HDL designs do not indicate the physical circuits directly. Here, we generate the SPICE netlists by EDA tools and components library, which can be converted into HDL netlists. A script tool is also designed to replace the basic components and generate new HDL netlists automatically.

The third step is to undertake simulations and analysis. Considering that the large-scale circuits may contain hundreds of inputs and outputs, the simulation

bench files are also generated by scripts automatically.

In this final step, the simulation test-bench codes contain three parts: 1) input generation, 2) SEEs injection and 3) error detection. The input generation part is used to generate specific input data streams to represent different software programs. The injection part is used to control the SEE injection. The error detection part is used to monitor the output and analyse the errors. In this scheme, two identical designs are used as a reference module and an experiment module respectively. When the simulation starts, both modules are monitored. By comparing the outputs, the errors can be then detected. The number of errors was noted for subsequent analysis.

3.2.2 The basic component for SEE simulation

In order to generate the SEE model of those basic logic components, a concept of the basic circuit unit is introduced in this work. The basic circuit unit is a "black box", with inputs and outputs and relationships between the inputs and outputs which can be represented by an equation. When SEEs occur in this unit, the effects of the SEE (i.e., digital pulses and bit-flips) can also be represented through this equation.

Therefore, we can build HDL SEE models of the integrated circuits by using equations. The model should include two parts: 1) the behaviours of the circuits without errors and 2) the effect of SEEs on operational circuits. In digital circuits (including combinational and sequential circuits), the output will only be affected by the current inputs and states. The equation of the normal behaviour of the circuits can be represented as follows:

$$O_n = f(S_n, I_n), \tag{3.1}$$

where O_n represents the current output of the circuit, I_n represent the current inputs and S_n represents the current state of the circuit.

As indicated in Chapter 2, SETs and SEUs are two main types of soft errors in SEEs. SETs can generate digital pulses that will propagate through the following circuits. SEUs can cause bit-flips and they will change the current state of the circuits. I represented SETs and SEUs in separate equations.

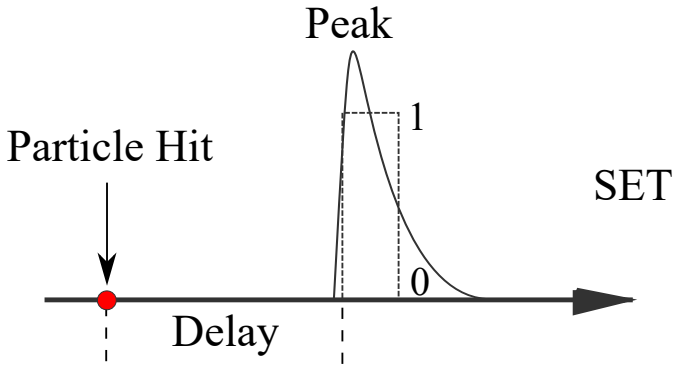


Figure 3.3: The original transient SEE pulse and the generated SEE digital pulses.

When charged particles strike a sensitive area in the circuit unit, a transient current will be generated, which will cause a voltage pulse. If the peak voltage exceeds the threshold voltage of value "1", then a digital pulse will be generated. The waveform of the voltage pulse and the corresponding digital pulse is shown in Fig. 3.3. When one charged particle strikes a transistor in the circuit, the pulse will appear at output ports after a delay. The duration time of the signal at the output ports is the width of the pulse. According to the different current states and the propagation paths, the output signal can be a positive pulse, a negative pulse or no pulse. For the output of one SET on transistor K, the equation can be then represented as follows:

$$O_{SET,k} = \begin{cases} 0 & 0 < t < T_d, \\ f_{SET,k}(S_n, I_n) & T_d \leq t < T_d + T_w, \\ 0 & T_d + T_w \leq t, \end{cases} \tag{3.2}$$

where $O_{SET,k}$ represents the output signal of SETs on one transistor, $f_{SET,k}(S_n, I_n)$ represents the outputs of pulses which can be positive pulse, negative pulse and no change and t represents the elapsed time of the events, T_d represents the time for the pulses to propagate to the outputs from the strike point and T_w represents

the width of the pulses.

The possibility of triggering the $O_{SET,k}$ depends on the radiation cross section of the transistors. To simplify the process of the analysis, we assume that the size of the area under the radiation for each transistor is the same. Therefore, if there are N transistors in the circuit unit, then the probability that each transistor will trigger the $O_{SET,k}$ will be $1/N$. When a SET occurs in the circuit unit, the equation for this circuit unit can be then represented as follows:

$$O_{SET} = f_c(O_{SET,1}, O_{SET,2}, ..., O_{SET,N}), \tag{3.3}$$

where f_c is a choice function, which indicates the transistors struck by the particles, and O_{SET} represents the output of SETs on this circuit.

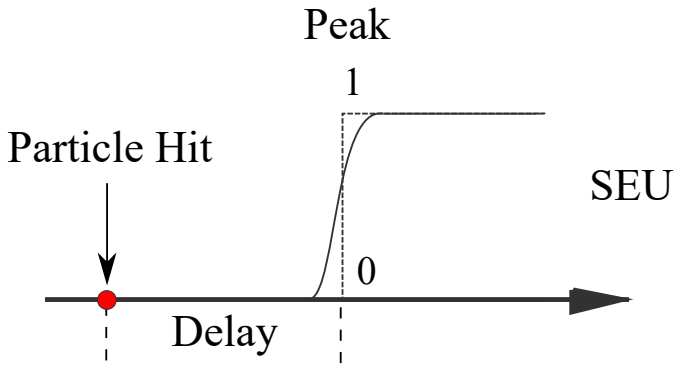


Figure 3.4: The voltage change of the bit flip.

Similar to the effects of SETs, the effects of the SEUs can also be represented mathematically. As shown in Fig. 3.4, when SEUs occur in the circuit unit, the state of the circuit will be changed, which may also change the output of the circuit unit. Therefore, the equation of the SEU on one transistor K can be then represented as follows:

$$S_{SEU,k} = \begin{cases} 0 & 0 < t < T_d, \\ f_{SEU,k}(S_n) & T_d \leq t, \end{cases} \tag{3.4}$$

where $S_{SEU,k}$ represents the new state of the circuit unit after the SEUs on transistor k, T_d represents the propagation time and $f_{SEU,k}$ represents the effects of

the SEU on transistor k.

Therefore, the effects of the SEUs on the circuit unit can be then represented as follows:

$$S_{SEU} = f_c(f_{SEU,1}, f_{SEU,2}, ..., f_{SEU,N}), \tag{3.5}$$

where S_{SEU} represents the new state of the circuit unit after the SEUs in this circuit unit, $f_{SEU,k}$ represents the effects of the SEU on transistor k and f_c represent a choice function, when SEUs occur, one of the $f_{SEU,k}$ functions will be triggered.

3.2.3 Propagation between multiple units

In physical circuits, the observed voltage changes depend on not only the hitted units themselves, but also the propagation path. When SEEs occur in logic gate chains, the width of the transient pulses may significantly increase or decrease in the propagation, which is called propagation-induced pulse broadening (PIPB) effect [181].

As shown in [182], the PIPB effect is induced by unbalanced propagation delay of the rising edge and the falling edge in logic gate chains, Fig. 3.5 shows an example of the PIPB effect. In this figure, the "Diving signal" is the output pulse of the previous circuit unit, assuming that there is a SET occurring in the previous unit. The observed pulse is then the input pulse of the next circuit unit. T_{LH} represents the time for the voltage to change from low to high and T_{HL} represents the time for voltage to change from high to low, which includes the propagation time, charging time and discharging time. When the rising edge time (T_{LH}) is longer than the falling edge time (T_{HL}), the detected width of the positive pulse will reduce, while the negative pulse will widen. When the SEE occurs in the long logic gate chains, the propagation effect will be accumulated, which may further affect the transient pulses significantly.

The rising and the falling edge delays depend on the capacitance of the nodes, which are related to the number of driven gates [182]. In this work, the detected

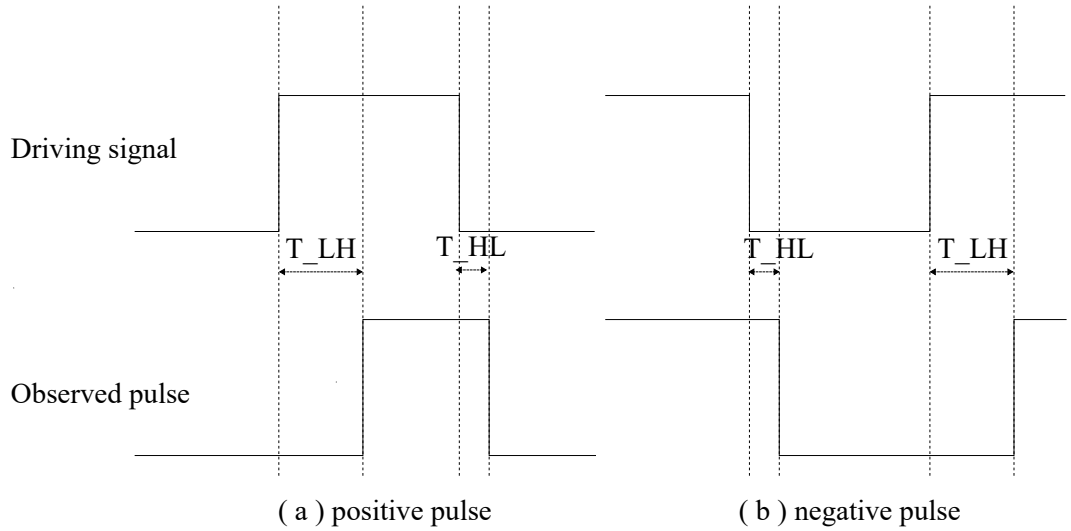


Figure 3.5: The propagation effects on positive and negative pulses, which caused by the difference between the rising and the falling edge propagation time.

propagation delay at input wires of unit K is represented as follows:

$$dt(I) = \begin{cases} T_{LH} & I = 0, \\ T_{HL} & I = 1, \end{cases} \tag{3.6}$$

where $dt(I)$ represents the detected propagation delay, I represents previous valid input values, T_{LH} represents rising edge delay and T_{HL} represents the falling edge delay.

Hence, the input of the unit can be represented as follows:

$$I'_n = I_{(t_n - dt(I))} \tag{3.7}$$

where I'_n represents the input value of the current state for operations, t_n represents the current moment and $dt(I)$ represents the detected propagation delay.

3.2.4 Large circuits

Large circuits are composed of many small circuits or circuit units. When the circuits are struck by the particles, SEEs may occur in any of the small circuits. The probability of a SEE occurring in the specific units corresponds to the SEE

cross section of the units and the size of the circuit units. The SEE cross section represents the number of events per unit of fluence. For the circuits in one chip, the SEE cross section should be identical. Therefore, the probability of the SEEs for each unit should correspond to the size of the circuits. In other words, the larger circuits may have higher chances to capture particles and generate more SEEs.

Fig. 3.6 shows an example of SEE injection in a large circuit, which is a TMR module with 4 sub-modules. There are three identical big functional modules (i.e., M1 M2 and M3) and a small voting module (i.e., M4). If a SEE occurs in the TMR module, then the transient pulse should occur in one of those modules. However, considering that the size of the functional module and the voting module are very different, it is likely that the pulse occurs in the bigger sub-modules: M1, M2 or M3. The size of the unit is relative to the probability of the SEE function of the unit that is triggered.

Therefore, when one SEE occurs in one complex circuit with M circuit units, the probability (P_i) that the unit i is struck can be represented as follows:

$$P_i = \frac{s_i}{s_1 + s_2 + ... + s_M} = \frac{s_i}{S}, \tag{3.8}$$

where $s_1, s_2..s_M$ represents the sizes of each circuit unit respectively in this large circuit, S represents the total size of the circuits and s_i represents the size of the unit i.

Integrated circuits consist of PMOS and NMOS transistors. The size of the circuits can be represented by the number of the PMOS and NMOS. The probability of the occurrence of SEEs can be represented as follows:

$$P_i = \frac{s_i}{S} = \frac{N_{pmos,i} + \lambda N_{nmos,i}}{N_{pmos} + \lambda N_{nmos}}, \tag{3.9}$$

where $N_{pmos,i}$ represents the number of the PMOS in unit i, $N_{nmos,i}$ represents the number of the NMOS in unit i, N_{pmos} represents the number of all PMOS in this

complex circuit, N_{nmos} represents the number of all NMOS and λ represents the ratio of the size of the PMOS and NMOS.

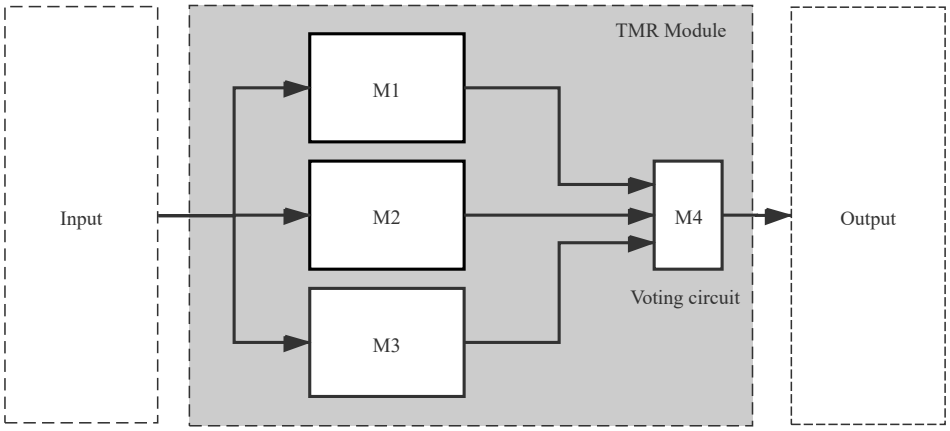


Figure 3.6: A TMR module for SEE injection. M1, M2 and M3 are three identical modules. M4 is the voting module. M1, M2 and M3 are bigger in size than M4.

We assume that the PMOS and NMOS transistors have an identical sensitive size that can catch particles to ease calculations. In that case, the λ is equal to 1 and the probability can be represented as follows:

$$P_i = \frac{s_i}{S} = \frac{N_{mos,i}}{N_{mos}}, \tag{3.10}$$

where $N_{mos,i}$ and $N_{mos,i}$ represent the numbers of the MOS in unit i and the number of all MOS in the large circuit respectively. By using the SEE models and the probability for each unit, the SEE simulation scheme for large circuits can be then implemented.

3.3 Implementation of SEE models

The SEE models of the circuit units form the foundation of the predication scheme. To obtain accurate models, the physical parameters were included in the circuit unit simulations. In this work, the circuit units are simulated in HSPICE [183], a Synopsys circuit simulator for accurate circuit simulation.

3.3.1 Selection of basic components

In this scheme, the basic units to generate SEE models can be a set of gate components or other circuit modules in the targeted designs. The SEE models will be more accurate if additional transistors are covered in the HSPICE simulations. However, the larger the units are, the more difficult the transistor simulation will be. Considering the complexity of the integrated circuits, the basic units with smaller size and higher reusability can ease both transistor level simulations and system level simulations. Thus, the gate components are better options.

Table 3.1: The netlist of the S27 circuits

The SPICE netlist:
.subckt S27 GND VDD CK G0 G1 G17 G2 G3
XDFF2 G7 CK G13 DFFQNX1M
XDFF0 G5 CK G10 DFFQNX1M
XDFF1 G6 CK G11 DFFQNX1M
XU10 G14 G0 INVX2M
XU11 G17 G11 INVX2M
XU12 G8 G14 G6 AND2X1M
XU13 G15 G12 G8 OR2X1M
XU14 G16 G3 G8 OR2X1M
XU15 G9 G16 G15 NAND2X1M
XU16 G10 G14 G11 NOR2X1M
XU17 G11 G5 G9 NOR2X1M
XU18 G12 G1 G7 NOR2X1M
XU19 G13 G2 G12 NOR2X1M

For instance, Table 3.1 shows the HSPCE netlist generated from the S27 circuit, which is one of the ISCAS89 benchmark circuits. QFFQX1M, INVX2M, AND2X1M, OR2X1M, NAND2X1M and NOR2X1M in the HSPICE code indicates different gates used in physical circuits, respectively. Those components are all units required for the S27 circuit. It is possible to combine parts of the components into a bigger unit. However, it will significantly increase the complexity of the following simulations. Considering the reusability and small size of those components, it will be suitable to choose those components as basic circuit units in the following simulations. In this work, 14 components from the gates library

were used in the simulation of circuits in ISCAS89 from the smallest one to the largest one.

3.3.2 The used circuit unit in simulation

The ISCAS89 [184] benchmark circuits are coded in HDL. The physical parameters (e.g., capacitance, resistance) are not included in the ISCAS89 files. In order to obtain the physical architecture of the basic units, HDL designs need to be firstly compiled and implemented. After the implementation, the SPICE netlists are generated and the logic devices in the HDL codes are replaced with the logic components in the physical gate libraries. In this work, a 180 nm gate library [179] from the SMIC was used to build SEE models.

Fig. 3.7 shows the circuit diagram of the DFFQNX1M, which is a flip-flop in the SIMIC 180 nm gate library. This is a sequential circuit that consists of 22 transistors. As mentioned in section 3.2, SEEs are affected by the current

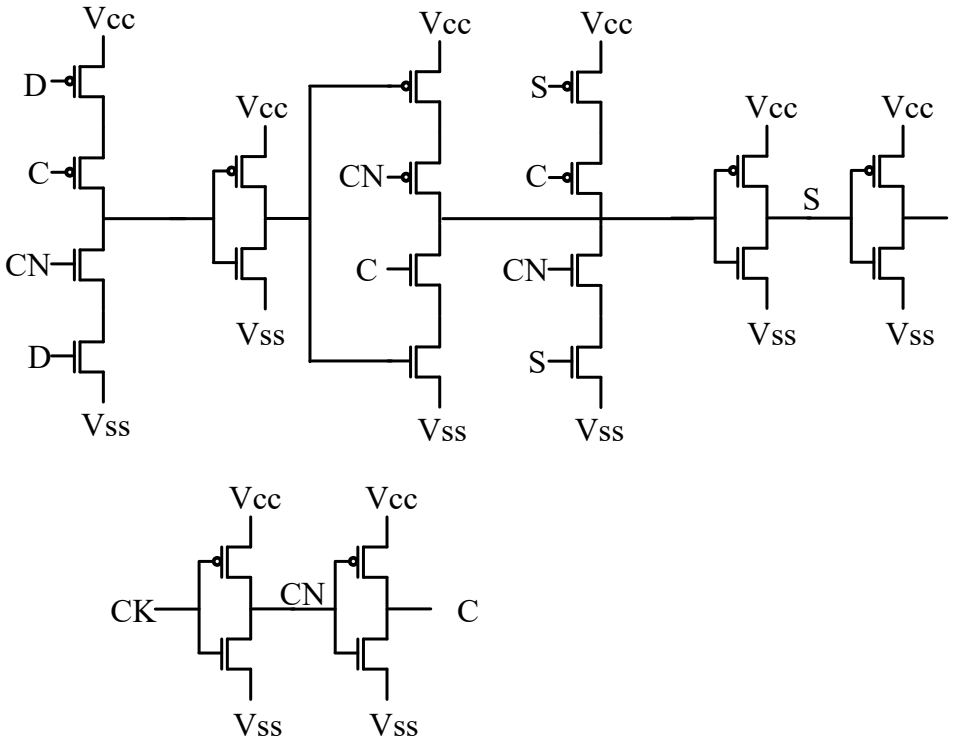


Figure 3.7: The circuit of the QFFQX1M in SMIC gate library.

inputs and states. In order to build accurate models, it is necessary to cover all circumstances in the SPICE simulation. The number of possibilities corresponds

with the number of inputs and the number of states. Therefore, The workload for building the SEE models can be estimated by the number of inputs and states. For a K-size circuit with N_i changeable inputs and N_s changeable states. the possibilities for the circuit can be represented as follows:

$$S_t = 2^{N_i} \times 2^{N_s} \times K, \tag{3.11}$$

where the S_t represents the number of the probabilities which need to be simulated and the K represents the number of the transistors in this circuit.

The probabilities can also be used to evaluate the time required to build SEE models. For example, QFFQX1M has 1 circuit state and 5 inputs including D, C, CK, VDD and VSS. When the circuit is working, the VDD and VSS are constant and the D, C and CK are changeable. According to Equation 3.11, the number of circumstances for all transistors is equal to 176, which is also the number of the simulation rounds required for building the SEE model.

The time required can be predicted by the probabilities and the time required for each simulation round. Table 3.2 shows the time required for the SPICE simulation for all units used in this work. The SEE models of small units can be generated in several minutes, while the SEE models of big circuits (e.g., QFFQX1M) will take several hours. In general, the SEE models for all units could be generated in hours.

3.3.3 Injection currents

Both SETs and SEUs are caused by ionizing particles and transient currents.. At present, the most common methods for simulating single-particle effects are to inject the transient currents to the target node, where the location is struck by high-energy particles. The current sources are used in this work to generate the transient currents to simulate single-particle effects. The intensity of the current reflects the intensity of radiation and the capability of energy absorption.

Table 3.2: Time required of the SPICE simulation to build SEE models

Unit	Size of the unit	Number of probabilities	Time required
INVX2M	2	4	1 s
OR2X1M	6	24	16 s
OR3X1M	8	32	31 s
OR4X1M	10	160	238 s
NOR2X1M	4	16	6 s
NOR3X1M	6	48	32 s
NOR4X1M	8	128	121 s
AND2X1M	6	24	16 s
AND3X1M	8	32	31 s
AND4X1M	10	160	238 s
NAND2X1M	8	32	6 s
NAND3X1M	10	160	32 s
NAND4X1M	12	192	480 s
QFFQX1M	25	176	196 min

In order to improve the accuracy of the SEE model, the injection currents should follow the trends of the physical effects. There are some current models for SEE simulation (e.g., the rectangular pulse model, the double exponential pulse model and the transient pulse model [185, 186]).

However, for the rectangular and the double exponential pulse, it has been proven that the peak of the currents could be 20% lower than real transient currents [185]. Here, we use a transient current model which is based on the quantity of the electricity generated by the ionisation effects. Compared to the other models, the current waves generated by this model are closer to the real currents of the SEEs [187].

The equation of the transient pulse model can be represented as follows:

$$I(t) = \frac{2Q}{\tau\sqrt{\pi}}\sqrt{\frac{t}{\tau}}\exp\left\{\frac{-t}{\tau}\right\}, \tag{3.12}$$

where the Q represents the quantity of the free electricity generated by the ionization effects, which is also the quantity of free electricity in the injection, τ represents the physical parameters related to the electricity absorption, which is affected by the materials of the semiconductors, physical shapes and architecture of the transistors. The higher value of τ means the slower current changes in the

circuits. Normally, Q is between 100 fC (femto-coulomb) and 150 fC [188] and τ is between 25 ps and 35 ps (picosecond) [30] for the circuits between 100 nm and 200 nm. We considered the circuits are from the 180 nm gate library, Q is set at 100 fC and τ is set at 25 ps.

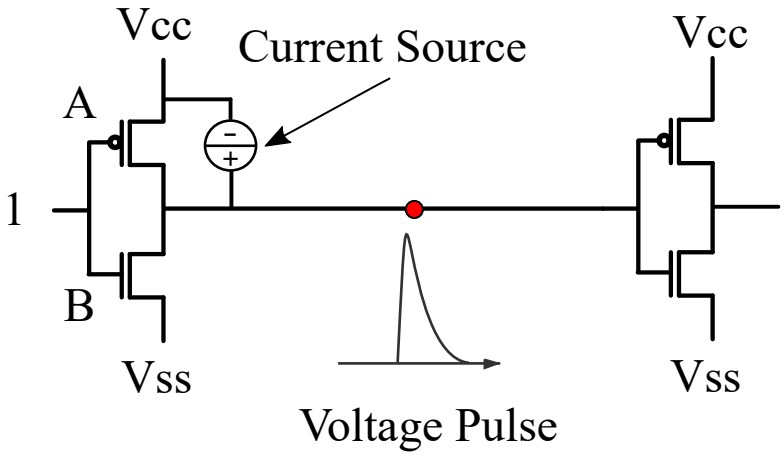


Figure 3.8: The location of injection current source in the inverter circuit, when the PMOS in the inverter is struck by the particle.

3.3.4 Simulation of SEE models

3.3.4.1 Implementation of current sources

The direction of the SEE transient currents are affected by the electric field direction. Therefore, the injection current source should correspond to the electric field. Fig. 3.8 shows the examples of the injection current source in the SMIC 180 nm inverter circuit. If the output of the inverter is "0", then the PMOS will be the sensitive gate. In this case, the SEE transient current flows from Vcc to the output port, which will cause a positive pulse. If the output is "1", the direction of the transient current is from the output port to Vss causing a negative pulse. Therefore, the current source will be connected to the sensitive gates to generate pulses in the simulation.

When transient currents are injected, corresponding voltage changes will be observed at output nodes. Fig. 3.9 shows the voltage change of the inverter output in the SPICE simulation. The peak of the voltage pulse is about 2.74 V

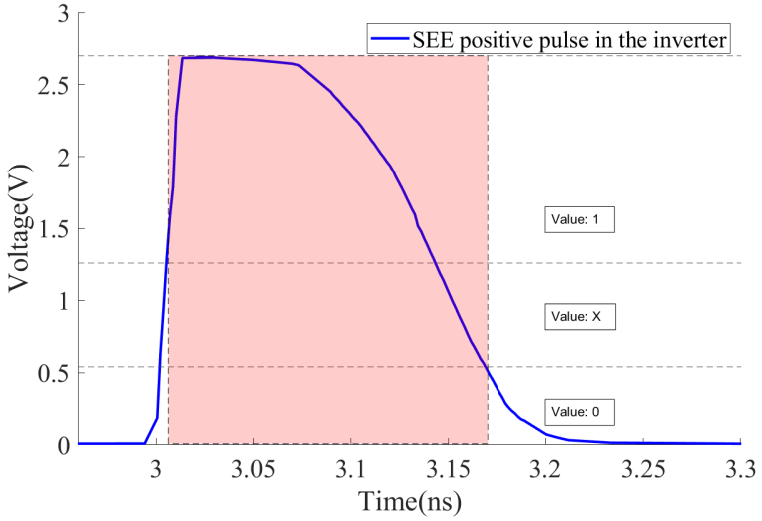


Figure 3.9: The simulation results of SEE positive pulse in the inverter circuit. The input of the inverter is "1" and the PMOS is struck by the particle.

and the time duration is about 0.2 ns. When it is converted to the digital model, the time duration is the width of the digital pulses.

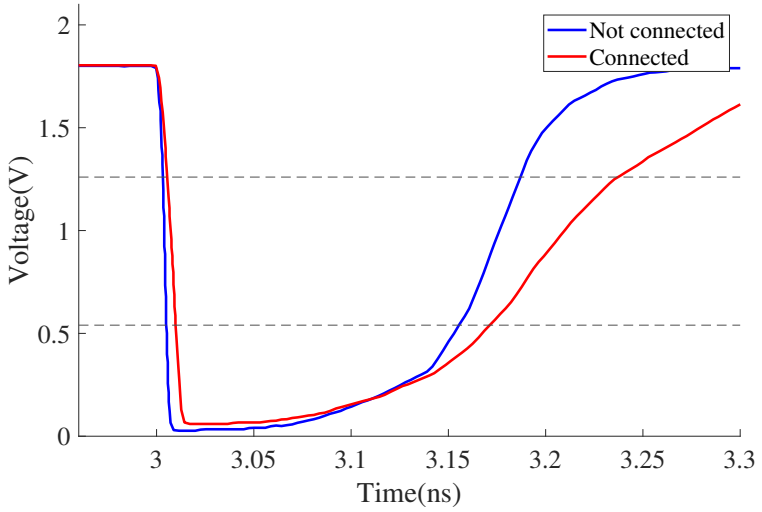


Figure 3.10: The comparison of the output voltages of the inverter with and without following circuits.

3.3.4.2 Propagation effects

Due to the different connections of the circuits, the voltage curve of transient pulses are not constant. Fig. 3.10 shows the voltage curve of transient pulses

Table 3.3: The lookup table with initial delay parameters

Number of units	Specifications (ps)	
	Rising edge delay	Falling edge delay
1	79	64
2	97	80
3	112	92
4	127	104
5	141	115
6	159	131
7	178	146

in the inverter circuit with one and without following circuits. Compared to the separated inverter without following circuits, the output voltage curve in inverter with following circuits is smoother, which causes propagation effects in the circuits.

As mentioned in Section III, increasing number of logic gates increases the capacitance of the connection node, which will increase rising edge and falling edge delay. In this chapter, a lookup table is built to represent Equation 3.7. Considering that the number of driven gates can be obtained by simply counting how many times the output wires are used in the netlist, T_{HL} and T_{LH} parameters can be initialised, when final netlists are generated. Table 3.3 shows the pre-built lookup table used in the simulation. For example, if a previous units is driving three following units, then the rising edge from the previous output should be detected by the following units 112 ps later.

3.3.4.3 The width of the digital pulses

The width of the digital pulses represents the time duration of the voltages, which can be sampled as incorrect values. In CMOS devices, the threshold voltage of logic "0" is normally $0.3(Vcc - Vss)$. When voltages are lower than $0.3(Vcc - Vss)$, the sampled values will be logic "0". The threshold voltage of logic "1" is $0.7(Vcc - Vss)$. When voltages are higher than $0.7(Vcc - Vss)$, the sampled values are "1". In this work, the threshold voltages are represented as follows:

$$V_H = \frac{Vcc - Vss}{\sqrt{2}} + Vss, V_L = \frac{Vcc - Vss}{2 + \sqrt{2}} + Vss, \tag{3.13}$$

where V_H represents the threshold of the high voltages and V_L represents the threshold of the low voltages. The width of the positive pulse is the duration of the voltage peaks which are higher than V_H. The width of the negative pulse is the duration of the voltage peaks which are lower than V_L.

Table 3.3 shows the specifications of the two pulses. The width of the pulse in the inverter with following circuits is 15% wider than the pulse in separate circuit.

Finally, the digital SEE models can be generated by using the SPICE simulation results. The digital SEE models of the INVX2M and DFFQNX1M are shown in Table 3.4 and Table 3.5, which include the trigger conditions, possibilities, the width of the pulse in different circumstances and the output delay.

Table 3.4: The SEE model of INVX2M unit

	Inputs	probability	Width (ns)	Delay (ns)
SET	0	50%	0.22	0.0
	1	50%	0.22	0.0
SEU	X	0%	/	/

Table 3.5: The SEE model of DFFQNX1M unit

	Inputs-States*	probability	Width (ns)	Delay (ns)
SET	XXX	8%	0.22	0.0
SEU	0X0	4%	/	0.10
	0X1	8%	/	0.12
	1X0	8%	/	0.11
	1X1	4%	/	0.10

* The data input, CLK and Stored bit

3.4 SEE simulation of large-scale circuits

In order to carry out simulations for large-scale circuits on the system level, the SEE models generated from the SPICE simulation need to be integrated into the HDL circuit units. Therefore, the original logic components (e.g., OR gates and AND gates) are replaced with the circuit unit modules that contains the digital SEE models. Considering the large number of logic components, Python script tools are developed to automatically generate HDL codes.

3.4.1 Implementation of HDL circuit units

Circuit units are used to replace the original logic components. Therefore, the function of the original logic components should also be covered by the circuit unit. The circuit units will include the function of the original logic components and the SEE models. There are three parts of the circuit unit models: 1) original logic, 2) configurable parameters and 3) the functions of the SEE models to generate the bit-flips and digital pulses.

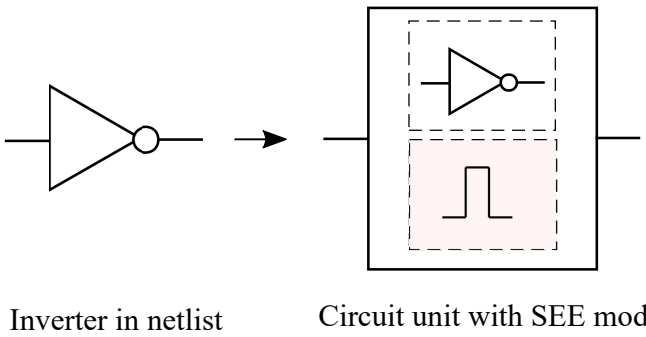


Figure 3.11: The implementation of SEE model of the inverter.

Fig. 3.11 shows the implementation of SEE model of the inverter. In the original netlists, the inverters are simple circuit blocks only with the "NOT" function. In the generated netlists, the inverters are replaced by the HDL modules which consist of the "NOT" function and the SEE model function to conduct SEE injection. If there is no SEE, the module will work as the inverter. When the SEE injection is conducted, the injection programme will try to call the functions of the SEE model to force the circuits to generate a digital pulse. As with the inverter, other logic gates will be also replaced. In this way, we can rebuild the netlist for the HDL simulation.

The SEE model is implemented as a task functions in Verilog. When the SEEs are triggered, the task functions will be called by the injection scripts. The task functions simulate the pulse caused by forcing signals on the output wire to be incorrect for a certain time and simulate the bit-flips by changing the circuit states.

Fig. 3.12 shows the example of the SEE output in the inverter in the HDL simulation. The "x" is the input of the inverter and "nx" is the output. The SEE

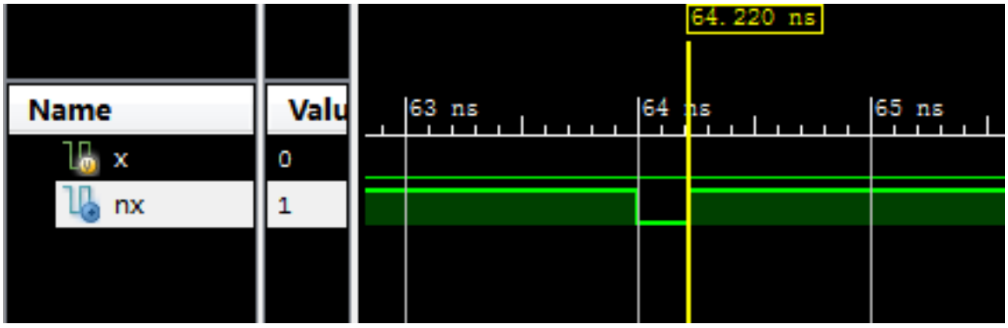


Figure 3.12: The transient pulse in HDL simulation. "x" is the input of the inverter and "nx" is the output of the inverter. The pulse is injected at the moment of 64 ns in the inverter unit.

occurs at 64 ns and the function of the SEE model is triggered at the same time. SPICE simulation also shows a negative pulse lasting for 0.22 ns.

In the HDL simulations, the SEEs are represented by injection functions in the units. Considering that there is only one unit struck by the particle in the single event, therefore only one of the injection functions should be triggered. We marked all circuit units and injection functions with unique IDs to indicate the circuit units. In this way, only one function will be accessed during SEE injections. The IDs can be calculated by the following equation:

$$ID_k = \sum_{i=1}^{k-1} s_i \tag{3.14}$$

where s_i represents the size of the marked circuit units, the number of the MOSFETs in the circuit unit. During SEE injections, the scripts will generate a random number under the range of the IDs. If the number is between the ID_k and $ID_k + s_k$, then the SEE function in the unit k will be triggered. The probability will be calculated using Equation 3.10.

3.4.2 Script tools for injection

Large-scale HDL designs contain many circuit units. For example, S38584, the largest circuits in ISCAS89, contains 11,448 logic units. It is difficult to design the test bench by handcraft. Hence, we design a tool based python script to generate

the test-bench codes automatically. The test-bench codes include three parts: 1) input generation, 2) SEE injection and 3) output analysing.

The design of the HDL simulation for large-scale circuits is shown in the Figure. 3.13. In the HDL simulation, the target circuits will be instantiated twice. Therefore, there will be two identical designs in the HDL simulation. One of the modules is the injection circuit where we inject the SEE. The other one is the reference circuit, which is used for error detection. Considering that the pulses caused by SETs could be easily blocked by registers, I added a set of registers as output buffers. If the pulses are blocked by the registers, then the pulses will not affect the following sequential circuits.

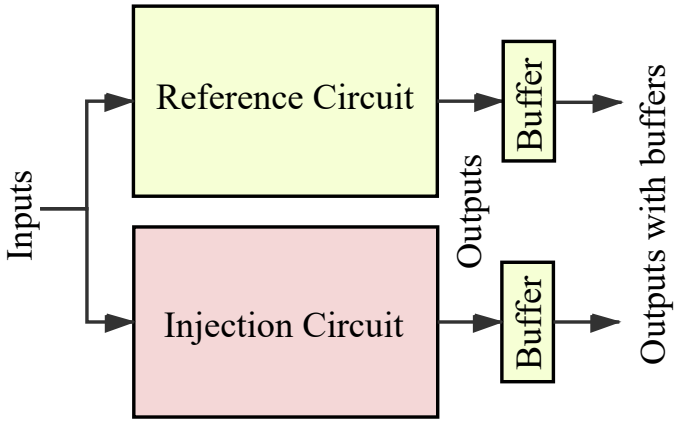


Figure 3.13: The HDL simulation for large-scale circuits.

In the HDL simulation, there are two types of inputs generated by the scripts, which are random inputs and specific inputs. In this scheme, the random inputs are generated based on the given switch probabilities of input signals. If the HDL bench codes just drive the target design with the random inputs, the difference in the data stream will not be observed. If the test-bench codes drive the circuits with specific input data streams, the SEE mitigation performance of the input streams can also be evaluated.

The test-bench codes call the functions in the SEE models to carry out injections. Calling the SEE function with the unique ID, the SEE injection can be conducted. If the error can finally reach the output of the circuit, it will be observed. By comparing the outputs, the errors can be detected and recorded. We

use scripts to calculate the number of errors, the number of SEEs and the error rates of the design.

3.4.3 Propagation of SEE induced errors

When SEEs occur in the large-scale circuits, some error may not be visible at the output ports of the circuits. The pulses may be blocked in the propagation path by the logic gates and registers.

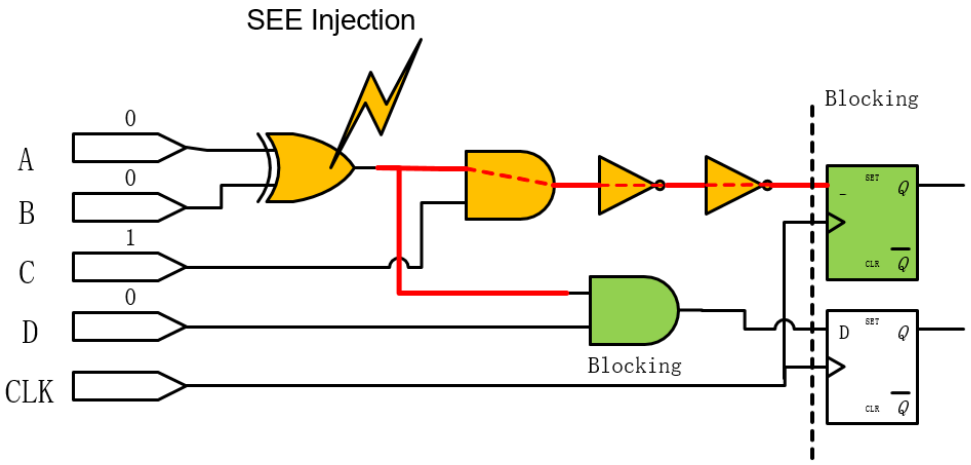


Figure 3.14: Errors blocked by logic gates and registers. SEE occurs in the XOR gate. If C or D is "0", the pulses will be blocked by AND gates. If CLK is not flipping, the pluses will be blocked by registers.

Fig. 3.14 shows the scenario where SEE errors are blocked by logic gates and registers. There are two different propagation paths of the errors in this figure. In the first propagation path, because the current input 'C' is "1", the error generated in the OR gate can reach the register. However, the registers only sample the input signals at the positive edge of the clock. There is a high chance that the pulse will not be sampled, which means that the error is blocked. In the second path, because the input signal 'D' is "0", the error cannot go through the AND gate, which means that the errors are blocked by the logic gate.

Considering the above cases, the scripts will also monitor the signal after buffers to see if adding buffers is helpful to mitigate errors. When generating bench codes,

scripts will automatically add registers connected to the output wires. The original outputs will be compared to the outputs of the buffers, which are used to analyse the effects of the errors on the following circuits.

3.5 Simulation results of large-scale circuits

We used the same generated SEE behaviour models to analyse the performance of more than 40 circuits from ISCAS89 benchmark circuits, which represents that the circuits are under the same circumstances. In addition, I compared the SEE mitigation performance of the same circuits using TMR and register space-time redundancy (STR) technology.

3.5.1 The simulation of the ISCA89 circuits

There are more than 40 circuits in ISCA89, among those, the S27 circuit is the smallest. It contains 13 circuit units. S28584 circuit is the largest one. It contains 11,448 circuit units. The simulation results in the circuits indicate the effects of SET and SEU effects between small-scale and large-scale circuits.

Fig. 3.15 shows the SEE injection in the S27 circuits, where the G0, G1, G2 and G3 are inputs of the S27 circuits. The G17 is the output of the S27, where the scripts do the SEE injection. The G17_ref is the output wire of the reference unit. The flag_wire indicates the SEE injection. At the positive edges of the flag_wire, the SEEs are injected. In this figure, there is a positive pulse at the G17 wire, which is different from the signal at G17_ref. The variable werr_cnt shows the number of the observed errors.

Fig. 3.16 shows the comparison of the circuits with and without output buffers. “G17” is the output of the injection circuit and “G17_ref” is the output of the reference circuit. “result” is the output of the buffer which is connected to the “G17” and “result_ref” is the output of the buffer which is connected to the “G17_ref”. In the simulation, the output buffers are used to filter out digital pulses to evaluate

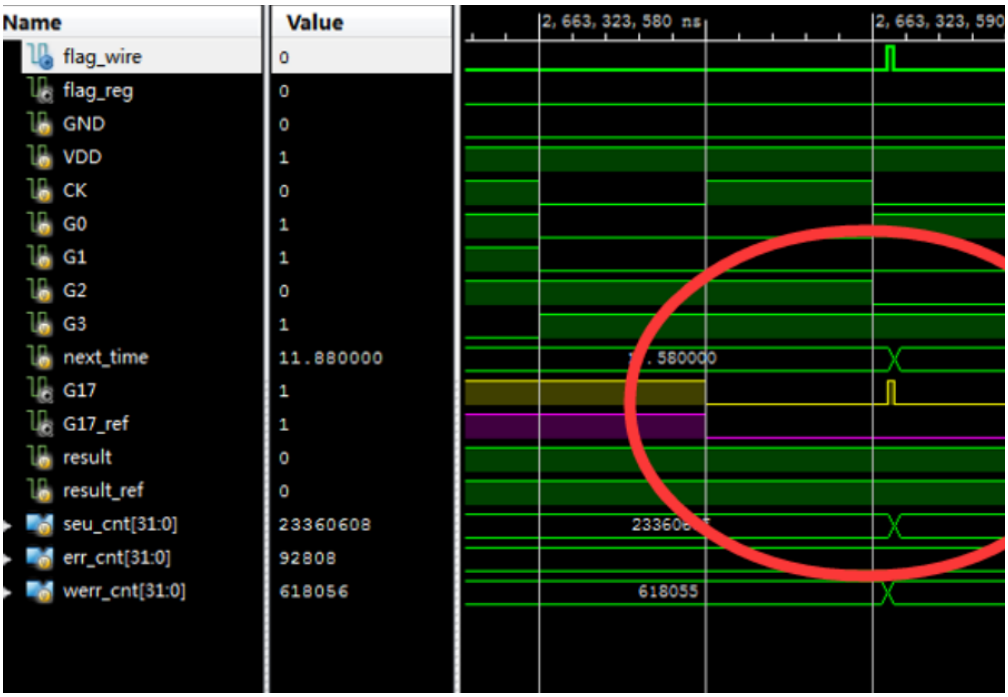


Figure 3.15: The outputs of the S27 in HDL simulation. The signal "G17" is the original output of S27. "G17_ref" is the reference output of S27. A transient pulse can be observed in "G17".

bit flip rates.

Fig. 3.17 shows the error rates of the ISCAS89 circuits with and without buffers. This result indicates the bit-flips rates and the total error rates, respectively. In this figure, the error rates of the small circuits without buffers drop significantly, while the error rates of the circuits with output buffers remain stable. This can be attributed to the fact that the SEE induced pulses are likely to be blocked by the registers and logic designs, while the bit-flips cannot be blocked by the register. Considering there are more registers in larger circuits, there could be a high chance that the pulses are blocked by the registers in the propagation path. Therefore, the error rates of circuits without buffers decrease along with decreases in the circuit size.

Additionally , when the circuits are becoming larger, there will be more unused propagation paths and invalid states, where the errors cannot affect the following circuits. That is why the observed error rates in the simulation decrease slightly with the circuit size. It is possible to utilise the redundant states and path to

Figure 3.16: The outputs of the S27 with buffers in HDL simulation. The signal "result" and "result_ref" are buffered "G17" and buffered "G17_ref".

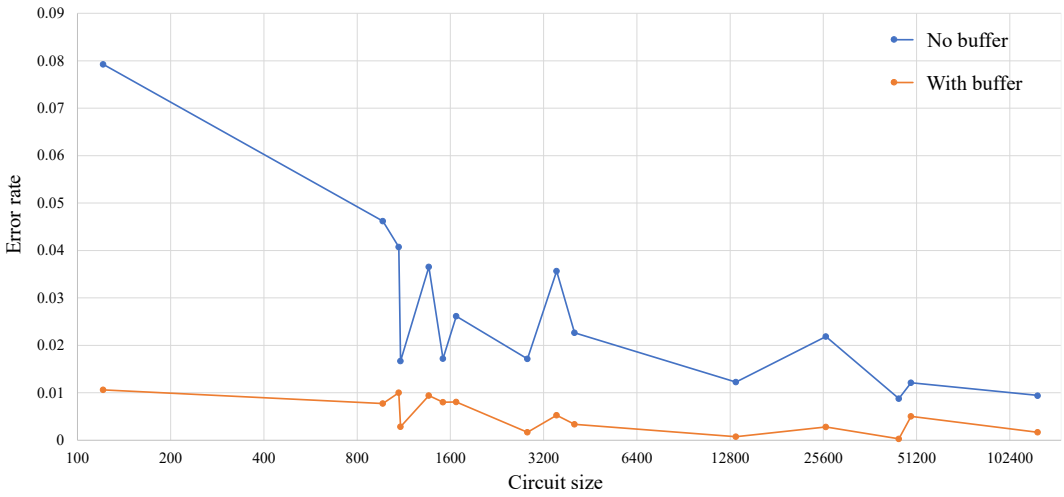


Figure 3.17: The error rates of the circuits in ISCAS89 without buffers and the circuits with buffers.

improve the SEE mitigation performance.

3.5.2 Simulation of the SEE mitiagtion circuits

The simulation results of the circuits without mitigation show that there are a large number of error bits caused by the pulses. Therefore, I evaluated the SEE mitigation performance of STR and TMR methods.

The STR method only hardens registers. With extra registers as redundancy, STR can fix the one-bit errors by itself. In addition, the registers will sample input signals at different moments, which means that it can filter out digital pulses. TMR is a popular method for SEE mitigation systems. It triples the modules for

Table 3.6: The error rates of the S27 circuits with different fault-tolerant methods

	Original Design	TMR	STR*
Inputs	4		
Outputs	1		
Transistors	121	389	184
Error rates without buffers	0.0792	0.0195	0.0620
Error rates with buffers	0.0106	0.00130	0.00131

* space-time redundancy

Table 3.7: The error rates of the S1423 circuits with different fault-tolerant methods

	Original Design	TMR	STR
Inputs	17		
Outputs	5		
Transistors	5,102	15,436	8609
Error rates without buffers	0.0226	0.0024	0.0091
Error rates with buffers	0.0033	0.00016	0.00027

redundancy and uses an extra voting module to select the correct outputs.

TMR has the best error mitigation performance with the highest hardware costs and the STR technology has a balance between performance and costs [189, 190, 191]. Therefore, we validated the proposed scheme by comparing the performance of both methods in the proposed scheme.

The Table 3.6, 3.7 and 3.8 shows the simulation results of S27, S1423 and S38584 with different hardening technologies. S27 represents the small circuits, S1423 represents the medium sized circuits and the S38584 represents the large circuits.

The simulation results show that the proposed scheme provides details analysis to evaluate and compare the SEE mitigation performance of different circuits. Firstly, the proposed scheme provides a general analysis includes 1) SET rates, 2) SEU rates, 3) changes of the circuit size and 4) error rates with buffers. It helps to exclude unnecessary hardening methods. For example, compared to TMR, STR could be a better option to harden the S27 circuit with the same performance and much lower costs. Secondly, with HDL simulations, the proposed scheme provides the waveform analysis, which can be used to address the vulnerable parts and the error propagation paths.

Table 3.8: The error rates of the S38584 circuits with different fault-tolerant methods

	Original Design	TMR	STR
Inputs	38		
Outputs	304		
Transistors	125,940	385,724	155,886
Error rates without buffers	0.0094	0.0040	0.0065
Error rates with buffers	0.0017	0.00027	0.00035

3.5.3 Time required for the simulation

In this chapter, the time cost of HDL simulation is elaborated. We can evaluate the time required for the HDL simulation by averaging time costs for each SEE injection. Fig. 3.18 shows the time required for the HDL simulation with one million SEE injections in the ISCAS89 circuits. By using HDL models, one million SEEs can be injected into the S27 circuit (121 transistors) in just 55 s, while it will take 55 hours in S38584 (125940 transistors). The time required for SEE simulation increases nearly linearly with the scale of the circuits.

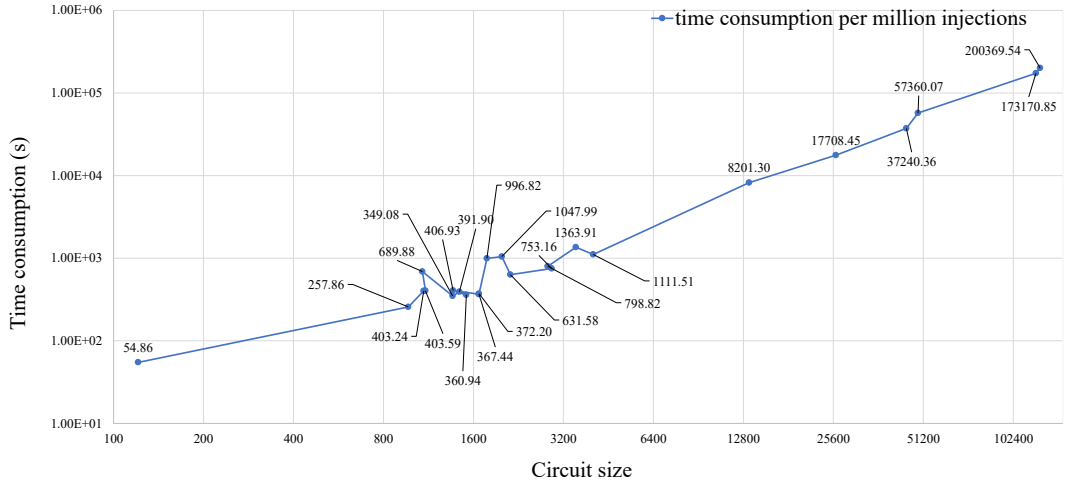


Figure 3.18: The time required to SEE HDL simulations for ISCAS89 benchmark circuits.

Table 3.9: The average time required to run the S27 circuit for 1 ms in different simulation environments

	Average time cost
HDL simulation without injection	0.473 s
The proposed simulation with SEE injection	0.676 s
SPICE simulation with SEE injection	67,000 s

In this chapter, the time requirements in different simulation environments are also tested. Due to the long time required for large circuits to run simulations in SPICE, S27, the smallest circuit in the benchmark circuits, is used for the comparison. There are three simulations including HDL simulation without SEE injection, SEE HDL simulation and SEE SPICE simulation.

Table 3.9 shows the average time required for HDL simulations and SPICE simulation to run for 1 ms. The time precision of HDL simulation and SPICE simulation is set to 1 ps, so that the required time could be compared directly. The HDL simulation with SEE injection for S27 takes 0.676 s to run for 1 ms. The time required increases slightly from 0.473 s to conduct SEE injections. Compared to SPICE simulation, which costs 67000 s (i.e., ≈18.6 hours) to run the simulation for 1 ms, the HDL SEE simulation shows a great advantage concerning simulation efficiency. Due to the high complexity and huge number of possibilities, it is unlikely to run HSPICE simulations to analyse SEE error rates of large integrated circuits. There are two reasons that accuracy is not included here. Firstly, it is too expensive to produce real chips in this study. Hence, there is no data from real-world radiation experiments as reference. Secondly, there SPICE simulation costs too much time. The SEE simulations for large circuits can not even be finished in SPICE. Therefore, there is no accuracy results for large circuits from SPICE simulations.

3.6 Conclusion

In this chapter, a fast and cost-efficient method is proposed to evaluate and compare the performance of large-scale circuits under the effect of radiation particles. Compared to typical methods, it requires less simulation time and computational resources. SPICE simulations are used to build a set of general SEE models to simplify the processes of the evaluation of SEE effects on large-scale circuits. The proposed scheme has been evaluated using 40 different circuits from the ISCAS89 benchmark circuits. It is shown that the scheme can analyse the circuits with

sizes varying from 100 transistors to 100k transistors. We have also compared and evaluated the simulation results of the circuits using TMR and STR technology. The results prove the correctness of the scheme. The main contributions of this work can be summarised as follows:

1) The proposed SEE simulation scheme provides a rapid, convenient and universal comparison method to evaluate the designs of circuits in the context of SEEs. Due to various manufacturing processes, physical layouts and radiation environments, the simulation tools and simulation environments may also vary in different SEE research. It is difficult to repeat or compare those experiments directly. The proposed SEE models can be easily integrated into the current circuit design workflow without significant cost. It can create a universal simulation environment to provide a quick analysis of the relative performance, which can significantly reduce the total simulation time for the time-consuming back-end simulation.

2) The proposed work introduces a range of new SEE behaviour models. Based on the transistor level simulations, the SEE behaviour models are firstly embedded into a range of digital functions in the HDL described circuits, the transient currents and voltages are then converted into digital pulses and bit-flips. Unlike the typical transistor level based SEE behaviour models that fully rely on low-level currents and voltages simulation inputs, the proposed SEE models use only high level digital functions in HDL, therefore it can offer lightweight and fast simulations for large-scale circuits.

3) The proposed scheme can offer a high level of flexibility in the design. All parts in this scheme including gate components, SPICE simulation and HDL simulation are decoupled. The gate components can be modified to adapt to different manufacturing processes, and the SEE spice model can be also modified to adapt to different radiation environments, as required. In this way, the proposed scheme can make full use of existing models to build

simulation environments and be adapted for various requirements.

Chapter 4

Self-adaptive SEU mitigation for RAM

4.1 Introduction

The previous chapter introduce a SEE simulation methods, which can be used to evaluate the performance of SEE mitigation design. To design an embedded system, it is critical implement SEE mitigation designs. In this work, a portable SEE mitigation scheme is proposed to correct errors and mitigate error accumulation in extreme radiation environments.

SRAM cells are susceptible to SEUs, as discussed in Chapter 1. For long-term operation, it is required to harden SRAM devices, especially for the devices that are intended to function in radiation environments. A variety of error mitigation solutions have been considered in order to mitigate faults in memory systems. TMR, ECC, and scrubbing are three typical error mitigation methods for RAMs. [33, 34, 35, 36, 37].

Existing hardening approaches have difficulty being applied to an FPGA system with several pre-designed hardware modules employing BRAMs. To begin with, due to the pre-designed FSM in the hardware modules, changing the time sequence of the hardware modules would be tough. Therefore we cannot simply

add the corrective operation. Second, due to the use of dual-port BRAMs, there will be insufficient RAM access ports to connect scrubbers. Third, because the TMR approaches will require treble the amount of resources, RAM space will be constrained.

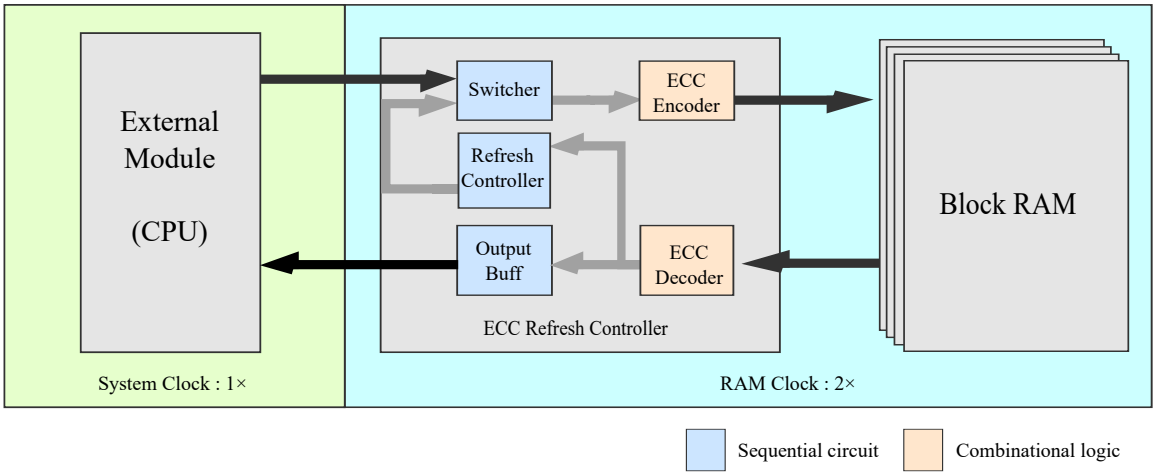


Figure 4.1: The scheme of the external scrubber platform.

To harden RAM systems based on CPUs and customised circuits in extreme environments, a self-refreshing scheme with ECC is presented [192]. It is designed to extend the life of commercial electronic equipment that are exposed to high levels of radiation and experience faults every second. It features high reliability, flexibility and low hardware costs. Considering the systems for radiation environments usually operate at a relatively low frequency, the frequency of the RAM can be doubled to fully use the bandwidth. In this way, the spare bandwidth can be used to operate data refreshing where the refresh operations can be performed simultaneously with the normal read and write operation. The refresh operations are performed by a small separate scrubber inserted between the modules and RAMs. The scrubber can transmit all writing and reading operations to the RAMs without additional delay. With this scheme, the scrubber is transparent to the external modules. The design provides flexibility in case of porting the design to different hardware platforms. Considering that, in the FPGA designs, many customised modules operate the RAMs with fixed reading or writing latency. The proposed scheme introduces no additional latency for error detection and correction which

means that it can be easily implemented in the consumable embedded platforms for special radiation tasks without modifications.

A Xilinx Virtex-5 XC5VLX110T [193] FPGA is utilised to develop a hardware simulation platform, while an Artix-7 XC7A15T-1CPG236C [194] FPGA is used to build a prototype for radiation experiments. The simulation platform, which consists of both hardware and software components, is presented for real-time SEU injection and performance verification. The hardware and software co-simulation shows that the proposed design can handle more than 99.9% and 99.97% of errors, while the SEU rates are 1×10^4 bit/s and 6.25×10^4 bit/s respectively. In the neutron radiation experiment, the observed error rate for unhardened RAM was 1.2 bit/(KB·h). The errors rates for conventional ECC ram are approximately 4.3×10^{-4} bit/(KB·h), while the self-scrubbing RAM is less than 8.7×10^{-5} bit/(KB·h).

4.2 Architecture of the self-refresh RAM

Fig. 4.1 shows the system block diagram of self-refresh ECC RAM. Unlike the scrubbing scheme with dual-port RAMs where scrubbers use separated ports, the self-refresh controller and user modules share the same RAM ports in the proposed system. In this scheme, the self-refresh controller is not only a scrubber used to "clean" errors in RAMs, but also a transmitter used to pass data from the user module to RAM devices. In addition, to minimise the effect on the read and write timing sequence of the user module. The controller and RAMs operate at double the frequency of the user clock, which means that the ECC refresh controller runs faster than user modules and is thus able to utilise extra clock cycles to perform additional tasks (e.g., fault detection and error correction) without interrupting the normal operations.

4.2.1 Switcher for operations

As mentioned, the purposes of the controller are 1) transmitting operations from user modules and 2) scrubbing RAMs. When the proposed scheme is applied in the system. The user module will access the controller with original read and write operations. For user modules, the controller will work just as a simple configured BRAM. The switcher will pass all controller signals from the user module to RAMs. By using classic RAM control circuits, the multiple features (e.g., enable signal and mask function) can be easily implemented for various hardware systems.

To achieve this purpose, the operations from user modules and the scrubber should be carefully arranged to ensure the timing sequence remains unchanged. Assuming that the outputs are ready in the next user clock cycle in the original timing sequence, the controller should also follow the same timing sequence. In the controller, a switcher module is designed to re-arrange the sequence of operations from two directions: 1) user module and 2) refresh controller. As shown in Fig. 4.2, the external operations from the user module and the refreshing operations are interlaced by the switcher. Considering that the controller is working at the double frequency of the user clock, there will be one clock cycle left (in the 2× clock domain) to transmit and return data.

However, one clock cycle is still very limited to conduct transmission and ECC coding. Therefore, in this scheme, Hamming code is used as the correcting code. Because of its simple calculations, the decoder and coder can be designed fully based on logic gates. In this way, the coding processes will not require additional cycles.

4.2.2 Refresh controller

The refresh controller is a core module of the proposed self-refresh ECC RAM. It has two operating modes: 1) the scan mode and 2) the refresh mode. When the system is working, the refresh controller continuously generates reading commands to read all memory units in the RAM periodically. Simultaneously, it checks the

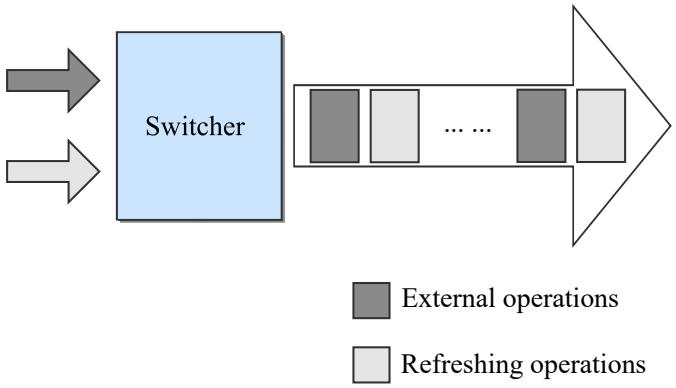


Figure 4.2: Sequence of the input operations stream. Operations from the user module and refresh controller will be placed one by one.

outputs of the ECC decoder, which is a combinational logic module to decode the outputs of the RAM. When SEUs occur, the ECC decoder can detect and fix the errors per byte. An error flag will also be asserted to indicate the occurrences of SEU. Then the refresh controller will be switched to refreshing mode and generate a writing command to refresh memory units.

4.2.3 Output buffer

In order to resolve the timing problems, a buffer module is set between the ECC decoder and output port. Because the operations sent to the RAM are interlaced, the output data stream is also interlaced. To ensure that the external module will not be affected, the sequence of the output data stream needs to be re-arranged accordingly. As shown in Fig. 4.3, the output buffer module will block the outputs of refreshing operations. In other words, the output buffer allows only the outputs of external operations to go through in order to prevent the external module access to the data of refreshing operations.

In this architecture, the external operations (e.g., reading and writing) and internal operations (e.g., refreshing) execute parallelly. The performance of the systems will therefore not be affected.

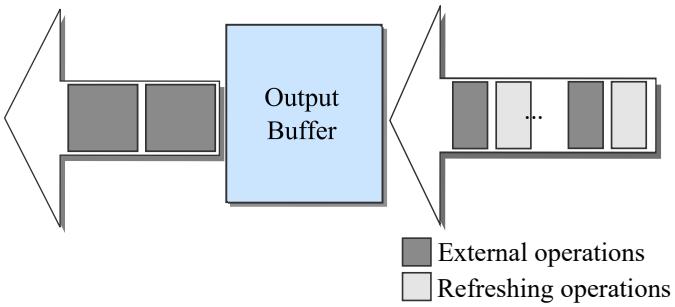


Figure 4.3: Sequence of the output data stream. The read data of the user module will be buffered, while the read data of the refresh controller will be blocked.

4.3 Hardware implementation

As mentioned in the last section, as all operations from the external modules (i.e., CPU) and the refresh controller are interlaced, the controller must arrange the sequence of the operation stream carefully. The details of the FSM that enables this and how to handle the conflicts between external modules and internal modules will also be discussed.

4.3.1 Design of the FSM

To distinguish from user module operations, in this work, "R/W operation" indicates read and write operations from user modules, while "scrubbing operation" indicates operations from the refresh controller. Considering the different clock domains of the user module and the refresh controller, the operation sequence needs to be carefully arranged. In this work, the cycles (2× clock domain) used by the refresh controller are called "refresh cycles", while the remaining cycles are called "user cycles". The refresh operations will be distributed into the "refresh cycles".

The state machine diagram of the refresh controller is shown in Fig. 4.4. The progress of refreshing starts from the rising edge of the user clock (1× clock domain). In this way, the output sequence can be synchronised with the user clock domain.

The refresh progresses start from state 0 (S0), indicating that it is in refresh

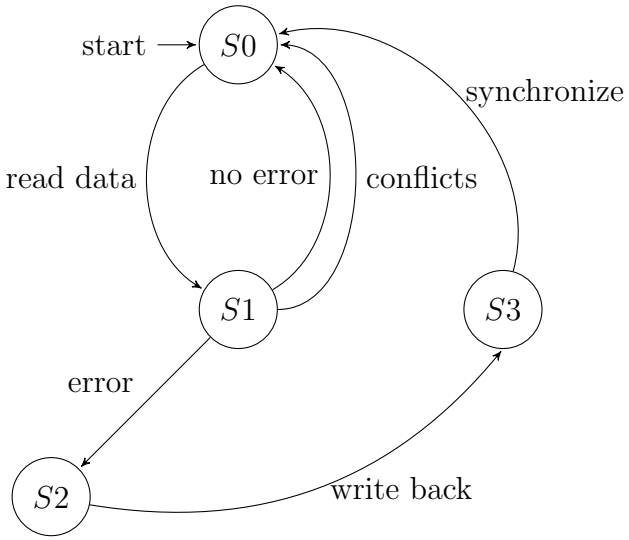


Figure 4.4: There are four states in this FSM: S0 (start,next), S1 (check), S2 (refresh) and S3 (synchronise). The read or write operations from the refresh controller can only be conducted in S0 and S2, because the RAM access port is occupied by the user module operations in S1 and S3.

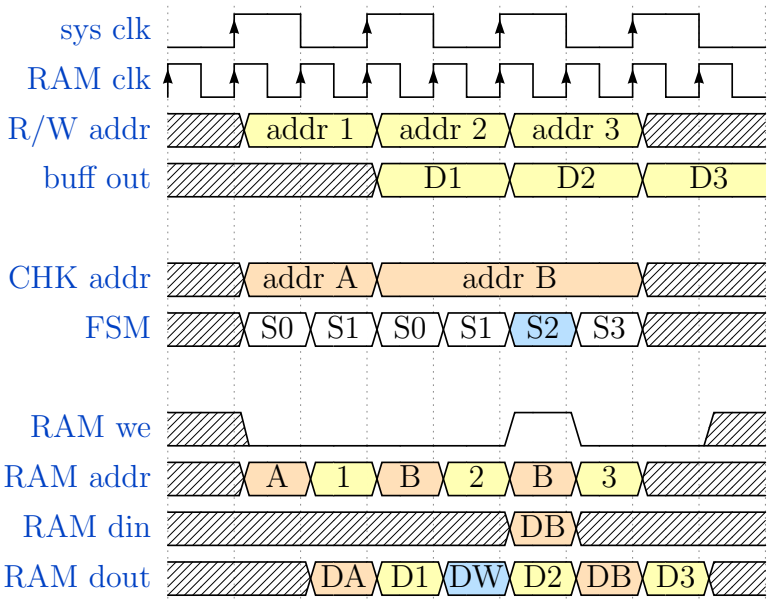


Figure 4.5: Example of the timing diagram without address conflicts. Addresses 1, 2 and 3 are different from address A and B. The refresh processes will not affect user operations.

cycles. In state 0 (S0), the RAM input port is occupied by the refresh controller for reading data from the target address (address A). Then in the next cycle (S1), output data from address A (DA) will be ready at the RAM output port. Because the decoder is combinational logic, the correcting results is also ready in the same clock cycle. If there is no error in the current address, the address controller will move to the next address, and the FSM will move to the next round and return to S0. If the decoding results show the data from the target address is incorrect (DW), the FSM will switch to the error processing state (S2). In this state, the corrected data will be written back to the targeted address. After writing back, the targeted memory unit will be refreshed. Finally, in state 3, the controller will be synchronised to ensure that S0 start from the next rising edge of the user clock.

Fig. 4.5 shows examples of the timing sequence of refreshing. "CHK addr" is the checking address, which is read by the refresh controller. "R/W addr" represent the address operated by the user module. "RAM addr" represent the actual RAM address in operation. The CHK addresses and the R/W addresses are represented by alphabet and numbers, respectively, to indicate that the user module and refresh controller are accessing the different addresses.

In this figure, "addr A" represents an address with correct data, while "addr B" represent an address with incorrect data. FSM start from S0 to read the data in address A. The output data and checking results are ready in the next state S1. Because there is no error, the FSM moves back to the S0 to read the data in address B. If there is an error in address B, then FSM will move to S2 to overwrite corrected data to address B. Finally, FSM is synchronised in S3 to ensures the S0 starts from the refresh clock cycle.

Furthermore, the buffer module will block the RAM outputs used by the refresh controller. The outputs of the buffer (D1, D2, D3) will correspond to the sequence of the user module operations (address 1, address 2, address 3). All the refreshing operations are invisible to the external modules. Following the shown timing sequence, the user R/W operations will not be interrupted by either reading or

rewriting processes. Hence, the self-refresh ECC RAM works like a normal single-port RAM for the user modules.

4.3.2 Conflicts of operations and strategy of scanning

Typically, the FSM works as shown in Fig. 4.5, the refreshing progress without errors lasts for two RAM cycles, and the refreshing progress with errors lasts for 4 RAM cycles. The user module may write new data to the same address that the refresh controller is rewriting. In this case, the new data may be covered by the out-of-date "corrected" data. In this work, cases in which the user module and refresh controller access the same address are called "address conflict".

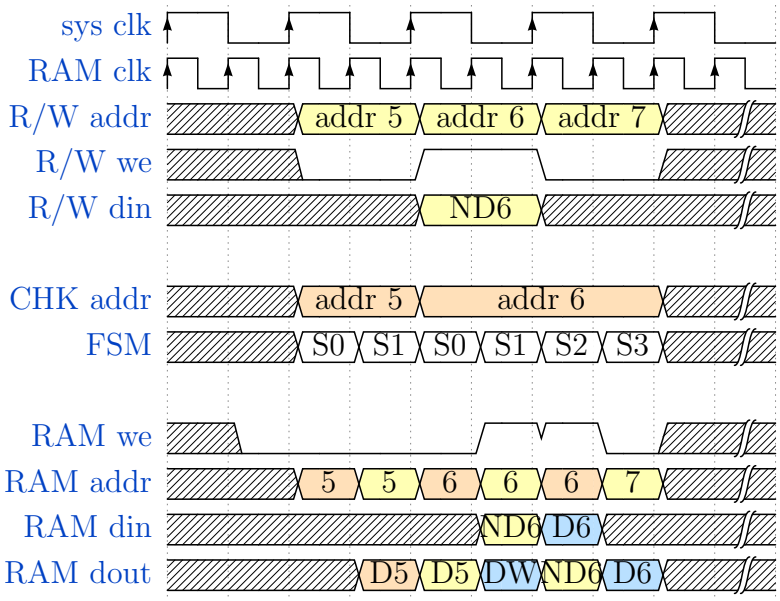


Figure 4.6: The errors caused by writing address conflicts. The user module is writing new data (ND) to address 6, while the refresh controller is writing the "corrected" data (DW is corrected to be D6) to the same address. The ND is overwritten by D6.

Fig. 4.6 shows the time sequence of the address conflict in the refreshing processes. In this figure, the user module and refresh controller are accessing the address 5 and 6 at the same time. In the shown case, the data in address 6 is incorrect. The user module is writing new data to address 6 (ND6), while the refresh controller is reading the old data from address 6 (D6). In the S1 for address 6, ND6 is written to address 6. However, out-of-data D6 is written subsequently

in S2 due to the previous checking results.

To solve this problem, the refresh controller mush monitor the address of user operations to ensure that there is no address conflict. In the proposed design, if the refresh controller accesses the address that is being written, it will give up the current operation and move to the next address directly, regardless there is an error or not. In this way, the refresh controller will not write out-of-date data back to the address in address conflicts by the costs of two cycles. However, considering that most external modules will access memory devices by order of address, there is a good chance that the refresh controller and the user module will continue accessing the same address. Thus, to lower the probability of the address conflict, the refresh controller accesses the memory units in reverse order.

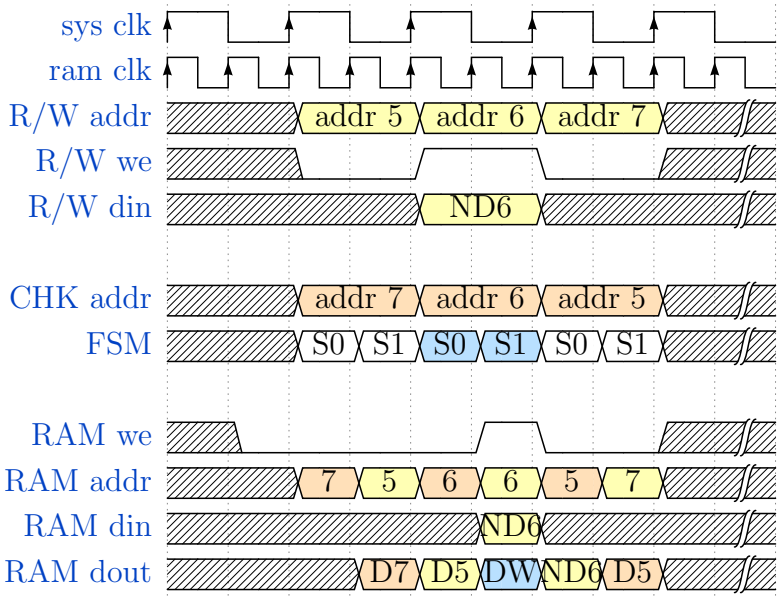


Figure 4.7: The actual timing Diagram of the proposed scheme. In address conflicts, the refresh controller gives up overwrite operations. Error bits are still corrected by the ECC decoder. The scan order of the address is also reversed to reduce address conflicts.

An example of the actual timing sequence of the proposed scheme is shown in Fig. 4.7. The user module accesses memory units by order of address 5, 6 and 7, while the refresh controller accesses memory units by address 7, 6 and 5. When there is an address conflict, the refresh controller will skip the current address (address 6), and check the next address (address 5), while the user module will

access a different address (address 7) in the next user clock cycles. In this way, there will be no continuous address conflicts.

4.3.3 Parallel architecture

Due to the difference between error refreshing and no error refreshing processes, the time of scanning all memory units is not constant. In this system, scanning time is based on the memory size, clock frequency and the number of detected errors. In a RAM where N memory units are under detection, the frequency of the working clock is f, and the scanning time T, can be represented by:

$$\boldsymbol{T} = \frac{2n_1 + 4n_2}{f} = \frac{2(N + n_2)}{f} \tag{4.1}$$

where n_1 represents the number of memory units without errors and n_2 represents the number of memory units with errors.

In order to correct errors before the occurrence of the sequential error, the scanning time should be less than the error generation time. Therefore, the maximal scanning time (T_{max}) for stable executing can be represented by:

$$\boldsymbol{T_{max}} = \frac{1}{NR} \tag{4.2}$$

where R represents the generation rate of bit flips and N represents the number of memory units under detection. For example, if the given generation rate of bit flips in an environment is 1×10^{-3} bit/(N·h), the scanning time for 1 MB RAM should be less than 1×10^{-3} hours.

$$\boldsymbol{T} = \frac{2(N + n_2)}{mf}, \tag{4.3}$$

where m represents the number of refresh controllers, n_2 represents the number of memory units with errors and N represents the number of total memory units under detection.

In this way, the number of units under detection can be altered to change the

scanning time. By using such multi refresh controllers, the time of scanning can decrease significantly, which decides the system's performance of SEU mitigation.

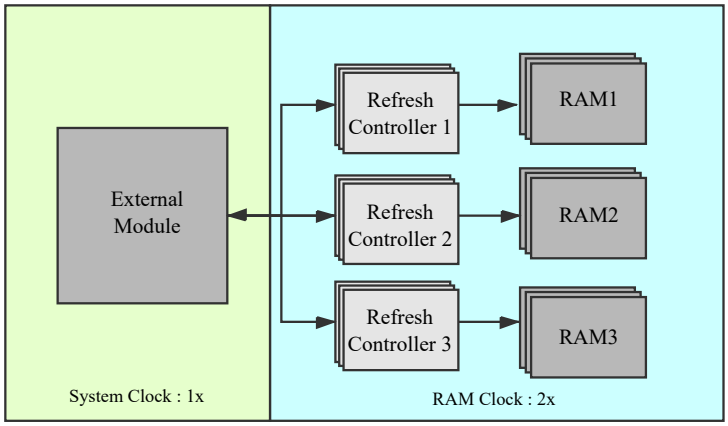


Figure 4.8: Architecture of multi refresh controllers.

The architecture of multi refresh controllers is shown in Fig. 4.8. The large RAM is divided into a set of smaller RAMs. The external module can access each small RAM by different addresses. Hence, different architecture can be used according to different environments. On the one hand, additional refresh controllers can be used in high radiation environments to compensate and achieve higher performance. On the other hand, the number of refresh controllers can also be reduced to save hardware resources at lower performance.

In addition, using small RAMs rather than large RAM means that the size of the refresh controller can also be small. Considering that RAMs run at twice the system clock frequency, it can also help resolve the setup time problems.

4.4 Fault injection platform and hardware simulation

A hardware simulation platform was built to carry out hardware SEU fault injection, verify the performance of the self-refresh ECC memory technology and conduct functional tests. The self-refresh ECC RAM was implemented in this platform to evaluate the performance.

4.4.1 Design of hardware fault injection platform

This platform includes an FPGA part performing SEU hardware simulation and a PC software for data analysis and human-computer interaction. Those two parts communicate via UART. The PC client part is mainly responsible for the operation control of the SEU simulation platform and the display of error correction results. The FPGA part is designed to implement SEU fault injection and ECC verification.

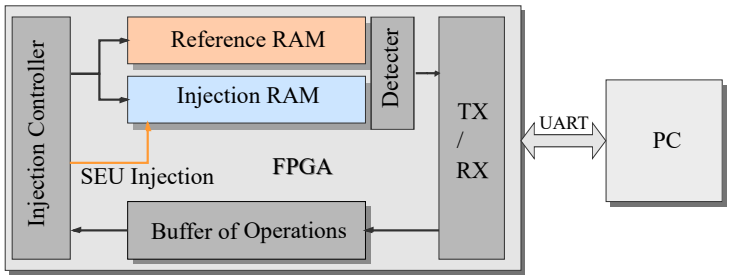


Figure 4.9: Architecture of Hardware Fault Injection Platform.

The main architecture of the hardware platform is shown in Fig. 4.9. The operations communicated from the PC are stored in a buffer. There are two RAMs under test: 1) a reference RAM which is a RAM without injection and 2) an injection RAM, where the SEU injection is conducted. In this work, both the reference RAM and the injection RAM will be replaced by the proposed design.

When a simulation test starts, the reference RAM and the injection RAM will be read or written simultaneously according to the pre-generated operations. At the same time, the injection controller will inject the error bits into the memory units of the injection RAM to simulate the occurrence of SEU. The platform can also simulate the intensity of radiation by adjusting the probability of the occurrence of the SEE and the frequency of SEU injection. This allows it to evaluate the effects of different factors and the performance of hardening design in different situations. Through an equivalent circuit without the injection, the simulation platform can simulate the module's state in both radiation and non-radiation environments.

In this work, the errors that may affect the system are called functional errors. In the unhardened RAMs, all the errors read by the system are functional errors,

while in the proposed systems, functional errors are the errors not corrected by the refresh controller. In the simulation, different outputs between the injection RAM and the reference RAM can suggest functional errors. The SEE mitigation performance is evaluated by comparing the functional error rates. All data generated in the RAM operation is exported to the PC to analyse the capability of SEU mitigation of the tested module in real-time.

4.4.2 Fault injection in the hardware platform

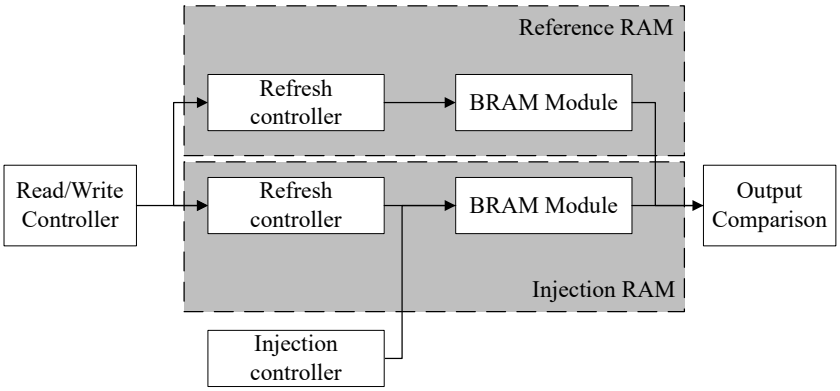


Figure 4.10: The injection controller will bypass the refresh controller and write error bits into RAMs directly. The BRAM module is modified to accept data from both the refresh controller and the injection controller.

Unlike the software simulation, the hardware simulation cannot simply simulate the occurrence of SEU by just specifying bit flip. The hardware simulation platform has incorrect bits written into the target memory units to simulate SEUs. To ensure that the RAM logic function will not change during execution, we adopted the idea of self-refresh ECC RAM and doubling the system clock to make the injection controller operate at a higher frequency. Hence, the proposed simulation system can make use of extra clock cycles to perform SEU injections.

Also, to detect the effects of SEUs on different instruction sequences in RAM, another RAM is used to store instructions. By reading or writing the RAM with a specific operation order, this platform can also simulate the code execution of different software programs. Hence, the capability of SEU mitigation in different programmes can also be evaluated.

Fig. 4.10 shows the block diagram of the injection modules for evaluating the proposed design. In the simulation, the injection controller is connected to the BRAM module directly. BRAM modules are not just BRAMs. It is a module with BRAMs inside and can conduct operations from the refresh controller and injection controller. By using a similar method (e.g., double frequency), the refreshing operations controller will not be affected.

During hardware simulations, the refresh controller should keep sending read or write operations to the BRAM module. The read/write controller will work as the user module and keep sending operations to refresh the controller. Meanwhile, the injection controller writes error bits into BRAMs from time to time. Inside the injection controller, there is a random number generator. After each injection, it will generate a random interval time for the next injection. The random seed, average injection time and floating range can be set by users. The actual interval will float randomly within the range around the average time.

4.4.3 Results of simulations

Table 4.1 shows the main technical specifications of the SEU hardware simulation platform, which is built on a Xilinx Virtex-5 XC5VLX110T FPGA. The system works at a frequency of 100 MHz and has the ability of SEU injection at a maximal frequency of 50 MHz. The executor programming simulated by the hardware platform works at a frequency of 25 MHz which equals the frequency of reading or write (R/W) operations. Hence, according to Equation 4.1, the scanning time is approximately 160 µs, and the SEU rate is 6.25×10^4 bit/s.

There are three simulation modes to satisfy different requirements: 1) Single-mode means performing all saving R/W operations for one round, which is designed to test certain programmes, 2) Loop mode means repeating R/W operations for specified times rounds, which is designed to test the performance of the systems in a specified time, 3) Unlimited mode means continuously R/W operations until it is stopped by users, which is designed to evaluate the error rates.

Table 4.1: Specifications of hardware fault injection platform

Description	Specifications
RAM size for SEU injection	4096
Size of command buffer	4096
Maximum number of R/W operations	4096 × 256
System clock	100 MHz
Frequency of Read or Write operation	25 MHz
Clock of refresh controller	50 MHz
Max frequency of SEU injection	50 MHz
Simulation mode	Single/Loop/Unlimited

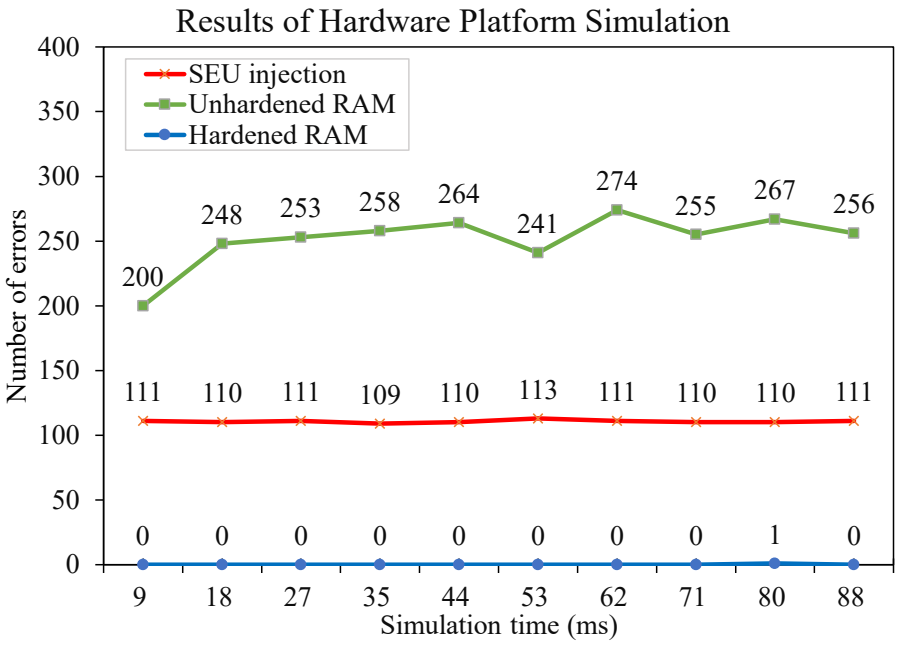


Figure 4.11: Results of simulations with 80 μs injection time.

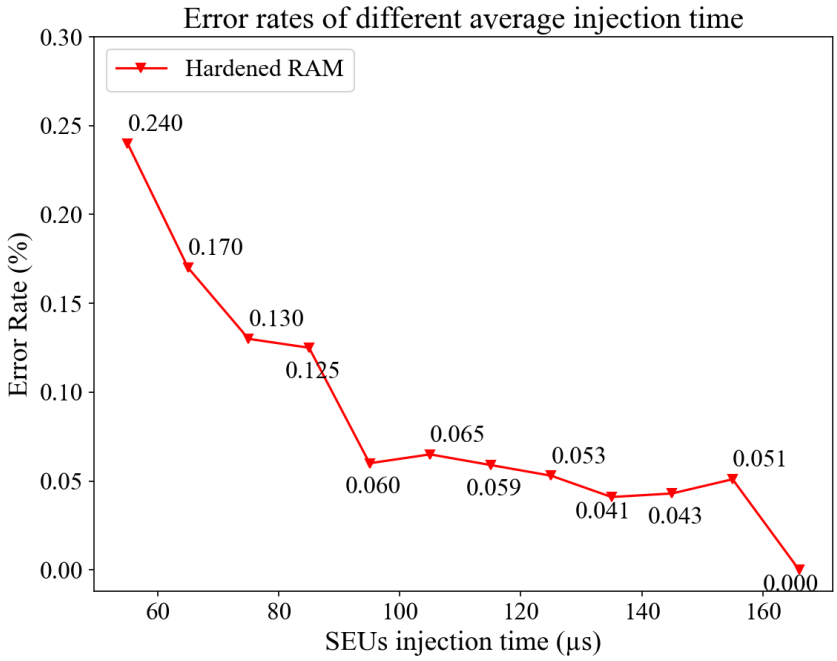


Figure 4.12: Error rates for different average injection time.

Fig. 4.11 shows the performance of the self-refresh ECC RAM and an unhardened RAM in hardware simulation. The injection time represents the average interval time between two SEUs. In this simulation, all the operations are generated by the PC in a randomly generated order. Because the number of operations is much higher than the number of memory units, the injection controller may access the same unit multiple times before the errors in this memory unit are covered or refreshed by new data. In other words, the number of detected functional errors may be greater than the number of SEU injections. As we can see, after using the self-refresh ECC technique, the number of functional errors is reduced to almost 0, which validates that the proposed design effectively avoids functional errors caused by SEU. Error rates of different average injection times are shown in Fig. 4.12. When the average injection time is more than 100 us, which means that the SEU rate is 1×10^4 bit/s, the self-refresh ECC RAM can handle 99.9% of errors. When the injection time is equal to the scanning time, which is 160 µs, the self-refresh controller can handle 99.97% of the errors.

The injection RAM and the reference RAM are changeable in this platform. By replacing the injection RAM and the reference RAM with other hardened

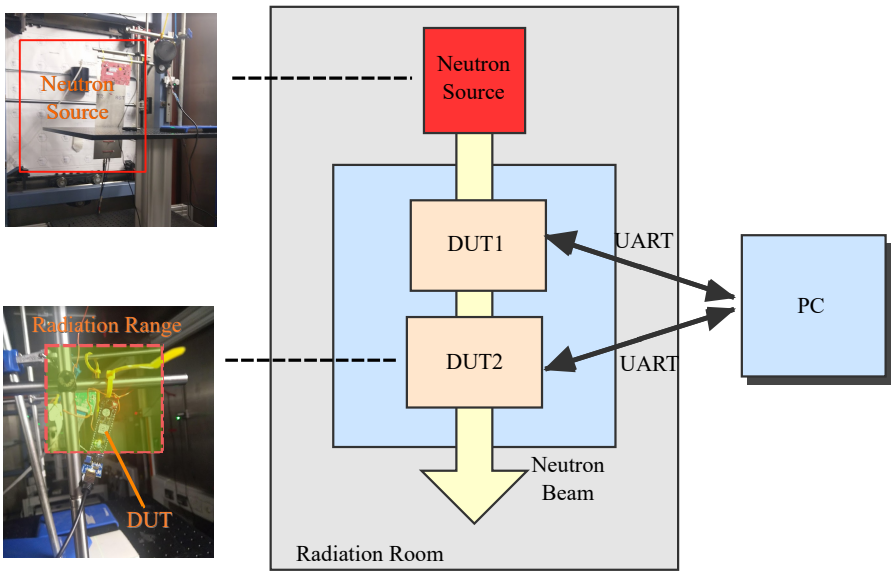


Figure 4.13: Setup of neutron radiation experiment.

RAMs, this platform can also be used to evaluate the performance of other SEU reinforcement designs. The reliability, functionality and effectiveness of self-refresh ECC RAM are verified by this platform. However, hardware simulation is not equivalent to testing with real radiation. Therefore, it is necessary to test the proposed systems in real radiation environments.

4.5 Neutron radiation experiments

In order to evaluate the real-world performance of the system, the experiments are conducted with neutron radiation. Neutron radiation was used to create an extreme environment to evaluate the SEU mitigation performance of self-refresh ECC RAM [195, 196, 197].

4.5.1 Setup of the neutron experiment

Radiation experiments were conducted at the ChipIr facility at ISIS, Didcot, UK [198]. ChipIr provides a neutron spectrum which is suitable to emulate the effects of terrestrial neutrons in electronic devices and systems. The ChipIr neutron flux (with $E_n > 10$ MeV) has been measured to be approximately $5 \times 10^6\ cm^2 s^{-1}$. The neutron flux at ChipIr is about 8 to 9 orders of magnitude higher than the

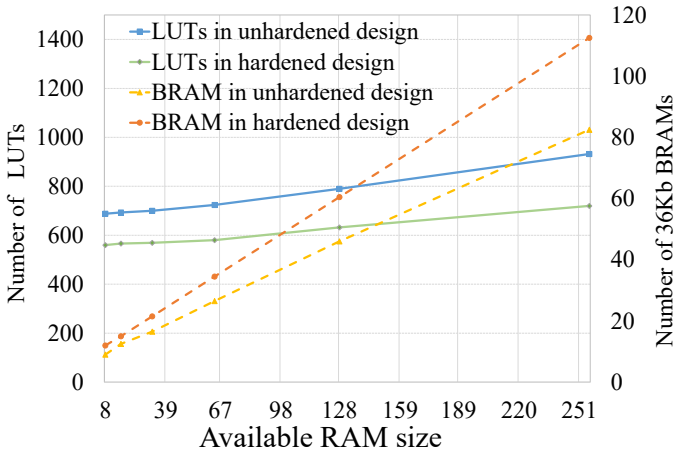


Figure 4.14: The hardware costs with the scale of the design.

terrestrial flux at sea level. As calculated by the scientists from ISIS, the radiation dose on water (human body) is about 20,000 mSv/h in our experiments.

Two experiments are designed to evaluate the proposed design: 1) comparison between unhardened RAMs and the self-refresh RAMs and 2) comparison between the conventional ECC RAMs and the self-refresh RAMs. In the experiments, the systems are placed under the neutron beams for several hours, which amounts to a neutron yield of 6.5×10^{10} per hour considering a 7 cm x 7 cm beam. This is equivalent to more than half a million years of natural exposure.

The setup of the neutron experiment is shown in Fig. 4.13. The devices under test (DUT) are placed in the radiation room. Because of the strong penetration of the neutron beam, it can penetrate all test boards. The DUTs in the radiation room are connected to the PC in the control room by long USB cables. In order to reduce the impact of other electronic components, the communication modules and power supplies used in the experiment were kept outside of the radiation range.

4.5.2 Hardware implementation of the proposed design

In the radiation experiment, the self-refresh ECC RAM is implemented on Digilent Cmod A7-15T which is a low-price entry-level FPGA development board. The chip on this board is an Artix-7 XC7A15T-1CPG236C FPGA with 112.5 KB block RAM inside.

Table 4.2: Specifications of Hardware Implementation

Description	Specifications	
	Unhardened RAM	self-refresh ECCRAM
Available RAM size	32 KByte	32 KByte
BRAM*	16.0	21.50
LUT	562(5.4%)	681(6.5%)
LUTRAM	8	8
FF	467	575

* The number of the 36Kb BRAM used in the FPGA

The basic architecture of DUT is similar to the hardware injection platform discussed above, but no injection controller and reference RAMs. The design of the experiment circuit includes two parts: 1) the UART controller and 2) the target RAM. When the boards are working, the PC client sends compressed write or read commands to the FPGA part via UART periodically. Subsequently, the UART controller operates target RAMs according to those commands. All outputs of the read operations will be sent to the PC immediately to avoid the impacts of radiation.

The specification of the hardware of the whole design is shown in Table 4.2. Both the reference RAM and the target RAM have the same available memory size, which is 32 KB and the same available bandwidth for external modules. Hence, the self-refresh ECC RAM consumes slightly more memory and requires a higher frequency of the RAM clock. The unhardened design and hardened design consume 16 and 21.5 BRAM units, respectively. The operating frequency of the unhardened RAM and the self-refresh ECC RAM are 50 MHz and 100 MHz, respectively. The number of the LUTs for the entire design included UART and operation parts. The unhardened design and hardened design use 562 and 681 LUTs, respectively. The additional 121 LUTs are used to build the self-refresh controller.

The scalability of the proposed design is shown in Fig. 4.14. It shows the trends of the utilisation of LUTs and BRAM with available RAM size. When the available RAM size is 8KB, the unhardened and hardened designs consume 560 and 688 LUTs, respectively. When the available RAM size was 256 KB, the unhardened and hardened designs consume 720 and 932 LUTs, respectively. The

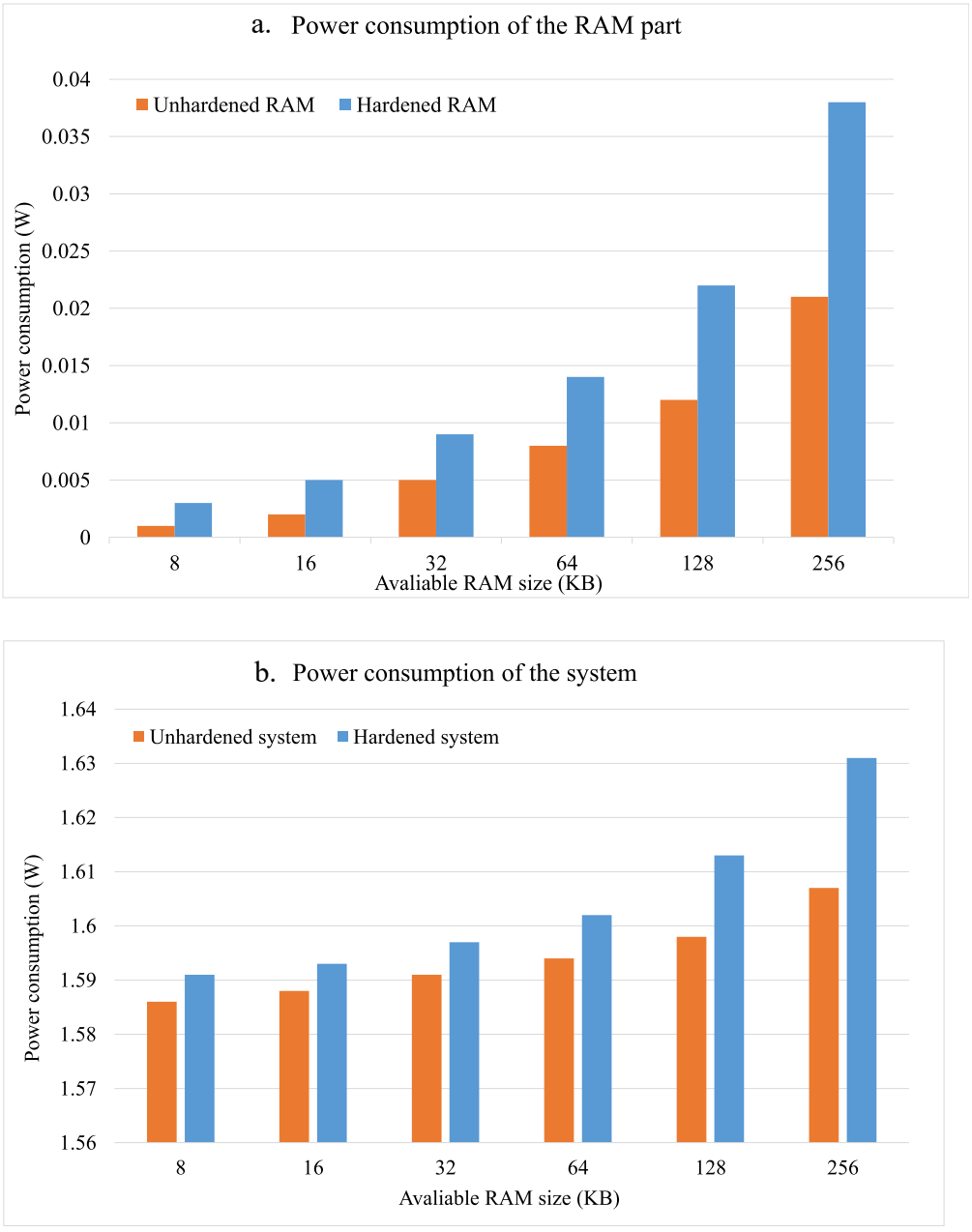


Figure 4.15: The power consumption with the scale of the design.

additional utilisation of the LUTs scales up slightly for wider bandwidths. The refreshing controller itself does not scale up. The utilisation of BRAM grows linearly with the available RAM size.

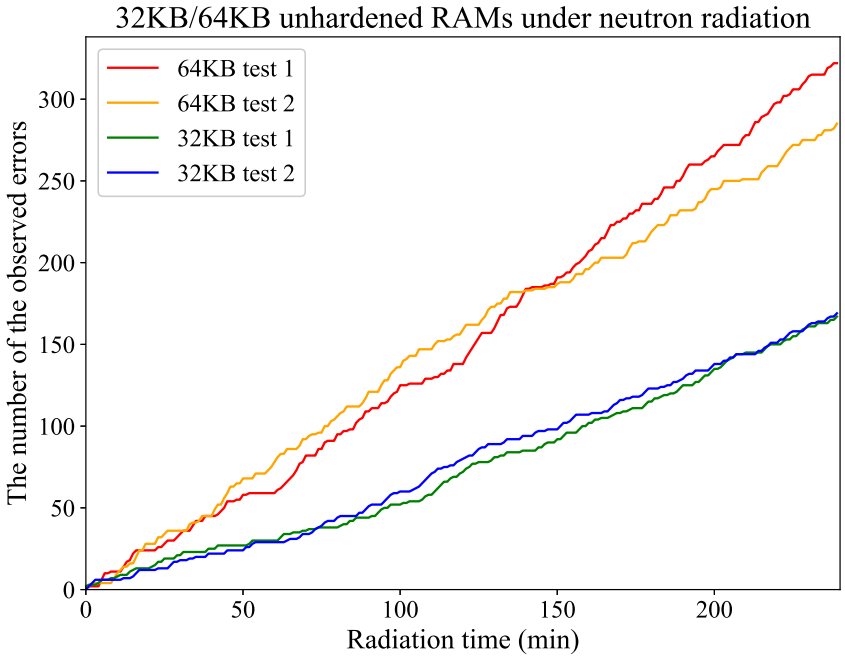


Figure 4.16: Number of the errors in the unhardened RAMs.

The power analysis generated by the Vivado analysis tool is shown in Fig. 4.15. Compared to the unhardened RAM, the power consumption of the hardened systems also slightly scale up with the RAM size. There are two parts to the additional power consumption. Firstly, the dynamic power consumption of the self-refresh ECC RAM increases for operating at a higher frequency. Secondly, the hardened design consumes more power for the additional control circuits.

When the available RAM size is 8 KB, the unhardened RAM and the hardened RAM consume 0.001 W and 0.003 W, respectively. When the available RAM size is 256 KB, the unhardened RAM and the hardened RAM consume 0.02 W and 0.038 W, respectively. However, the RAMs consume much less power than the processor in the designs. The total chip power consumption does not increase significantly. When the RAM size is 256 KB, the hardened system consumes 1.4% more power than the unhardened system.

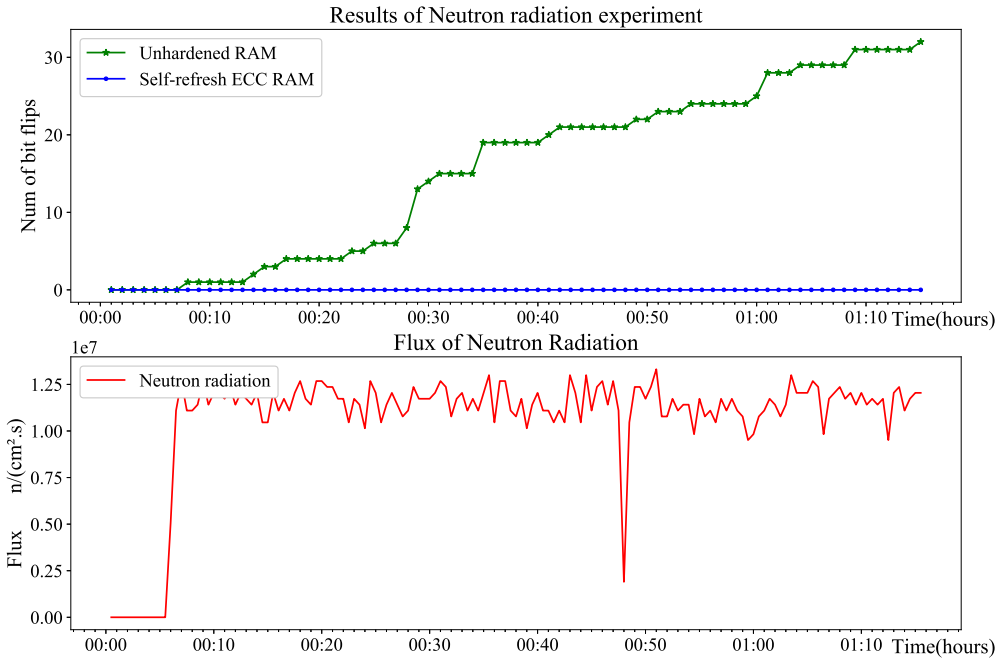


Figure 4.17: Observed errors in hardened and unhardened RAMs during the one hour neutron experiment.

4.5.3 Analysis of the return data

The entire FPGA board was placed in the radiation room during the radiation experiments. Hence the communication and control modules implemented in FPGAs were also irradiated. It is possible that the detected error bits are false-positive results instead of the actual errors in BRAMs. Fortunately, it is possible to identify the error types in the systems. In the experiments, the PC will read back all the raw data from the FPGAs every 5 seconds. the current read back data with the next data will be compared to see if there are some changes. According to our experiment, it can be found that the error bits can be categorised into three types: 1) bit flips that can be read back continuously, 2) transient bit flips that show up once and 3) many unexpected error bits in line. In the proposed design, if there are too many multiple error bits in a byte unit for the controller to fix, the refresh controller will leave it alone and rewrite the correcting codes to avoid repeating detection. Hence, if there are errors in BRAMs that refreshing cannot correct, behaviours of errors should fall in the first category. Because the communication modules keep receiving and sending data, it is likely that errors in the commu-

nication modules are refreshed by the new data instead of continuously showing up in the location of the read back data. Finally, if there are a large number of unexpected errors, it is most likely that a failure happened in the controller systems. Therefore, the errors in the proposed design can be identified if the errors are repeatedly presented in the reading back data.

4.5.4 Results of the radiation experiment

4.5.4.1 Unhardened RAMs

The number of errors in the unhardened RAMs with different memory size is shown in Fig. 4.16. In the experiments, the experiments are conducted with the 32 KB RAMs and 64 KB RAMs for 4 hours. The number of the observed errors in both 32 KB and 64 KB RAMs increases linearly with the radiation time. After 4 hours of radiation experiments, the average number of the observed errors in the 32 KB RAMs is 159. The average number of observed errors in the 64 KB RAMs is 303, which is approximately double the number of the errors in the 32 KB. The errors rates of the RAM in the given radiation environments can be represented as follow:

$$\boldsymbol{R} = \frac{N_{error}}{T_{radiation} \cdot S_{RAM}}, \tag{4.4}$$

where the R represents the error rates of the RAM in the given environment, N_{errors} represents the number of the error bits in total, $T_radiation$ represents the radiation time and the S_{ram} represents the size of the RAM. Therefore, it can calculated that the observed error rates of the RAMs under the neutron radiation in this paper is approximately 1.2 bit/(KB·h).

4.5.4.2 Unhardened and self-refresh RAMs

The results of the neutron radiation experiments of the comparison between unhardened RAMs and the self-refresh RAMs are shown in Fig. 4.17. The number of bit flips in self-refresh ECC RAM remains at zero during the entire experiment,

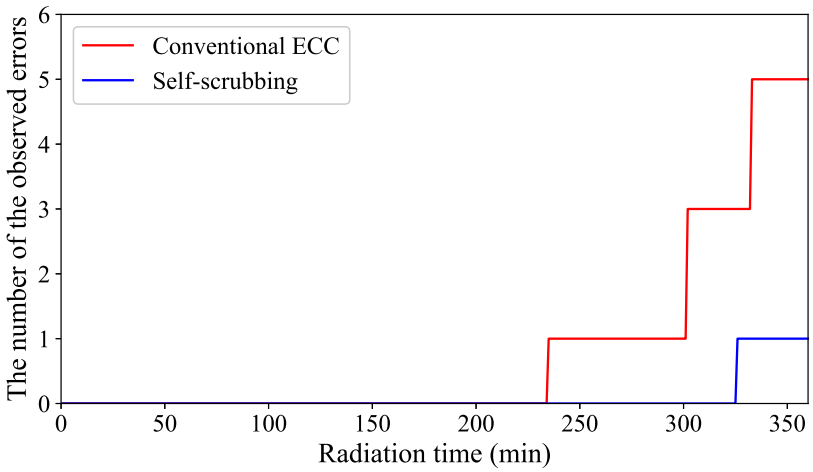


Figure 4.18: Comparison between the conventional ECC RAMs and the self-refresh ECC RAMs.

while the number of bit flips in unhardened RAM rises to 32 in the initial 1.5 hours. As both RAMs are working in the same radiation environment, it proves that the design of self-refresh ECC RAM is effective for SEU mitigation.

4.5.4.3 ECC and self-refresh RAMs

The comparison results between conventional ECC RAM and self-refresh RAM are shown in Fig. 4.18. In this experiment, the conventional ECC RAM is the RAM hardened by Xilinx official ECC modules [199], which is used as the reference RAM. After the 360 minutes radiation experiment, the total number of observed errors in the conventional ECC RAM is five while the number of errors in the self-refresh RAM is only one.

Table 4.3 shows the SEU cross-section of BRAMs on Artix-7 FPGAs in different neutron radiation experiments. Despite the experiments performed in different radiation environments, the SEU cross-section of unhardened RAMs is about the same order of magnitude. Compared to the unhardened RAMs and conventional ECC RAMs, the self-refresh RAMs achieve better error mitigation performance in neutron radiation environments. In our experiment, the error rates of the self-refresh RAMs are 8.7×10^{-5} bit/(KB $\cdot$ h) and the calculated SEU cross-section is

Table 4.3: SEU cross sections of SRAM on Artix-7 FPGAs in neutron radiation environments

Radiation facilities	Integrated flux (MeV)	Hardening	Cross section ($cm^2 \cdot bit^{-1}$)
ChipIr (this work)	10	None	2.22×10^{-14}
ChipIr (this work)	10	ECC	5.79×10^{-16}
ChipIr (this work)	10	SR*	1.16×10^{-16}
CERN [200]	20	None	3.18×10^{-14}
GENEPI2 [201]	14.2	None	1.2×10^{-14}
TTEA [202]	1	None	7.6×10^{-15}

* SR indicates self-refresh.

Table 4.4: Comparison between existing scrubbers

	processor	scrubbing rate	RAM size
Proposed	no	20us	32KB
internal [23]	no	19.32 us	161 B
external [203]	yes	360s	96-1392* KB

* The scrubber is designed to scrub the whole Virtex-4 devices. The RAM size of Virtex-4 ranges from 96KB to 1392KB

1.16×10^{-16} $cm^2 \cdot bit^{-1}$, which is one-fifth of the error rate of the ECC RAM.

When the experiment continues, there will be noticed errors in the RAM with self-scrubbing methods. The radiation effects can cause only soft errors but also hard errors. The hard errors can not be corrected. If the number of hard errors exceeds the capability of error correction. The errors will be observed. Normally, the RAM units should be banned for accessing, which is not discussed in this thesis.

4.5.4.4 Internal and external scrubbers

Table 4.4 shows the comparison between internal and external scrubbers. Compared with internal scrubbers, the proposed scrubber has a higher scrubbing rate. Compared with external scrubbers, the proposed design require neither processor to conduct scrubbing operation nor dedicated components to access RAMs. Therefore, the proposed scheme has advantages in flexibility and SEE mitigation

performance.

4.6 Conclusion

This chapter proposes a scheme that combines ECC and refreshing methods to mitigate SEUs for the devices supposed to work in extreme radiation environments. Compared to the conventional refreshing method, it can refresh memory units separately with high frequency without interrupting operations from the user module. The proposed scheme requires no additional RAM ports so that it can be applied in a wide range of hardware systems. In addition, by modifying parallel architecture according to the density of radiation, this design can achieve a balance between performance and hardware costs.

The experiments are conducted in neutron radiation environments. It is shown that the error rates remain robust irrespective of the RAM size. The comparison of the radiation experiments also shows that the self-refresh scheme is an effective strategy for hardening embedded systems and the error rate of the self-scrubbing RAM is one-fifth of the conventional ECC RAM.

The main contributions of this work are stated as follows:

1) The design is highly flexible. Compared to conventional external scrubbers [61, 62, 63], the work is transparent to other modules. No additional latency is introduced in the system. There is no need to modify the designs to adapt to the hardware changes, hence, it can be easily applied in various embedded systems.

2) The design is an area-efficient design, which can be used to harden low-cost computer systems. Compared to conventional internal scrubbers [64], this design requires no dedicated components (e.g., internal configuration access port). In systems with multiple customised modules which operate separated RAMs, the proposed design can be deployed multiple times to protect one or more modules.

3) The SEU mitigation design can achieve high SEU correction rates in various conditions. The results of the simulation and the radiation environment test follow the same trend. In the simulation, the proposed design can correct more than 99.97% of SEUs errors at the SEU injection rate of 6.25×10^4 bit/s. In the neutron radiation experiment, the SEU correction rate achieves 100%, when the flux of neutron radiation is 5×10^6 $cm^2 s^{-1}$.

Chapter 5

Hardware acceleration for multiple tasks

5.1 Introduction

Due to the requirement of substantial computational resources, it is practically difficult to execute multiple challenging hardware tasks simultaneously in resource-constrained systems. For example, a typical FPGA-based satellite system [17] is required to execute multiple algorithms (tasks), including general spacecraft operations (e.g., automated control and navigation), the analysis of payload data (e.g., weather and atmospheric monitoring [204]) and radiation hardening (e.g., SEU mitigation [17]). Such systems are constrained by processing resources, memory, power and even communication bandwidth. Moreover, high accuracy and timeliness are required to execute the respective mission-critical tasks. Therefore, it is challenging to ensure the efficient execution of multiple applications on the basis of dynamic performance, low hardware costs and low power consumption.

As an important feature of modern FPGAs, PR [162] is capable of dynamically reconfiguring the specific areas of an FPGA after its initial configuration [205]. A typical application of PR is the dynamical deployment of multiple identical modules (circuits) for TMR systems [163] to mitigate transient faults. The PR

feature improves flexibility and enables the hardware reuse of systems based on FPGAs.

However, due to the challenges arising from the PR technique (e.g., difficulties in the design for both hardware circuit design and software drivers), the application of PR is still limited. As mentioned in Chapter 2, Xilinx launched a new technology known as DFX [206]. It represents a comprehensive solution involving many parts such as PR features, hardware IPs and software run-time, which contributes a convenient methodology to the design and dynamic implementation of applications. Under this framework, Xilinx also offers standard DPU IP to accelerate AI inference tasks, which facilitates the building of PR systems.

In this work, an adaptive hardware system intended for DL tasks [207] is proposed to manage the hardware resources seamlessly according to the exact system requirements. By using DFX, the executing system is capable of dynamically allocating the hardware resources according to the exact requirements, e.g., performance and power consumption. Through the deployment of hardware accelerators with different configurations, it is achievable to dynamically adjust the performance, power consumption and available FPGA resources for various tasks. Under the proposed scheme, the hardware accelerators are grouped into accelerator pools to accept acceleration tasks, so the reconfiguration can be conducted seamlessly, without disrupting the executed programmes.

The power consumption and performance of the proposed system are evaluated in this work by implementing multiple hardware configurations which range from one DPU at 100 MHz to two DPUs running at 300 MHz. According to the experimental results, the proposed scheme is effective in improving the efficiency of the resource-constrained systems operated under multiple scenarios. Compared to the conventional system, the proposed system consumes 38% and 82% of power in the case of low working loads and high working loads, respectively. Under extreme scenarios, the proposed system can reduce energy consumption by as much as 75.8%.

5.2 The system with dynamic management

Considering the tasks and working loads related to resource-constrained embedded systems that may vary from one scenario to another, the accelerators are typically designed to accommodate maximum working loads. Thus, there exist redundant hardware resources because the maximum given resources are not required for most tasks. Considering the multiple task system, the redundant resources of one task cannot be used directly for other tasks, which will lead to the waste of hardware resources. In order to fully utilise the hardware resources for improving the efficiency of the system, an adaptive hardware system is proposed in this work to dynamically manage the hardware resources. In this system, the accelerators will be flexibly allocated according to different workloads and task scenarios.

5.2.1 Software architecture

Fig. 5.1 shows the overall system architecture of the proposed scheme consisting of three layers: 1) the management of acceleration requests from applications, 2) the mapping of the acceleration tasks on hardware accelerators and 3) the management of reconfiguration in physical partitions.

In this system, the top layer is the software layer, which is conducted by PS. “APP1”, “APP2” and “APP3” represent the respective applications, which may require hardware acceleration. When the system is executed, the requests from applications will be sent to the acceleration tasks for management. Depending on the type of their corresponding accelerator, these tasks will be placed in a number of task queues. When there are free accelerators in the system, the task manager will send them to the hardware accelerators.

Intended to manage the hardware accelerators and hardware resources, the middle layer includes both processing system (PS) and programmable logic (PL). There are two parts in this layer: configuration manager and accelerator pool manager. The configuration manager aims to manage the hardware configurations and conduct partial reconfiguration. It is capable of dynamically deploying new

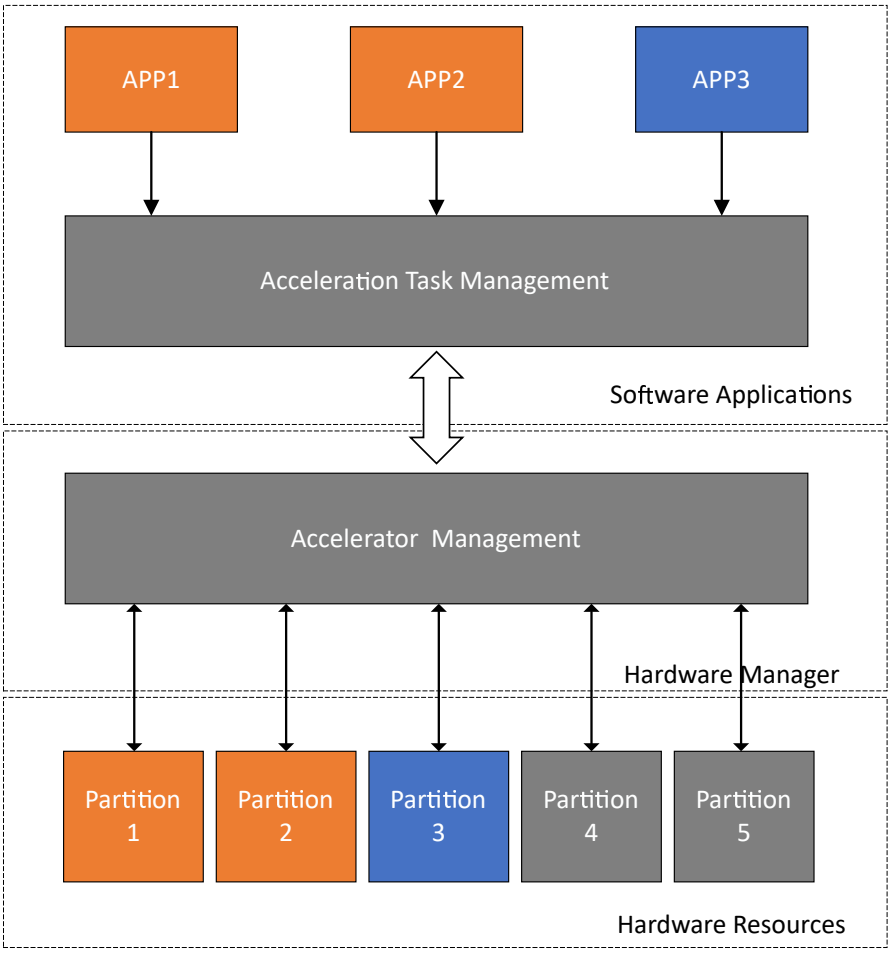


Figure 5.1: The system architecture of the proposed scheme. "APP1" and "APP2" have the same orange color, which means that they can share the same accelerators. The blue color of "APP3" indicates that it requires another accelerator.

accelerators to achieve better performance or releasing unused hardware resources to reduce power consumption. The accelerator pool manager is purposed to manage the accelerators. Considering the changes to hardware configurations, it is unrealistic to send the acceleration tasks from the top layer to the accelerators directly. Therefore, the accelerators with the same functionalities are categorised into the same groups, which are referred to as accelerator pools. During the reconfiguration, there will be no change to the pools, which allows the acceleration tasks to be sent to the pools regardless of the changes in the hardware.

The bottom layer is the hardware resources layer which is aimed to manage the partitions in the PL part. In this scheme, the dynamic hardware resources are divided into multiple partitions in advance, with the physical partitions fixed in the system. In each partition, there will be some requisite hardware resources for deploying the accelerators, including LUTs, RAMs and DSPs. When the applications are running, those partitions can be reconfigured separately. In case additional performance is required for the acceleration tasks, free partitions will be deployed to build the accelerators in run-time. If the performance is satisfactory, the partitions can also be reconfigured as empty partitions. In this way, the amount of power consumption could be reduced.

5.2.2 Hardware architecture

The system proposed in this work is targeted at Zynq UltraScale+ devices. In general, there are two major parts in this architecture: an ARM-based PS and PL, which contains LUTs, RAMs, DSPs and other dedicated hardware blocks. The PL part is divided into two regions: a static region and a dynamic region. Fig. 5.2 shows the basic hardware architecture of the proposed system. The static region is the FPGA region, where the basic components (e.g., PLL) are implemented. However, these components cannot be reconfigured in the executed systems. For example, the PLL module provides stable clocks for all the running modules. Because it works even during the reconfiguration operations, it is implemented in the

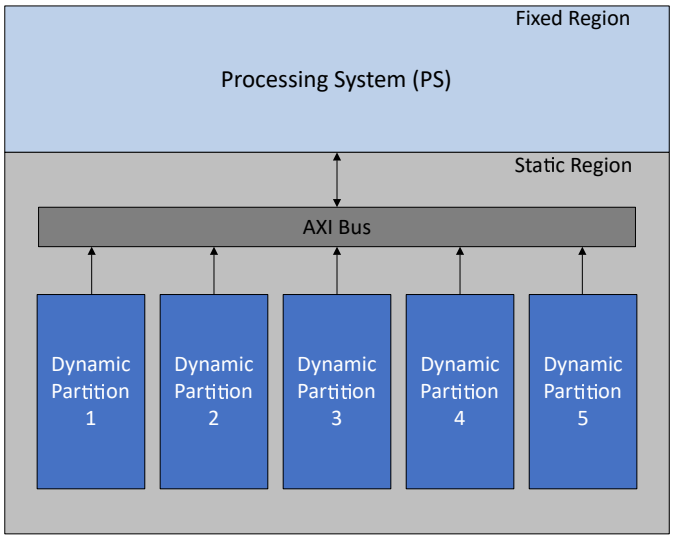


Figure 5.2: The hardware architecture of the proposed design and the reconfiguration partitions.

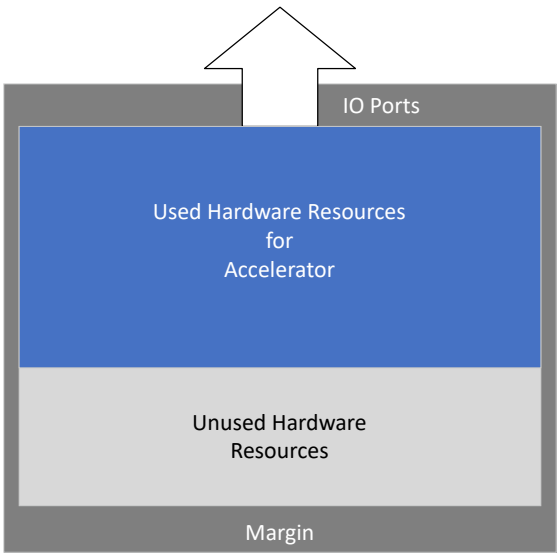


Figure 5.3: The hardware architecture of dynamic partition.

static region.

The dynamic region is the region where the accelerators are implemented, including a number of reconfiguration independent partition blocks for deploying accelerators. During the reconfiguration operations, there will be new accelerator circuits built by reconfiguring the entire area of the partition. Fig. 5.3 illustrates the basic architecture of the dynamic region. The "used region" indicates the hardware resources in the partitions that have been utilised to build accelerators, while the "unused region" indicates the unused resources. To continue the operation of the circuits in the static region, a margin area is required. It occupies a small amount of hardware resources to ensure that the static region will be unaffected during the reconfiguration operations. Finally, there will be fixed Advanced eXtensible Interfaces (AXIs) involved to transmit the data through the margin.

5.3 Working flow of the proposed system

As introduced in section 5.2, the dynamic changes to the hardware accelerators impede the applications from any direct communication with specific accelerators. Under the proposed scheme, those physical accelerators with the same functionalities will be managed in the accelerator pools. The tasks will be sent to the pools for assignment by the accelerator manager, which limits the changing of the hardware accelerators in the accelerator group, thus avoiding the potential impact on applications. In this section, a discussion will be conducted about the workflow and the strategies for deploying accelerators in detail.

5.3.1 Acceleration tasks

In the proposed system, there are multiple applications running simultaneously. Due to the constraints on hardware resources, the same accelerator may be used to conduct hardware acceleration for some applications. For this reason, the requests from applications are packaged into the accelerator tasks which can be sent to the

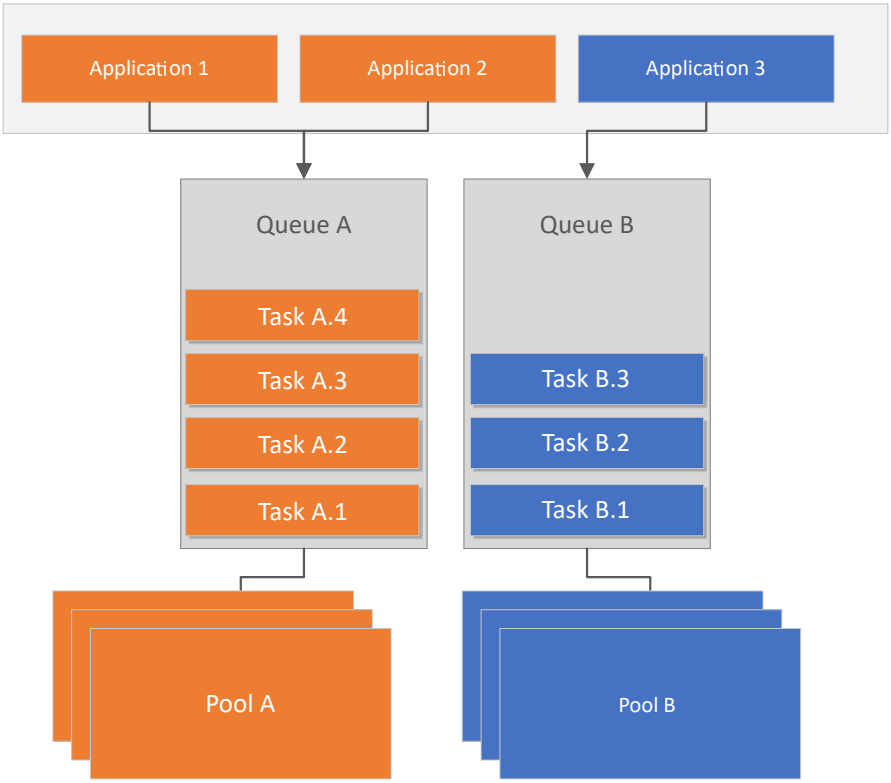


Figure 5.4: The acceleration tasks are sent to the hardware accelerator pools.

hardware accelerator directly. Because the type of tasks is associated with the type of hardware accelerators, the hardware resources can be allocated to the specific acceleration tasks instead of specific applications, thus achieving the maximum efficiency.

There are three steps in sending the acceleration tasks to the hardware accelerations. Firstly, the applications send acceleration requests to the systems. Then the requests will be packaged into the acceleration tasks, with the necessary information (e.g., the identities of the source applications, priorities, estimated execution time). Secondly, those tasks will be placed into a task queue according to the packaged information. The number of task queues is determined by the number of task types. There will be multiple applications sharing the queues if the same types of accelerators are required. In the third step, the task manager will be deployed to check the corresponding accelerators for each queue. If there are accelerators available, then the task manager will select the tasks from the front of the queue and send them to the accelerators.

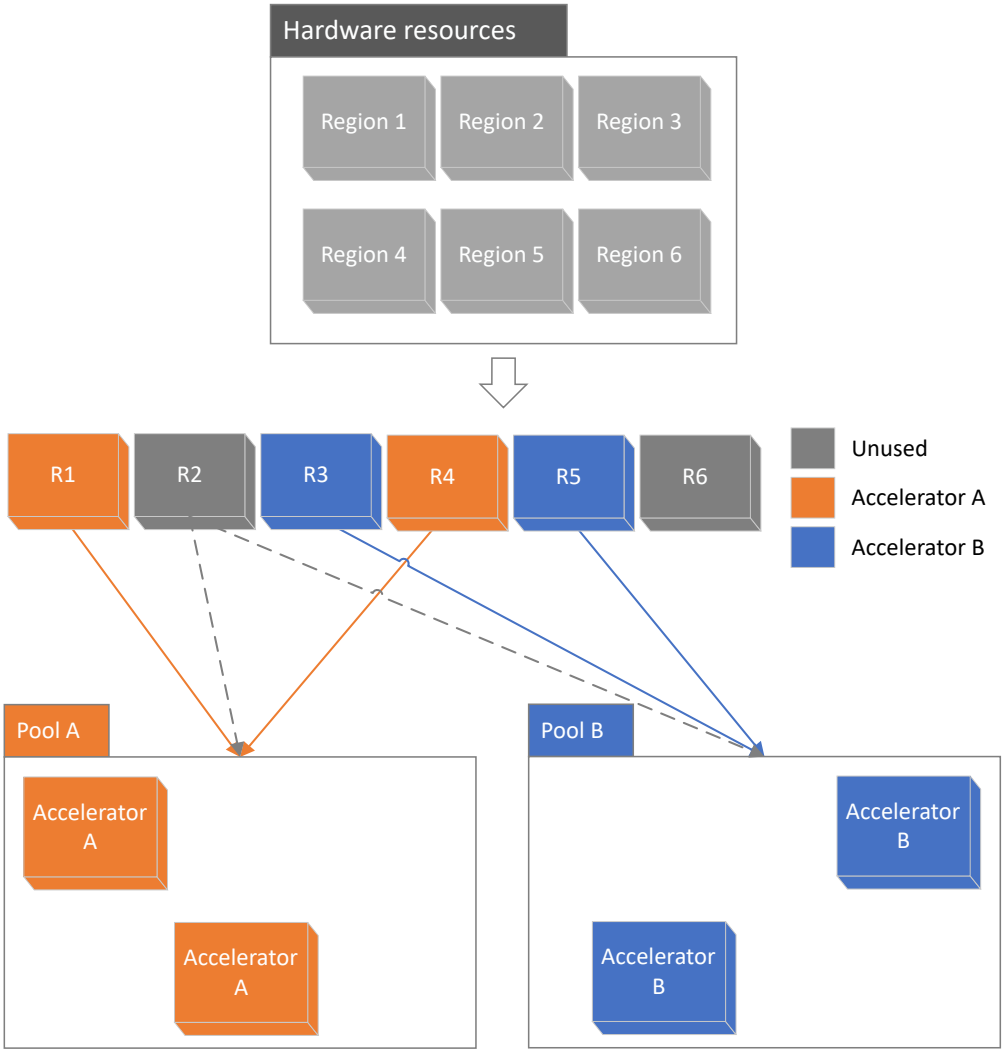


Figure 5.5: The deployment of hardware accelerators.

Fig. 5.4 shows an example of acceleration task management. In this example, there are three applications (i.e., application 1, 2 and 3) running in the system, to send two types of acceleration requests (i.e., A and B). Applications 1 and 2 will send tasks to accelerator A, While Application 3 will send tasks to accelerator B. The tasks are placed in separate queues according to their types. Pool A and Pool B represent the combination of accelerators A and B, both of which are managed by the pool manager. If there are accelerators available in the pools, then the pool manager will request the task manager to send the new tasks.

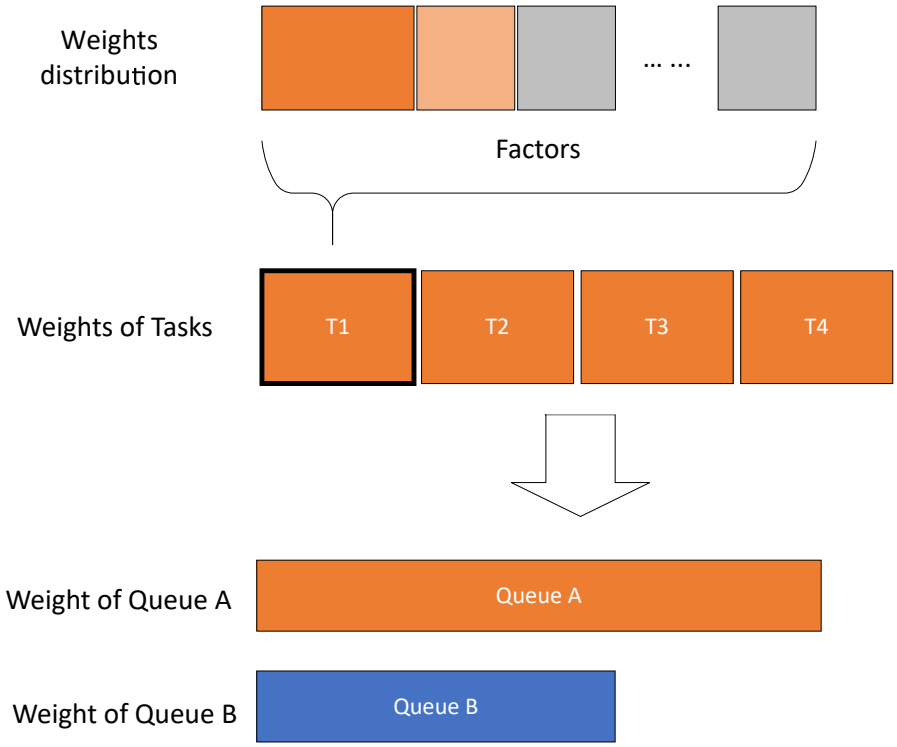


Figure 5.6: The comparison of the weights between queue A and B. The weights of queues indicate the congestion of the tasks.

5.3.2 The management of the accelerator pool

Fig. 5.2 provides an example of how the two types of accelerators are deployed. There are six dynamic regions pre-designed in this system, each of which can be reconfigured to deploy accelerator A or B. There are two regions used to deploy accelerator A and two regions used to deploy accelerator B, with the remaining regions awaiting further deployment. The proposed strategy will allocate the unused region dynamically to deploy new accelerators or remove the regions from the accelerator pools.

5.3.3 The strategies of reconfiguration

To achieve the maximum power efficiency of the system, our strategy dynamically manages the accelerators in the pools according to the exact system requirements. In a multi-task system, the requirements for different tasks could vary. For example, improved overall performance may be required for some applications to deal with a large number of tasks, while a lower delay time may be required for each

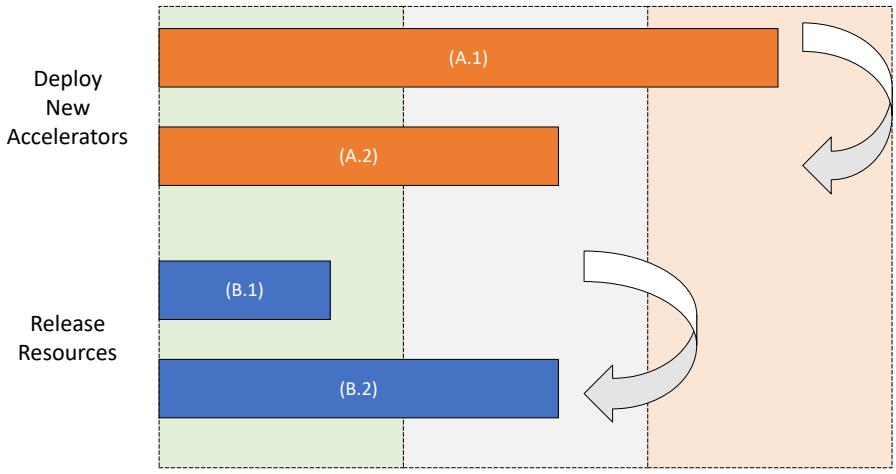


Figure 5.7: Weight changes when applying the strategies. The new accelerator A is deployed due to the heavy weight. The accelerator B is released for low power consumption.

task in some other applications. In this work, a "weight" variable is used to evaluate the requirements. If the "weight" falls short of the requirements, the lowest power consumption configuration will be selected by the system.

Herein, the requirements of applications are represented by a number of factors (e.g., execution time and data throughput). As shown in Fig. 5.6, assume that there are 'N' factors, the weights of the tasks can be expressed as follows:

$$W^T = \sum_{i=1}^{N} f_i(t_i), \tag{5.1}$$

where W^T indicates the weight of tasks, t_i represents the factor i and $f_i(t_i)$ refers to the effects of factor i.

Similarly, assume that there are M tasks in the queue. Then the queue weight can be expressed as follows:

$$W^Q = \sum_{i=1}^{M} W_i^T + b, \tag{5.2}$$

where W^Q represents the weight of the queue, W^T represents the weight of the tasks in the queue and b indicates the fixed costs (e.g., the time required to create the queue) of the queue.

As shown in Fig. 5.7, if the queue is "heavy", there are probably a large num-

ber of tasks that are waiting for acceleration. In this circumstance, the system will allocate more resources to improve the processing speed. If the queue is "light", there are more computing resources available for this queue than required. In this case, the system should switch to a lower power configuration mode. Besides, the trigger weight for adjustments will be adjusted according to the actual requirements of applications.

5.4 Experiments of the dynamic deployment

5.4.1 The setup of the experiments

Herein, the experiments are designed to evaluate the proposed scheme from two perspectives: performance and power efficiency. A prototype system was implemented on a Xilinx ZCU102 development board with two hardware dynamic regions. In the software part, face detection applications are running to send acceleration tasks to the hardware. In the hardware part, DPUs are deployed in the dynamic regions to build the accelerator pools for hardware acceleration in face detection applications. In the experiments, the dedicated benchmark program launched by Xilinx for Densebox is taken as the test application [208] to validate the DPUs in the proposed scheme. The model used in the experiments is Densebox, with a model resolution of 320×320.

Fig. 5.8 shows the configuration of the experiments. In order to simulate the working cases in working systems, there are a number of performance targets set for the system. According to the targets, the proposed system will dynamically adjust the number and frequency of DPUs in the accelerator pool. As shown in Table 5.1, the frequency of DPU ranges from 100 MHz to 300 MHz, while the maximum number of DPUs ranges from 1 to 2.

The ZCU102 boards include a Maxim PMBus based power system. There are a number of voltage regulators used to manage the onboard currents and voltages, which can be read back in real time via a PMBus interface. In the experiments,

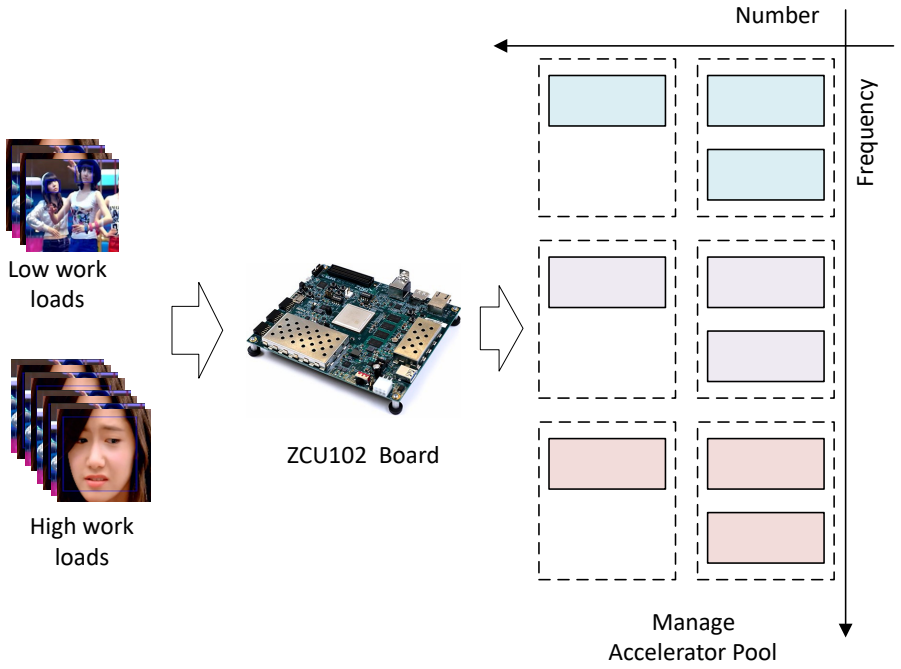


Figure 5.8: The setup of the experiments. The numbers and frequencies of accelerators in pools can be adjusted.

the power consumption by the system is determined by the currents and voltages from the power pins of the chip.

Table 5.1: The configuration of accelerator pools.

Configuration	Frequency of DPU 0	Frequency of DPU 1
DPU_L	100	N/A
DPU_M	200	N/A
DPU_H	300	N/A
DPU_LL	100	100
DPU_MM	200	200
DPU_HH	300	300

5.4.2 The power consumption and performance of different configurations

In order to verify the proposed strategy, the power consumption and performance were evaluated in different configurations. Fig. 5.9 shows the experimental results. “Static power” and “Dynamic power” indicate the power consumption by the system when the face detection programme is executed and not executed, respectively. “Dynamic power” indicates the maximum power consumption of the

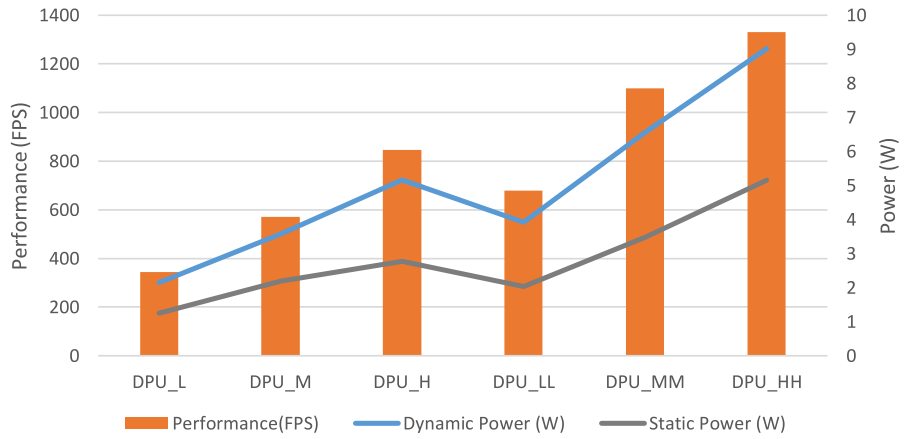


Figure 5.9: The power consumption and the performance of the system in different configurations.

configuration, while "static power" indicates the minimum power consumption of the configuration.

As shown in Fig. 5.9, the best performance and maximum dynamic power consumption increase at the same time as the frequency of the DPUs, which suggests the consistent power efficiency of the accelerator with the highest utilisation rate. In addition, the static power consumption shows an increasing trend with the frequency of DPUs. Even without hardware acceleration in the system, more power is still consumed by the system with higher frequency DPUs. When there is a DPU operating at 100 MHz, the static power consumption of the system is 1.25 W. When there are two DPUs running at 300 MHz, the static power consumption is 5.155 W for the system. In case of extremely low workloads for the current tasks, the power consumption of the system will be comparable to the level of static power consumption. In this case, there is a 75.8% power waste by the configuration of two DPUs with 200 MHz.

5.4.3 The results of the proposed hardware adaptive system

A number of cases with different task distributions were designed to simulate the applications in practice. Table 5.2 shows the comparison of power consumption between the normal design and the proposed scheme. Three cases of low loads,

Table 5.2: The performance of the proposed system in different cases.

Workloads	Low	Medium	High
Task distribution			
100 FPS	70%	25%	10%
800 FPS	20%	50%	20%
1,200 FPS	10%	25%	70%
Average power consumption			
Normal design (W)	6.11	7.12	8.03
Proposed scheme (W)	2.32	3.52	6.59

medium loads and high loads represent three different scenarios. For example, in the case of medium load, 50% of tasks require 800 FPS performance, 25% of the tasks require 100 FPS performance and 25% of the tasks require 1,200 FPS. The power consumption is 7.12 W for the normal design, while it is 3.52 W under the proposed scheme. The proposed scheme can reduce power consumption by 50% in this case. Similarly, in the cases of low loads and high loads, the proposed system consumes 32% and 82% less power respectively compared to the normal configurations. Note that the hardware switching is based on the partial reconfiguration, which consume some time, to write bitstream into configuration memory. In addition, the number of hardware switching will also affect the overall performance. If the system keeps switching between two states, the performance will be significantly affected. To simplify the results, the system in the experiment starts from low workloads state and moves to the high workloads states. It will not switch back to the same states again. Therefore, compared to the task time, it can be ignored.

5.5 Conclusion

In this scheme, the acceleration tasks are conducted by accelerator pools directly instead of specific accelerators. Compared with other works using PR features, the partial regions in this work are grouped for tasks to share, which make it easy to adjust the performance of specific tasks. The reconfiguration could be performed when applications are running. Given the different requirements in

different scenarios, a coping strategy is proposed to better manage the hardware resources. In this scheme, the acceleration tasks are placed in queues according to the types of acceleration. By calculating the weights of queues, the constraint on the acceleration can be identified to trigger hardware management. In addition, the trigger conditions can be modified by adjusting the coefficient of factors. The experimental implementation achieved a reduction by over 75% for the evaluated experimental system requirement. Although the prototype design is based on Xilinx FPGAs, this architecture can be applied to any FPGAs with PR features. In the future, specific algorithms will be explored to optimise the efficiency of the tasks and hardware deployment automatically.

This chapter focuses on hardware reconfiguration. However, the concept of "reconfiguration" can be further extended to software and DL models. To increase the power and computing efficiency of both DL model/software and hardware, an improved flexible DL software framework was also proposed. It is presented in the next chapter.

Chapter 6

AI system with adaptive DL inference

6.1 Introduction

Although the adaptive hardware system presented in Chapter 5 can improve the system efficiency, there are still some limitations. Firstly, hardware resources cannot be allocated at a fine-grained level. Because different reconfigured partitions cannot share resources, sufficient resources such as LUTs, DSPs, clocks and RAM must be pre-allocated to each partition in order to deploy hardware accelerators. This results in the number of partitions being limited by the size of the accelerator. In addition, when using partial reconfiguration techniques, each partition needs to lock up a margin area. An excessive number of partitions may lock up too many resources and lead to a reduction in hardware efficiency. Secondly, hardware reconfiguration is not a cost-free operation. On the one hand, it takes some time to reconfigure the hardware. During reconfiguration, the hardware resources in PR are not available for acceleration. On the other hand, the energy cost for the reconfiguration will increase with the number of reconfiguration operations. Therefore, the number of reconfiguration operations needs to be limited to reduce excessive energy consumption. If acceleration tasks or system workloads do not

change dramatically, then the hardware reconfiguration may not be efficient

As detailed in Chapter 1, despite the massive potential demonstrated by such new DNNs architectural concepts to improve on the current DNNs techniques, they are likely to have an influence on the type of hardware and software required to deliver such capabilities efficiently in the future. In this chapter, both acceleration technique and design optimisation technique are addressed.

When there are multiple AI tasks running, the variation in the workloads of the system can be complex. Adopting the concept of "dynamic reconfiguration", the dynamic DNNs may also be taken into account. Muliple DNNs can be dynamically implemented for different conditions. Because the dynamic adjustment of DNN models in the software layer are more fine-grained and less consuming, it can be used to further improve the flexibility, performance and efficiency of the overall system.

Therefore, in this chapter, to further improve the performance of the adaptive hardware system, a flexible DL software framework is proposed [2, 209]. It can provide a significant level of adaptability support for various DL algorithms on an FPGA-based edge computing platform. The platform can dynamically configure hardware and software processing pipelines to achieve better cost, power and processing efficiency for the dedicated application requirements and operating environments. In this work, a practical adaptive system was implemented. It may help to develop following optimisation and scheduling algorithms.

To demonstrate the effectiveness of the proposed solution, the framework is implemented for a DNN based real-time video processing pipeline on a Xilinx ZCU104 platform, where a set of comprehensive experiment tests are carried out to evaluate the performance of the proposed scheme. The achieved results (e.g., cars detection scenarios) show that when compared to deploying the original model on an FPGA board, the proposed scheme can reduce the latency by 12.9%, 23.9% and 36.5% as well as the total energy consumption by 18.9%, 38.4% and 53.8% for three different DNN models respectively.

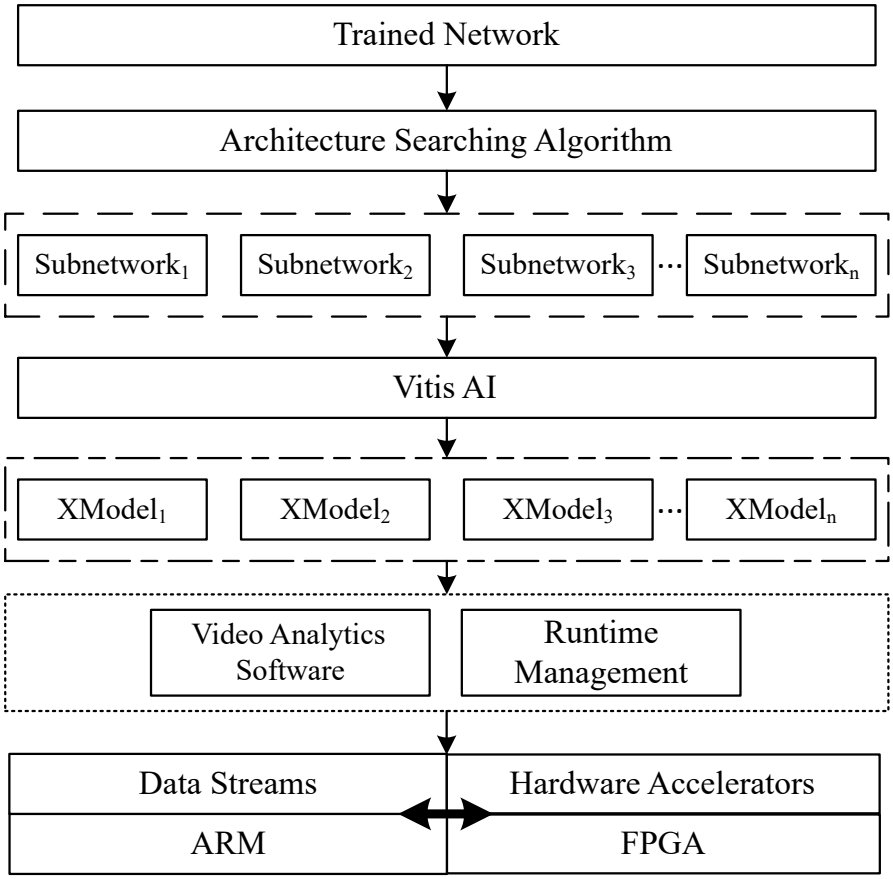


Figure 6.1: The system architecture of the proposed scheme.

6.2 Overview of the proposed system

The proposed scheme can support one to n implementations, which improves flexibility and run-time efficiency for run-time video analytics applications. It consists of three main software components: 1) NAS algorithms, which can generate different sub-networks under the given constraints, 2) the neural network model compilation that can convert sub-networks into FPGA focused executing formats and 3) the run-time management that can support the dynamic execution of sub-networks on embedded devices. A high-level overview diagram is illustrated in Fig. 6.1.

6.2.1 Neural network architecture search

NAS refers to a popular technology that can be applied to reduce the sizes of neural networks. In general, the NAS algorithm ignores the target hardware architecture

and run-time conditions directly due to the lack of accurate cost information for feedback to the NAS algorithm. For example, one-on-one NAS optimisation is capable of generating a network architecture Net_1 that meets the design requirements of accuracy $a_1 > A$, power consumption $p_1 < P$ and latency $l_1 < L$. However, during run-time inference, it might be challenging to deal with the input data. The accuracy falls short of the designed parameters, e.g., $a_1 < A$.

Consequently, there could be an increase in power consumption and latency accordingly due to a longer processing time required. In a recent work proposed by [60], an interesting NAS method was introduced, along with the OFA that can build a variety of different network architectures under the constraints of latency and accuracy. In this chapter, OFA is integrated into a joint optimisation toolchain which takes advantage of this approach to construct a one to n inference model for meeting various needs at the run-time. This optimisation approach will be detailed in Section 6.3.

6.2.2 Neural network model compilation

As for the network models developed in the mainstream frameworks, there is a necessity to map them into a high-efficient instruction set and data flow for the targeted hardware platform. Herein, Vitis-AI is applied to establish a complied network model, where 32-bit floating-point weights and activations are converted into 8-bit fixed-point format[210]. Finally, the AI model is mapped onto a high-efficient instruction set and data flow along with sophisticated optimisations as much as possible by Vitis-AI, such as layer fusion, instruction scheduling and reusing on-chip memory.

6.2.3 Software and hardware run-time management

The run-time management of this system is implemented using Vitis AI Runtime (VART), which makes the applications suitable for the unified high-level run-time API. VART enables the asynchronous submission and collection of jobs for the

accelerator and supports multi-threading and multi-process execution [208]. The software and hardware run-time reconfiguration would generate some additional overheads for the system. For example, when hardware fabrics are reconfigured, two types of costs should be considered: additional time and power consumption. The additional time consumption is majorly needed for rewriting the bitstream and updating software drivers, and the time for rewriting bitstream is varied by the sizes of the reconfiguration stream. In addition, the extra power consumption maybe caused by rewriting the RAMs, and the frequency of rewriting would dominate the values. This work focuses on evaluating model/software run-time reconfiguration.

6.3 DNN model optimisation

6.3.1 Brief introduction to once-for-all network

OFA [60] consists mainly of 5 blocks and in each block, depth, width and weight kernel size can vary as per the following example: depth $D = \{2, 3, 4\}$, width $W = \{3, 4, 6\}$, kernel size $K = \{3, 5, 7\}$, where D, W and K represents the number of convolution layers and channels, size of filters in a single block respectively. It is assumed that each variable is independent of each other, which means the number of subnetworks will be $((3 \times 3)^2 + (3 \times 3)^3 + (3 \times 3)^4)^5 \approx 2 \times 10^9$. In OFA, it is possible to utilise and train any model like ResNet [211] and Mobilenet [212] progressively while maintaining the variability in depth, width, or kernel size. In order to identify a subnetwork from this vast number of subnetworks, both latency and accuracy were treated as a constraint in the random search and evolutionary search algorithms.

6.3.2 Model generation and optimisation

Herein, the OFA trained network is taken as a super network and its searching algorithms are applied to generate multiple subnetworks as per our requirements.

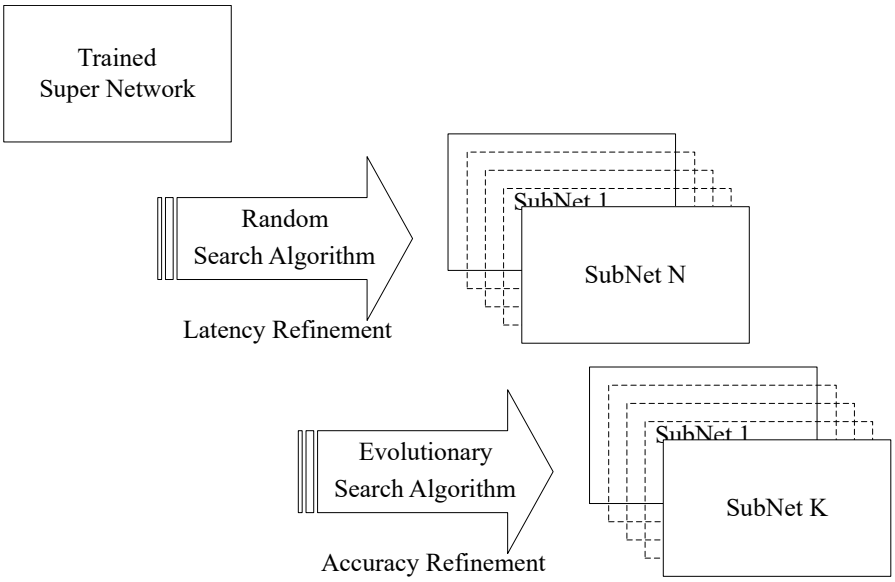


Figure 6.2: The model generation and optimisation technique.

The latency is first treated as an input parameter in the search algorithm. Fig. 6.2 illustrates the model generation technique, through which the model is optimised in terms of both latency and accuracy. Under the OFA framework, random search is first conducted to determine a set of subnetworks (Subnet N) that are close to the defined latency. Then, evolutionary search is conducted to identify the subnetworks (Subnet K) with the maximum accuracy among the previously selected subnetworks.

6.4 System hardware/software co-design

6.4.1 Hardware architecture

In general, a real-time DNN based video analytic system consists of four parts: 1) *video decoding*, 2) *pre-processing* (e.g., resize and normalisation), 3) *DNN inference* and 4) *post-processing*. because DNN, inference and other processes require a significant amount of computational resources, it is necessary to consider DNN inference and other processing tasks for the acceleration design. In addition, the hardware accelerators intended for video decoding and pre-processing should also be deployed to deploy the DNN inference. However, the requirements of post-

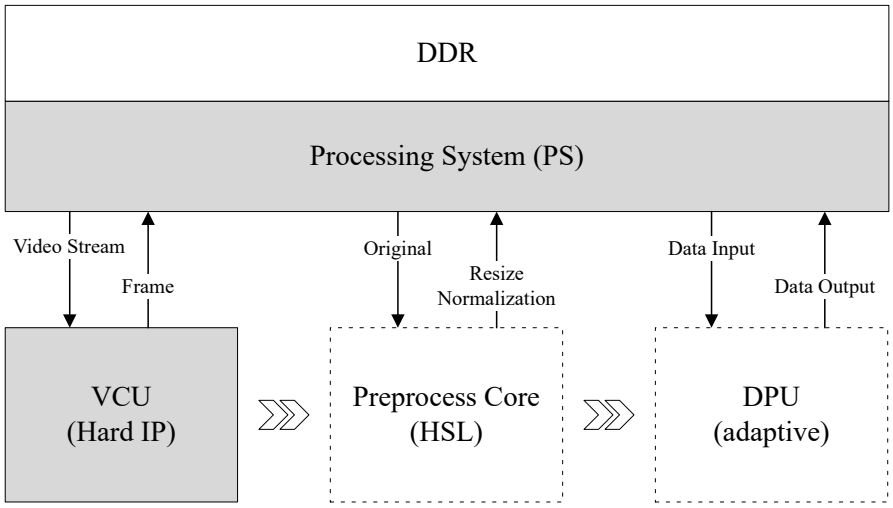


Figure 6.3: The hardware architecture of the proposed platform.

processing algorithms vary between different DNN models. Therefore, the post-processing tasks are performed in software.

Fig. 6.3 shows the overall system architecture which mainly includes three types of accelerators: 1) Xilinx H.264/H.265 video codec unit (VCU)[213], which is a hardware IP used for video coding and decoding tasks; 2) pre-processing module, which is a hardware module implemented by high-level synthesis (HLS) and dedicated to the resizing and normalisation tasks; and 3) *DPUs*, which is intended for DL inference tasks and can be reconfigured in different scenarios at run-time.

In order to process video streams in real time, the input video stream will be decompressed by the VCU at first, for the convertion of video stream into separate frames. Then, the pre-processing core will be used to carry out resize and pixel value normalisation on each frame. The VCU and pre-processing modules in this system can process up to $3,840 \times 2,160$ pixels at 60 frames per second, which has been set to 1080p video streams in our experimental scenarios. Therefore, the constraint on this system is supposed not to be the VCU or pre-processing modules. Instead, it is most likely that the system performance will be restricted by the DPUs and other processes. Allowing for this, our ultimate goal is to reconfigure them at run-time for the optimal performance of the entire system.

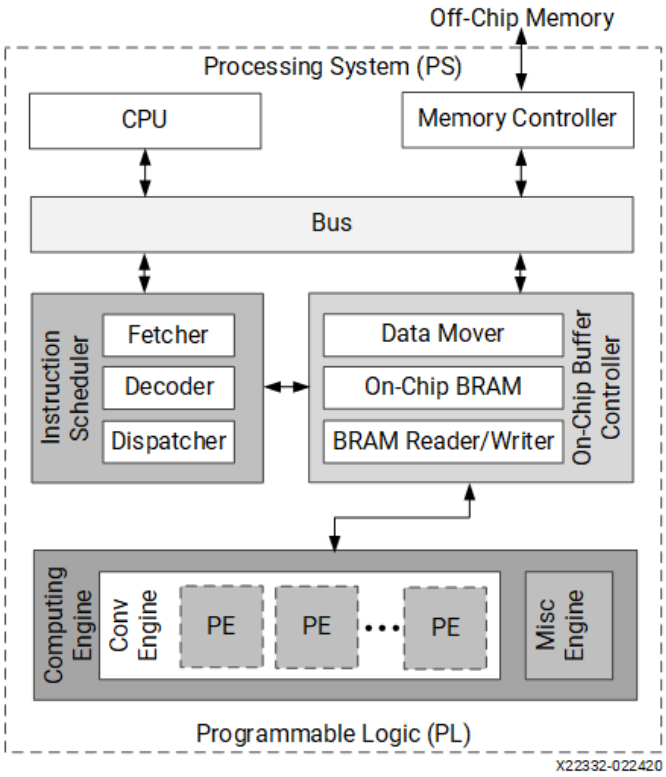


Figure 6.4: The hardware architecture of the proposed platform.

6.4.1.1 Deep learning processing unit

The DPU IP in our system is DPUCZDX8G, a dedicated unit designed for the Zynq UltraScale+ MPSoC. It is a configurable computation engine optimised for convolutional neural networks. The degree of parallelism utilised in the engine is a design parameter and can be selected according to the target device and application. It includes a set of highly optimised instructions and supports most convolutional neural networks, such as VGG, ResNet, GoogLeNet, YOLO, SSD, MobileNet, FPN and others.

The detailed hardware architecture of the DPUCZDX8G is shown in Fig. 6.4. In the beginning, the DPU unit gets instructions to control the operation. According to instructions, data in off-chip RAM will be buffered in on-chip memory for high throughput and efficiency. The calculation is conducted by the processing elements that take advantage of the hardware resources in FPGAs.

There are many parameters affecting the performance of the DPU. Chapter 5 has shown that the number and frequency of DPUs have great impacts on systems

performance. The DPU itself can also be configured with various convolution architectures, which are related to the parallelism of the convolution unit. The architectures for the DPUCZDX8G IP include B512, B800, B1024, B1152, B1600, B2304, B3136 and B4096, where the number indicates the peak operations per clock. Table. 6.1 shows the performance in different configurations.

Table 6.1: Performance of Different Models [214]

Device	Configuration	Frequency (MHz)	Peak Theoretical Performance (GOPS)
ZU2	B1152x1	370	426
ZU3	B2304x1	370	852
ZU5	B4096x1	350	1,400
ZU7EV	B4096x2	330	2,700
ZU9	B4096x3	333	4,100

6.4.2 Software implementation

As for the software part, video analytic applications were developed using Vitis Video Analytics SDK (VVAS) [208], which is a GStreamer-based plugin development framework. Because the Gstreamer runs video processing pipelines in multiple threads, the DNN inference processes can be precisely controlled by introducing several customised plugins for multiple video analytic applications.

Fig. 6.5 shows two types of pipelines representing different video analytic applications. "Pipeline (a)" represents a typical one-stage video analytic application (e.g., object detection and segmentation), where a single DNN model is applied to conduct an inference once per frame. "Pipeline (b)" represents a two-stage video analytic application (e.g., tracking, re-identification and car plate detection), where two DNN models are applied simultaneously. Besides, the second one may be executed multiple times, due to the detection results of the first one.

6.4.3 Dynamic DNN model switching method

A pair of communication interfaces (e.g., read and write communications) were designed in DNN inference plugins, to report DNN inference information and control

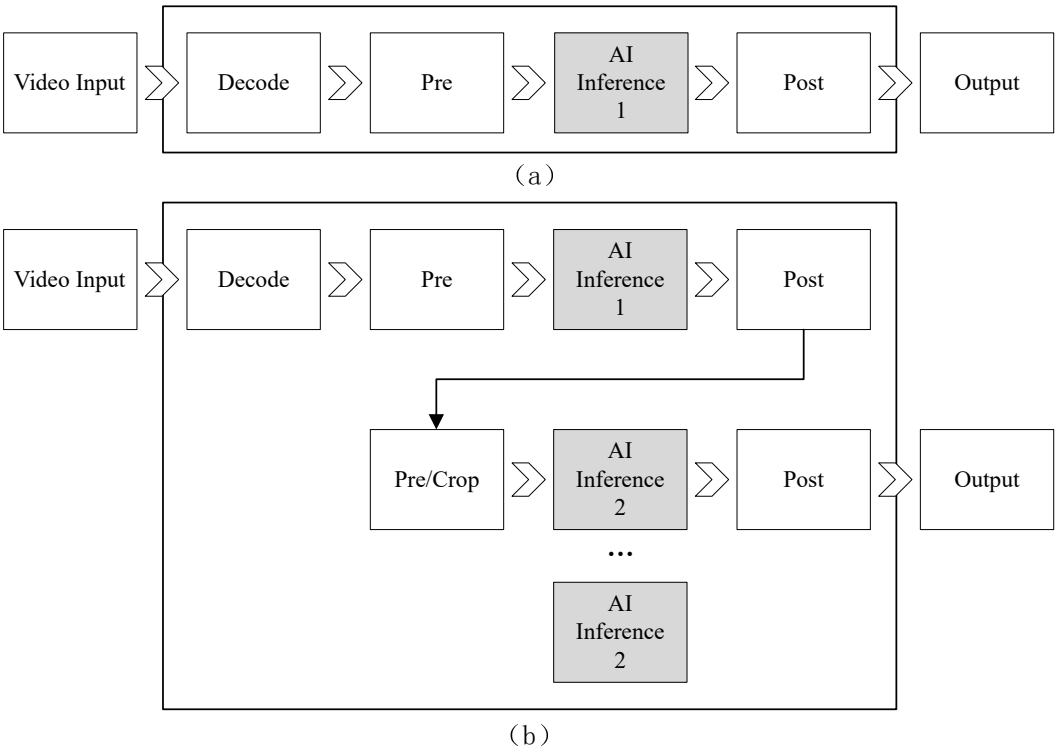


Figure 6.5: Video processing pipelines in the proposed system. (a) represents an application with a one stage AI inference task. (b) represents an application with a two-stage AI inference task.

its run-time DNN model process. When the pipeline is running, such information as the execution time of the DNN model, processing time of the pipeline and power consumption of the entire system will be sent to a separate system management thread simultaneously. According to the real-time performance metrics from the system, the processing pipeline can be reconfigured accordingly.

The proposed system will be reconfigured in two main aspects: 1) *Hardware configuration (DPUs)* and 2) *DNN model execution*. Reagrding DPU reconfigurations, DNN inference performance is determined by DPU design specifications (e.g., number of DPUs, maximum frequency and size). The power and computing efficiency of the proposed system can be further improved by using the method as introduced in Chapter 5, when the workloads placed on the system are increased.

Concerning DNN model switching, DNN can be adjusted at run-time in each video frame through the developed communication interfaces. This feature allows

both types of video analytic applications and the sizes of DNN inference models to be updated according to the exact performance specification and its execution environment. In the proposed system, there are three types of inter-process communication interfaces available for data transfer between the host programme and Gstreamer video pipeline:

- **Named pipe (an interprocess communication method)**: it represents the main method applied to send the control commands to the video pipeline. The customised plugin reads new commands from the named pipe, before the next frame is processed. Users can transfer data between*dpuinfer* and *drawing* plugins via the named pipe. This mechanism contributes to a stable and flexible method of communication between plugins.

- **File IO**: this interface supports direct file output to report the status of the video pipeline. For example, the proposed plugin can output the DPU inference results into a file for offline analysis.

- **Shared memory**: As Python does not support shared memory naively, a shared memory mechanism is developed to share information between different video pipeline branches in the Gstreamer framework.

The design of the communication framework is shown in Fig. 6.6. There are three software layers, including 1) Python management interfaces for user control, 2) Gstreamer applications to run AI inference and 3) system info (e.g., hardware temperature and power consumption). In the work, the system info is recorded in the Proc file system (a virtual file system in Linux). During runtime, Gstreamer applications continuously read the virtual via file IO interfaces. Each time when a virtual file is read, a function will be triggered to read sensor data. The history data points are stored in applications. If necessary, the data can also be passed to the Python management program via the name pipe interface. The named pipe interface is the main method to transfer data between applications and management programs. Applications can send real-time status to or get commands

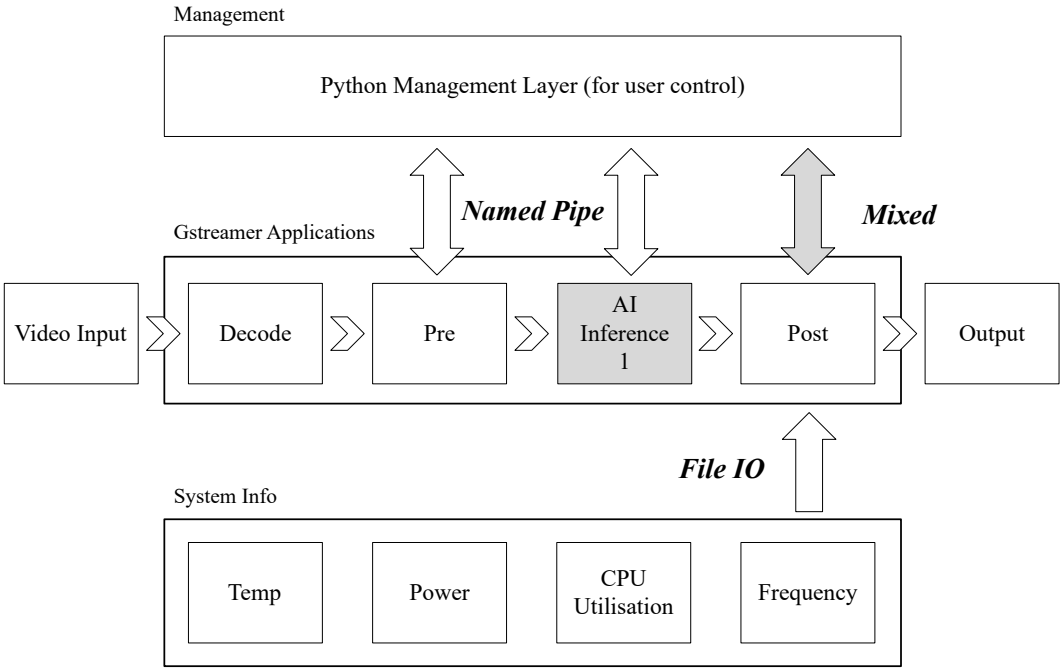


Figure 6.6: Design of the communication framework.

from the management program. The "post" plugin in Gstreamer applications is a key module to collect system info (e.g., power) and running status (e.g., FPS). It is also responsible for transferring inference results. If the output result contains massive data (e.g., semantic segmentation), it will use file IO to store output results.

6.5 Experiments

In this section, the experimental configuration and results are reported, with a typical real-time video processing pipeline implemented on a Xilinx ZCU104 (XCZU7EV) for the detection and classification of both cars and pedestrians via analytic applications.

6.5.1 Overall system setup

The proposed video analytics framework was implemented on a Xilinx ZCU104 (XCZU7EV- 2FFVC1156 MPSoC) development platform, as shown in Fig. 6.7, with the main modules as follows:

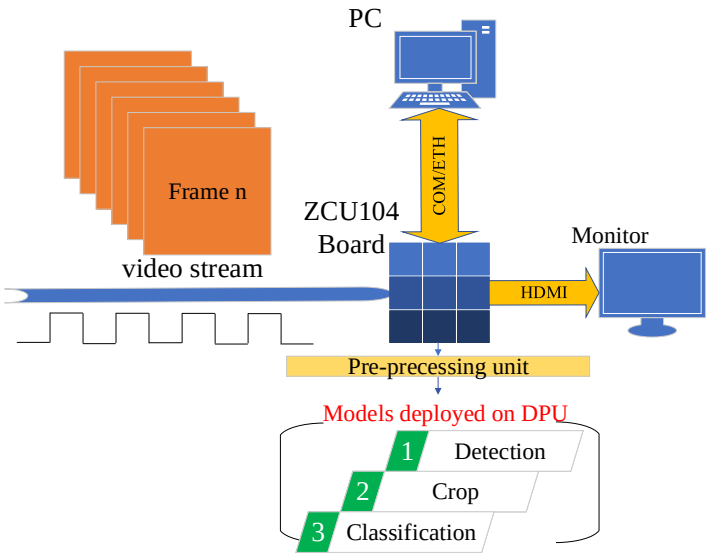


Figure 6.7: System setup diagram

1) *The input*: the video streams need to be processed; 2)*Pre-processing unit*: carries out resizing and normalisation functions to allow the processed video data streams to meet the input requirements of DNN models; 3) *Object detection DNN inference*: deployment of object-detect DNN model on DPUs (e.g., YOLOv3 for cars and Refinet for pedestrians detection respectively [215, 216]); 4) *Image cropping*: cropping detected objects in each video frame and send them to a second AI inference module for more precise classification tasks; 5) *Object classification*: deployment of Resnet-50 based backbone networks, and the network models are generated with different sizes based on the OFA algorithm.

6.5.2 Hardware configuration

The proposed design is implemented using Xilinx Vivado 2021.1 and PetaLinux 2021.1 on a Xilinx ZCU104 evaluation platform, video streams ($1920\times1080@30FPS$) are used as input for testing. Two DPUs (i.e., B3136) are deployed and integrate in the video processing pipeline (i.e., Fig. 6.3) The detailed hardware utilization are reported in Table 6.2.

Table 6.2: Sub-module resource utilisation

Sub module	LUT	Register	BRAM	DSP
DPU	47667	85,778	210	436
VPU	105	24	0	0
Pre-processing unit	13,147	17,390	12	40

6.5.3 Software configuration

6.5.3.1 Software pipeline

The proposed software pipeline is implemented under the VVAS 1.0 framework applied to control the video data flow. Besides, the Gstreamer plugins used in our experiments include *ivas-xabrscaler*, *ivas-xfilter* and *ivas-xmetaaffixer* [217].

ivas-xabrscaler: this plugin takes an input stream and outputs multiple output streams with different resolutions and colour space conversion per config- uration. It is mainly used in a pre-processing pipeline.

ivas-xfilter: this plugin is used to control data stream for DPU. It relies on a JSON configuration file to initialise the DPU module for the following tasks:

- Specify the acceleration software and other utility libraries.
- Interpret the acceleration AI library and select a suitable acceleration software library class (e.g., Vitis-AI Library) for DNN inference and post-processing.

ivas-xmetaaffixer: this plugin is used to merge multiple incoming streams into one, and the bounding box of the detection results is remapped on a data stream with higher resolution.

6.5.3.2 DNN model management

Xilinx Vitis-AI 1.4.1 tool-chain is used to convert Pytorch DNN models into xmodel files. By scaling DNN models in line with the OFA searching strategy, there are three different sizes of the OFA-ResNet-50 models obtained: OFA700, OFA1000 and OFA2000. The number after each OFA model represents million floating-point operations per second (MFLOPs), which is used as the threshold of the

subnet searching in the OFA algorithm. Three parameters are used to represent a wide range of model sizes in our experiments. Table 6.3 shows a list of the models used in our experiments, including a number of sub-networks created by using the OFA network. To update the DNN models at frame level dynamically, the communication interfaces as introduced in Section 6.4 is implemented.

Table 6.3: Parameters of the used DNN models

Model	Parameter size (MB)	Workload (MOPS)	Accuracy (Top1/Top5 ImageNet-1k)
OFA700	10.75	1340.61	74.9%/92.4%
OFA1000	18.02	1905.48	77.0%/92.8%
OFA2000	32.88	3805.47	79.7%/94.7%
ResNet-50	26.22	7360.32	83.2%/96.5%

In our experiment, this management scheme was verified by using a DNN powered car/pedestrian re-identification application, where a two-stage DNN pipeline is implemented. In the first stage of the DNN pipeline, Yolov3 and Refinet are used for car/pedestrian detection, respectively. In the second stage of the pipeline, ResNet-50 is used as a backbone network for car/pedestrian classification.

6.5.4 Results and analysis

As shown in Fig. 6.8, our framework is tested for two different video analytic applications: car and pedestrian re-identification. Through the tests conducted in both scenarios, the proposed framework shows the capability to handle the videos and identify objects correctly.

(a) (b)

Figure 6.8: Testing scenarios. (a) Car re-identification; (b) Pedestrian re-identification

In the proposed experiment, the same video processing pipeline is used to handle different video input streams, while monitoring the performance metrics of the system through the proposed customised communication interference plugin continuously, such as frame rate per second (FPS), power consumption and DPU latency. In the proposed work, the real-time power consumption of the entire ZCU104 board is measured from the onboard registers provided by PetaLinux, to determine the total energy consumption for the same video stream when the DNN models of different sizes are used, as shown in Fig. 6.9. The total energy consumption can be calculated using the following equation:

$$E = \sum_{i=1}^{n} P_i / f \tag{6.1}$$

where E denotes the total energy consumption in Joules (J), P denotes power consumption in Watt (W) at time i, f denotes sampling frequency in Hz. The total energy consumption is reduced by 18.9%, 38.4% and 53.8% in the car scenarios respectively, and similarly reduced by 25.4%, 41.1% and 61.6% respectively in the pedestrian scenarios.

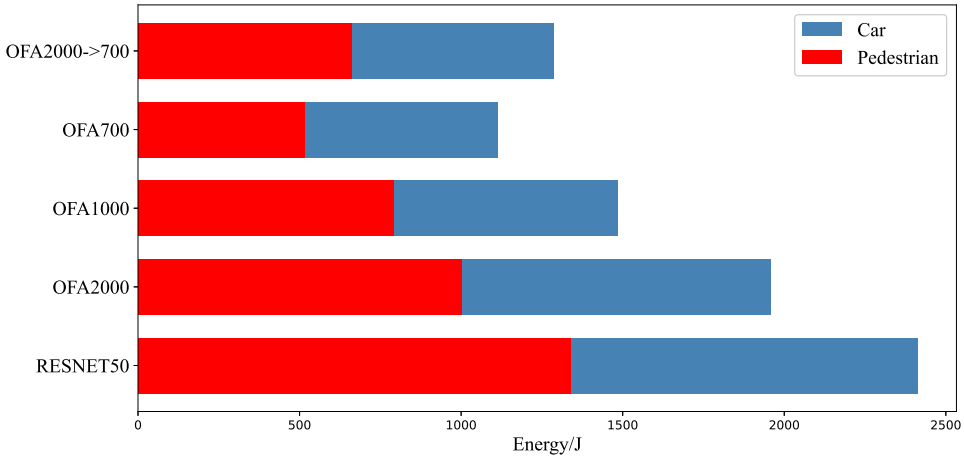


Figure 6.9: Total energy consumption for running different models.

The DPU inference latencies for each OFA network model are detailed in Fig.

6.10. Because the object detection model is not changed at run-time, the latencies are thus stable, which are 24∼25 ms and 19∼20 ms for car and pedestrian scenarios respectively. In general, the DNN models of a smaller size achieve a better DPU inference latency due to fewer operations being required. As a result, the entire video pipeline will take less time to complete than if a larger model is deployed. Therefore, a dynamic model-switch strategy can be implemented to identify a suitable DNN model based on the real-time performance metrics for improved performance of the entire system. By comparing the original model, the proposed system can reduce the latency of DPUs for the whole video pipeline by 12.9%, 23.9% and 36.5% in the car scenario and 14.0%, 25.9% and 38.6% in the pedestrian scenario.

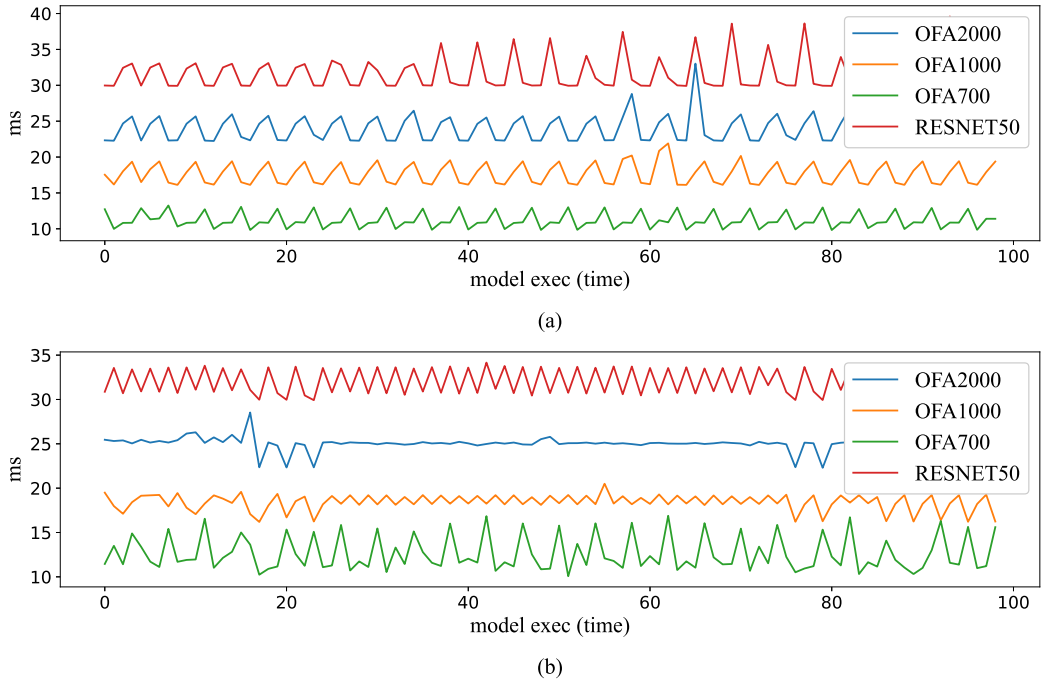


Figure 6.10: Latency in different scenarios. (a) represents Latency in car scenarios. (b) represents Latency in pedestrian scenarios.

Fig. 6.11. (a) and (c) present the FPS results of OFA700, OFA1000, OFA2000 and ResNet-50 under the car and pedestrian scenarios, respectively. By comparing the FPS of ResNet-50, it can be discovered that the overall FPS is improved by 26.3%, 65.6% and 113.0% in the car scenarios by using OFA700, OFA1000 and OFA2000 models respectively. Similarly, it is also improved by 27.1%, 65.7%

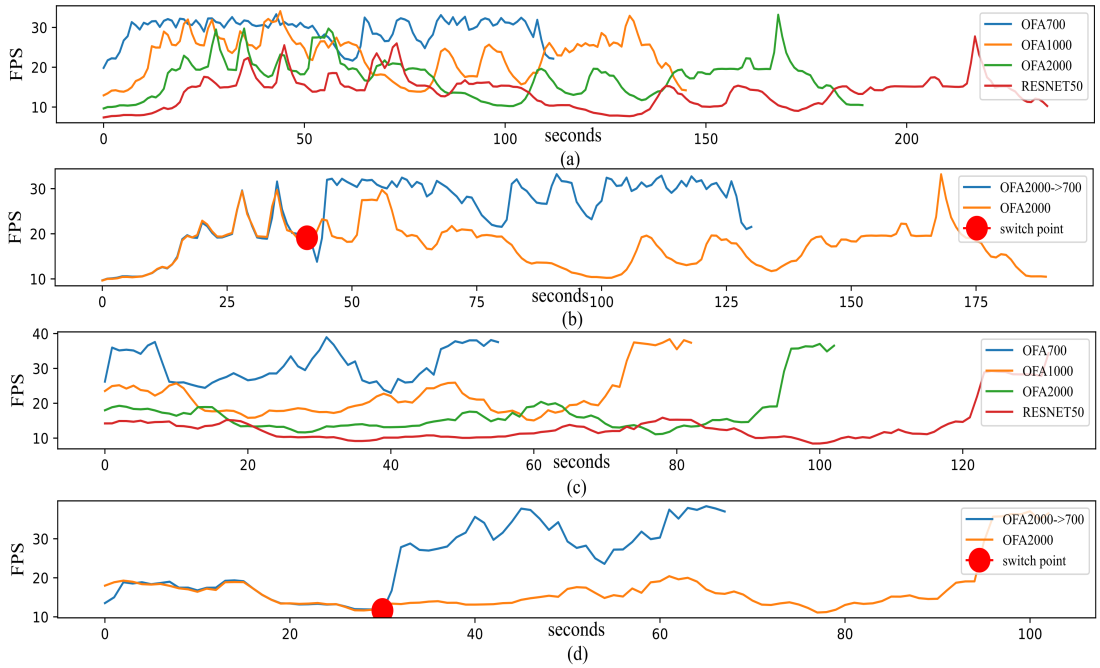


Figure 6.11: FPS of the proposed system in different scenarios. (a) Car scenarios without model switching. (b) Car scenarios with model switching. (c) Pedestrian scenarios without model switching. (d) Pedestrian scenarios with model switch

and 132.1% FPS in the pedestrian scenarios respectively. The video-stream is processed in a frame-by-frame manner, and each frame follows the architecture of the pipeline shown in Fig. 6.7. In the proposed test scenarios, the performance of the system will be affected by the number of objects that have been detected in the first stage of the pipeline (i.e., car/pedestrian de- tection). Each detected object will be cropped and then sent to the second stage of the pipeline (e.g., ResNet-50 or OFA-Resnet-50 networks) for the classification work. The workloads will vary significantly in the second stage of the pipeline, because they are affected by the number of objects detected in each frame at run-time. Therefore, the classification task is accelerated by applying a compressed DNN model, which means the processing time required for the entire frame will be reduced as a result. In the experiment, I also evaluate the overhead of the model switching. Compared with the frame without using the model switching approach, the average processing time for the frames using the model switching approach is 1.5 ms longer. This is

mainly because the model files need to be reloaded from the external memory. Considering that the frame time is around 30 ms in a 30 FPS video stream, it is acceptable.

In certain running environments (e.g., simple scenarios but with varying amounts of objects), it can dynamically switch DNN models by monitoring run-time performance metrics. Thus, it could further increase overall power and computing efficiency with an acceptable loss of classification accuracy. For example, as shown in Fig. 6.11 (b) and (d), by switching the DNN model to a smaller one at run-time, the average frame rates are increased from 17.04 FPS to 29.4 FPS in the car scenarios and from 16.9 FPS to 30.8 FPS in the pedestrian scenarios respectively. Meanwhile the energy consumption is also dropped by 34.2 % and 34.0 % respectively for the two scenarios (see Fig. 6.9). Accordingly, a proper model-switch strategy can be defined to dynamically locate a suitable DNN model based on the current resource, task and battery requirements at run-time.

6.6 Conclusion

In this chapter, it is proposed to design a new flexible hardware accelerator framework [1] for adaptive support offered to various DNN algorithms on an FPGA-based edge computing platform. According to the obtained results (i.e. for the cars scenarios), the proposed system can reduce the latency of DPUs by 12.9%, 23.9% and 36.5% depending on the sizes of the models, with the total energy consumption lowered by 18.9%, 38.4% and 53.8% for three different DNN-inference models with accuracy (Top5) at 94.7%, 92.8% and 92.4%, respectively.

By using a dynamical model-switch strategy, the frame rates are increased immediately at the switching point. The average frame rates are improved from 17.04 FPS to 29.4 FPS in the car scenarios and from 16.9 FPS to 30.8 FPS in the pedestrian scenarios, respectively. Further combined with a dynamic-reconfiguration

[1] This work won the prize of Xilinx University Program (XUP) in Adaptive Computing Challenge 2021: `https://www.hackster.io/contests/xilinxadaptivecomputing2021#challengeNav`

strategy in hardware modules the proposed system presents an unprecedented opportunity to build new adaptable architectures and algorithm models using the hybrid-computing units and resources, which is expected to play a significant role in improving energy efficiency, performance and flexibility.

In this system, I first combine hardware reconfiguration and software dynamic models. The main novel contributions in this work can be concluded as follow:

- An improved flexible DNN hardware accelerator framework that can be applied to configure the hardware and software processing pipelines dynamically is proposed to improve the power consumption and latency performance metrics.

- A comprehensive evaluation of DNN model sizes and inference performance is conducted, with Xilinx DPUs used in video analytic applications.

- A practical FPGA-based test platform for real-time software and hardware management is designed and implemented in this work. It can help to develop subsequent optimisation algorithms for hardware and software scheduling.

- This framework allows run-time reconfiguration to increase the power and computing efficiency of both the DNN model/software and hardware, to meet the requirements of dedicated application specifications and operating environments.

Chapter 7

Conclusions and future work

7.1 Summary of Research

In this thesis, the challenges for deploying an AI hardware system in extreme environments are addressed concerning SEE hardening, power efficiency and performance.

In Chapter 3, a SEE simulation scheme is proposed to evaluate the effects of SEE on large-scale circuits. Due to the complexity of SEE and the increasingly large size of existing circuits, it has become a challenge to conduct SEE simulations. On the one hand, accurate simulation tools (e.g., SPICE) require a large amount of computing power. In addition, the fast simulation tools (e.g., HDL) lack detailed SEE information in logic paths. In this work, the advantages of transistor simulation and HDL simulation are taken to achieve high accuracy and efficiency. Firstly, the SPICE simulation generates the SEE model for gate devices. Secondly, the SEE model is converted into the HDL model. Thirdly, HDL simulation for large-scale circuits is conducted. In this work, the SEE model is reusable, which significantly reduces the time cost to evaluate multiple HDL designs and improves the efficiency of circuit design. The gate library from SMIC and the ISCAS89 benchmark circuits are used to implement the scheme in the experiments. The results of the experiments show that the scheme can evaluate circuits with more than 100,000 transistors, which proves its' capabilities for evaluating large-scale

circuits.

In Chapter 4, an SEU mitigation scheme is proposed for RAMs in extreme Radiation environments because the RAM is one of the most SEE-sensitive elements in embedded systems. There are two features in this work. The first one is the capability to mitigate accumulated errors in extreme radiation environments. The second one is the high adaptability to RAM modules requiring no additional ports or timing modification. On the one hand, the use of the single port extends the range of applications in hardware systems. On the other hand, a well-organised state machine avoids the impact of detection and refreshing on the original timing sequence. Furthermore, parallel architecture can be configured according to the density of radiation so that it can balance the performance and hardware costs.

The experiments are conducted in the ISIS neutron source facility. Unhardened RAMs, normal ECC RAMs and the RAMs hardened by the proposed scheme are evaluated under the same conditions. It is shown that the error rates remain robust irrespective of the RAM size. The comparison of the radiation experiments also shows that the proposed scheme is an effective strategy for hardening embedded systems and the error rate of the self-scrubbing RAM is one-fifth of the conventional ECC RAM in 6-hour neutron radiation tests.

The third work was divided into two hardware and software parts. In chapter 5, a dynamic management scheme is proposed for hardware acceleration based on DFX. In radiation environments, AI systems face challenges in not only AI deployment but also in radiation resistance. Because FPGAs are RAM-based devices, inspired by the concept of “refreshing”, the reconfiguration feature of FPGA is adopted to build the system. In this work, I used DFX, which is a kind of PR, to build the hardware system with DFX used to dynamically configure the regions in the FPGAs, there are multiple types of accelerators that can be deployed for various tasks, thus improving both power efficiency and flexibility for multiple tasks systems. In order to dynamically manage onboard resources, accelerator pools were used to hold different types of accelerators during hardware

reconfiguration. When there are multiple AI applications running in the system, the acceleration tasks will be sent to corresponding pools instead of accelerators. In this way, the tasks will not be interrupted, if accelerators are changed during run-time. The trigger conditions for hardware reconfiguration was also discussed in this chapter. The bottlenecks of the system were evaluated by the acceleration task queue. By calculating the weights of queues, the constraint on the acceleration can be identified to trigger hardware management. Factors of the queue weights can also be modified manually for high flexibility.

In chapter 6, an adaptive DL software framework is built to enable adaptive support for various DNN algorithms on a FPGA-based edge computing platform. This work further improves AI systems' adaptability by deploying a range of sub-networks for different scenarios. By combining with the hardware reconfiguration in Chapter 5, the system can be more precisely controlled. This work implements a demo system supporting multi-channel video analytics on ZCU104. Hardware accelerators, including VCU, DPU and pre-process module, comprise the hardware base. VVAS and Gstreamer framework are used to build applications. In each video process channel, there are a number of sub-networks generated from OFA for different requirements. Python management interfaces are also built for real-time control.

7.2 Novel Contributions

The main contributions of this work in this thesis are stated as follows:

The first work in Chapter 3 is a new, rapid and convenient SEE simulation scheme. It can provide a universal comparison method to evaluate the designs of circuits in the context of SEEs. Compared to other works, the proposed scheme adopts the advantages of SPICE simulation and HDL simulations. In this work, a range of new SEE behaviour models is introduced for SEE simulations. Those models are based on the SPICE simulations. The transient current and voltage pulses are converted into digital signal changes. Therefore it can offer lightweight

and fast simulations for large-scale circuits. In addition, can offer a high level of flexibility in the design. Since the simulation steps in this scheme are decoupled. The existing SEE models can be reused in different circuits without modification.

In the second work, the SEE refreshing controller is a highly flexible, area-efficient design for scrubbing RAMs. Compared to conventional external scrubbers [61, 62, 63], It requires neither processor to conduct scrubbing options nor dedicated modules (e.g., ICAP) to access RAMs. There it can be easily used in various embedded systems. Furthermore, the design can achieve high SEU correction rates, which can significantly mitigate error accumulation in harsh radiation environments.

The third work is an adaptive system. To improve the efficiency of embedding systems for DL tasks, hardware reconfiguration and software dynamic DL models are first combined together. In hardware, partial reconfiguration is used. Compared with other works using PR features, the partial regions in this work are grouped for tasks to share, which make it easy to adjust the performance of specific tasks. Normally, acceleration systems use PR regions to build specific accelerators for corresponding works. Instead of deploying the same accelerator for specific acceleration tasks, the proposed systems will dynamically change the size, number and frequency of the accelerators. In software, I propose an improved flexible DNN hardware accelerator framework that can dynamically configure the hardware and software processing pipelines to achieve improved power consumption and latency performance metrics. Based on the work in hardware and software, a complete DNNs based real-time video processing pipeline is built to evaluate the effectiveness of the proposed framework.

7.3 Future works

There are still some further works that can be done to improve works in this thesis.

n Chapter 3, the work of SEE simulation can be improved. Firstly, more components should be discussed. In this work, the library was simplified and only

14 models were finally generated. However, in practice, libraries contain more components and layouts, which may affect the accuracy of SEE simulation. For example, OR gates may use different layouts when they are supposed to run at different frequencies. Secondly, an automation script will help to generate digital SEE models. Due to the inconvenience of SPICE, the injection of SEEs was conducted by handcraft, which is slow and inefficient. Thirdly, automation can be developed to analyse the SEE mitigation performance and provide improving guides. It will help to design large-scale circuits.

This work in Chapter 4 can also be improved. Firstly, doubling frequency is used to extend bandwidth for refreshing operation. This method is suitable, when the original frequency of the target device is not too high, which was 100 MHz in this work. However, current embedded systems for the general purpose may use a higher frequency than radiation dedication devices. In this case, doubling the frequency may waste bandwidth and energy. Therefore, the controller can be improved by using adaptive frequency. If the system requires high SEE mitigation performance, then a double frequency can be used. If the system requires low power consumption, then a frequency slightly higher than the original frequency can be used for refreshing. Although there may be difficulties in designing a clock domain crossing, it will bring some additional benefits to the compatibility of the design.

Further work for the third part, the adaptive DL system, can be done as follow. First, there is always a consumption (e.g., time costs and power consumption), whether it is a software or hardware reconfiguration. During switching, it is worth developing an algorithm to evaluate the trade-off between the increase in performance and the consumption caused by the switch. Secondly, although software and hardware reconfiguration increases flexibility, it also incurs management complexity. If the software and hardware switching are not well coordinated, then the switch can potentially degrade the system's performance. An evaluation algorithm is thus essential. Thirdly, the FPGA resources are pre-allocated to sev-

eral blocks when using DFX or PR. If too many resources are per-allocated to a block, then the accelerator may not be able to utilise all of them. If too few resources are per-allocated to a block, then the accelerator may not be able to be implemented. Therefore, an allocation algorithm for hardware resources can be developed to design suitable PR regions. Fourthly, DPUs have various architectures. In different DPU's architectures, the parallelisms of the core calculation matrix differ concerning input, pixel and out channels. In Chapter 6, the performance and efficiency are assumed to be only related to the "Peak Ops" (peak operations per second). However, DPU's architectures may affect the efficiency of specific modules. Therefore, the algorithm in progress will consider the effects of DPU's architectures on different types of models. Finally, A scheduling algorithm may also be necessary. There are multiple software tasks and hardware accelerators in the proposed system. How software maps to the hardware may significantly affect system efficiency.

To sum up, a number of the works referenced in this thesis are related to adaptive intelligent systems for extreme environments. The works can be divided into two aspects: the first two works are related to radiation hardening and the last two works are related to AI inference on an embedded platform. To mitigate SEEs in RAM systems, I used the refreshing method to correct errors. The concept was then extended to FPGA devices. By using the reconfiguration feature, an adaptive hardware system was proposed to improve flexibility and efficiency. Subsequently, the "reconfiguration" concept was also used in software to further improve performance and efficiency. The works in this thesis are at the research prototype stage and still need to be improved. Currently, I am working on the scheduling algorithm for the adaptive hardware and software system.

Appendix A

Demo of adaptive AI system

This part includes some figures of an adaptive AI system for smart city. It is a system based on the works in Chapter 6 and 5. The project is called "All-in-one Self-adaptive Computing Platform for Smart City" and it won the prize of Xilinx University Program (XUP) in Adaptive Computing Challenge 2021 with AMD-Xilinx (`https://www.hackster.io/contests/xilinxadaptivecomputing2021`). The source code can be find in Github repository: `https://github.com/luyufan498/Adaptive-Computing-Challenge-2021`.

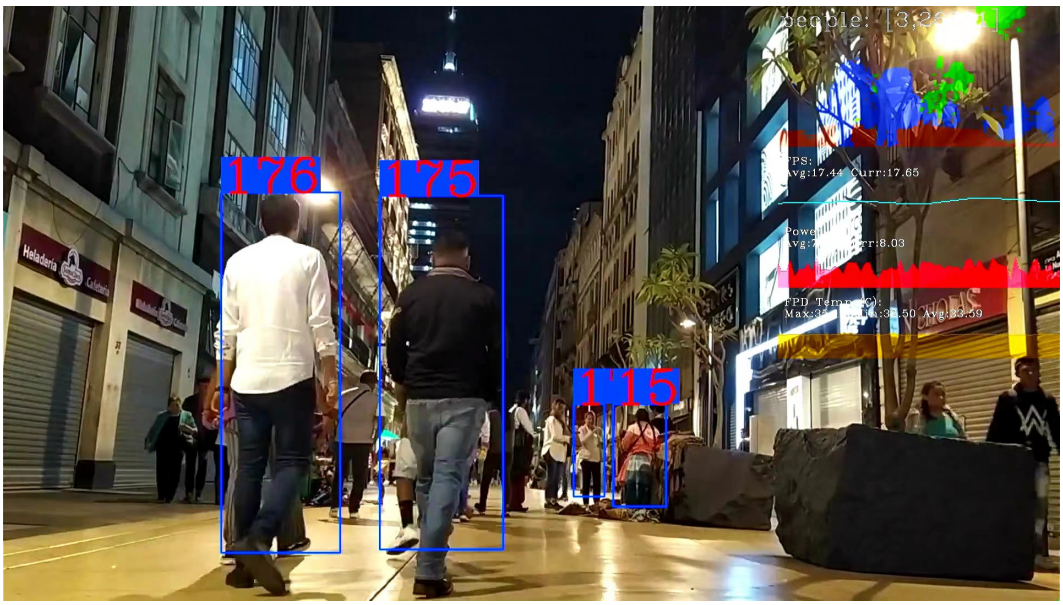


Figure A.1: ReID for pedestrians

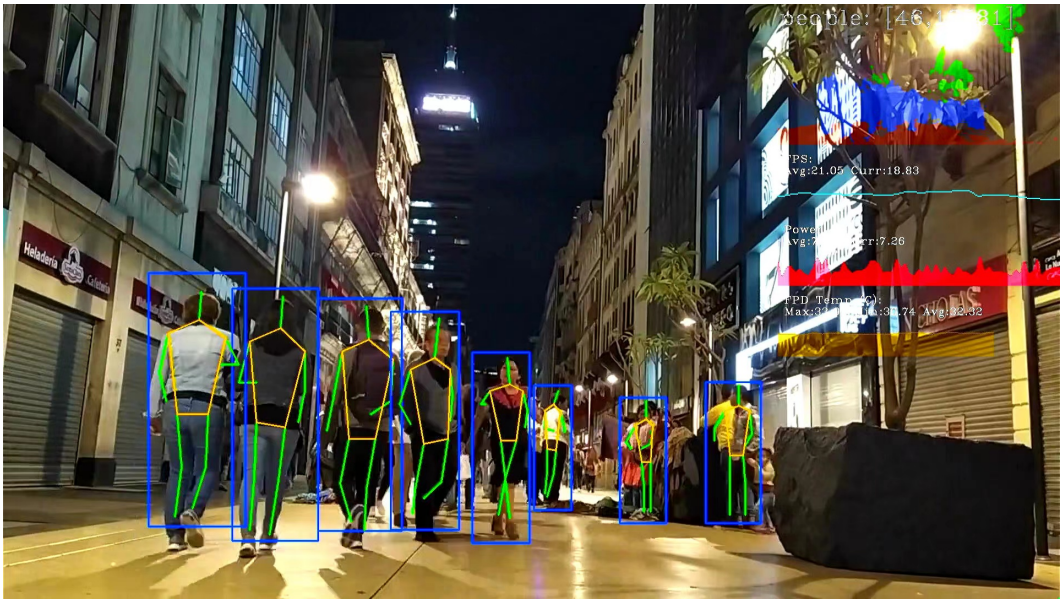


Figure A.2: Pose detection

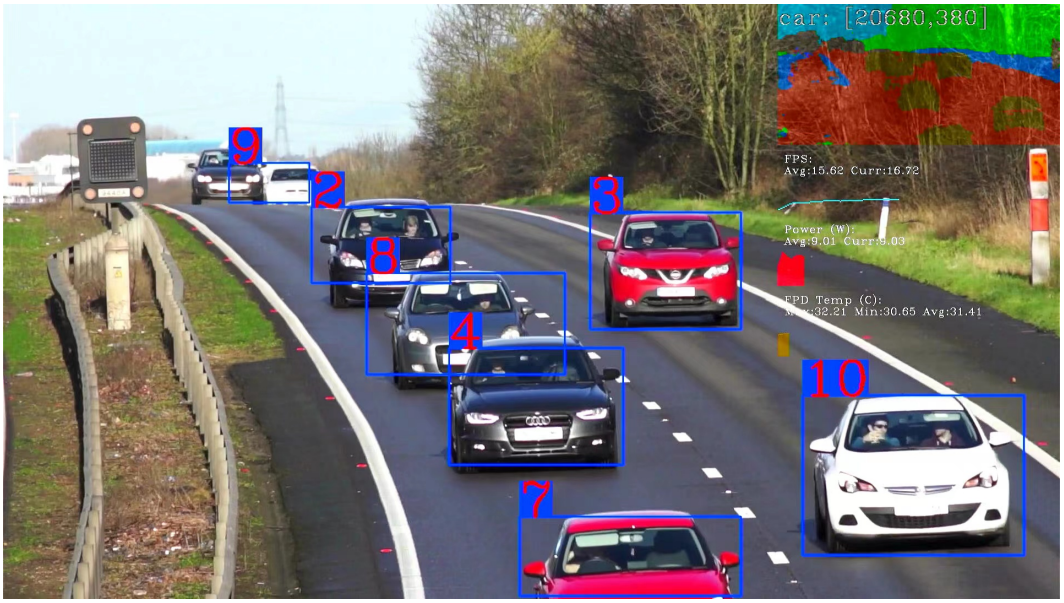
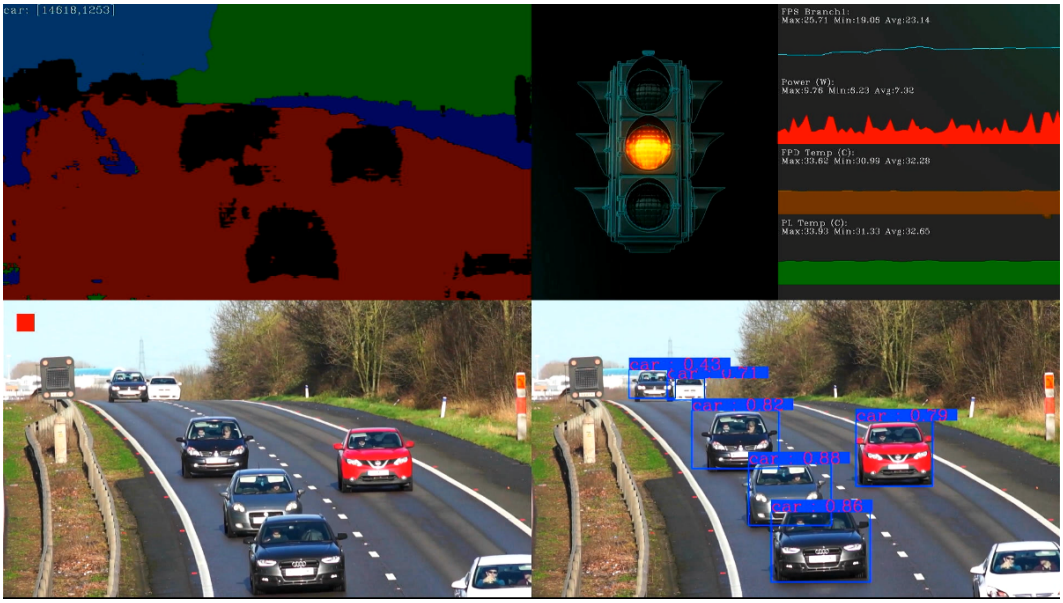


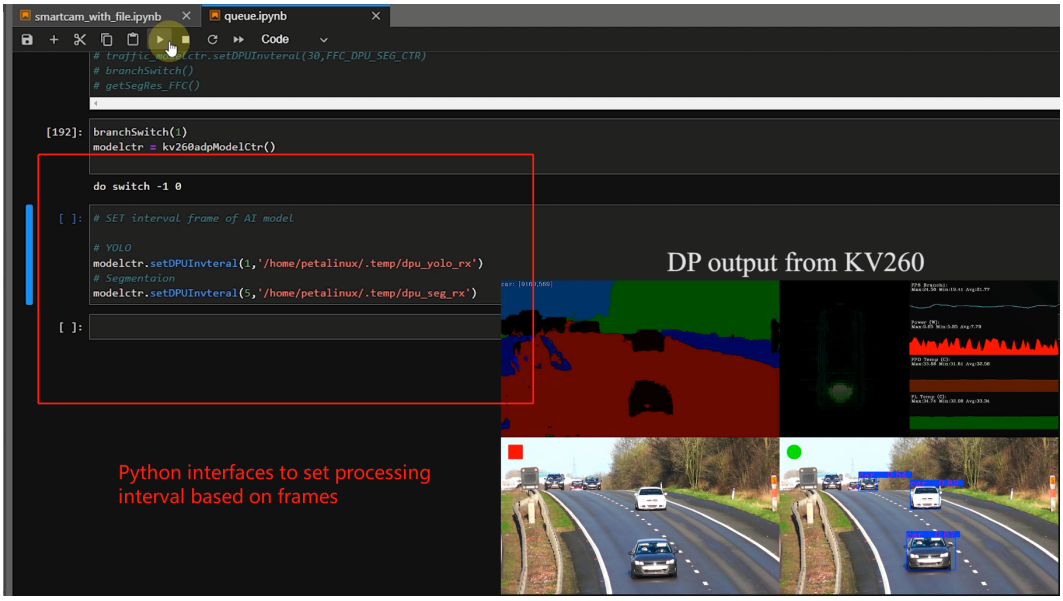
Figure A.3: ReID task for cars

(a) pedestrian detection and pose detection

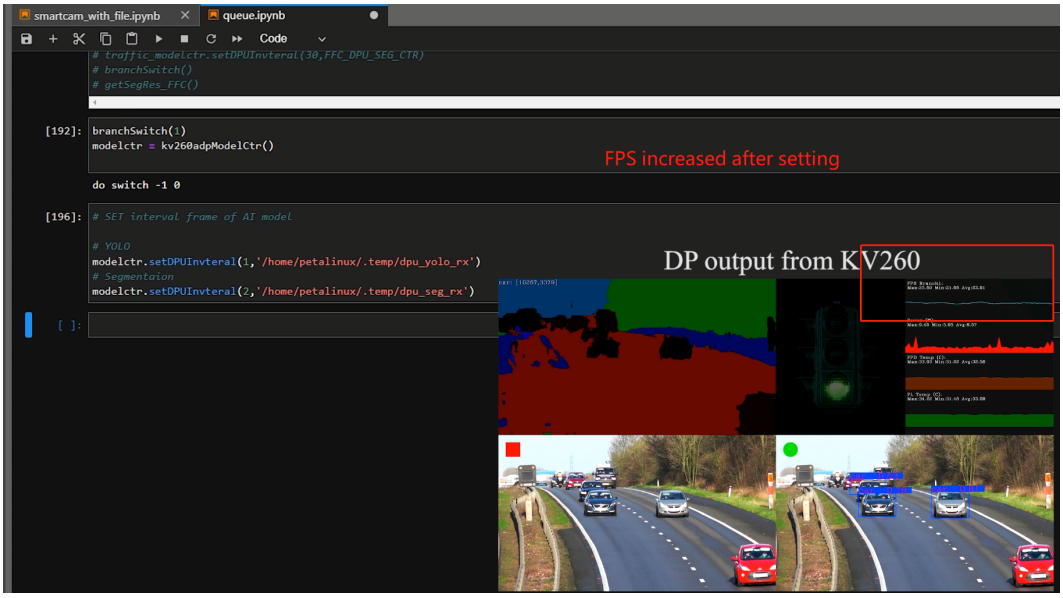


(b) car detection

Figure A.4: There are 3 AI inference branches integrated in this system for different tasks including scene recognition, pedestrian related AI inference and car related AI inference. In this system, a segmentation algorithm is used to conduct scene recognition tasks and the results are shown at the top left corner. The results of pedestrian related algorithms are shown on the bottom left corner. The results of car related algorithms are shown on the bottom right corner.

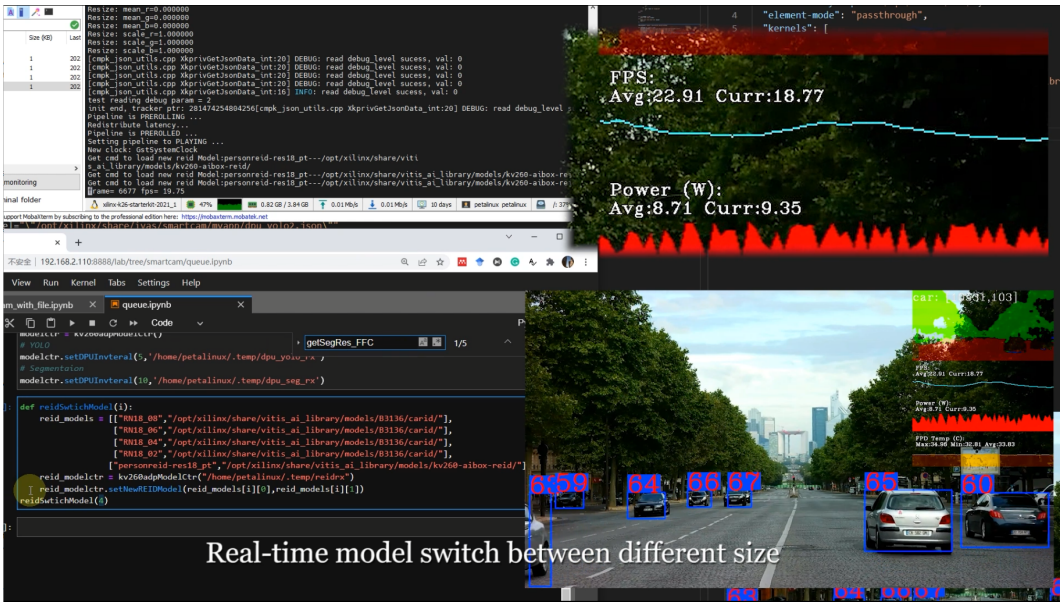


(a) inference interval for segmentation is set to be 5

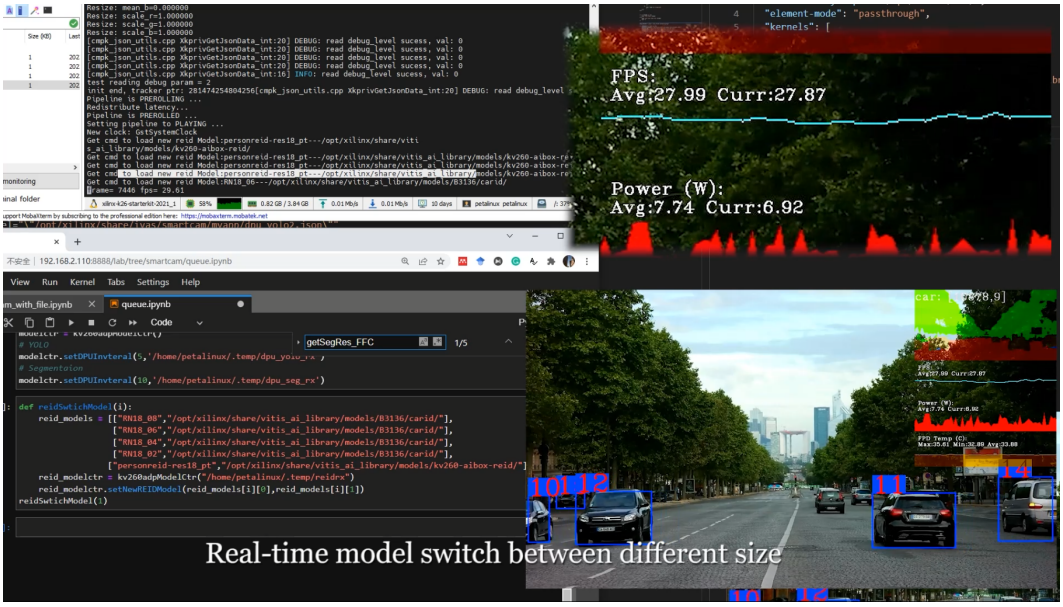


(b) inference interval for segmentation is set to be 2

Figure A.5: Using the designed python interfaces to adjust processing intervals to improve the system performance.

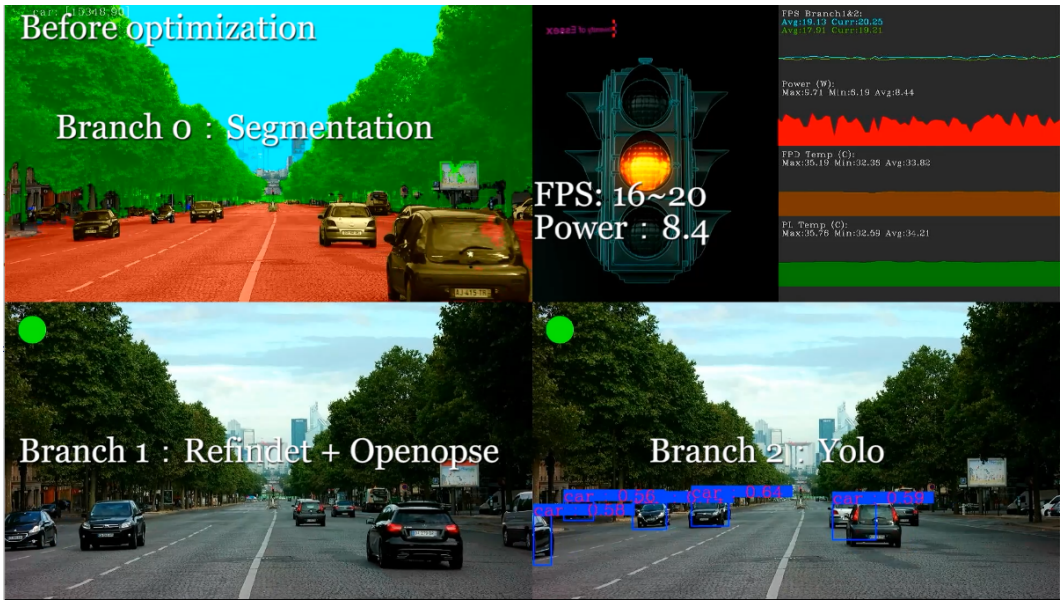


(a) AI inference with RN18 model in large size. The frame rate is 22.91 fps.

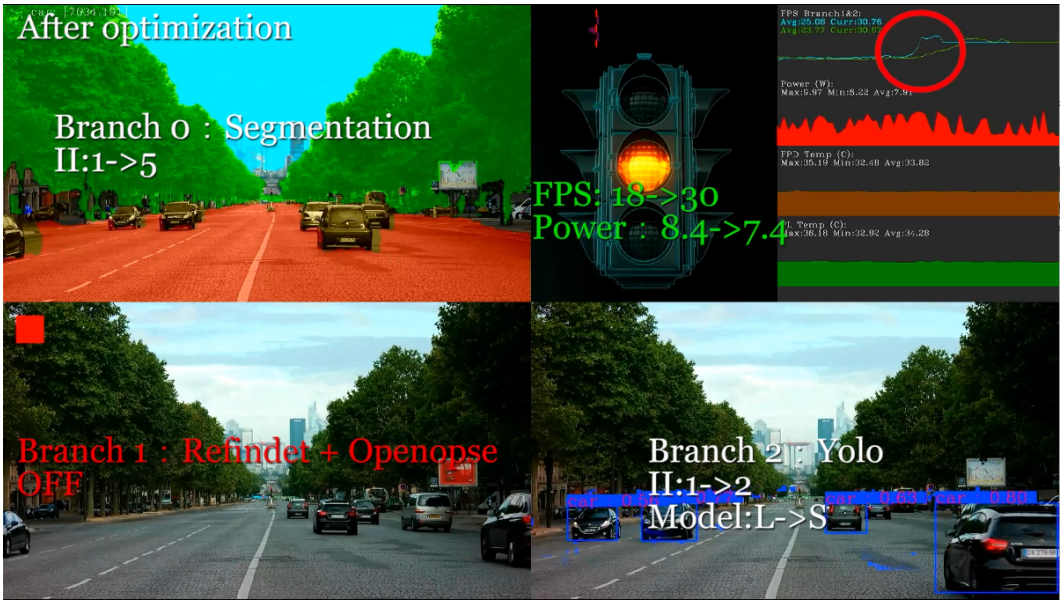


(b) AI inference with RN18 model in small size. The frame rate is 27.99 fps.

Figure A.6: Using the designed python interfaces, the running model can be changed dynamically.



(a) adaptive management is disabled



(b) adaptive management is enabled

Figure A.7: System performance with and without adaptive management. When adaptive management is enable, the AI inference branches are modified to improve the system performances. The branch 0 runs a segmentation algorithm for scene recognition, the frame based inference interval ('II') is set to be 5. The branch 1 runs Refindet and Openpose algorithms for pedestrian detection. It is disabled because there is no pedestrian. The branch 2 runs a YOLO algorithm for car detection. The inference interval is set to be 2 and the large model is replaced with a small ('L' to 'S') model. As a result, the frame rate is increased to be 30fps and the system power consumption drops from 8.4w to 7.4w.

Appendix B

Source Code

B.1 Python program: generate SEE models in netlists

```
import  re
#'input.txt'
def readFromFile(file_name):
    f = open(file_name,'r')
    txt = f.read()
    f.close()
    return  txt

def writeIntoFile(file_name,txt):
    f = open(file_name,'w')
    f.write(txt)
    f.close()

def getHead(verilog_txt):
    # define
    # get names
    match    =  re.search(r'module\s+(?P<Modname>\w+).+?;',
    verilog_txt,re.S)
    mode_name = match.group('Modname')
```

```
19      define_module = match.group()
20      #print(define_module,'\n',mode_name)
21      # get signals
22      match = re.search(r'input\s+(?P<Iname>[\w,\s]+?);',
        verilog_txt,re.S)
23      define_input = match.group()
24      input_names = re.sub(r'\s','',match.group('Iname')).split(',',
        re.S)
25      #print(define_input,' ',input_names)
26      # get outputs
27      match = re.search(r'output\s+(?P<Oname>[\w,\s]+?);',
        verilog_txt,re.S)
28      define_output = match.group()
29      output_names = re.sub(r'\s','',match.group('Oname')).split(','
        )
30      #print(define_output)
31      # get wires
32      match = re.search(r'wire\s+(?P<Wname>[\w,\s]+?);\n',
        verilog_txt,re.S)
33      define_wire = match.group()
34      wires = re.sub(r'\s','',match.group('Wname')).split(',')
35      print('name:{0},\n{2}inputs:{1},\n{4}outputs:{3}\n{6}wires:{5}
        '.format(mode_name,input_names,input_names.__len__(),
        output_names,output_names.__len__(),wires,wires.__len__()))
36      #return head
37      head_txt = define_module+'\n' +define_input + '\n' +
        define_output  +'\n'+define_wire
38      return head_txt ,mode_name,input_names,output_names
39
40  def genNewExample(verilog_txt):
41      device_id = 0;
42      unit_id = 0;
43      mod_txt = ''
44      mods_size = {'dff': 25, 'not': 2, 'and': 6, 'or': 6, 'nand':
        8, 'nor': 4}
```

```
45      mode_rpname = {'dff': 'DFF_S', 'not': 'NOT_S', 'and': 'AND_S',
        'or': 'OR_S', 'nand': 'NAND_S', 'nor': 'NOR_S'}
46      keep_words = ['module']
47      p = re.finditer(r'(?P<old_mod>\w+)\s+(\w+)\((?P<pin>[\w,]+)\)'
       , verilog_txt)
48      for u in p:
49          if (u.group('pin')):
50              if (u.group('old_mod') in mode_rpname.keys()):
51                  type = u.group('pin').count(',')
52                  size_addition = 0
53                  if (type > 2):
54                      new_mod = mode_rpname[u.group('old_mod')][:-2]
        + str(type) + mode_rpname[u.group('old_mod')][-2:]
55                      size_addition = type * 2 - 4
56                  else:
57                      new_mod = mode_rpname[u.group('old_mod')]
58
59                  mod_txt += '    {3}   #(.DEVICE_ID({0} ))     U
       {1}({2});\n'.format(device_id, unit_id, u.group('pin'),
60
                    new_mod)
61                  device_id += mods_size[u.group('old_mod')] +
       size_addition
62                  unit_id += 1
63              elif(u.group('old_mod') not in keep_words):
64                  print('WARNING:undefined submodule', u.group())
65      print('Circuit Size:{0},units:{1}\n'.format(device_id,unit_id)
       )
66      return mod_txt,device_id,unit_id
67
68  def genInjectionTask(device_id,unit_id):
69      task_txt = ''
70      tab = "    "
71      task_txt += tab +'reg [31:0] rand_id;\n'
72      task_txt += tab +'task SEU_INJECTION(seed);\n'
```

```
73      task_txt += tab * 2 + 'begin\n'
74      task_txt += tab * 3 + 'rand_id = {1}%{0};\n'.format(device_id,
    '{$random(seed)}')
75      for i in range(unit_id):
76          task_txt += tab*3 + 'U{0}.SEU_INJECTION(seed,rand_id);\n'.
    format(i)
77
78      task_txt += tab * 2 + 'end\n'
79      task_txt += tab * 1 + 'endtask\n'
80      return task_txt
81
82  def genNewModVerilogTxt(verilog_txt):
83      head_txt ,mode_name,input_names,output_names = getHead(
    verilog_txt)
84      print('_' * 100)
85      mod_txt, device_id, unit_id = genNewExample(verilog_txt)
86
87      task_txt = genInjectionTask(device_id, unit_id)
88      return head_txt + mod_txt + task_txt + 'endmodule' ,mode_name
    ,input_names ,output_names
89
90  def genNewTestVerilogTxt(mode_name ,input_names ,output_names):
91      testbench = 'module test_{0};\n'.format(mode_name)
92      tab = "    "
93      for inputpin in input_names:
94          testbench += tab + "reg {0};\n".format(inputpin)
95
96      for outputpin in output_names:
97          testbench += tab +"wire {0};\n".format(outputpin)
98          testbench += tab +"wire {0}_f;\n".format(outputpin)
99
100     testbench += tab + '{0} uut (\n'.format(mode_name)
101     for inputpin in input_names:
102         testbench += tab*2 + ".{0}({1}),\n".format(inputpin,
    inputpin)
```

```
103     index = 0
104     for outputpin in output_names:
105         if(index == 0):
106             testbench += tab * 2 + ".{0}({1})".format(outputpin,
    outputpin)
107         else:
108             testbench += ",\n" + tab * 2 + ".{0}({1})".format(
    outputpin, outputpin)
109         index += 1
110     testbench += "\n" + tab + ');\n'
111
112     testbench += tab + '{0} uut_f (\n'.format(mode_name)
113     for inputpin in input_names:
114         testbench += tab*2 + ".{0}({1}),\n".format(inputpin,
    inputpin)
115     index = 0
116     for outputpin in output_names:
117         if(index == 0):
118             testbench += tab * 2 + ".{0}({1})".format(outputpin,
    outputpin)
119         else:
120             testbench += ",\n" + tab * 2 + ".{0}({1})".format(
    outputpin, outputpin)
121         index += 1
122     testbench += "\n" + tab + ');\n'
123
124     testbench += tab + 'initial begin\n'
125     for inputpin in input_names:
126         testbench += tab*2 + "{0} = {1};\n".format(inputpin,
    inputpin == 'VDD'and 1 or 0)
127     testbench += tab + 'end\n'
128
129     for outputpin in output_names:
130         testbench += tab + "reg r{0},r{0}_f;\n".format(outputpin)
131
```

```
132     for inputpin in input_names:
133         if(inputpin == 'CK'):
134             testbench += tab + 'always #5 CK = ~CK;\n'
135         elif(inputpin in ['GND','VDD']):
136             pass
137         else:
138             testbench += tab + 'always #10 begin\n'
139             testbench += tab*2 + 'if(({1}%100) < 50) {0} = ~{0};\n
    '.format(inputpin,'{$random()}')
140             testbench += tab + ' end\n'
141 
142     testbench += tab + 'reg [31:0]  seu_cnt = 0;\n'
143     testbench += tab + 'real  next_time = 0;\n'
144     testbench += tab + 'always #(64 + next_time) begin\n'
145     testbench += tab*2 + 'uut.SEU_INJECTION(0);\n'
146     testbench += tab*2 + 'next_time = ({$random()}%10000)/100.0;\n
    '
147     testbench += tab*2 + 'seu_cnt <= seu_cnt + 1;\n'
148     testbench += tab + 'end\n'
149 
150     testbench += tab + 'always @ (posedge CK) begin\n'
151     for outputpin in output_names:
152         testbench += tab*2 + "r{0} <= {0};\n".format(outputpin)
153         testbench += tab*2 + "r{0}_f <= {0}_f;\n".format(outputpin
    )
154     testbench += tab + 'end\n'
155 
156     testbench += tab + 'wire flag_reg ='
157     for outputpin in output_names:
158         testbench += " (r{0}^r{0}_f) |".format(outputpin)
159 
160     testbench = testbench[:-1] + ";\n"
161 
162     testbench += tab + "reg [31:0]  err_cnt = 0;\n"
163     testbench += tab + "reg [31:0]  err_cnt2 = 0;\n"
```

```
164     testbench += tab + "always @ (posedge flag_reg) begin\n"
165     testbench += tab*2 + "err_cnt = err_cnt + 1;\n"
166 
167     testbench += tab*2 + "err_cnt2 = err_cnt2"
168     for outputpin in output_names:
169         testbench += '+ (r{0}^r{0}_f)'.format(outputpin)
170 
171     testbench += ";\n"
172 
173     testbench += tab + "end\n"
174     testbench += 'endmodule\n'
175 
176     #print(testbench)
177     return testbench
178 
179 print('='*100)
180 mod_txt,mode_name,input_names,output_names = genNewModVerilogTxt(
    readFromFile(r'.\input.v'))
181 
182 test_txt = genNewTestVerilogTxt(mode_name,input_names,output_names
    )
183 
184 writeIntoFile(mode_name + '.v',mod_txt)
185 print(mode_name + '.v' + 'generated,file size:{0}KB'.format(
    mod_txt.__len__()/1024))
186 writeIntoFile('test_'+mode_name+'.v',test_txt)
187 print('test_'+mode_name+'.v' + 'generated,file size:{0}KB'.format(
    test_txt.__len__()/1024))
188 print('='*100)
```

B.2 Verilog code: Design of FSM in self-refreshing RAM

```
1 module ECC_RAM_byHM#(
```

```
2   parameter   ADDR_WIDTH   =   'd19
3 )
4 (
5   input          clk,
6   input          clk_double,//double clk
7   input          rst_n,
8   input      [7:0] din,
9   input      [ADDR_WIDTH-1:0]   addr,
10  input          wea,
11  output       [7:0] dout
12    );
13
14
15
16
17  wire   [5:0] Code_HM;
18  wire   [13:0]   Ext_din_ready = {din,Code_HM};
19
20
21  (*mark_debug = "true"*)reg     [13:0]   ramport_dina = 'd0;
22  (*mark_debug = "true"*)reg     [ADDR_WIDTH -1:0] ramport_addra =
     'd0;
23  (*mark_debug = "true"*)reg         ramport_wea = 'd0;
24
25  (*mark_debug = "true"*)wire [13:0]   ramport_douta;
26  (*mark_debug = "true"*)wire [13:0]   ramport_CHK_dout;
27  (*mark_debug = "true"*)reg  [13:0]   ramport_CHK_dout_d;
28
29  (*mark_debug = "true"*)reg     [13:0]   ecc_data = 'd0;
30  (*mark_debug = "true"*)reg         ecc_ctrl = 'd0;
31  (*mark_debug = "true"*)reg     [ADDR_WIDTH-1:0]   ecc_addr = 'd0;
32  (*mark_debug = "true"*)wire      ecc_err_flg;
33
34  reg   [13:0]    Ext_din_ready_d;
35  reg         wea_d;
```

```
36  reg   [ADDR_WIDTH -1:0] addr_d;
37
38  always @ (posedge clk_double) begin
39    ramport_CHK_dout_d <= ramport_CHK_dout;
40
41    if(clk) begin
42      Ext_din_ready_d <= Ext_din_ready;
43      wea_d <= wea;
44      addr_d <= addr;
45    end
46  end
47
48  //RAMPORT IN CONTROL
49  always @(posedge clk_double or negedge rst_n)
50  begin
51    if(~rst_n)
52    begin
53       ramport_dina <= 14'd0;
54       ramport_wea  <= 1'd0;
55       ramport_addra  <= 12'd0;
56    end
57    else
58    begin
59       if(~clk)//posedege
60       begin
61        ramport_dina  <= Ext_din_ready_d;
62        ramport_wea   <= wea_d;
63        ramport_addra <= addr_d;
64       end
65       else
66       begin
67        ramport_dina  <= ecc_data;
68        ramport_wea   <= ecc_ctrl;
69        ramport_addra <= ecc_addr;
70       end
```

```
71    end
72  end
73
74  //PORT OUT CONTROL
75  reg   [13:0]  dout_crt ;
76  assign  dout = dout_crt[13:6];
77
78
79  always @(posedge clk_double or negedge rst_n)
80  begin
81    if(~rst_n)
82    begin
83      dout_crt  <=  14'd0;
84    end
85    else
86    begin
87      if(clk)
88        dout_crt <= ramport_CHK_dout_d;
89    end
90  end
91
92  //ECC ADDR CONTRIL
93  parameter IDE         = 4'd0;
94  parameter CHECK_CRC     = 4'd1;
95  parameter REWRITE_CRC     = 4'd2;
96  parameter ENDCHECK_NOERROR  = 4'd3;
97  parameter ENDCHECK_CORERROR = 4'd4;
98  parameter ENDCHECK_WAITCLK  = 4'd5;
99
100 (*mark_debug = "true"*)reg    [3:0]   fsm_state;
101 reg   [ADDR_WIDTH -1:0]   last_repair_addr;
102 always @(posedge clk_double or negedge rst_n)
103 begin
104   if(~rst_n)
105   begin
```

```
106       ecc_addr  <= 'd0;
107       ecc_ctrl  <= 'b0;
108       fsm_state <= IDE;
109       ecc_data  <= 'd0;
110     end
111     else
112     begin
113       case (fsm_state)
114         IDE:
115           begin
116             if(~clk && {wea,addr} != {1'b1,ecc_addr})
117               fsm_state <= CHECK_CRC;
118             else
119             if({wea,addr} == {1'b1,ecc_addr})
120             begin
121               ecc_addr <= ecc_addr - 'd1;
122             end
123             else
124               fsm_state <= IDE;
125             ecc_ctrl <= 1'b0;
126           end
127         CHECK_CRC:
128           begin
129             if({wea,addr} == {1'b1,ecc_addr} || ecc_err_flg == 1'
    b0)
130               fsm_state <= ENDCHECK_NOERROR;
131             else
132             if(ecc_err_flg) begin
133               ecc_data <= ramport_CHK_dout; // save the crt value

134               fsm_state <= REWRITE_CRC;
135             end
136           end
137         ENDCHECK_NOERROR:
138           begin
```

```
139             ecc_addr <= ecc_addr - 'd1;
140             fsm_state <= IDE;
141           end
142         REWRITE_CRC:
143           begin
144             last_repair_addr <= ecc_addr;
145             ecc_ctrl <= 1'b1;
146             fsm_state <= ENDCHECK_CORERROR;
147           end
148         ENDCHECK_CORERROR:
149           begin
150             ecc_ctrl <= 1'b0;
151             ecc_addr <= ecc_addr - 'd1;
152             fsm_state <= ENDCHECK_WAITCLK;
153           end
154         ENDCHECK_WAITCLK:
155           begin
156             fsm_state <= IDE;
157           end
158         default : fsm_state <= IDE;
159       endcase
160     end
161   end
162
163
164   Gen_CodeHM GEN_HMcode (
165     .din(din),
166     .code_hm(Code_HM)
167     );
168
169
170   CoreRam_4K CORE_RAM (
171     .clka(clk_double),
172     .wea(ramport_wea),
173     .addra(ramport_addra),
```

```
174     .dina(ramport_dina),
175     .douta(ramport_douta)
176     );
177
178   Check_CodeHM CHK_HMcode (
179     .din(ramport_douta),
180     .dout(ramport_CHK_dout),
181     .err(ecc_err_flg)
182     );
183
184 endmodule
```

B.3 Shell scripts: build Gstreamer pipelines

```
1 #! /bin/sh
2 # this is for 4k and 4 channel
3 video=/home/petalinux
4 segback="ori"
5 branch1="reid"
6 sync="false"
7
8 conf_pre_onlyresize="\"/opt/xilinx/share/ivas/cmpk/preprocess/
    resize_reid.json\""
9 conf_pp1_status="\"/opt/xilinx/share/ivas/cmpk/runstatus/pp1status
    .json\""
10 conf_pp2_status="\"/opt/xilinx/share/ivas/cmpk/runstatus/pp2status
    .json\""
11 conf_pp1_recordfps="\"/opt/xilinx/share/ivas/branch1/fpsbranch1.
    json\""
12 conf_pp2_recordfps="\"/opt/xilinx/share/ivas/branch2/fpsbranch2.
    json\""
13 conf_dpu_seg="\"/opt/xilinx/share/ivas/cmpk/segmentation/dpu_seg.
    json\""
14 conf_draw_seg="\"/opt/xilinx/share/ivas/cmpk/segmentation/
```

```
    drawSegmentation.json\""

while getopts f:br:sh opt
do
    case $opt in
        f)
            video=$OPTARG
            ;;
        b)
            segback="black"
            ;;
        r)
            branch1="$OPTARG"
            ;;
        s)
            sync="true"
            echo $sync
            ;;

        :)
            echo "-$OPTARG needs an argument"
            ;;
        h)
            echo ""
            echo "Help:"
            echo "-f video file source"
            echo "-b (optional) segmentation use black background"
            echo "-r (optional) model for branch 1  [(reid),
    openopse]"
            echo ""
            ;;
        *)
            echo "-$opt not recognized"
            ;;
    esac
```

```
48 done
49
50 if [ -f $video ]; then
51     echo "find video: $video"
52 else
53     echo "cant find video file: $video"
54     exit -1
55 fi
56
57
58 if [ $sync == 'false' ]; then
59     videosrc_cmd="multifilesrc location=\"${video}\" ! h264parse !
       queue ! omxh264dec ! video/x-raw, format=NV12"
60     tee1_name="maintee1"
61     tee2_name="maintee2"
62     teeseg_name="tseg"
63     tee1_cmd=$videosrc_cmd"!tee name=$tee1_name"
64     tee2_cmd=$videosrc_cmd"!tee name=$tee2_name"
65     teeseg_cmd=$videosrc_cmd"!tee name=$teeseg_name"
66     echo $tee1_cmd
67     echo $tee2_cmd
68     echo $teeseg_cmd
69 else
70     echo "sync video pipeline (fps will drops)"
71     tee1_name="maintee"
72     tee2_name=$tee1_name
73     teeseg_name=$tee1_name
74     tee1_cmd=$videosrc_cmd"!tee name=$tee1_name"
75     tee2_cmd="$tee2_name."
76     teeseg_cmd="$teeseg_name."
77     echo tee2_cmd
78 fi
79
80
81 if [ $segback == "black" ]; then
```

```
82      segbackcmd="multifilesrc location=\"/home/petalinux/videos/
    black.nv12.h264\" ! h264parse ! queue ! omxh264dec ! video/x-
    raw, format=NV12"
83      echo "use black background for segmentation."
84 else
85      segbackcmd="$teeseg_name."
86      echo "use original background for segmentation."
87 fi
88
89 if [ $branch1 == "reid" ]; then
90      branch1crop="\"/opt/xilinx/share/ivas/aibox-reid/crop.json\""
91      branch1model="\"/opt/xilinx/share/ivas/aibox-reid/reid.json\""
92      branch1draw="\"/opt/xilinx/share/ivas/aibox-reid/draw_reid.
    json\""
93      echo "branch 1: use reid"
94 elif [ $branch1 == "openpose" ]; then
95      branch1crop="\"/opt/xilinx/share/ivas/cmpk/openpose/crop.json
    \""
96      branch1model="\"/opt/xilinx/share/ivas/cmpk/openpose/openpose.
    json\""
97      branch1draw="\"/opt/xilinx/share/ivas/cmpk/openpose/draw_pose.
    json\""
98      echo "branch 1: use openopse"
99 else
100     echo "branch 1: unsported model: $branch1 [(reid), openpose]"
101     exit 2
102 fi
103
104
105
106
107
108
109 echo | modetest -M xlnx -D b0000000.v_mix -s 52@40:3840x2160@NV16
110 gst-launch-1.0 \
```

```
111  multifilesrc location=\"${video}\" ! h264parse ! queue !
     omxh264dec ! video/x-raw, format=NV12  \
112  ! tee name=$tee1_name  \
113      ! queue \
114      ! ivas_xmultisrc kconfig=$conf_pre_onlyresize  \
115      ! queue ! ivas_xfilter name=refinedet kernels-config="/opt
     /xilinx/share/ivas/aibox-reid/refinedet.json"  \
116      ! queue ! ivas_xfilter name=crop kernels-config=
     $branch1crop  \
117      ! queue ! ivas_xfilter kernels-config=$branch1model \
118      ! ima.sink_master ivas_xmetaaffixer name=ima ima.
     src_master ! fakesink  \
119  $tee1_name. \
120  ! queue \
121  ! ima.sink_slave_0 ima.src_slave_0  \
122  ! queue ! ivas_xfilter kernels-config=$branch1draw  \
123  ! queue ! ivas_xfilter kernels-config=$conf_pp1_status    \
124  ! queue ! ivas_xfilter kernels-config=$conf_pp1_recordfps \
125  ! queue ! kmssink bus-id=b0000000.v_mix plane-id=34 render-
     rectangle="<0,1080,1920,1080>" show-preroll-frame=false sync=
     false \
126  \
127  $tee2_cmd  \
128      ! queue ! ivas_xmultisrc kconfig=$conf_pre_onlyresize  \
129      ! queue ! ivas_xfilter kernels-config="/opt/xilinx/share/
     ivas/branch2/dpu_yolo2.json"  \
130      ! imacar.sink_master ivas_xmetaaffixer name=imacar imacar.
     src_master ! fakesink  \
131  $tee2_name. \
132      ! queue \
133      ! imacar.sink_slave_0 imacar.src_slave_0  \
134      ! queue ! ivas_xfilter kernels-config="/opt/xilinx/share/
     ivas/branch2/drawbox.json"  \
135      ! queue ! ivas_xfilter kernels-config=$conf_pp2_status
     \
```

```
136         ! queue ! ivas_xfilter kernels-config=$conf_pp2_recordfps
     \
137         ! queue ! kmssink bus-id=b0000000.v_mix plane-id=36 render
    -rectangle="<1920,1080,1920,1080>" show-preroll-frame=false
    sync=false \
138     \
139     $teeseg_cmd  \
140         ! queue ! ivas_xmultisrc kconfig=$conf_pre_onlyresize  \
141         ! queue ! ivas_xfilter kernels-config=$conf_dpu_seg  \
142         ! imaseg.sink_master ivas_xmetaaffixer name=imaseg imaseg.
    src_master ! fakesink  \
143     $segbackcmd \
144         ! queue \
145         ! imaseg.sink_slave_0 imaseg.src_slave_0  \
146         ! queue ! ivas_xfilter kernels-config=$conf_draw_seg  \
147         ! queue ! kmssink bus-id=b0000000.v_mix plane-id=35 render
    -rectangle="<0,0,1920,1080>" show-preroll-frame=false sync=
    false \
148     \
149     multifilesrc location="/home/petalinux/videos/back_logo.nv12.
    h264" \
150     ! h264parse ! queue ! omxh264dec ! video/x-raw, format=NV12 !
    queue \
151     ! ivas_xfilter kernels-config="/opt/xilinx/share/ivas/cmpk/
    analysis/4K/drawPower.json"  ! queue \
152     ! ivas_xfilter kernels-config="/opt/xilinx/share/ivas/cmpk/
    analysis/4K/drawTemp.json" ! queue \
153     ! ivas_xfilter kernels-config="/opt/xilinx/share/ivas/cmpk/
    analysis/4K/drawPLTemp.json" ! queue \
154     ! ivas_xfilter kernels-config="/opt/xilinx/share/ivas/cmpk/
    analysis/4K/drawfpsB1.json" ! queue \
155     ! ivas_xfilter kernels-config="/opt/xilinx/share/ivas/cmpk/
    analysis/4K/drawfpsB2.json" ! queue \
156     ! kmssink bus-id=b0000000.v_mix plane-id=37 render-rectangle
    ="<1920,0,1920,1080>" show-preroll-frame=false sync=false \
```

```
#! /bin/sh
# this is for 1080P output
video=/home/petalinux/videos/cars1900.nv12.h264
branch1="reid"
source=file
conf_pre_onlyresize="\"/opt/xilinx/share/ivas/cmpk/preprocess/
    resize_smartcam.json\""
conf_pre_seg="\"/opt/xilinx/share/ivas/cmpk/segmentation/
    preprocess_seg_smartcam.json\""
conf_pre_seg=$conf_pre_onlyresize
conf_dpu_seg="\"/opt/xilinx/share/ivas/cmpk/segmentation/dpu_seg.
    json\""
conf_draw_seg="\"/opt/xilinx/share/ivas/cmpk/segmentation/
    drawSegmentationTR.json\""

while getopts f:br:i:sh opt
do
    case $opt in
        f)
            video=$OPTARG
            ;;
        b)
            segback="black"
            ;;
        r)
            branch1="$OPTARG"
            ;;
        s)
            sync="true"
            echo $sync
            ;;
        i)
            source=$OPTARG
            ;;
        :)
```

```
32              echo "-$OPTARG needs an argument"
33              ;;
34          h)
35              echo ""
36              echo "Help:"
37              echo "-f video file source"
38              echo "-b (optional) segmentation use black background"
39              echo "-r (optional) model for branch 1  [(reid),
    openopse]"
40              echo ""
41              ;;
42          *)
43              echo "-$opt not recognized"
44              ;;
45      esac
46 done
47
48 if [ $source == "usb" ]; then
49      source_cmd=""
50      source_cmd=$source_cmd"! video/x-raw, width=1920, height=1080"
51      source_cmd=$source_cmd"! videoconvert ! video/x-raw, format=
    NV12"
52 elif [ $source == "mipi" ]; then
53      source_cmd="mediasrcbin media-device=/dev/media0 v4l2src0::io-
    mode=dmabuf v4l2src0::stride-align=256 !video/x-raw, width
    =1920, height=1080, format=NV12, framerate=30/1"
54 elif [ $source == "file" ]; then
55      source_cmd="multifilesrc location=\"${video}\" ! h264parse !
    queue ! omxh264dec ! video/x-raw, format=NV12, framerate=30/1"
56 else
57      echo "unsupport video source :$source [usb,mipi,file]."
58      exit -1
59 fi
60
61 echo $source_cmd
```

```
62
63 if [ -f $video ]; then
64     echo "find video: $video"
65 else
66     echo "cant find video file: $video"
67     exit -1
68 fi
69
70
71 ivas_xfilter="! queue ! ivas_xfilter kernels-config="
72
73 if [ $branch1 == "reid" ]; then
74     branch1firstmodel="\"/opt/xilinx/share/ivas/aibox-reid/
    refinedet.json\""
75     branch1crop="\"/opt/xilinx/share/ivas/aibox-reid/crop.json\""
76     branch1model="\"/opt/xilinx/share/ivas/cmpk/reid/reid.json\""
77     branch1draw="\"/opt/xilinx/share/ivas/cmpk/reid/draw_reid.json
    \""
78     branch1cmd="$ivas_xfilter $branch1firstmodel $ivas_xfilter
    $branch1crop $ivas_xfilter $branch1model"
79     echo "branch 1: use reid"
80 elif [ $branch1 == "carid" ]; then
81     branch1firstmodel="\"/opt/xilinx/share/ivas/smartcam/myapp/
    dpu_yolo2.json\""
82     branch1crop="\"/opt/xilinx/share/ivas/aibox-reid/crop.json\""
83     branch1model="\"/opt/xilinx/share/ivas/cmpk/reid/reid.json\""
84     branch1draw="\"/opt/xilinx/share/ivas/cmpk/reid/draw_reid.json
    \""
85     branch1cmd="$ivas_xfilter $branch1firstmodel $ivas_xfilter
    $branch1crop $ivas_xfilter $branch1model"
86     echo "branch 1: use reid"
87 elif [ $branch1 == "openpose" ]; then
88     branch1firstmodel="\"/opt/xilinx/share/ivas/aibox-reid/
    refinedet.json\""
89     branch1crop="\"/opt/xilinx/share/ivas/cmpk/openpose/crop.json
```

```
    \""
90      branch1model="\"/opt/xilinx/share/ivas/cmpk/openpose/openpose.
    json\""
91      branch1draw="\"/opt/xilinx/share/ivas/cmpk/openpose/draw_pose.
    json\""
92      branch1cmd="$ivas_xfilter $branch1firstmodel $ivas_xfilter
    $branch1crop $ivas_xfilter $branch1model"
93      echo "branch 1: use openopse"
94  elif [ $branch1 == "yolo" ]; then
95      branch1firstmodel="\"/opt/xilinx/share/ivas/smartcam/myapp/
    dpu_yolo2.json\""
96      branch1cmd="$ivas_xfilter $branch1firstmodel"
97      branch1draw="\"/opt/xilinx/share/ivas/smartcam/myapp/drawbox.
    json\""
98  else
99      echo error
100     exit -2
101 fi
102
103
104
105
106
107
108 gst-launch-1.0 \
109 $source_cmd \
110 ! tee name=t  \
111     ! queue ! ivas_xmultisrc kconfig=$conf_pre_onlyresize  \
112     $branch1cmd  \
113     ! ima.sink_master ivas_xmetaaffixer name=ima ima.src_master !
    fakesink  \
114 t. \
115     ! queue \
116     ! ivas_xmultisrc kconfig=$conf_pre_seg  \
117     ! queue \
```

```
    ! ivas_xfilter kernels-config=$conf_dpu_seg  \
    ! ima2.sink_master ivas_xmetaaffixer name=ima2 ima2.src_master
    \
    ! fakesink  \
t. \
    ! queue \
    ! ima.sink_slave_0 ima.src_slave_0  \
    ! queue \
    ! ivas_xfilter kernels-config=$branch1draw \
    ! queue  \
    ! ima2.sink_slave_0 ima2.src_slave_0  \
    ! queue ! ivas_xfilter kernels-config=$conf_draw_seg  \
! queue ! ivas_xfilter kernels-config="/opt/xilinx/share/ivas/
    smartcam/myapp/drawPower.json"  \
! queue ! ivas_xfilter kernels-config="/opt/xilinx/share/ivas/
    smartcam/myapp/drawTemp.json" \
! queue ! ivas_xfilter kernels-config="/opt/xilinx/share/ivas/
    smartcam/myapp/drawPerformance.json" \
! queue ! kmssink driver-name=xlnx plane-id=39 sync=false
    fullscreen-overlay=true
```

B.4 Python management interfaces for adaptive platform

```
import os, sys
import time
import threading
def getFPS():
    result= []
    read_path = "/home/petalinux/.temp/pf_tx"
    rf = os.open(read_path, os.O_RDONLY)
    s = b''
    while True:
        try:
```

```
11              s += os.read(rf, 1024)
12              if(len(s) >= 1024):
13                  continue
14
15              print("received msg:",len(s))
16              for string in s.split():
17                  info = str(string,encoding = "utf-8").split(',')
18                  if(info[0] == 'reportFPS'):
19                      result.append(float(info[1]))
20
21              break
22          except NameError as error:
23              print(error)
24              break
25
26      os.close(rf)
27      os.remove(read_path)
28      return result
29  class kv260adpFPS():
30      _instance_dict = {}
31      busy = False
32      def __new__(cls,read_path, *args, **kw):
33
34          if(read_path in cls._instance_dict.keys()):
35              print("warning:",read_path," already in program!")
36              return cls._instance_dict[read_path]
37          cls._instance_dict[read_path] = object.__new__(cls)
38          return cls._instance_dict[read_path]
39
40
41      def __init__(self, read_path,len = 5):
42          self.read_path = read_path
43          self.len =len
44
45      def get_bytesVersion(self,timeout = None):
```

```
46          if(self.busy):
47              print("busy")
48              return
49
50          self.result = []
51          self.busy = True
52          if(not  os.path.exists(self.read_path)):
53              os.mkfifo(self.read_path)
54          def readfifodata():
55              self.rf = os.open(self.read_path, os.O_RDONLY)
56              s = b''
57              while True:
58                  s = os.read(self.rf, 1024)
59                  if(len(s) >= 1024):
60                      continue
61  #                 print("received msg:",len(s))
62
63                  for string in s.split():
64
65                      fps = self.praseFPSStr(str(string,encoding = "
    utf-8"))
66                      if(not fps== None):
67                          self.result.append(fps)
68                  s = b''
69                  if(len(self.result)>= self.len):
70                      break
71                  time.sleep(0.1)
72
73              os.close(self.rf)
74
75          t1 = threading.Thread(target=readfifodata,args=())
76          t1.start()
77          t1.join(timeout)
78          os.remove(self.read_path)
79          self.busy = False
```

```
80          return self.result
81
82
83      def praseFPSStr(self,msg):
84          info = msg.split(',')
85          if(info[0] == 'reportFPS'):
86              try:
87                  fps = float(info[1])
88              except:
89                  print("invaild data:",info[1])
90                  fps = None
91          return fps
92
93
94      def get_fileVersion(self,timeout = None):
95          if(self.busy):
96              print("busy")
97              return
98          self.results = []
99          self.busy = True
100         if(not os.path.exists(self.read_path)):
101             os.mkfifo(self.read_path)
102         def func():
103             fifo_read_fd  = open(self.read_path,'r')
104             maxcnt = self.len
105             while(maxcnt>0):
106                 line = fifo_read_fd.readline()
107                 if len(line) == 0:
108                     time.sleep(0.1)
109                     continue
110                 maxcnt-=1
111 #                 print(line)
112                 self.results.append(self.praseFPSStr(line))
113             fifo_read_fd.close()
114         t1 = threading.Thread(target=func,args=())
```

```
115         t1.start()
116         t1.join(timeout)
117         os.remove(self.read_path)
118         self.busy = False
119         return self.results
120 class FIFOSendObj():
121     _instance_dict = {}
122     busy = False
123     timeout = 1
124 
125     def __new__(cls, write_path, *args, **kw):
126         if(write_path in cls._instance_dict.keys()):
127             return cls._instance_dict[write_path]
128         cls._instance_dict[write_path] = object.__new__(cls)
129         return cls._instance_dict[write_path]
130 
131     def __init__(self,write_path, timeout = 1, *args, **kw):
132         self.write_path =  write_path
133         self.timeout = timeout
134 
135     def writeData2FIFO(self,msg):
136         if(self.busy):
137             print("Bus is busy")
138             return
139 
140         self.busy = True
141         self.istimeout = True
142         def writefifodata(write_path,msg):
143             if(not os.path.exists(write_path)):
144                 os.mkfifo(write_path)
145             fifo_write_fd = open(write_path, 'w', 1 )
146             protect = open(write_path, 'r')
147             fifo_write_fd.write(msg)
148             fifo_write_fd.flush()
149             fifo_write_fd.close()
```

```
150             time.sleep(0.2)
151             protect.close()
152             self.istimeout = False
153 #             print("write finished")
154 
155         self.t1 = threading.Thread(target=writefifodata,args=(self
    .write_path,msg))
156         self.t1.start()
157         self.t1.join(self.timeout)
158         if(os.path.exists(self.write_path)):
159             os.remove(self.write_path)
160         self.busy = False
161         if(self.istimeout):
162             print("write fifo timeout!")
163             return False
164 #
165         else:
166             return True
167 class kv260adpModelCtr(object):
168 
169     modelname = ""
170     modelpath = ""
171     modelclass = ""
172     enable_str = ["ON","1"]
173     disable_str = ["OFF","0"]
174 
175     def __init__(self,write_path="",*args, **kw):
176         self.write_path = write_path
177         self.timeout = 1;
178 
179 
180     def checkModelfile(self):
181         if(self.modelname == "" or self.modelpath==""):
182             assert self.write_path != "" ,"model file path unset"
183 
```

```
184         if(not os.path.exists(os.path.join(self.modelpath,self.
    modelname,self.modelname+'.xmodel'))):
185             assert self.write_path != "" ,"xmodel file does not
    exist!"
186 
187         if(not os.path.exists(os.path.join(self.modelpath,self.
    modelname,self.modelname+'.prototxt'))):
188             assert self.write_path != "" ,"prototxt file does not
    exist!"
189         return True
190 
191     def checkModelClass(self):
192         vaildclasses=['YOLOV3','FACEDETECT','CLASSIFICATION','SSD'
    ,'REID','REFINEDET','TFSSD','YOLOV2','ROADLINE','SEGMENTATION']
193         if(self.modelclass not in vaildclasses):
194             assert self.write_path != "" ,"invalid class, please
    use valid calss name:{}".format(vaildclasses)
195 
196         return True
197 
198     def checkWritePath(self):
199         assert self.write_path != "" ,"please set write path"
200 
201     def resetWritePath(self, write_path):
202         if(write_path != ""):
203             self.write_path = write_path;
204         self.checkWritePath()
205 
206 #
    ================================================================================

207     def setNewModel(self,modelname, modelclass, modelpath,
    write_path = ""):
208         self.resetWritePath(write_path)
209         header = "switch2model"
```

```
210         self.modelname = modelname
211         self.modelpath = modelpath
212         self.modelclass = modelclass
213         self.checkModelfile()
214         self.checkModelClass()
215         cmd = "{},{},{},{}".format(header,self.modelname,self.
    modelclass,self.modelpath)
216         fifosend = FIFOSendObj(self.write_path,self.timeout)
217         fifosend.writeData2FIFO(cmd)
218 
219     def setNewREIDModel(self,modelname,modelpath,write_path = ""):
220         self.resetWritePath(write_path)
221         header = "switch2reidmodel"
222         self.modelname = modelname
223         self.modelpath = modelpath
224 
225         if(not self.checkModelfile()):
226             return
227 
228         cmd = "{},{},{}".format(header,self.modelname,self.
    modelpath)
229         fifosend = FIFOSendObj(self.write_path,self.timeout)
230         fifosend.writeData2FIFO(cmd)
231 
232     def setDPUInvteral(self,inverteral,write_path = ""):
233         self.resetWritePath(write_path)
234         inverteral = int(inverteral)
235         header = "pluginCtr_invteral"
236         if(inverteral<1):
237             print("invalid value:",inverteral)
238             return
239         fifosend = FIFOSendObj(self.write_path,self.timeout)
240         fifosend.writeData2FIFO('{},{}'.format(header,inverteral))
241 
242 
```

```
243     def setDPUenable(self,enable,write_path = ""):
244         self.resetWritePath(write_path)
245         header = "pluginCtr_DPUenable"
246 
247 
248         if str(enable).upper() in self.enable_str:
249             ctr = 1
250         elif str(enable).upper() in self.disable_str:
251             ctr = 0
252         else:
253             print("invaild",str(enable).upper(),)
254             return
255 
256         fifosend = FIFOSendObj(self.write_path,self.timeout)
257         fifosend.writeData2FIFO('{},{}'.format(header,ctr))
258 
259     def setIndicaterUI(self,on,write_path = ""):
260         self.resetWritePath(write_path)
261 
262         if(on == "ON" or on == 1 or on == "on"):
263             status = 1;
264         else:
265             status = 0;
266 
267 
268         header = "runindicater"
269         fifosend = FIFOSendObj(self.write_path,self.timeout)
270         fifosend.writeData2FIFO('{},{}'.format(header,status))
271 
272     def getFPSfromFile(self, file):
273         f = open(file)
274         line = f.readline()
275 #         print("fps:",line)
276         return float(str(line))
277 
```

```
278     def getSegmentationResult(self,file):
279         f = open(file)
280         line = f.readline()
281         rescnt = []
282         p_sum = 0
283         for s in line.split(','):
284             rescnt.append(int(s))
285             p_sum += int(s)
286 
287         res_normal = [x/p_sum for x in rescnt]
288 
289         car_related = rescnt[13] + rescnt[14] + rescnt[15] +
    rescnt[17]
290         people_related = rescnt[18] + rescnt[11] + rescnt[12]
291 
292         k1 = 0.1
293         k2 = 0.1
294         k3 = 0.2
295         k4 = 1.5
296         if(car_related < k1 and people_related < k2 ):
297             if(rescnt[0]> k3):
298                 return 'car'
299             else:
300                 return None
301 
302 
303         if(car_related > people_related * 1.5):
304             return 'car'
305         else:
306             return 'people'
307 
308 
309 
310 def getSegRes_FFC():
311         classification = {-1:"unknown",0:"people",1:"car",2:"road"
```

```
    }
312         result = -1
313         read_path = "/home/petalinux/.temp/segresults"
314         rf = os.open(read_path, os.O_RDONLY)
315         s = b''
316         while True:
317             try:
318                 s += os.read(rf, 1024)
319                 if(len(s) >= 1024):
320                     continue
321     #              print("received msg:",len(s),s)
322                 for string in s.split():
323                     info = str(string,encoding = "utf-8").split(',
    ')
324                     if(info[0] == 'reportSeg' and int(info[1]) in
    classification.keys()):
325                         result = int(info[1])
326                 break
327             except NameError as error:
328                 print(error)
329                 break
330
331         os.close(rf)
332         os.remove(read_path)
333         return result
334
335
336
337 def reidSwtichModel():
338     reid_models = [["carid","/opt/xilinx/share/vitis_ai_library/
    models/B3136/"],
339                    ["personreid-res18_pt","/opt/xilinx/share/
    vitis_ai_library/models/kv260-aibox-reid/"],
340                    ["RN18_08","/opt/xilinx/share/vitis_ai_library/
    models/B3136/carid/"],
```

```
341                     ["RN18_06","/opt/xilinx/share/vitis_ai_library/
    models/B3136/carid/"],
342                     ["RN18_04","/opt/xilinx/share/vitis_ai_library/
    models/B3136/carid/"],
343                     ["RN18_02","/opt/xilinx/share/vitis_ai_library/
    models/B3136/carid/"]]
344     reid_modelctr = kv260adpModelCtr("/home/petalinux/.temp/reidrx
    ")
345     reid_modelctr.setNewREIDModel(reid_models[5][0],reid_models
    [5][1])
346
347 def segmentationSwtichModel():
348     modelctr = kv260adpModelCtr("/home/petalinux/.temp/dpu_seg_rx"
    )
349     modelctr.setNewModel("SemanticFPN_cityscapes_256_512","
    SEGMENTATION","/opt/xilinx/share/vitis_ai_library/models/B3136/
    ")
350     modelctr.setNewModel("ENet_cityscapes_pt","SEGMENTATION","/opt
    /xilinx/share/vitis_ai_library/models/B3136/")
351     modelctr.setDPUInvteral(10)
352
353
354
355
356
357 def branchSwitch(maxcnt = -1):
358
359     # READ
360     FFC_SEG_RES = "/home/petalinux/.temp/segresults"
361     FFC_FPS_HB = '/home/petalinux/.temp/pf_tx'
362     FILE_FPS = '/home/petalinux/.temp/fps'
363     FILE_FPS1 = '/home/petalinux/.temp/fps_branch1'
364     FILE_FPS2 = '/home/petalinux/.temp/fps_branch2'
365     FILE_SEGMENTATION = '/home/petalinux/.temp/segres'
366     # WRITE
```

```
367     FFC_UI_BRANCH1 = '/home/petalinux/.temp/runstatus1_rx'
368     FFC_UI_BRANCH2 = '/home/petalinux/.temp/runstatus2_rx'
369     FFC_DPU_BRANCH_CAR_CTR = '/home/petalinux/.temp/dpu_yolo_rx'
370     FFC_DPU_BRANCH_PEO_CTR ='/home/petalinux/.temp/
    dpu_refinedet_rx'
371     FFC_DPU_SEG_CTR = '/home/petalinux/.temp/dpu_seg_rx'
372
373     lastseg = -1
374     traffic_modelctr = kv260adpModelCtr()
375     traffic_modelctr.setDPUInvteral(30,FFC_DPU_SEG_CTR)
376     while(maxcnt != 0):
377 #         fps1 = traffic_modelctr.getFPSfromFile(FILE_FPS1)
378 #         fps2 = traffic_modelctr.getFPSfromFile(FILE_FPS2)
379         maxcnt -= 1
380         seg = getSegRes_FFC()
381 #         seg = traffic_modelctr.getSegmentationResult(
    FILE_SEGMENTATION)
382         if lastseg != seg:
383             print("do switch",lastseg,seg)
384             lastseg = seg
385             if(seg in [0]):
386                 traffic_modelctr.setIndicaterUI('on',
    FFC_UI_BRANCH2)
387                 traffic_modelctr.setIndicaterUI('off',
    FFC_UI_BRANCH1)
388                 traffic_modelctr.setDPUenable('on',
    FFC_DPU_BRANCH_CAR_CTR)
389                 traffic_modelctr.setDPUenable('off',
    FFC_DPU_BRANCH_PEO_CTR)
390
391             elif(seg in [1]):
392                 traffic_modelctr.setIndicaterUI('on',
    FFC_UI_BRANCH1)
393                 traffic_modelctr.setIndicaterUI('off',
    FFC_UI_BRANCH2)
```

```
394                 traffic_modelctr.setDPUenable('off',
    FFC_DPU_BRANCH_CAR_CTR)
395                 traffic_modelctr.setDPUenable('on',
    FFC_DPU_BRANCH_PEO_CTR)
396
397         time.sleep(1)
398
399 # traffic_modelctr.setDPUenable('0',FFC_DPU_BRANCH_PEO_CTR)
400 # traffic_modelctr.setDPUenable('on',FFC_DPU_BRANCH_CAR_CTR)
401 # traffic_modelctr.setDPUenable('on',FFC_DPU_BRANCH_CAR_CTR)
402 # traffic_modelctr.setDPUInvteral(30,FFC_DPU_SEG_CTR)
403 # branchSwitch()
404 # getSegRes_FFC()
405 branchSwitch()
```

B.5 C++ program: VVAS plugins

For more source codes of VVAS pulgins, please refer to `https://github.com/luyufan498/VVAS_CMPK`.

B.5.1 Parse commands from the management program

```
1  int fifoComCtr_DPUInvteral(ivas_xkpriv * kpriv)
2  {
3      std::string header = "pluginCtr_invteral";
4
5      cmpk::fifocom *ffc = &kpriv->ffc;
6      if(ffc->lines_buffer.size()<(1+1))
7      {
8          return -1;
9      }
10
11     if(ffc->lines_buffer[0].compare(header))
12     {
```

```
13          return -1;
14      }
15
16      string num_string = ffc->lines_buffer[1];
17      int value = atoi(num_string.c_str());
18
19
20      if(value <= 0)
21      {
22
23          LOG_MESSAGE (LOG_LEVEL_ERROR, kpriv->log_level, "ffc:
    invail interval value %s --> %d",ffc->lines_buffer[1].c_str(),
    value);
24          return -1;
25      }
26
27      kpriv->interval_frames = value;
28
29      cout<< "reset interval value "<<ffc->lines_buffer[1]<<" "<<
    value<<endl;
30      // LOG_MESSAGE (LOG_LEVEL_DEBUG, kpriv->log_level, "reset
    interval value %d",value);
31
32      return 1;
33  }
34
35
36  int fifoComCtr_DPUenable(ivas_xkpriv * kpriv)
37  {
38      std::string header = "pluginCtr_DPUenable";
39
40      cmpk::fifocom *ffc = &kpriv->ffc;
41      if(ffc->lines_buffer.size()<(1+1))
42      {
43          return -1;
```

```
44      }
45
46      if(ffc->lines_buffer[0].compare(header))
47      {
48          return -1;
49      }
50
51      string num_string = ffc->lines_buffer[1];
52      int value = atoi(num_string.c_str());
53
54      if(value < 0 or value > 1)
55      {
56          LOG_MESSAGE (LOG_LEVEL_ERROR, kpriv->log_level, "ffc:
    invail interval value %s --> %d",ffc->lines_buffer[1].c_str(),
    value);
57          return -1;
58      }
59
60      if(value == 1)
61      {
62          cout<< "enable dpu inference:"<<kpriv->modelname<<endl;
63          kpriv->enable = true;
64      }
65      else
66      {
67          cout<< "disable dpu inference:"<<kpriv->modelname<<endl;
68          kpriv->enable = false;
69      }
70
71
72      LOG_MESSAGE (LOG_LEVEL_DEBUG, kpriv->log_level, "DPU enable: %
    d",value);
73
74      return 1;
75  }
```

```
76
77
78 int fifoComCtr_DynamicModel(fifocom *ffc){
79     std::string header = "switch2model";
80
81     if(ffc->lines_buffer.size()<(3+1))
82     {
83         return false;
84     }
85
86     if(ffc->lines_buffer[0].compare(header))
87     {
88         cout<<"woring header:"<<ffc->lines_buffer[0]<<endl;
89         return false;
90     }
91
92
93     ffc->modelinfo.model_name = ffc->lines_buffer[1];
94     ffc->modelinfo.model_class = ffc->lines_buffer[2];
95     ffc->modelinfo.model_path = ffc->lines_buffer[3];
96
97
98
99     if (!fileexists (ffc->modelinfo.model_path))
100    {
101        //check path
102        cout<< "ERROR Model Read:"<<ffc->modelinfo.model_name<<" "
    <<ffc->modelinfo.model_class
103    <<" "<<ffc->modelinfo.model_path<<endl;
104        return false;
105    }
106
107    cout<< "FIFO Model Read:"<<ffc->modelinfo.model_name<<" "<<ffc
    ->modelinfo.model_class
108    <<" "<<ffc->modelinfo.model_path<<endl;
```

```
109
110     return true;
111 }
112
113
114 int loadDynamicModelfromFFC(ivas_xkpriv * kpriv)
115 {
116
117     if(!kpriv->run_time_model)
118     {
119         return 0;
120     }
121
122     cmpk::fifocom *ffc = &kpriv->ffc;
123
124     if(fifoComCtr_DynamicModel(ffc))
125     {
126
127       ivas_xkpriv *tmpxkpriv = (ivas_xkpriv *) calloc (1, sizeof (
    ivas_xkpriv));
128     //    ivas_xkpriv tmpxkpriv;
129       // add params
130       tmpxkpriv->modelname = ffc->modelinfo.model_name;
131       tmpxkpriv->modelpath = ffc->modelinfo.model_path;
132       tmpxkpriv->modelclass = ivas_xclass_to_num((char*)ffc->
    modelinfo.model_class.c_str());
133
134
135       tmpxkpriv->elfname =  modelexits (tmpxkpriv);
136       if (tmpxkpriv->elfname.empty ()) {
137         LOG_MESSAGE (LOG_LEVEL_ERROR, kpriv->log_level,"dynamic
    model:%s check failed\n",tmpxkpriv->modelname.c_str());
138         return -1;
139       }
140
```

```
141     tmpxkpriv->need_preprocess = kpriv->need_preprocess;
142
143     // create
144     tmpxkpriv->model = ivas_xinitmodel (tmpxkpriv, tmpxkpriv->
    modelclass);
145
146  //   LOG_MESSAGE (LOG_LEVEL_ERROR, kpriv->log_level,"enter %p
    ",tmpxkpriv->model);
147
148
149     if(tmpxkpriv->model == NULL){
150       LOG_MESSAGE (LOG_LEVEL_ERROR, kpriv->log_level,"dynamic
    model:%s intt failed\n",tmpxkpriv->modelname.c_str());
151       return -1;
152     }
153
154     cout<< tmpxkpriv->modelpath <<endl;
155     cout<< tmpxkpriv->modelname <<endl;
156     cout<< tmpxkpriv->modelclass <<endl;
157     cout<< tmpxkpriv->elfname <<endl;
158
159     //clear model and label
160     ivas_clean_currentmodel(kpriv);
161
162     // change model
163     kpriv->modelname = tmpxkpriv->modelname;
164     kpriv->modelpath = tmpxkpriv->modelpath;
165     kpriv->modelclass = tmpxkpriv->modelclass;
166     kpriv->elfname = tmpxkpriv->elfname;
167
168     // change label
169     kpriv->labelptr = tmpxkpriv->labelptr;
170     kpriv->labelflags = tmpxkpriv->labelflags;
171     kpriv->max_labels =tmpxkpriv->max_labels;
172
```

```
173
174         kpriv->model =    tmpxkpriv->model;
175
176         free(tmpxkpriv);
177         // kpriv->modelfmt = tmpxkpriv.modelfmt;
178
179         // // change priority
180         // kpriv->priority = tmpxkpriv.priority;
181
182     }
183
184     return 1;
185
186 }
187
188 void fifoComCtrAll(ivas_xkpriv * kpriv){
189     cmpk::fifocom *ffc = &kpriv->ffc;
190     cmpk::fifoComRead(ffc);
191
192
193     loadDynamicModelfromFFC(kpriv);
194     fifoComCtr_DPUInvteral(kpriv);
195     fifoComCtr_DPUenable(kpriv);
196
197 }
```

B.5.2 Conduct inference with DPU

```
1 #include <opencv2/core.hpp>
2 #include <opencv2/highgui.hpp>
3 #include <opencv2/imgproc.hpp>
4 #include <opencv2/opencv.hpp>
5 #include <sys/time.h>
6 #include <sys/stat.h>
7 #include <unistd.h>
```

```
8  #include <string>
9  #include <fstream>
10
11 #include <vitis/ai/bounded_queue.hpp>
12 #include <vitis/ai/env_config.hpp>
13
14 extern "C"
15 {
16 #include <ivas/ivas_kernel.h>
17 }
18 #include <gst/ivas/gstinferencemeta.h>
19 #include <gst/ivas/gstivasinpinfer.h>
20
21
22 #include "ivas_xdpupriv.hpp"
23 #include "ivas_xdpumodels.hpp"
24
25 #include "../../cm_package/cmpk_segmentation.hpp"
26 #include "../../cm_package/cmpk_json_utils.hpp"
27
28 #include "dpuinfer_partial_ffc.hpp"
29 #include "dpuinfer_partial_model.hpp"
30
31
32
33 using namespace cv;
34 using namespace std;
35 using namespace cmpk;
36
37 ivas_xdpumodel::~ivas_xdpumodel ()
38 {
39 }
40
41 /**
42  * fileexists () - Check either file exists or not
```

```
43  *
44  * check either able to open the file whoes path is in name
45  *
46  */
47 inline bool
48 fileexists (const string & name)
49 {
50   struct stat buffer;
51   return (stat (name.c_str (), &buffer) == 0);
52 }
53
54 int
55 performanceTestStart(ivas_xkpriv * kpriv)
56 {
57   ivas_perf *pf = &kpriv->pf;
58   if (kpriv->performance_test && !kpriv->pf.test_started) {
59       pf->timer_start = get_time ();
60       pf->last_displayed_time = pf->timer_start;
61       pf->test_started = 1;
62     }
63   return 0;
64 }
65
66
67 int
68 performanceTestRecord(ivas_xkpriv * kpriv)
69 {
70   ivas_perf *pf = &kpriv->pf;
71
72   pf->frames++;
73
74   if(!kpriv->performance_test)
75     return 0;
76
77   if(!kpriv->pf.test_started)
```

```
78      return 0;
79
80
81    if (get_time () - pf->last_displayed_time >= 1000000.0) {
82      long long current_time = get_time ();
83      double time = (current_time - pf->last_displayed_time) /
       1000000.0;
84      pf->last_displayed_time = current_time;
85      double fps = (time > 0.0) ? ((pf->frames - pf->
       last_displayed_frame) / time) : 999.99;
86      pf->last_displayed_frame = pf->frames;
87
88      pf->avgFPS = fps;
89
90      if (kpriv->performance_test && kpriv->pf.test_started) {
91
92          char buff[20] = {0};
93          sprintf(buff,"FPS:%f \r",fps);
94          // cmpk::ivas_fifocommuncation_send_raw(kpriv->ffc, buff,
       strlen(buff));
95          printf ("\rframe=%5lu fps=%6.*f        \r", pf->frames,(
       fps < 9.995) ? 3 : 2, fps); fflush (stdout);
96      }
97    }
98
99
100
101   return 0;
102 }
103
104
105
106
107 extern "C"
108 {
```

```
109
110 int32_t xlnx_kernel_init (IVASKernel * handle)
111 {
112   ivas_xkpriv *kpriv = (ivas_xkpriv *) calloc (1, sizeof (
    ivas_xkpriv));
113     kpriv->handle = handle;
114
115   json_t *jconfig = handle->kernel_config;
116   json_t *val,*karray = NULL,*jmodel = NULL;                /*
    kernel config from app */
117
118   XkprivGetJsonData_int(jconfig,&(kpriv->log_level),"debug_level
    ",0,3);
119   XkprivGetJsonData_bool(jconfig,&(kpriv->run_time_model),"
    run_time_model",false, kpriv->log_level);
120   XkprivGetJsonData_bool(jconfig,&(kpriv->performance_test),"
    performance_test",false, kpriv->log_level);
121   XkprivGetJsonData_bool(jconfig,&(kpriv->need_preprocess),"
    need_preprocess",true, kpriv->log_level);
122   XkprivGetJsonData_bool(jconfig,&(kpriv->enable),"enable",true,
    kpriv->log_level);
123   XkprivGetJsonData_bool(jconfig,&(kpriv->buff_en),"buff_en",
    true, kpriv->log_level);
124
125
126   string tmp_videofmt;
127   XkprivGetJsonData_string(jconfig,&(tmp_videofmt),"model-format
    ","BGR", kpriv->log_level);
128   kpriv->modelfmt = ivas_fmt_to_xfmt (tmp_videofmt.data());
129   if (kpriv->modelfmt == IVAS_VMFT_UNKNOWN) {
130     LOG_MESSAGE (LOG_LEVEL_ERROR, kpriv->log_level,"SORRY NOT
    SUPPORTED MODEL FORMAT %s",(char *) json_string_value (val));
131     goto err;
132   }
133
```

```
134     XkprivGetJsonData_string(jconfig,&(kpriv->modelpath),"model-
        path","/usr/share/vitis_ai_library/models/", kpriv->log_level);
135     if (!fileexists (kpriv->modelpath)) {
136       LOG_MESSAGE (LOG_LEVEL_ERROR, kpriv->log_level,
137           "model-path (%s) not exist", kpriv->modelpath.c_str ());
138       goto err;
139     }
140
141     LOG_MESSAGE (LOG_LEVEL_DEBUG, kpriv->log_level, "mid");
142     XkprivGetJsonData_string(jconfig,&(kpriv->ffc.txpath),"
        ffc_txpath",FIFO_WRITE, kpriv->log_level);
143     XkprivGetJsonData_string(jconfig,&(kpriv->ffc.rxpath),"
        ffc_rxpath",FIFO_READ, kpriv->log_level);
144     XkprivGetJsonData_int(jconfig,&(kpriv->ffc.tx_frame_interval),
        "tx_frame_interval",30,kpriv->log_level);
145     XkprivGetJsonData_int(jconfig,&(kpriv->ffc.rx_frame_interval),
        "rx_frame_interval",30,kpriv->log_level);
146     XkprivGetJsonData_int(jconfig,&(kpriv->target_fps),"target_fps
        ",30,kpriv->log_level);
147     XkprivGetJsonData_int(jconfig,&(kpriv->interval_frames),"
        interval_frames",1,kpriv->log_level);
148
149
150     // typedef int (*xkprivStringProcessAPI)(char *);
151     XkprivGetJsonData_string2Int(jconfig,&(kpriv->modelclass),"
        model-class",ivas_xclass_to_num,IVAS_XCLASS_NOTFOUND, kpriv->
        log_level);
152     if (kpriv->modelclass == IVAS_XCLASS_NOTFOUND) {
153       LOG_MESSAGE (LOG_LEVEL_ERROR, kpriv->log_level,"SORRY NOT
        SUPPORTED MODEL CLASS %s",(char *) json_string_value (val));
154       goto err;
155     }
156     XkprivGetJsonData_string(jconfig,&(kpriv->modelname),"model-
        name","", kpriv->log_level);
157     kpriv->elfname = modelexits (kpriv);
```

```
158     if (kpriv->elfname.empty ()) {
159       goto err;
160     }
161 
162     XkprivGetJsonData_int(jconfig,&(kpriv->priority),"priority",0,
      kpriv->log_level);
163     //fifo communication
164 //-----------------------------------------------------
165 
166     LOG_MESSAGE (LOG_LEVEL_INFO, kpriv->log_level, "model-name = %
      s",
167         (char *) json_string_value (val));
168     LOG_MESSAGE (LOG_LEVEL_DEBUG, kpriv->log_level, "model class
      is %d",
169         kpriv->modelclass);
170     LOG_MESSAGE (LOG_LEVEL_DEBUG, kpriv->log_level, "elf class is
      %s",
171         kpriv->elfname.c_str ());
172 
173     kpriv->model = ivas_xinitmodel (kpriv, kpriv->modelclass);
174     ivas_xsetcaps(kpriv,kpriv->model);
175     if (kpriv->model == NULL) {
176       LOG_MESSAGE (LOG_LEVEL_ERROR, kpriv->log_level,
177           "Init ivas_xinitmodel failed for %s", kpriv->modelname.
      c_str ());
178       goto err;
179     }
180 
181     handle->kernel_priv = (void *) kpriv;
182     return true;
183 
184   err:
185     free (kpriv);
186     return -1;
187   }
```

```
188
189 uint32_t xlnx_kernel_deinit (IVASKernel * handle)
190 {
191   ivas_xkpriv *kpriv = (ivas_xkpriv *) handle->kernel_priv;
192   if (!kpriv)
193     return true;
194   LOG_MESSAGE (LOG_LEVEL_DEBUG, kpriv->log_level, "enter");
195
196   ivas_perf *pf = &kpriv->pf;
197
198   if (kpriv->performance_test && kpriv->pf.test_started) {
199     double time = (get_time () - pf->timer_start) / 1000000.0;
200     double fps = (time > 0.0) ? (pf->frames / time) : 999.99;
201     printf ("\rframe=%5lu fps=%6.*f        \n", pf->frames,
202         (fps < 9.995) ? 3 : 2, fps);
203   }
204   pf->test_started = 0;
205   pf->frames = 0;
206   pf->last_displayed_frame = 0;
207   pf->timer_start = 0;
208   pf->last_displayed_time = 0;
209
210   if (!kpriv->run_time_model) {
211     for (int i = 0; i < int (kpriv->mlist.size ()); i++) {
212       if (kpriv->mlist[i].model) {
213         kpriv->mlist[i].model->close ();
214         delete kpriv->mlist[i].model;
215         kpriv->mlist[i].model = NULL;
216       }
217       kpriv->model = NULL;
218     }
219   }
220   kpriv->modelclass = IVAS_XCLASS_NOTFOUND;
221
222   if (kpriv->model != NULL) {
```

```
223       kpriv->model->close ();
224       delete kpriv->model;
225       kpriv->model = NULL;
226     }
227     if (kpriv->labelptr != NULL)
228       free (kpriv->labelptr);
229
230     ivas_caps_free (handle);
231     free (kpriv);
232
233     return true;
234   }
235
236   uint32_t xlnx_kernel_start (IVASKernel * handle, int start,
237       IVASFrame * input[MAX_NUM_OBJECT], IVASFrame * output[
      MAX_NUM_OBJECT])
238   {
239     ivas_xkpriv *kpriv = (ivas_xkpriv *) handle->kernel_priv;
240     cmpk::fifocom *ffc = &kpriv->ffc;
241     ivas_perf *pf = &kpriv->pf;
242     GstInferenceMeta *infer_meta = NULL;
243     GstIvasInpInferMeta *ivas_inputmeta = NULL;
244     IVASFrame *inframe = input[0];
245     char *indata = (char *) inframe->vaddr[0];
246     int ret, i;
247
248     LOG_MESSAGE (LOG_LEVEL_DEBUG, kpriv->log_level, "enter");
249
250     fifoComCtrAll(kpriv);
251     infer_meta = (GstInferenceMeta *) gst_buffer_add_meta ((
      GstBuffer *)inframe->app_priv, gst_inference_meta_get_info (),
      NULL);
252
253     if (infer_meta == NULL) {
254       LOG_MESSAGE (LOG_LEVEL_ERROR, kpriv->log_level,"ivas meta
```

```
data is not available for dpu");
255       return -1;
256     }
257
258
259     if(!kpriv->enable)
260     {
261       infer_meta->prediction->reserved_5 = (void*) -1;
262       return true;
263     }
264
265
266     cv::Mat image;
267     if (input[0]->props.fmt == IVAS_VFMT_BGR8 || input[0]->props.
fmt == IVAS_VFMT_RGB8)
268       image = cv::Mat (input[0]->props.height, input[0]->props.
width, CV_8UC3, indata, input[0]->props.stride);
269     else {
270       LOG_MESSAGE (LOG_LEVEL_ERROR, kpriv->log_level, "Not
supported format %d\n", input[0]->props.fmt);
271       return -1;
272     }
273
274     unsigned int width = kpriv->model->requiredwidth ();
275     unsigned int height = kpriv->model->requiredheight ();
276     if (width != inframe->props.width || height != inframe->props.
height) {
277       LOG_MESSAGE (LOG_LEVEL_WARNING, kpriv->log_level,"input
image size is [%d,%d], model required size is [%d,%d]",
278       inframe->props.width,inframe->props.height, width, height);
279       // return false; //TODO
280     }
281
282     ret = ivas_xrunmodel (kpriv, image, infer_meta, inframe);
283     performanceTestStart(kpriv);
```

```
284     performanceTestRecord(kpriv);
285
286     return ret;
287   }
288
289   int32_t xlnx_kernel_done (IVASKernel * handle)
290   {
291
292     ivas_xkpriv *kpriv = (ivas_xkpriv *) handle->kernel_priv;
293     LOG_MESSAGE (LOG_LEVEL_DEBUG, kpriv->log_level, "enter");
294     return true;
295   }
296
297 }
```

Bibliography

[1] AMD-Xilinx. All-in-one self-adaptive computing platform for smart city. `https://www.hackster.io/contests/xilinxadaptivecomputing2021`, 2022. Accessed: June 15, 2022.

[2] Yufan Lu. All-in-one self-adaptive computing platform for smart city. `https://www.hackster.io/yufan-lu/all-in-one-self-adaptive-computing-platform-for-smart-city-933ff2`, 2022. Accessed: June 15, 2022.

[3] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[4] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[5] George E Dahl, Dong Yu, Li Deng, and Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on audio, speech, and language processing*, 20(1):30–42, 2011.

[6] RCNN Faster. Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 9199(10.5555):2969239–2969250, 2015.

[7] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional

networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[8] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, pages 1045–1048. Makuhari, 2010.

[9] Zhida Bao, Yun Lin, Sicheng Zhang, Zixin Li, and Shiwen Mao. Threat of adversarial attacks on dl-based iot device identification. *IEEE Internet of Things Journal*, 9(11):9012–9024, 2021.

[10] Kaiqiang Zhang, Chris Hutson, James Knighton, Guido Herrmann, and Tom Scott. Radiation tolerance testing methodology of robotic manipulator prior to nuclear waste handling. *Frontiers in Robotics and AI*, 7:6, 2020.

[11] Ying Yang, Kun Yao, Matthew P Repasky, Karl Leswing, Robert Abel, Brian K Shoichet, and Steven V Jerome. Efficient exploration of chemical space with docking and deep learning. *Journal of Chemical Theory and Computation*, 17(11):7106–7119, 2021.

[12] Lijun Zhao, Qingsheng Li, and Guanhua Ding. Wireless control industrial robot processing irradiation system based on artificial intelligence technology. *Wireless Communications and Mobile Computing*, 2022, 2022.

[13] Edgar Liberis and Nicholas D Lane. Neural networks on microcontrollers: saving memory at inference via operator reordering. *arXiv preprint arXiv:1910.05110*, 2019.

[14] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[15] Travis DeWolf, Pawel Jaworski, and Chris Eliasmith. Nengo and low-power ai hardware for robust, embedded neurorobotics. *Frontiers in Neurorobotics*, 14:568359, 2020.

[16] Dmitrii Shadrin, Alexander Menshchikov, Dmitry Ermilov, and Andrey Somov. Designing future precision agriculture: Detection of seeds germination using artificial intelligence on a low-power embedded system. *IEEE Sensors Journal*, 19(23):11573–11582, 2019.

[17] Vivek Kothari, Edgar Liberis, and Nicholas D Lane. The final frontier: Deep learning in space. In *Proceedings of the 21st international workshop on mobile computing systems and applications*, pages 45–49, 2020.

[18] Jaques Reifman. Survey of artificial intelligence methods for detection and identification of component faults in nuclear power plants. *Nuclear Technology*, 119(1):76–97, 1997.

[19] Richard H Maurer, Martin E Fraeman, Mark N Martin, and David R Roth. Harsh environments: space radiation. *Johns Hopkins APL technical digest*, 28(1):17, 2008.

[20] H Vanmarcke. Unscear 2000: sources of ionizing radiation. *Annalen van de Belgische vereniging voor stralingsbescherming*, 27(2):41–65, 2002.

[21] George C. Messenger and Milton S. Ash. *The effects of radiation on electronic systems*. Van Nostrand Reinhold Co, 1986.

[22] A Campbell, P McDonald, and K Ray. Single event upset rates in space. *IEEE Transactions on Nuclear Science*, 39(6):1828–1835, 1992.

[23] Wassim Mansour and Raoul Velazco. An automated seu fault-injection method and tool for hdl-based designs. *IEEE Transactions on Nuclear Science*, 60(4):2728–2733, 2013.

[24] M Nicolaidis and R Perez. Measuring the width of transient pulses induced by ionising radiation. In *2003 IEEE International Reliability Physics Symposium Proceedings, 2003. 41st Annual.*, pages 56–59. IEEE, 2003.

[25] P Fernández-Martínez, I Cortés, S Hidalgo, D Flores, and FR Palomo. Simulation of total ionising dose in mos capacitors. In *Proceedings of the 8th Spanish Conference on Electron Devices, CDE'2011*, pages 1–4. IEEE, 2011.

[26] Daisuke Kobayashi. Scaling trends of digital single-event effects: A survey of seu and set parameters and comparison with transistor performance. *IEEE Transactions on Nuclear Science*, pages 1–1, 2020.

[27] Paul E Dodd. Physics-based simulation of single-event effects. *IEEE Transactions on Device and Materials Reliability*, 5(3):343–357, 2005.

[28] Boyang Du, Josie E Rodriguez Condia, M Sonza Reorda, and Luca Sterpone. On the evaluation of seu effects in gpgpus. In *2019 IEEE Latin American Test Symposium (LATS)*, pages 1–6. IEEE, 2019.

[29] Yi Sun, Hong-Wei Zhang, Zhi-Chao Wei, Qing-Kui Yu, Min Tang, Chen Shen, and Ding Gong. Heavy ion-and proton-induced seu simulation and error rates calculation in 0.15 um sram-based fpga. In *2019 3rd International Conference on Circuits, System and Simulation (ICCSS)*, pages 84–88. IEEE, 2019.

[30] Daniela Munteanu and J-L Autran. Modeling and simulation of single-event effects in digital devices and ics. *IEEE Transactions on Nuclear science*, 55(4):1854–1878, 2008.

[31] Robért Glein, Bernhard Schmidt, Florian Rittner, Jürgen Teich, and Daniel Ziener. A self-adaptive seu mitigation system for fpgas with an internal block ram radiation particle sensor. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 251–258. IEEE, 2014.

[32] Adam Jacobs, Grzegorz Cieslewski, Alan D George, Ann Gordon-Ross, and Herman Lam. Reconfigurable fault tolerance: A comprehensive framework

for reliable and adaptive fpga-based space computing. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 5(4):1–30, 2012.

[33] Po-Yuan Chen, Chin-Lung Su, Chao-Hsun Chen, and Cheng-Wen Wu. Generalization of an Enhanced ECC Methodology for Low Power PSRAM. *IEEE Transactions on Computers*, 62(7):1318–1331, jul 2013.

[34] Enrico Petritoli and Fabio Leccese. Reliability andSEE mitigation in memories for space applications. In *2016 IEEE Metrology for Aerospace (MetroAeroSpace)*, pages 136–140. IEEE, jun 2016.

[35] Salvatore Bianchi, Roberto Paggi, Gian Luca Mariotti, and Fabio Leccese. Why and when must the preventive maintenance be performed? In *2014 IEEE Metrology for Aerospace (MetroAeroSpace)*, pages 221–226. IEEE, may 2014.

[36] Shahin Golshan, Hessam Kooti, and Elaheh Bozorgzadeh. Seu-aware high-level data path synthesis and layout generation on sram-based fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(6):829–840, 2011.

[37] Mohammad Reza Rohanipoor, Behnam Ghavami, and Mohsen Raji. Improving combinational circuit reliability against multiple event transients via a partition and restructuring approach. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[38] F Lima Kastensmidt, Luca Sterpone, Luigi Carro, and M Sonza Reorda. On the optimal design of triple modular redundancy logic for sram-based fpgas. In *Design, Automation and Test in Europe*, pages 1290–1295. IEEE, 2005.

[39] W Cary Huffman and Vera Pless. *Fundamentals of error-correcting codes*. Cambridge university press, 2010.

[40] I. Herrera-Alzu and M. Lopez-Vallejo. Design Techniques for Xilinx Virtex

FPGA Configuration Memory Scrubbers. *IEEE Transactions on Nuclear Science*, 60(1):376–385, feb 2013.

[41] Tianming Jiang, Ping Huang, and Ke Zhou. Scrub unleveling: Achieving high data reliability at low scrubbing cost. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1403–1408. IEEE, 2019.

[42] Namhyung Kim and Kiyoung Choi. A design guideline for volatile STT-RAM with ECC and scrubbing. In *ISOCC 2015 - International SoC Design Conference: SoC for Internet of Everything (IoE)*, pages 29–30. IEEE, nov 2016.

[43] Gian Mayuga, Yasuo Sato, and Michiko Inoue. Highly reliable memory architecture using adaptive combination of proactive aging-aware in-field self-repair and ecc. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[44] Wooyoung Jang. Error-correcting code aware memory subsystem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(11):1706–1717, 2014.

[45] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: a large-scale field study. *ACM SIGMETRICS Performance Evaluation Review*, 37(1):193–204, 2009.

[46] Jangwoo Kim, Nikos Hardavellas, Ken Mai, Babak Falsafi, and James Hoe. Multi-bit error tolerant caches using two-dimensional error coding. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 197–209. IEEE, 2007.

[47] Raed Mesleh, Omar Hiari, and Abdelhamid Younis. Generalized space modulation techniques: Hardware design and considerations. *Physical Communication*, 26:87–95, 2018.

[48] Wei-Kai Cheng, Xin-Lun Li, and Jian-Kai Chen. Integration scheme for retention-aware DRAM refresh. In *2017 International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, pages 1–2. IEEE, oct 2017.

[49] Nour Sayed, Sarath Mohanachandran Nair, Rajendra Bishnoi, and Mehdi B. Tahoori. Process variation and temperature aware adaptive scrubbing for retention failures in STT-MRAM. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 203–208. IEEE, jan 2018.

[50] Wei-Chen Wang, Chien-Chung Ho, Yuan-Hao Chang, Tei-Wei Kuo, and Ping-Hsien Lin. Scrubbing-Aware Secure Deletion for 3-D NAND Flash. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2790–2801, nov 2018.

[51] Hooman Farzaneh, Ladan Malehmirchegini, Adrian Bejan, Taofeek Afolabi, Alphonce Mulumba, and Precious P Daka. Artificial intelligence evolution in smart buildings for energy efficiency. *Applied Sciences*, 11(2):763, 2021.

[52] Gianluca Furano, Gabriele Meoni, Aubrey Dunne, David Moloney, Veronique Ferlet-Cavrois, Antonis Tavoularis, Jonathan Byrne, Léonie Buckley, Mihalis Psarakis, Kay-Obbe Voss, et al. Towards the use of artificial intelligence on the edge in space systems: Challenges and opportunities. *IEEE Aerospace and Electronic Systems Magazine*, 35(12):44–56, 2020.

[53] Yanjiao Chen, Baolin Zheng, Zihan Zhang, Qian Wang, Chao Shen, and Qian Zhang. Deep learning on mobile and embedded devices: State-of-the-art, challenges, and future directions. *ACM Computing Surveys (CSUR)*, 53(4):1–37, 2020.

[54] Claire L Parkinson. Aqua: An earth-observing satellite mission to examine water and other climate variables. *IEEE Transactions on Geoscience and Remote Sensing*, 41(2):173–183, 2003.

[55] Robert Bogue. Robots in the nuclear industry: a review of technologies and applications. *Industrial Robot: An International Journal*, 2011.

[56] David Sands. Cost effective robotics in the nuclear industry. *Industrial Robot: An International Journal*, 2006.

[57] Dongjoon Park, Yuanlong Xiao, and André DeHon. Fast and flexible fpga development using hierarchical partial reconfiguration. In *2022 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–10. IEEE, 2022.

[58] Ben Taylor, Vicent Sanz Marco, Willy Wolff, Yehia Elkhatib, and Zheng Wang. Adaptive deep learning model selection on embedded systems. *ACM SIGPLAN Notices*, 53(6):31–43, 2018.

[59] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

[60] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once for all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*, 2020.

[61] Mahesh Kumar, Durga Digdarsini, Neeraj Misra, and T. V.S. Ram. SEU mitigation of Rad-Tolerant Xilinx FPGA using external scrubbing for geostationary mission. In *2017 4th International Conference on Signal Processing and Integrated Networks, SPIN 2017*, pages 414–418. IEEE, feb 2017.

[62] Ju-Yueh Lee, Cheng-Ru Chang, Naifeng Jing, Juexiao Su, Shijie Wen, Rich Wong, and Lei He. Heterogeneous configuration memory scrubbing for soft error mitigation in fpgas. In *2012 International Conference on Field-Programmable Technology*, pages 23–28. IEEE, 2012.

[63] Raffaele Giordano, Sabrina Perrella, Vincenzo Izzo, Giuliana Milluzzo, and Alberto Aloisio. Redundant-configuration scrubbing of sram-based fpgas. *IEEE Transactions on Nuclear Science*, 64(9):2497–2504, 2017.

[64] Rong-Sheng Zhang, Li-Yi Xiao, Xue-Bing Cao, Jie Li, Jia-Qiang Li, and Lin-Zhe Li. A Fast Scrubbing Method Based on Triple Modular Redundancy for SRAM-Based FPGAs. In *2018 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, pages 1–3. IEEE, oct 2018.

[65] Wilmot N Hess. Van allen belt protons from cosmic-ray neutron leakage. *Physical Review Letters*, 3(1):11, 1959.

[66] John E Naugle and Donald A Kniffen. Flux and energy spectra of the protons in the inner van allen belt. *Physical Review Letters*, 7(1):3, 1961.

[67] James Alfred Van Allen, George H Ludwig, Ernest Clark Ray, and Carl E McIlwain. Observation of high intensity radiation by satellites 1958 alpha and gamma. *Journal of Jet Propulsion*, 28(9):588–592, 1958.

[68] SR Elkington, MK Hudson, and AA Chan. Enhanced radial diffusion of outer zone electrons in an asymmetric geomagnetic field. In *AGU Spring Meeting Abstracts*, volume 2001, pages SM32C–04, 2001.

[69] YY Shprits and RM Thorne. Time dependent radial diffusion modeling of relativistic electrons with realistic loss rates. *Geophysical research letters*, 31(8), 2004.

[70] Stefano Gabici, Carmelo Evoli, Daniele Gaggero, Paolo Lipari, Philipp Mertsch, Elena Orlando, Andrew Strong, and Andrea Vittino. The origin of galactic cosmic rays: Challenges to the standard paradigm. *International Journal of Modern Physics D*, 28(15):1930022, 2019.

[71] JA Simpson. Elemental and isotopic composition of the galactic cosmic rays. *Annual Review of Nuclear and Particle Science*, 33(1):323–382, 1983.

[72] Giuseppe Di Sciascio. Measurement of energy spectrum and elemental composition of pev cosmic rays: Open problems and prospects. *Applied Sciences*, 12(2):705, 2022.

[73] M Kachelrieß, O Kalashev, S Ostapchenko, and DV Semikoz. Minimal model for extragalactic cosmic rays and neutrinos. *Physical Review D*, 96(8):083006, 2017.

[74] Sébastien Bourdarie and Michael Xapsos. The near-earth space radiation environment. *IEEE transactions on nuclear science*, 55(4):1810–1832, 2008.

[75] Isabelle Lange and Scott E Forbush. Further note on the effect on cosmic-ray intensity of the magnetic storm of march 1, 1942. *Terrestrial Magnetism and Atmospheric Electricity*, 47(4):331–334, 1942.

[76] Silvia Mollerach and Esteban Roulet. Progress in high-energy cosmic ray physics. *Progress in Particle and Nuclear Physics*, 98:85–118, 2018.

[77] Clive Dyer and David Rodgers. Effects on spacecraft & aircraft electronics. In *Proceedings ESA Workshop on Space Weather, ESA WPP-155*, pages 17–27, 1998.

[78] Daniel C Wilkinson, Stuart C Daughtridge, John L Stone, Herbert H Sauer, and Phil Darling. Tdrs-1 single event upsets and the effect of the space environment. *IEEE Transactions on Nuclear Science*, 38(6):1708–1712, 1991.

[79] L Adams, EJ Daly, R Harboe-Sorensen, R Nickson, J Haines, W Schafer, M Conrad, H Griech, J Merkel, T Schwall, et al. A verified proton induced latch-up in space (cmos sram). *IEEE Transactions on Nuclear Science*, 39(6):1804–1808, 1992.

[80] DT Bartlett. Radiation protection aspects of the cosmic radiation exposure of aircraft crew. *Radiation protection dosimetry*, 109(4):349–355, 2004.

[81] Erich Regener and Georg Pfotzer. Vertical intensity of cosmic rays by threefold coincidences in the stratosphere. *Nature*, 136(3444):718–719, 1935.

[82] Timothy J O'Gorman, John M Ross, Allen H Taber, James F Ziegler, Hans P Muhlfeld, Charles J Montrose, Huntington W Curtis, and James L Walsh.

Field testing for cosmic ray soft errors in semiconductor memories. *IBM Journal of Research and Development*, 40(1):41–50, 1996.

[83] James F Ziegler and William A Lanford. Effect of cosmic rays on computer memories. *Science*, 206(4420):776–788, 1979.

[84] Janet L Barth, CS Dyer, and EG Stassinopoulos. Space, atmospheric, and terrestrial radiation environments. *IEEE Transactions on nuclear science*, 50(3):466–482, 2003.

[85] VA Naumov and TS Sinegovskaya. Simple method for solving transport equations describing the propagation of cosmic-ray nucleons in the atmosphere. *Physics of Atomic Nuclei*, 63(11):1927–1935, 2000.

[86] James F Ziegler, Huntington W Curtis, Hans P Muhlfeld, Charles J Montrose, B Chin, Michael Nicewicz, CA Russell, Wen Y Wang, Leo B Freeman, P Hosier, et al. Ibm experiments in soft fails in computer electronics (1978–1994). *IBM journal of research and development*, 40(1):3–18, 1996.

[87] M Caldwell and P D'Antonio. A study of using electronics for nuclear weapon detonation safety. In *34th AIAA/ASME/SAE/ASEE Joint Propulsion Conference and Exhibit*, page 3465, 1998.

[88] Energy Department for Business and Industrial Strategy. UK Energy in Brief 2021. `https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/1032260/UK_Energy_in_Brief_2021.pdf`, 24 January 2022. Accessed: 2022-06-17.

[89] Boris M Bolotovskii. Vavilov–cherenkov radiation: its discovery and application. *Physics-Uspekhi*, 52(11):1099, 2009.

[90] N Ezell, Kyle Reed, and Milton Ericson. Radiation-hard electronics for nuclear instrumentation in terrestrial reactors. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2021.

[91] L. Gonella, F. Faccio, M. Silvestri, S. Gerardin, D. Pantano, V. Re, M. Manghisoni, L. Ratti, and A. Ranieri. Total Ionizing Dose effects in 130-nm commercial CMOS technologies for HEP experiments. *Nuclear Instruments and Methods in Physics Research, Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 582(3):750–754, dec 2007.

[92] Yan Liu, Wei Chen, Chaohui He, Chunlei Su, Chenhui Wang, Xiaoming Jin, Junlin Li, and Yuanyuan Xue. Analysis of displacement damage effects on bipolar transistors irradiated by spallation neutrons. *Chinese Physics B*, 28(6):067302, 2019.

[93] Sangeet Saha, Shoaib Ehsan, Adrian Stoica, Rustam Stolkin, and Klaus McDonald-Maier. Real-Time Application Processing for FPGA-Based Resilient Embedded Systems in Harsh Environments. In *2018 NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2018*, pages 299–304. Institute of Electrical and Electronics Engineers Inc., nov 2018.

[94] Edward Petersen. *Single event effects in aerospace*. John Wiley & Sons, 2011.

[95] Rémi Gaillard. Single event effects: Mechanisms and classification. In *Soft errors in modern electronic systems*, pages 27–54. Springer, 2011.

[96] Paul E Dodd, Marty R Shaneyfelt, James A Felix, and James R Schwank. Production and propagation of single-event transients in high-speed digital logic ics. *IEEE Transactions on Nuclear Science*, 51(6):3278–3284, 2004.

[97] Johan Karlsson, Peter Liden, Peter Dahlgren, Rolf Johansson, and Ulf Gunneflo. Using heavy-ion radiation to validate fault-handling mechanisms. *IEEE micro*, 14(1):8–23, 1994.

[98] Alexey O Akhmetov, Andrey V Yanenko, and Anatoliy I Bazhan. Proton accelerator with adjustable energy for ics radiation test. In *2013 14th Euro-*

pean Conference on Radiation and Its Effects on Components and Systems (RADECS), pages 1–3. IEEE, 2013.

[99] S Buchner, D McMorrow, J Melinger, and AB Camdbell. Laboratory tests for single-event effects. *IEEE Transactions on Nuclear Science*, 43(2):678–686, 1996.

[100] TK Sanderson, D Mapper, JH Stephen, and J Farren. Use of a 252cf source in cosmic ray simulation studies on cmos memories. *Electronics Letters*, 19(10):373–374, 1983.

[101] Simon Platt, ZoltÁn Torok, Chris D Frost, and Stuart Ansell. Charge-collection and single-event upset measurements at the isis neutron source. *IEEE Transactions on Nuclear Science*, 55(4):2126–2132, 2008.

[102] J Baggio, D Lambert, V Ferlet-Cavrois, P Paillet, C Marcandella, and O Duhamel. Single event upsets induced by 1–10 mev neutrons in static-rams using mono-energetic neutron sources. *IEEE Transactions on Nuclear Science*, 54(6):2149–2155, 2007.

[103] JS Melinger, S Buchner, D McMorrow, WJ Stapor, TR Weatherford, AB Campbell, and H Eisen. Critical evaluation of the pulsed laser method for single event effects testing and fundamental studies. *IEEE Transactions on Nuclear Science*, 41(6):2574–2584, 1994.

[104] RI Smith, S Hull, MG Tucker, HY Playford, DJ McPhail, SP Waller, and ST Norberg. The upgraded polaris powder diffractometer at the isis neutron source. *Review of scientific instruments*, 90(11):115101, 2019.

[105] Terry Ma, Victor Moroz, Ricardo Borges, Karim El Sayed, Plamen Asenov, and Asen Asenov. Future perspectives of tcad in the industry. In *2016 International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*, pages 335–339. IEEE, 2016.

[106] Xuebing Cao, Liyi Xiao, Jie Li, Rongsheng Zhang, Shanshan Liu, and Jinxiang Wang. A layout-based soft error vulnerability estimation approach for combinational circuits considering single event multiple transients (semts). *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(6):1109–1122, 2018.

[107] Mojtaba Ebrahimi, Hossein Asadi, Rajendra Bishnoi, and Mehdi B Tahoori. Layout-based modeling and mitigation of multiple event transients. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(3):367–379, 2015.

[108] Jinshun Bi, Bo Li, Zhengsheng Han, Jiajun Luo, Li Chen, and Xuefang Lin-Shi. 3d tcad simulation of single-event-effect in n-channel transistor based on deep sub-micron fully-depleted silicon-on-insulator technology. In *2014 12th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, pages 1–3. IEEE, 2014.

[109] AF Petrie and Charles Hymowitz. A spice model for igbts. In *Proceedings of 1995 IEEE Applied Power Electronics Conference and Exposition-APEC'95*, volume 1, pages 147–152. IEEE, 1995.

[110] Chang Hoon Jeon, Jae Gwang Um, Mallory Mativenga, and Jin Jang. Fast threshold voltage compensation amoled pixel circuit using secondary gate effect of dual gate a-igzo tfts. *IEEE Electron Device Letters*, 37(11):1450–1453, 2016.

[111] Omid Kavehei, Said F Al-Sarawi, and Derek Abbott. An automated approach for evaluating spatial correlation in mixed signal designs using synopsys hspice®. Citeseer, 2009.

[112] Roy W Goody. *OrCAD Pspice for Windows Volume 1: DC and AC Circuits*. Prentice Hall PTR, 2000.

[113] Nisha Yadav, Shireesh Kumar Rai, and Rishikesh Pandey. New grounded and floating memristor emulators using ota and cdba. *International Journal of Circuit Theory and Applications*, 48(7):1154–1179, 2020.

[114] K Castellani-Coulie, Marc Bocquet, Hassen Aziza, Jean Michel Portal, Wenceslas Rahajandraibe, and Christophe Muller. Spice level analysis of single event effects in an oxrram cell. In *2013 14th Latin American Test Workshop-LATW*, pages 1–5. IEEE, 2013.

[115] Neil Rostand, Sebastien Martinie, Joris Lacord, Olivier Rozeau, Olivier Billoint, Jean-Charles Barbé, Thierry Poiroux, and Guillaume Hubert. Compact modelling of single event transient in bulk mosfet for spice: Application to elementary circuit. In *2018 International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*, pages 364–368. IEEE, 2018.

[116] Tze Sin Tan and Bakhtiar Affendi Rosdi. Verilog hdl simulator technology: a survey. *Journal of Electronic Testing*, 30(3):255–269, 2014.

[117] Wassim Mansour, Raoul Velazco, Rafic Ayoubi, Haissam Ziade, and Wassim El Falou. A method and an automated tool to perform set fault-injection on hdl-based designs. In *2013 25th International Conference on Microelectronics (ICM)*, pages 1–4. IEEE, 2013.

[118] Mohammad Shokrolah-Shirazi and Seyed Ghassem Miremadi. Fpga-based fault injection into synthesizable verilog hdl models. In *2008 Second International Conference on Secure System Integration and Reliability Improvement*, pages 143–149. IEEE, 2008.

[119] Stan D Phillips, Kurt A Moen, Laleh Najafizadeh, Ryan M Diestelhorst, Akil Khamsi Sutton, John D Cressler, Gyorgy Vizkelethy, Paul E Dodd, and Paul W Marshall. A comprehensive understanding of the efficacy of n-

ring see hardening methodologies in sige hbts. *IEEE Transactions on Nuclear Science*, 57(6):3400–3406, 2010.

[120] JM Benedetto, PH Eaton, DG Mavis, M Gadlage, and T Turflinger. Variation of digital set pulse widths and the implications for single event hardening of advanced cmos processes. *IEEE Transactions on Nuclear Science*, 52(6):2114–2119, 2005.

[121] M Sonza Reorda, Massimo Violante, Cristina Meinhardt, and Ricardo Reis. A low-cost see mitigation solution for soft-processors embedded in systems on pogrammable chips. In *2009 Design, Automation & Test in Europe Conference & Exhibition*, pages 352–357. IEEE, 2009.

[122] Felix Siegle, Tanya Vladimirova, Jørgen Ilstad, and Omar Emam. New voter design enabling hot redundancy for asynchronous network nodes. In *2014 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 23–30. IEEE, 2014.

[123] Jonathan Johnson, William Howes, Michael Wirthlin, Daniel L McMurtrey, Michael Caffrey, Paul Graham, and Keith Morgan. Using duplication with compare for on-line error detection in fpga-based designs. In *2008 IEEE Aerospace Conference*, pages 1–11. IEEE, 2008.

[124] Byonghyo Shim, Srinivasa R Sridhara, and Naresh R Shanbhag. Reliable low-power digital signal processing via reduced precision redundancy. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(5):497–510, 2004.

[125] Kuang-Hua Huang and Jacob A Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers*, 100(6):518–528, 1984.

[126] Felix Siegle, Tanya Vladimirova, Jørgen Ilstad, and Omar Emam. Mitiga-

tion of radiation effects in sram-based fpgas for space applications. *ACM Computing Surveys (CSUR)*, 47(2):1–34, 2015.

[127] Fernanda Lima, C Carmichael, J Fabula, R Padovani, and Ricardo Reis. A fault injection analysis of virtex fpga tmr design methodology. In *RADECS 2001. 2001 6th European Conference on Radiation and Its Effects on Components and Systems (Cat. No. 01TH8605)*, pages 275–282. IEEE, 2001.

[128] MJ Wirthlin, Nathan Rollins, PS Graham, and MP Caffrey. Hardness by design technique for field programmable gate arrays. Technical report, Citeseer, 2003.

[129] Keith S Morgan, Daniel L McMurtrey, Brian H Pratt, and Michael J Wirthlin. A comparison of tmr with alternative fault-tolerant design techniques for fpgas. *IEEE transactions on nuclear science*, 54(6):2065–2072, 2007.

[130] Manoj Franklin and Kewal K Saluja. Pattern sensitive fault testing of rams with built-in ecc. In *Digest of Papers. Fault-Tolerant Computing: The Twenty-First International Symposium*, pages 385–386. IEEE Computer Society, 1991.

[131] J. Yamada, T. Mano, J. Inoue, S. Nakajima, and T. Matsuda. A submicron 1 Mbit dynamic RAM with a 4-bit-at-a-time built-in ECC circuit. *IEEE Journal of Solid-State Circuits*, 19(5):627–633, oct 1984.

[132] T. Yamada, H. Kotani, J. Matsushima, and M. Inoue. A 4-Mbit DRAM with 16-bit concurrent ECC. *IEEE Journal of Solid-State Circuits*, 23(1):20–26, feb 1988.

[133] J. Yamada. Selector-line merged built-in ECC technique for DRAMs. *IEEE Journal of Solid-State Circuits*, 22(5):868–873, oct 1987.

[134] Juhyung Hong, Jeongbin Kim, Sangwoo Han, and Eui-Young Chung. A locality-aware compression scheme for highly reliable embedded systems.

IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 38(3):453–465, 2018.

[135] Panagiota Papavramidou and Michael Nicolaidis. Iterative diagnosis approach for ecc-based memory repair. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[136] M. Asakura, Y. Matsuda, H. Hidaka, Y. Tanaka, and K. Fujishima. An experimental 1-Mbit cache DRAM with ECC. *IEEE Journal of Solid-State Circuits*, 25(1):5–10, 1990.

[137] M. Y. Hsiao. A Class of Optimal Minimum Odd-weight-column SEC-DED Codes. *IBM Journal of Research and Development*, 14(4):395–401, jul 1970.

[138] Umberto Martínez-Peñas and Frank R Kschischang. Reliable and secure multishot network coding using linearized reed-solomon codes. *IEEE Transactions on Information Theory*, 2019.

[139] Matthew Walters and Sujoy Sinha Roy. Constant-time bch error-correcting code. *IACR Cryptology ePrint Archive*, 2019:155, 2019.

[140] Jen-Wei Hsieh, Chung-Wei Chen, and Han-Yi Lin. Adaptive ecc scheme for hybrid ssd's. *IEEE Transactions on Computers*, 64(12):3348–3361, 2015.

[141] Felix Siegle, Tanya Vladimirova, Jorgen Ilstad, and Omar Emam. Availability analysis for satellite data processing systems based on sram fpgas. *IEEE Transactions on Aerospace and Electronic Systems*, 52(3):977–989, 2016.

[142] Amr M.S. Tosson, Mohab Anis, and Lan Wei. RRAM Refresh Circuit. In *Proceedings of the 26th edition on Great Lakes Symposium on VLSI - GLSVLSI '16*, pages 227–232, New York, New York, USA, 2016. ACM Press.

[143] Luca Sterpone and Massimo Violante. A new partial reconfiguration-based fault-injection system to evaluate seu effects in sram-based fpgas. *IEEE Transactions on Nuclear Science*, 54(4):965–970, 2007.

[144] Andrew M Keller and Michael J Wirthlin. Benefits of complementary seu mitigation for the leon3 soft processor on sram-based fpgas. *IEEE Transactions on Nuclear Science*, 64(1):519–528, 2016.

[145] A Moopenn and AP Thakoor. Programmable synaptic devices for electronic neural nets. *Control and Computers*, 18(2):37–41, 1990.

[146] Yann LeCun. 1.1 deep learning hardware: Past, present, and future. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 12–19. IEEE, 2019.

[147] Ahmad Shawahna, Sadiq M. Sait, and Aiman El-Maleh. Fpga-based accelerators of deep learning networks for learning and classification: A review. *IEEE Access*, 7:7823–7859, 2019.

[148] Xiaowei Xu, Yukun Ding, Sharon Xiaobo Hu, Michael Niemier, Jason Cong, Yu Hu, and Yiyu Shi. Scaling for edge inference of deep neural networks. *Nature Electronics*, 1(4):216–222, 2018.

[149] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. Automated systolic array architecture synthesis for high throughput cnn inference on fpgas. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[150] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

[151] Qingxia Zhang, Zihao Meng, Xianwen Hong, Yuhao Zhan, Jia Liu, Jiabao Dong, Tian Bai, Junyu Niu, and M Jamal Deen. A survey on data center cooling systems: Technology, power consumption modeling and control strategy optimization. *Journal of Systems Architecture*, 119:102253, 2021.

[152] Carlo Puliafito, Enzo Mingozzi, Francesco Longo, Antonio Puliafito, and Omer Rana. Fog computing for the internet of things: A survey. *ACM Transactions on Internet Technology (TOIT)*, 19(2):1–41, 2019.

[153] Manar Abu Talib, Sohaib Majzoub, Qassim Nasir, and Dina Jamal. A systematic literature review on hardware implementation of artificial intelligence algorithms. *The Journal of Supercomputing*, 77(2):1897–1938, 2021.

[154] Jeremy Hsu. Ibm's new brain [news]. *IEEE spectrum*, 51(10):17–19, 2014.

[155] Kizheppatt Vipin and Suhaib A Fahmy. Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Computing Surveys (CSUR)*, 51(4):1–39, 2018.

[156] Tobias Becker, Wayne Luk, and Peter YK Cheung. Enhancing relocatability of partial bitstreams for run-time reconfiguration. In *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, pages 35–44. IEEE, 2007.

[157] Jean-Philippe Delahaye, Jacques Palicot, Christophe Moy, and Pierre Leray. Partial reconfiguration of fpgas for dynamical reconfiguration of a software radio platform. In *2007 16th IST Mobile and Wireless Communications Summit*, pages 1–5. IEEE, 2007.

[158] Julien Delorme, Jérôme Martin, Amor Nafkha, Christophe Moy, Fabien Clermidy, Pierre Leray, and Jacques Palicot. A fpga partial reconfiguration design approach for cognitive radio based on noc architecture. In *2008 Joint 6th International IEEE Northeast Workshop on Circuits and Systems and TAISA Conference*, pages 355–358. IEEE, 2008.

[159] Hanaa M Hussain, Khaled Benkrid, Ali Ebrahim, Ahmet T Erdogan, and Huseyin Seker. Novel dynamic partial reconfiguration implementation of k-means clustering on fpgas: Comparative results with gpps and gpus. *International Journal of Reconfigurable Computing*, 2012, 2012.

[160] Hanaa Hussain, Khaled Benkrid, and HÜSEYİN ŞEKER. Novel dynamic partial reconfiguration implementations of the support vector machine classifier on fpga. *Turkish Journal of Electrical Engineering & Computer Sciences*, 24(5):3371–3387, 2016.

[161] Christopher Claus, Florian H Muller, Johannes Zeppenfeld, and Walter Stechele. A new framework to accelerate virtex-ii pro dynamic partial self-reconfiguration. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–7. IEEE, 2007.

[162] Björn Osterloh, Harald Michalik, Sandi Alexander Habinc, and Björn Fiethe. Dynamic partial reconfiguration in space applications. In *2009 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 336–343. IEEE, 2009.

[163] Cristiana Bolchini, Antonio Miele, and Marco D Santambrogio. Tmr and partial dynamic reconfiguration to mitigate seu faults in fpgas. In *22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007)*, pages 87–95. IEEE, 2007.

[164] Jonathan Heiner, Benjamin Sellers, Michael Wirthlin, and Jeff Kalb. Fpga partial reconfiguration via configuration scrubbing. In *2009 International Conference on Field Programmable Logic and Applications*, pages 99–104. IEEE, 2009.

[165] Shanker Shreejith, Suhaib A Fahmy, and Martin Lukasiewycz. Reconfigurable computing in next-generation automotive networks. *IEEE embedded systems letters*, 5(1):12–15, 2013.

[166] Reza Taghipour, Tristan Perez, and Torgeir Moan. Hybrid frequency–time domain models for dynamic response analysis of marine structures. *Ocean Engineering*, 35(7):685–705, 2008.

[167] Rakan Khraisha and Jooheung Lee. A scalable h. 264/avc deblocking filter architecture using dynamic partial reconfiguration. In *2010 IEEE In-*

ternational Conference on Acoustics, Speech and Signal Processing, pages 1566–1569. IEEE, 2010.

[168] Manish Birla and Krishna N Vikram. Partial run-time reconfiguration of fpga for computer vision applications. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–6. IEEE, 2008.

[169] Esam El-Araby, Ivan Gonzalez, and Tarek El-Ghazawi. Performance bounds of partial run-time reconfiguration in high-performance reconfigurable computing. In *Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications: held in conjunction with SC07*, pages 11–20, 2007.

[170] Neil Joseph Steiner. *Autonomous computing systems*. PhD thesis, Virginia Tech, 2008.

[171] Xabier Iturbe, Khaled Benkrid, Chuan Hong, Ali Ebrahim, Raul Torrego, Imanol Martinez, Tughrul Arslan, and Jon Perez. R3tos: a novel reliable reconfigurable real-time operating system for highly adaptive, efficient, and dependable computing on fpgas. *IEEE Transactions on computers*, 62(8):1542–1556, 2013.

[172] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 109–116. IEEE, 2014.

[173] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24. IEEE, 2014.

[174] Suhaib A Fahmy, Kizheppatt Vipin, and Shanker Shreejith. Virtualized fpga accelerators for efficient cloud computing. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 430–435. IEEE, 2015.

[175] Christoforos Kachris and Dimitrios Soudris. A survey on reconfigurable accelerators for cloud computing. In *2016 26th International conference on field programmable logic and applications (FPL)*, pages 1–10. IEEE, 2016.

[176] Jim Torresen, Geir Aarstad Senland, and Kyrre Glette. Partial reconfiguration applied in an on-line evolvable pattern recognition system. In *2008 NORCHIP*, pages 61–64. IEEE, 2008.

[177] Yufan Lu, Xin Chen, Xiaojun Zhai, Sangeet Saha, Shoaib Ehsan, Jinya Su, and Klaus McDonald-Maier. A fast simulation method for analysis of see in vlsi. *Microelectronics Reliability*, 120:114110, 2021.

[178] Yufan Lu, Xin Chen, Xiaojun Zhai, Sangeet Saha, Shoaib Ehsan, Jinya Su, and Klaus D McDonald-Maier. A simulation and evaluation scheme for single event effects in vlsi. 2021.

[179] Semiconductor Manufacturing International Corporation (SMIC). Smic foundry solutions 90nm,130/110nm,150nm,180nm,250nm,350nm. `https://www.smics.com/en/site/mature_logic`, 2022. Accessed: June 4, 2022.

[180] MD Shazzad Hossain and Ioannis Savidis. Reusing leakage current for improved energy efficiency of multi-voltage systems. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2019.

[181] V Ferlet-Cavrois, P Paillet, D McMorrow, N Fel, J Baggio, S Girard, O Duhamel, JS Melinger, M Gaillardin, JR Schwank, et al. New insights into single event transient propagation in chains of inverters—evidence for propagation-induced pulse broadening. *IEEE Transactions on Nuclear Science*, 54(6):2338–2346, 2007.

[182] Gilson Wirth, Fernanda L Kastensmidt, and Ivandro Ribeiro. Single event transients in logic circuits—load and propagation induced pulse broadening. *IEEE Transactions on Nuclear Science*, 55(6):2928–2935, 2008.

[183] M Karimian, M Dousti, M Pouyan, and R Faez. An improved macro-model for simulation of single electron transistor (set) using hspice. In *2009 IEEE Toronto International Conference Science and Technology for Humanity (TIC-STH)*, pages 1000–1004. IEEE, 2009.

[184] Scott Davidson. Itc'99 benchmark circuits-preliminary results. In *International Test Conference 1999. Proceedings (IEEE Cat. No. 99CH37034)*, pages 1125–1125. IEEE, 1999.

[185] Alireza Kasnavi, Joddy W Wang, Mahmoud Shahram, and Jindrich Zejda. Analytical modeling of crosstalk noise waveforms using weibull function. In *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, pages 141–146. IEEE, 2004.

[186] LR Rockett. An seu-hardened cmos data latch design. *IEEE Transactions on Nuclear Science*, 35(6):1682–1687, 1988.

[187] Leo B Freeman. Critical charge calculations for a bipolar sram array. *IBM Journal of Research and Development*, 40(1):119–129, 1996.

[188] Quming Zhou and Kartik Mohanram. Cost-effective radiation hardening technique for combinational logic. In *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, pages 100–106. IEEE, 2004.

[189] Luca Sterpone and Massimo Violante. Analysis of the robustness of the tmr architecture in sram-based fpgas. *IEEE Transactions on Nuclear Science*, 52(5):1545–1549, 2005.

[190] Roystein Oliveira, Aditya Jagirdar, and Tapan J Chakraborty. A tmr scheme

for seu mitigation in scan flip-flops. In *8th International Symposium on Quality Electronic Design (ISQED'07)*, pages 905–910. IEEE, 2007.

[191] Lorena Anghel, Dan Alexandrescu, and Michael Nicolaidis. Evaluation of a soft error tolerance technique based on time and/or space redundancy. In *Proceedings 13th Symposium on Integrated Circuits and Systems Design (Cat. No. PR00843)*, pages 237–242. IEEE, 2000.

[192] Yufan Lu, Xiaojun Zhai, Sangeet Saha, Shoaib Ehsan, and Klaus McDonald-Maier. A self-scrubbing scheme for embedded systems in radiation environments. In *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–4. IEEE, 2020.

[193] Xilinx. Space-grade virtex-5qv fpga. `https://www.xilinx.com/products/silicon-devices/fpga/virtex-5qv.html`, 2018. Accessed: June 15, 2022.

[194] Xilinx. Ds197 artix-7 fpgas data sheet: Overview (v1.3). `https://www.xilinx.com/support/documentation/data_sheets/ds197-xa-artix7-overview.pdf`, 2017. Accessed: June 15, 2022.

[195] Bilal Aslam, Sangeet Saha, et al. Degradation Measurement of Commercial Camera Sensors Under Fast Neutron Beamline. In *SEE MAPLD, In press.*, 2019.

[196] Zeba Khanam, Sangeet Saha, et al. Degradation Measurement of Kinect Sensor Under Fast Neutron Beamline. In *Radiation Effect Data Workshop, REDW, In press.*, jul 2019.

[197] L Obermueller, C Cazzaniga, S Kulmiya, and CD Frost. A fast neutron monitor based on single event effects in srams using commercial off-the-shelf components. In *2018 18th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, pages 1–5. IEEE, 2018.

[198] Carlo Cazzaniga and Christopher D. Frost. Progress of the Scientific Commissioning of a fast neutron beamline for Chip Irradiation. In *Journal of Physics: Conference Series*, 2018.

[199] Xilinx. Xilinx 7 series fpgas memory resources. `https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf`, 2019. Accessed: June 15, 2022.

[200] G Tsiligiannis, S Danzeca, R García Alía, A Infantino, A Lesea, M Brugger, A Masi, S Gilardoni, and F Saigné. Radiation effects on deep submicrometer sram-based fpgas under the cern mixed-field radiation environment. *IEEE Transactions on Nuclear Science*, 65(8):1511–1518, 2018.

[201] Juan Carlos Fabero, Hortensia Mecha, Francisco J Franco, Juan Antonio Clemente, Golnaz Korkian, Solenne Rey, Benjamin Cheymol, Maud Baylac, Guillaume Hubert, and Raoul Velazco. Single event upsets under 14-mev neutrons in a 28-nm sram-based fpga in static mode. *IEEE Transactions on Nuclear Science*, 67(7):1461–1469, 2020.

[202] Yu Nakazawa, Yuki Fujii, Eitaro Hamada, MyeongJae Lee, Yuta Miyazaki, Akira Sato, Kazuki Ueno, Hisataka Yoshida, and Jie Zhang. Radiation study of fpgas with neutron beam for comet phase-i. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 936:351–352, 2019.

[203] Mahesh Kumar, Durga Digdarsini, Neeraj Misra, and TVS Ram. Seu mitigation of rad-tolerant xilinx fpga using external scrubbing for geostationary mission. In *2016 IEEE Annual India Conference (INDICON)*, pages 1–6. IEEE, 2016.

[204] Corey Baker, Rick L Lawrence, Clifford Montagne, and Duncan Patten. Change detection of wetland ecosystems using landsat imagery and change vector analysis. *Wetlands*, 27(3):610–619, 2007.

[205] Wang Lie and Wu Feng-Yan. Dynamic partial reconfiguration in fpgas. In *2009 Third International Symposium on Intelligent Information Technology Application*, volume 2, pages 445–448. IEEE, 2009.

[206] Xilinx. Vivado design suite user guide dynamic function exchange (v2020.2). `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug909-vivado-partial-reconfiguration.pdf`, 2020. Accessed: June 4, 2022.

[207] Yufan Lu, Xiaojun Zhai, Sangeet Saha, Shoaib Ehsan, and Klaus D McDonald-Maier. Fpga based adaptive hardware acceleration for multiple deep learning tasks. In *2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 204–209. IEEE, 2021.

[208] Xilinx. Vitis-ai 1.4 release. `https://github.com/Xilinx/Vitis-AI`, 2021. Accessed: June 15, 2022.

[209] Yufan Lu, Cong Gao, Rappy Saha, Sangeet Saha, Klaus D McDonald-Maier, and Xiaojun Zhai. Fpga-based dynamic deep learning acceleration for real-time video analytics. In *Architecture of Computing Systems: 35th International Conference, ARCS 2022, Heilbronn, Germany, September 13–15, 2022, Proceedings*, pages 68–82. Springer, 2022.

[210] Vinod Kathail. Xilinx vitis unified software platform. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 173–174, 2020.

[211] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, Los Alamitos, CA, USA, jun 2016. IEEE Computer Society.

[212] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[213] Xilinx. H.264/H.265 Video Codec Unit v1.2. `https://www.xilinx.com/support/documentation/ip_documentation/vcu/v1_2/pg252-vcu.pdf`, 2021. Accessed: June 15, 2022.

[214] Xilinx. DPUCZDX8G for Zynq UltraScale+ MPSoCs Product Guide. `https://docs.xilinx.com/r/en-US/pg338-dpu/`, 2022. Accessed: June 15, 2022.

[215] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[216] Keren Fu, Qijun Zhao, and Irene Yu-Hua Gu. Refinet: A deep segmentation assisted refinement network for salient object detection. *IEEE Transactions on Multimedia*, 21(2):457–469, 2018.

[217] Xilinx. Vitis Video Analytics SDK (VVAS) plug-ins. `https://xilinx.github.io/VVAS/main/build/html/docs/common/common_plugins.html`, 2022. Accessed: June 15, 2022.