Performance-aware Design-space Optimization and Attack Mitigation for Emerging Heterogeneous Architectures

A THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF

Doctor of Philosophy

BY

MITALI SINHA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDRAPRASTHA INSTITUTE OF INFORMATION TECHNOLOGY DELHI

NEW DELHI– 110020

APRIL, 2022

Acknowledgements

I wish to express sincere gratitude to my advisor, Dr. Sujay Deb for providing me with the opportunity to conduct effective research work. He is a visionary leader and is excellent at presenting and explaining research work in a rational form. He gave me enough freedom in choosing my research domain and heard me patiently while providing his guidance through all these years. In addition to the technical inputs and crucial research suggestions, he was always present to boost my morale and support me emotionally as well. I also extend my gratitude to Dr. Suman Deb (National Institute of Technology Agartala) and Dr. Jhunu Debbarma (Tripura Institute of Technology) for their valuable guidance during my under-graduation and post-graduation programs. Their kind words and belief have encouraged me to pursue the path of a researcher and achieve greater heights in my career.

I am grateful to have a loving family who stood beside me in every important decision in my life. I thank my mother, whose constant love and support have given me the confidence to push my limits and taught me to be kind to others. I thank my brother for his presence and support. I am grateful to have a caring husband who constantly boosts my energy and gives me a fresh perspective on any problem or challenge at hand. I sincerely thank my in-laws, who took care of me like their daughter and always supported me to successfully complete my final years during the PhD program.

I would also like to thank all the people I interacted with at IIITD. I had amazing friends and colleagues who helped me through all these years. I wish to thank Hemanta Mondal, Wazir Singh, Harsha Gade, Sidhartha Rout, Sneha Agarwal, Deepank Grover, and Tarun Sharma, for their time in several research discussions and for keeping up a positive environment in the lab. I am grateful to have great friends like Mansi, Niharika, Nidhi, Puspita, Neha, and Chitrita, who were present to cheer me up during difficult times and share my happy moments. I would also like to thank the undergraduate and graduate students who have worked with me on different projects.

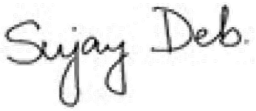
Mitali Sinha

Mitali Sinha (PhD17001)

Certificate

This is to certify that the thesis titled "Performance-aware Design-space Optimization and Attack Mitigation for Emerging Heterogeneous Architectures" being submitted by Mitali Sinha to INDRAPRASTHA INSTITUTE OF INFORMATION TECHNOLOGY DELHI, for the award of the DOCTOR OF PHILOSOPHY, is an original research work carried out under my supervision. In my opinion, the thesis has reached the standards fulfilling the requirements of the regulations relating to the degree.

The results contained in this thesis have not been submitted in part or full to any other university or institute for the award of any degree/diploma.



Dr. Sujay Deb

DEPARTMENT OF ELECTRONICS AND COMMUNICATIONS ENGINEERING

INDRAPRASTHA INSTITUTE OF INFORMATION TECHNOLOGY DELHI

Abstract

In recent years, as multi-core processors are emerging as a solution to the limitations of power scaling, the software domain is also experiencing a surge of compute and data-intensive applications. These applications are often targeted for real-time processing systems which demand high performance and energy efficiency. To meet these needs of the real-time processing architectures, the embedded systems are moving toward heterogeneous System-on-Chip (SoC) design. Alongside the general-purpose cores, the heterogeneous SoCs include application-specific customized designs, commonly known as hardware accelerators, which accelerate the performance of specific functions and provide energy efficiency. All the processing elements share the on-chip resources like memory etc., and communicate with each other through a shared Network-on-chip (NoC).

The overwhelming performance improvement achieved from hardware acceleration has spurred a growing number of fixed-function accelerators in modern heterogeneous SoCs. The growing system sizes and time-to-market pressure of accelerator-rich heterogeneous SoCs compel the chip designers to analyze only part of the design space, leading to suboptimal Intellectual Property (IP) designs. Hence, the accelerators are generally designed as standalone IP blocks by third-party vendors and chip designers often over-provision the amount of on-chip resources required to add flexibility to each accelerator design. Although this modularity simplifies IP design, integrating these off-the-shelf accelerator blocks into a single SoC may overshoot the resource budget of the underlying system. Furthermore, the integration of third-party accelerator IPs alongside other on-chip modules makes the system vulnerable to security threats. This is due to the lack of design details from third-party vendors, which makes it difficult to verify the design. Even in the presence of the design, it becomes infeasible to carry out exhaustive simulations and find any malicious logic or hardware trojan.

It is promising to target accelerator memory subsystem for on-chip resource optimization as memory is a significant part of an accelerator design. A straightforward approach will be to allow small chunks of private memories for each accelerator core, so that the budget constraints are met. But, this approach might lead to excessive off-chip memory accesses as the data-intensive accelerators try to bring their respective data on chip.

While sharing the accelerators' on-chip memory is a viable solution for meeting the budget constraints and reducing off-chip memory accesses, it also poses a few challenges as follows. Excessive sharing of accelerator on-chip memory would result in less space for accelerator private data, resulting in more off-chip memory accesses, and, increased NoC contention due to remote shared memory accesses. Another major challenge imposed by the integration of third-party accelerator is the threat of mounting an attack on the on-chip resources and degrading the overall application performance. One such attack is a flooding attack, where a malicious IP injects frequent useless packets into the network to create congestion and block legitimate communication, resulting in a Denial-of-Service (DoS) attack. The distributed nature of NoCs and the dynamic behaviour of the accelerator-rich heterogeneous systems make it difficult to detect and localize such flooding attacks.

We address the challenges involved in designing efficient accelerator-rich SoCs by optimizing the utilization of on-chip resources and mitigating the aforementioned performance-based security threats. We propose a design exploration framework to provide an optimal memory configuration for multi-accelerator systems to maintain overall system performance under a given resource budget constraint. Our framework formulates a model for the optimization problem that captures an application's memory access patterns and provides the best-suited memory system configurations, taking on-chip network contention into consideration. It also employs an efficient method for shared data allocation across the distributed shared memory banks and reduces communication overhead. To minimize the performance degradation from flooding-based DoS attack, we propose an attack detection framework, which employs state-of-the-art machine learning algorithms to study the communication behaviour and accurately raise a flag in case of a flooding attack. An attack localization framework, called Sniffer, is also proposed which is able to localize one or multiple malicious IPs creating the DoS attack. Sniffer employs a perceptron-based and collaborative approach in tracing back the attack path and find out all the attacker nodes. Finally, we also study the design challenges in developing convolution neural network (CNN) accelerators that are widely integrated in heterogeneous SoCs for a range of application domains like image processing, speech recognition, search engines etc. The large number of input parameters and weights involved in CNNs impose a major challenge in data movement between the processing elements of a CNN accelerator, and on-chip and/or off-chip memory. We address this communication bottlenecks by extensively studying the application behaviour and propose an efficient accelerator architecture with fused wired and wireless interconnection network along with a data-flow scheduling algorithm.

List of publications

Book Chapters

1. Mitali Sinha, Sidhartha Sankar Rout, Sujay Deb, "DoS Attack Models and Mitigation Frameworks for NoC-based SoCs", Frontiers of Electronic Design, ISQEDFED, Publisher: Springer, 2022. (accepted)
2. Sidhartha Sankar Rout, Mitali Sinha, Sujay Deb, "NoC Post-Silicon Validation and Debug", Network-on-Chip Security and Privacy, Prabhat Mishra and Subodha Charles (Editors), Publisher: Springer Nature, p.339, 2021.

Journals

1. Mitali Sinha, Pramit Bhattacharyya, Sidharth Sankar Rout, Neha Prakriya, Sujay Deb, "Securing an Accelerator-rich System from Flooding-based Denial-of-Service Attacks", IEEE Transactions on Emerging Topics in Computing (TETC), Jan 8, 2021.
2. Mitali Sinha, Sri Harsha Gade, Pramit Bhattacharyya, Sujay Deb, "Design Space Optimization of Shared Memory Architecture in Accelerator-rich Systems", ACM Transactions on Design Automation of Electronic Systems (TODAES), 26(4), 1-31, 2021.
3. Mitali Sinha, Setu Gupta, Sidharth Sankar Rout, Sujay Deb, "Sniffer: A Machine Learning Approach for DoS Attack Localization in NoC-based SoCs", IEEE Journal on Emerging and Selected Topics in Circuits and Systems (JETCAS), May 24, 2021.
4. Sidharth Sankar Rout, M. Badri, Mitali Sinha, Sujay Deb, "ReDeSIGN: Reuse of Debug Structures for Improvement in Performance Gain of NoC based MPSoCs". (under revision)
5. Mitali Sinha, Shubham Roy, Sujay Deb, "Packet tampering-based HT localization in NoC-based SoC". (under preparation)
6. Sidharth Sankar Rout, Mitali Sinha, Sujay Deb, "A Sustainable and Efficient Channel Access Mechanism for Future Wireless NoCs". (under revision)

Conferences

1. Sri Harsha Gade, Mitali Sinha, Madhur Kumar, Sujay Deb, "Scalable Hybrid Cache Coherence Using Emerging Links for Chiplet Architectures", 35th International Conference on VLSI Design and 21st International Conference on Embedded Systems (VLSID), 2022.

2. Sidhartha Sankar Rout, Akshat Singh, Suyog Bhimrao Patil, Mitali Sinha, Sujay Deb, "Security Threats in Channel Access Mechanism of Wireless NoC and Efficient Countermeasures", IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1-5, 2020.

3. Mitali Sinha, Sri Harsha Gade, Wazir Singh, Sujay Deb , "Data-flow Aware CNN Accelerator with Hybrid Wireless Interconnection", IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 1-4, 2018.

4. Mitali Sinha, Sidhartha Sankar Rout, Sri Harsha Gade, Sujay Deb, "Near Threshold Last Level Cache for Energy Efficient Embedded Applications", The Ninth International Green and Sustainable Computing Conference (IGSC), pp. 1-6, 2018.

5. Sri Harsha Gade, Sidhartha Sankar Rout, Mitali Sinha, Hemanta Kumar Mondal, Wazir Singh, Sujay Deb, "A utilization aware robust channel access mechanism for wireless nocs", IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1-5, 2018.

6. Sri Harsha Gade, Mitali Sinha, Sidhartha Sankar Rout, Sujay Deb, "Enabling Reliable High Throughput On-Chip Wireless Communication for Many Core Architectures, IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pp. 591-596, 2018.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

To leverage the benefits from advanced CMOS technologies and address the limitations of power scaling, the chip multi-processors are shifting towards multi-core and many-core processor architectures. The presence of multiple cores provides significant performance improvement by allowing multiple application tasks to run concurrently on different cores. However, as the computing industry grows, the demand for high performance and energy efficiency increases to meet the need for real-time processing systems. These processing systems are employed as full-fledged embedded solutions in various domains like traffic monitoring, national security, etc. Furthermore, with the advent of the Internet of Things (IoT), they have also found their application in edge devices like wearable health monitoring systems, autonomous vehicles etc. As a result, the computing industries have adapted heterogeneous system architectures to address the ever-increasing performance and energy demands of these wide ranges of application domains.

1.1 Heterogeneous System-on-Chip

The heterogeneous system-on-chips (SoCs) integrate different types of processing cores like general-purpose cores (e.g. x86, ARM), Graphics Processing Units (GPUs), application-specific hardware accelerators, etc. in a single system. While GPUs provide better performance for a domain of applications, the hardware accelerators provide 100$\times$ improvement in performance and energy efficiency as compared to the general-purpose cores. The hardware accelerators achieve such performance improvement through custom-designed architectures that target specific functions ranging from accelerating simple tasks or basic blocks (e.g., multiply and accumulate) to moderate application kernels (e.g., Fast Fourier Transforms) to complex and compute-intensive tasks (e.g., encoding/decoding, machine learning algorithms). Hence, in a heterogeneous SoC, different processing cores work together concurrently on different application kernels to improve the overall system performance. All

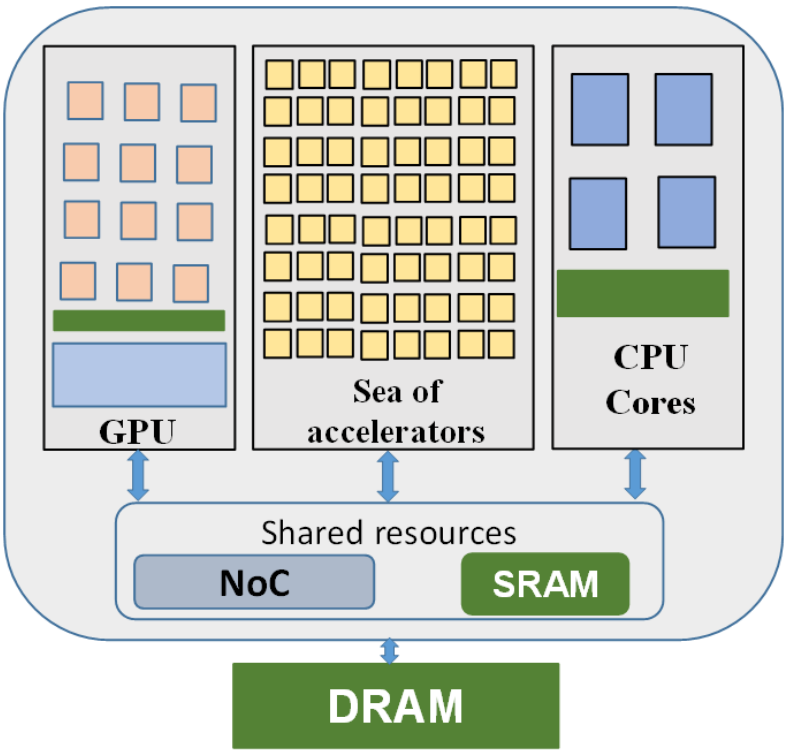


Figure 1.1: Heterogeneous system-on-chip architecture.

the computing cores uses shared on-chip resources such as, memory and interconnection frameworks to communicate with each other during execution of different applications. Figure 1.1 shows a typical heterogeneous SoC with different processing cores and memory modules connected by an on-chip network. Examples of such heterogeneous systems include Apple SoCs, which dedicate more than half of the chip area to accelerators [1]. Other examples include Myriad 2 vision processing unit [2] that incorporate more than 20 accelerators to enhance vision processing and TI OMAP5430 [3] that comprises cores for general computations, digital signal processing, video processing and graphics processing. According to an ITRS prediction [4], more and more accelerators will be employed, resulting in a plethora of different on-chip heterogeneous cores in the future SoCs.

1.2 Challenges in Heterogeneous SoCs

The increased performance and energy efficiency provided by the heterogeneous mix of processing cores, while becoming widespread, still poses multiple challenges for developing a pervasive and robust design. These design challenges are present across various abstraction layers, starting from the high-level application layer for efficient task mapping to the lower circuit level for low power integration. Along with the principal design choices, the inherent characteristics of resource sharing in the heterogeneous SoCs may also give rise to newer

challenges from the security and safety perspective. We present a list of potential challenges in designing the heterogeneous SoCs as follows.

- Application layer: Unlike the homogeneous architectures, the heterogeneous SoCs comprise of multiple choices of mapping application tasks across different available computing cores. It becomes more challenging during the application execution, when multiple tasks are arriving dynamically and competing for the same type of resource. Hence, it is very difficult for the application programmer to perform optimal task scheduling on a heterogeneous system to achieve the utmost performance.
- Instruction Set Architecture (ISA): In a heterogeneous environment, different computing architectures may have different ISAs. Each processing element interprets the instruction and memory according to their ISAs. As a result, it is extremely important to consider the compatibility issues to allow seamless interaction of application tasks across different heterogeneous architectures.
- Lower level design: At the lower circuit level abstraction, the computing architectures can be implemented in different ways that may lead to varied performance and power profiles. In a heterogeneous system, it becomes even more challenging to make a trade-off between the micro-architectural and circuit-level designs, and achieve the desired power and performance goals.
- Memory subsystem: Each computing architecture in a heterogeneous platform have varying requirement of memory interactions. While a computing element may need more processing power and minimal memory interaction, others might be memory-intensive and need to frequently access the memory. As a result, one has to take a holistic decision while designing the memory subsystem for a heterogeneous SoC that will cater to the requirements of all the computing elements.
- On-chip interconnection: The shared interconnection network in a heterogeneous SoC poses another major challenge in handling the diverse traffic requirements of various computing architectures. While a simple interconnection structure like a bus-based system would be sufficient for some architectures, others might require faster and larger bandwidth for frequent long-distance on-chip communication.
- Security threats from third-party IPs: The integration of various third-party IP blocks within a single SoC makes the system vulnerable to various security threats. Due to the infeasibility of extensively verifying all the third-party IPs, a malicious IP block can mount an attack on the system and degrade the overall application performance.

In our work, we primarily target two of the most important challenges, namely, memory subsystem optimization and security aspects, to improve the performance of the applications running on the heterogeneous

SoCs. Here, we mostly use accelerator-rich systems to demonstrate these challenges in heterogeneous SoCs and our proposed solutions. However, most of the challenges are prevalent in any heterogeneous system architecture. We discuss these challenges in detail in the following sections.

1.2.1 Memory System Optimization Challenges in Heterogeneous SoCs

A naive approach to address the memory optimization problem of an accelerator-rich heterogeneous system is to *provide small private memory for each computing core within the given budget*. While such a design approach easily meets the memory constraints and saves overall on-chip area, the smaller private memories often become the performance bottleneck for memory-intensive cores, like hardware accelerators, due to frequent off-chip memory accesses. The ever-increasing number of on-chip accelerators only exacerbates the problem and potentially even result in overall performance loss as the inefficient memory access performance overshadows the gains from various custom-designed cores. The alternative and viable solution is to *share the memory subsystem among the cores* and reduce the overall area and energy consumption. The memory subsystem is designed to provide small to moderate private memory for each core while allowing a pool of shared memory accessed by all the on-chip computing cores. Compared to the former approach, shared memory-based accelerator-rich heterogeneous systems provide the following advantages:

- The private/shared memory system provides sufficient on-chip memory for each core within the given budget to mitigate frequent off-chip memory accesses and maintain performance.
- It eliminates data replication and reduces memory usage for multi-accelerator applications, especially for producer-consumer kind by writing/reading data directly to shared memory.
- Inter-core communication through shared memory further reduces unwanted accesses to off-chip memory and instructions involved to share data between the cores.

However, shared memory subsystem, under inefficient hierarchy design, can lead to performance pitfalls like

- High data access latency over the on-chip network to remote shared memory as the on-chip network size and number of on-chip cores increases.
- Increased contention on the on-chip network and at shared memory as the heterogeneous system is stressed to generate a high volume of memory requests at each memory-intensive accelerator core.

Hence, though shared memory systems provide performance improvement, impetuous memory sharing may also lead to performance degradation due to higher access latency of shared data and contention in the network.

Therefore, an informed decision has to be taken while designing a shared memory system, and an efficient memory hierarchy design outweighs any adverse impacts to achieve maximum performance under the given resource constraints. Estimating the memory configurations earlier in the design phase would save a significant amount of chip area bringing down the cost of freeing up space for integrating more logic elements. However, the time-to-market pressure and tedious simulation process to explore the vast design space of memory optimization, allow chip designers to analyze only part of the design space. Hence, optimizing memory subsystems of accelerator-rich heterogeneous architectures at the design stage is time-consuming and error-prone, which leads to inefficient system designs. Therefore, it is of utmost importance to address these challenges and propose an efficient design-time solution for memory subsystem optimization of an accelerator-rich heterogeneous system.

1.2.2 Security analysis of Heterogeneous SoCs

As the demand for high performance and energy efficiency increases, the number of custom-designed Intellectual Property (IP) blocks are also increasingly integrated into a single heterogeneous SoC. In such systems, Network-on-chip (NoC) has gained traction as an effective interconnection platform due to its ability to provide high bandwidth and low latency communication [5]. While the integration of various heterogeneous cores and efficient interconnect improve performance, the growing system sizes and time-to-market demand for the SoCs do not allow the chip designers to analyze the whole design space, resulting in suboptimal designs. Hence, there is a tendency to integrate customized architectures from the third party vendors. These third-party IPs can bring in vulnerability in the system due to the presence of a malicious intent within their design. As a result, various security threats may arise due to the shared resources among all the IP blocks within the system. For instance, malicious code running on an IP block can attack other on-chip modules through the shared resources. These security threats get even more aggravated with the integration of third-party IPs in the SoC. Firstly, the lack of design details from the third-party vendors makes it hard to validate the IP blocks. Secondly, even with the design details, it becomes infeasible to exhaustively explore the millions of logic elements to find a possible design bug or malicious modifications like Hardware Trojans (HTs) [6].

In this work, we target the security threats generated by a malicious third-party IP that results in application performance degradation. One such attack that hinders the application execution is a flooding-based Denial-of-Service (DoS) attack. A DoS attack is an attack that diminishes a system's ability to provide appropriate services. A Malicious IP (MIP) generates frequent useless packets for one or more Victim IPs (VIPs) through the shared interconnection network. It results in the unavailability of the network for other legitimate on-chip modules and degrades the entire system performance. Due to the injection of a large number of useless

attack packets, the number of benign packets transferred is reduced significantly, which directly translates to application performance degradation. Hence, it is of utmost importance to address the security threats from MIPs and prevent extreme performance degradation of the applications running on the SoCs.

1.3 Research Contributions

The contributions of this thesis are primarily towards performance improvement of emerging heterogeneous SoCs through design-space optimization and performance-based security analysis. Figure 1.2 summarize the problems and solutions proposed in the thesis.

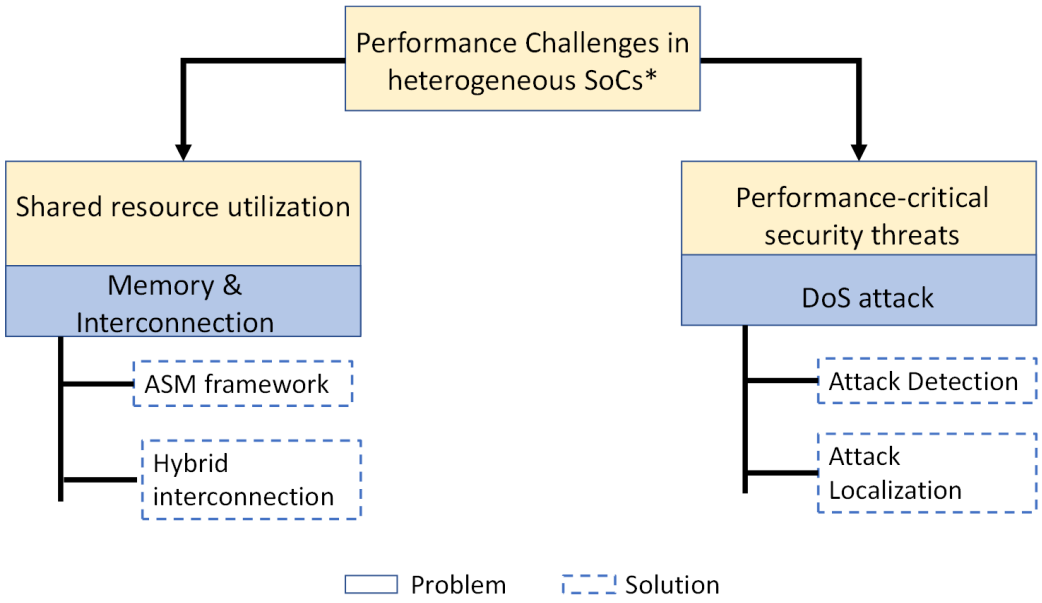


Figure 1.2: Overview of the problem statement and the proposed solutions.

1.3.1 Efficient design-space optimization of heterogeneous architectures

We propose a framework to facilitate design exploration of shared memory-based multi-accelerator heterogeneous systems. Our solution provides a best-effort design optimization framework in terms of both computational and human-resource efficiency. It addresses the problem as a two-stage process. In the first stage, the framework analyzes the accelerator memory accesses and provides the private/shared memory hierarchy as a design choice that minimizes the application runtime under given resource constraints. Since multi-accelerator embedded systems are generally designed for domain-specific applications, it allows us to study targeted applications at design time and gain insights about the application behaviour to provide optimal system design. The

Here, we highlight the performance challenges in heterogeneous SoCs targeted in this thesis.

second stage performs efficient shared data allocation on the shared memory architecture to further optimize the system performance. Our contributions are as follows:

- We extensively study memory access behaviour of multi-accelerator heterogeneous systems and provide a formal definition to the on-chip resource sharing problem for domain-specific applications at the design time.
- We propose a design exploration framework to optimally share the memory subsystem in a multi-accelerator environment to minimize the overall execution time under a given resource constraint. We formulate the memory sharing problem to capture the application's private and shared data information and their effect on the on-chip network to provide the optimal configuration. To further optimize the system performance, we employ an efficient method for shared data allocation over the on-chip memory subsystem.
- We demonstrate the feasibility and effectiveness of our method through performance analysis of multi-accelerator heterogeneous systems using representative domain-specific benchmarks. We also validate our framework against a pareto-optimal analysis for performance improvements and energy savings over the baselines.

1.3.2 Mitigation against performance-aware security attacks

We propose attack detection and localization frameworks to minimize the application performance degradation due to a MIP creating a flooding-based DoS attack. We incorporate state-of-the-art machine learning techniques to detect an ongoing attack and localize the MIPs within the system. The contributions are summarized as follows.

1.3.2.1 DoS attack detection

The primary goal of our proposed solution is to provide an accurate and reliable flooding-based DoS attack detection framework. The main part of our framework is an ML classifier which performs binary classification for attack and non-attack network traffic. The features chosen to train the ML classifier are carefully selected to include the network flow characteristics as well as the current system states. A two-level detection approach is introduced that keeps a balance between accurate attack detection and incurred overheads, which is crucial for time-critical systems. Our major contributions are summarized as follows:

- We present a two-step framework for detecting flooding-based DoS attacks in accelerator-rich heterogeneous SoCs. An ML classifier is employed in the second step for accurate attack detection, while the first

step minimizes the overheads incurred in the detection process.

- We introduce important feature metrics for attack detection by extensively studying the system behaviour. These features are dynamically tuned by an ML classifier for accurate attack detection in contrast to inefficient manual tuning.
- We incorporate various techniques like hierarchical feature data aggregation and prioritization of feature packets to curtail performance degradation and achieve early attack detection.
- We demonstrate the efficacy of our framework in detecting a flooding-based DoS attack using real accelerator benchmarks.

1.3.2.2 DoS attack localization

We present a localization framework called Sniffer, which successfully localizes one or multiple MIPs creating a flooding-based DoS attack. Our framework employs a low-overhead ML-based approach that tunes multiple network parameters to appropriate thresholds and helps in accurate MIP localization. A collective decision-making strategy is also introduced to increase the accuracy of localization, wherein, the framework considers the opinions of all the adjacent nodes at every intermediate node during the localization process. The ML-based approach, along with collective decision-making strategy enable Sniffer to successfully localize single or multiple MIPs with the least communication disruption. Our major contributions are summarized as follows:

- We propose Sniffer, an MIP localization framework for NoC-based heterogeneous systems which is able to promptly localize attacker nodes in varied scenarios with multiple malicious and/or victim IPs.
- We employ an ML model which is trained with carefully selected feature metrics to flag the congestion status of a router port as an attack, which helps in locating the MIPs with high accuracy.
- We introduce multiple design aspects within Sniffer and use a collective decision-making strategy to further improve accuracy for congestion checks, and minimize localization time by significantly reducing the search space for attack localization.
- We demonstrate the efficacy of Sniffer in localizing MIPs using real-world benchmark applications.

1.3.3 Efficient hardware accelerator design

We also studied the design challenges of the Convolutional Neural Network (CNN) accelerators which are employed in various application domains. We introduced an efficient custom-designed dual-layered network

to reduce the high latency incurred due to long distant broadcast or multicast communications in CNN accelerators. The proposed dual-layered network takes into account the movement of data among the Processing Elements (PEs) and provides optimal use or reuse of existing infrastructure combined with emerging interconnections. We proposed a data network that is responsible for propagating input and output neurons and it communicates over a wired-mesh design. The second network layer is coined as the weight network that makes use of wireless interconnection for efficiently transmitting the kernel weights. Each PE in the system connects to both data network and weight network layers through its network interface. At each clock cycle, an input weight and an input activation are available to every PE, that performs a convolution operation and stores the result in its local memory. Furthermore, we also propose a data-flow scheduling algorithm to take advantage of the data-reuse capability of CNN algorithms. The data-flow algorithm effectively make use of the data inputs and weights once brought into the on-chip memory and reuses them between the PEs. Our data-flow algorithm also reuses the partial results generated by a PE and forward them appropriately through the neighboring PEs. The proposed hybrid interconnection along with the data-flow scheduling algorithm provides a significant improvement in bandwidth, latency reduction and network energy saving as compared to baseline wired infrastructure with traditional topologies. Our major contributions are summarized as follows:

- Extensive analysis of application data-flow and communication patterns of CNN algorithms in hardware accelerators.
- A scalable fused-interconnection platform, that combines wired and wireless links to overcome the communication bottlenecks in traditional accelerator designs.
- Efficient data-flow management with wireless links for weight broadcasting and wired links for result propagation to either PEs or off-chip memory.

1.4 Organization of Thesis

The remainder of this thesis is organized as follows:

1. In Chapter 2, we present the background and literature works related to the domain of problems targeted by our proposed frameworks.
2. In Chapter 3, we present a detailed discussion on the challenges of memory resource optimization in case of accelerator-rich heterogeneous systems. We also present the proposed design space optimization framework which provides an efficient on-chip memory utilization by taking into account the accelerators' private and shared data information, and their effect on the on-chip network contention.

3. Chapter 4 discusses the vulnerabilities introduced due to the integration of third-party IP blocks within the heterogeneous systems. A machine-learning based framework is also presented that efficiently detects a flooding-based DoS attack induced by the third-party malicious IPs within the systems.

4. In Chapter 5, we present our proposed perceptron-based framework for localizing one or multiple malicious IP blocks that are creating a flooding-based DoS attack in the heterogeneous systems.

5. In Chapter 6, we propose an efficient accelerator architecture for convolutional neural networks that uses a fused wired and wireless interconnection to address the communication bottlenecks of the traditional accelerator designs.

6. Chapter 7 finally concludes the contributions of this thesis and discusses future directions of our work.

Chapter 2

Background and Related Work

This chapter discusses the topics required to enhance the understanding of the contributions of this thesis. It also provides a literature review and briefly discusses the research gaps and scope of improvements in the related works.

2.1 Background

In this Section, we present a brief background on the accelerator architectures, memory subsystems, interconnection networks, and security issues in accelerator-rich systems. We also introduce the Convolution Neural Networks (CNNs) and briefly discuss the opportunities and challenges in designing efficient accelerators for such algorithms.

2.1.1 Accelerator Architectures

The accelerator-rich heterogeneous SoCs comprise of different types of accelerator cores along with the general-purpose cores. These accelerator cores varies in terms of functionality and fundamental design choices. Depending upon the placement with respect to the general-purpose cores, the on-chip hardware accelerators can be classified into Tightly-Coupled Accelerators (TCAs) and Loosely-Coupled Accelerators (LCAs), as shown in figure 2.1. The TCAs consist of specialized hardware units to accelerate part of an application running on the host cores. They are placed inside or very close to the host cores and communicate with the host processors to access the system memory. Figure 2.1.a shows one such implementation of TCA, where the accelerator logic is placed parallel to the ALU unit of CPU pipeline, and they share key resources like registers, memory-management unit etc. However, unlike TCAs, LCAs are placed away from the host cores, which allow them to have complex datapaths to accelerate an entire application without affecting the processor core design. Since

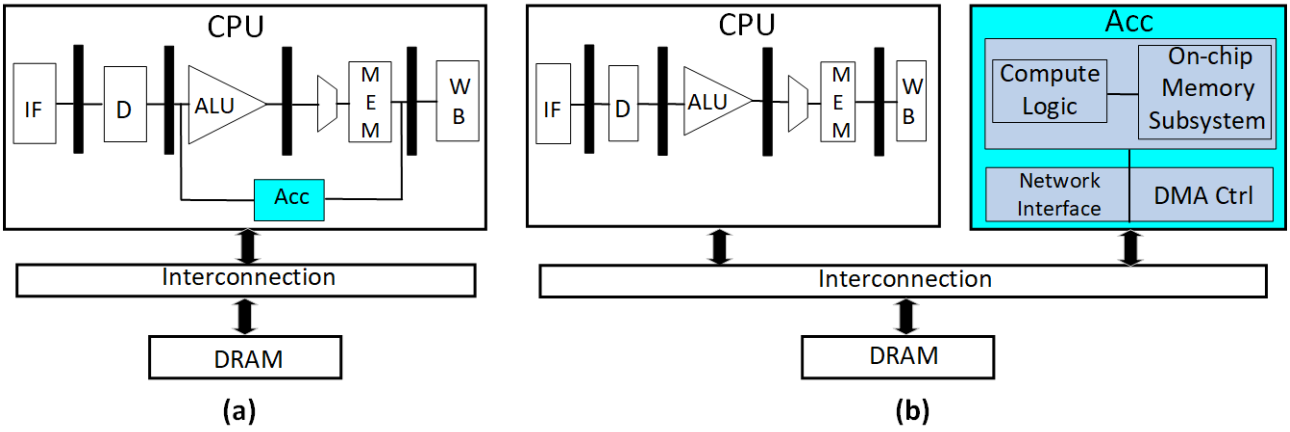


Figure 2.1: (a) Tightly-coupled accelerator (TCA); (b) Loosely-coupled accelerator (LCA);

LCAs are implemented as separate units out-of-core, it consists of private memories to store the input data, intermediate outputs and final output data to be written back to off-chip memory. As shown in Figure 2.1.b, an interconnection network along with Direct Memory Access (DMA) controllers and network interfaces facilitate communication between the accelerators, CPU cores and the memory modules. The independent design of LCAs provides more opportunities for design-reuse as well as higher flexibility by allowing CPU cores to perform other tasks parallelly while the LCAs are working on a part of the application.

2.1.2 Accelerator Memory Subsystem

Unlike the general-purpose cores, accelerators need to perform multiple loads/stores every clock cycle. Hence, they employ Scratchpad Memory (SPM) to achieve energy efficiency and timing predictability [7, 8]. SPMs are comprised of SRAM arrays with simple decoding and column circuitry, where the decoder selects the data row to be read/written based on the requested address. On the contrary, caches comprise of SRAM arrays, tag storage, column and comparator circuitry. As SPMs do not incur tag and comparator overheads, they are more energy-efficient than caches. Since SPMs are plain memories, the compiler/programmer needs to explicitly manage data allocation on SPMs. However, it provides timing predictability and reduces starvation as compared to cache-based systems that suffer from multiple cache misses. The DMA engine associated with each accelerator core facilitates data-transfer between off-chip memory and on-chip SPMs.

Since the on-chip SPM memory is limited and mostly insufficient to hold all the application data at a time, there is a need for SPM data management to control the data flow. Conventionally, data management in SPMs is done by code transformation, where compiler instructions are inserted within the program code to indicate points of data transfers between on-chip and off-chip memory [9, 10]. A software SPM Manager (SPMM) is responsible for performing this data management without incurring any hardware overhead [10]. The SPMM

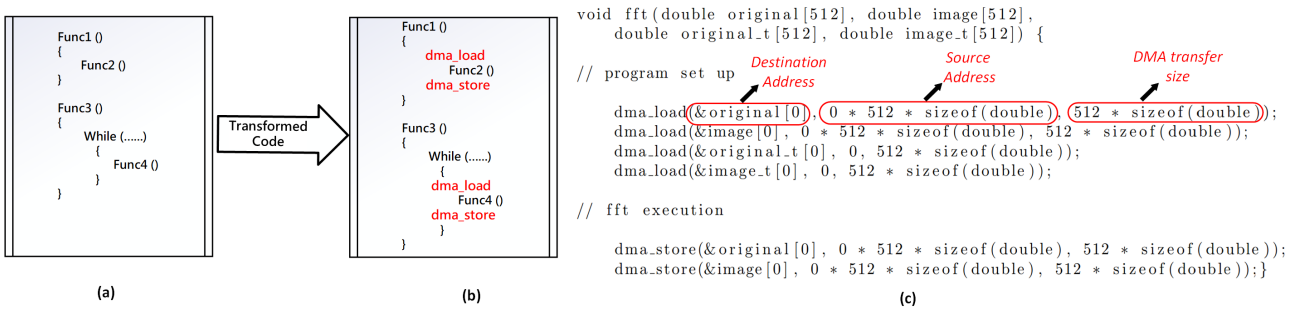


Figure 2.2: a) Application pseudocode; b) Transformed pseudocode with compiler instructions inserted for DMA transfer; c) Example C code snippet of FFT kernel highlighting APIs to facilitate DMA load/store operations.

maintains the information regarding each basic block including their addresses, sizes etc. obtained during compile time. Here, a basic block represents the granularity at which program data are transferred from the off-chip DRAM to the on-chip SPMs. [10] performed data management at the granularity of function stack frames. On the other hand, the data management in [9] is done at the whole stack space granularity. The SPMM keeps track of data-transfer to avoid overflow of the on-chip SPM and keep accommodating new basic blocks within the available space. This is accomplished by specific compiler instructions or APIs that facilitate the fetching and eviction of basic blocks from the SPMs. We explain the process using an example with two basic APIs as follows. Figure 2.2.a shows an example program code, and Figure 2.2.b shows the corresponding transformed code. The compiler instructions *dma_load* and *dma_store* are inserted before and after a basic block of the program code. The instruction *dma_load* signifies the point in the code where the data variables in a particular basic block need to be brought in the on-chip SPM. Whereas a *dma_store* signifies the point when the on-chip SPM (part or whole) needs to be flushed, and the resultant data is written back to the off-chip memory. Figure 2.2.c shows an example code snippet for FFT kernel with *dma_load* and *dma_store* APIs, highlighting the basic parameters for DMA transfer i.e., source address, destination address and granularity of transfer. In such a data-management technique, the granularity of data transfers between the off-chip and on-chip memory plays an important role in the overall system performance. As the number of accelerators increases, the memory access latency of each accelerator will also increase due to the large number of memory requests generated. In case of a fine-grained data-management method, the number of DMA requests from each accelerator will increase drastically, and the accelerators may starve to get a chance to access the memory. On the other hand, a coarse-grained data-management method will relatively produce a small number of DMA requests and hence reduce the starvation time. As a result, we consider LCAs with SPMs as on-chip memory, and the whole SPM stack space represents the data-transfer granularity as considered in [9].

2.1.3 Security Threat from Third-party accelerator IPs

In an accelerator-rich heterogeneous SoC comprising of many accelerator cores, a few CPU cores, and memory elements, the interactions between all the on-chip modules are typically accomplished by a Network-on-Chip (NoC) [11]. In such a scenario, a Malicious Third-party Accelerator node can initiate an attack on other benign on-chip modules through the commonly shared NoC. Hence, while an M3PA can affect the performance of the applications running locally, it may also have a severe impact on the entire system performance. It can lead to information leakage, data alteration, system performance degradation, and memory corruption. A taxonomy of accelerator vulnerabilities is presented in [12], which highlights the need for efficient security mechanisms in ArSoCs.

A Denial-of-Service (DoS) attack in a network is an attack that diminishes a network's capacity to provide services to legitimate users. In accelerator-rich SoCs, an M3PA can trigger a DoS attack by continuously injecting useless packets on the NoC and flooding the network. As a result, the network exhibits high congestion and extends communication latency between the legitimate on-chip modules. Since accelerators are typically employed for real-time applications, a flooding attack by a malicious accelerator resulting in DoS will severely degrade the overall system performance defeating the purpose of using accelerators in the first place. Therefore, it is imperative to accurately detect flooding attacks at an early stage and meet the timing constraints of the applications running on the SoC. Moreover, the nodes creating such attacks also need to be traced back and appropriate actions need to be taken to curtail further performance degradation.

2.1.4 Convolution Neural Network Accelerators

Deep convolution neural networks (CNNs) have evolved as promising machine learning algorithms, in several fields like image recognition, search engines, speech recognition, etc. [13, 14, 15, 16, 17]. The state-of-the-art CNN algorithms consists of multiple layers which is usually a combination of four types of layers namely, convolution layer(CONV), pooling layer(POOL), normalization layer(NORM) and fully connected layer(FC). Each layer generates output feature maps by accumulating the partial sums, which becomes the input feature map of the next layer. Most of the CNN computation is carried out in the CONV layer, where filters are applied on input feature maps to identify the presence of desired characteristics. The POOL layer is used to scale down the feature maps and the NORM layer performs square-sum of the sliding window elements followed by a non-linear function, which is multiplied with the input elements to be normalized. The FC layer computes the final classification results, where the output neurons are fully connected to the input neurons. In the recent CNN models, the number of CONV layers is found to be rapidly increasing, ranging from merely five to several hundreds of layers [13, 15]. Rapid advancements have been made in

the prediction accuracy of these CNN models over the past years further expanding their use. However, the increasing accuracy and adoption of CNNs come at the cost of a tremendous increase in training data involved (ranging from 60 million [13] to 10 billion [17] weights) and computation capabilities required. Also, the large number of weights and multiply & accumulate (MAC) operations involved in CONV and FC layers of recent CNN models [13, 14, 15] impose high pressure on network bandwidth and on-chip memory. It is crucial to transfer the computation load into specialized architectures to improve the overall performance. Hence, to meet such an extensive computational demand, implementation of CNNs is gradually shifted towards accelerator based designs. The inherent computation characteristics of deep CNN algorithms open up opportunities to achieve data-level, task-level and layer-level parallelism. For instance, computation of each output neuron can be mapped to a processing element (PE) thus achieving data-level parallelism. While multiple prior works have worked on the data reuse and architecture design of the CNN accelerators, the communication platform for CNN accelerators have not been appropriately addressed. Hence, the design space of interconnection networks in CNN accelerators need to be explored to provide an efficient framework for the PEs to communicate with each other and achieve better performance.

2.2 Related Work

In recent years, the hardware accelerators have been adopted at various domains like machine learning [18, 19, 20], robotics [21], web-search [22], etc. These fixed-function accelerators are available at different granularity levels to accelerate various parts of an application [23, 24, 25, 26]. The profound performance achieved by hardware acceleration has boosted the integration of more number of on-chip accelerators alongside general-purpose cores in an SoC. This growing trend has triggered various research works toward building a platform to support multi-accelerator systems. In [27], the authors developed an allocation protocol for multiple fixed-function accelerators to provide architectural support for accelerator-rich chip multiprocessors. An experimental study carried out on this accelerator-rich platform with a shared last-level-cache showed the effectiveness of such a platform [11]. Another approach includes the work in [28] wherein an accelerator-store framework is provided to allow dynamic memory sharing between multiple accelerators. Although these proposed works provide a platform to support multi-accelerator systems, they do not present any formal methodology to find the best-suited memory configurations for such platforms. [29] proposed a method to derive memory systems for accelerators by only analyzing application codes to be executed on the accelerators. Resource optimization for heterogeneous many-core architectures is also proposed for clusterised systems [30, 31]. A DNN mapping tool called MAESTRO [32] was proposed which analyze the design space of deep learning accelerators and provides an accelerator design with better performance and energy efficiency. The framework accepts the

dataflow description of neural network algorithms, hardware resource description, and neural network layer descriptions as inputs, and outputs an efficient deep learning accelerator design. Although the Synopsis Platform Architect tool [33] explores various design space with hardware-software partitioning and multi-core SoC optimization, it does not focus on accelerator-rich system design. As compared to traditional multi-core systems, in a multi-accelerator system, different accelerator cores may have different resource requirements to maintain performance and undergo multiple levels of interactions among themselves. Thus, the presence of various fixed-function accelerators with a varied range of resource requirements, makes the system design even more complex. While the Synopsys Platform Architect Ultra [34] presents Artificial Intelligence (AI) centric hardware libraries, its primary focus is on mapping AI workloads into the SoC architecture. In this thesis, we present a framework that targets the resource optimization problem for heterogeneous SoCs where different fixed-function accelerators co-exist. It analyzes the data sharing behaviour among the given set of accelerators and provides a comprehensive model which captures the SoC communication and memory access patterns of accelerator-rich systems.

While integrating different accelerator IPs improves overall application performance, multiple works have also highlighted the vulnerability introduced by the integration of these accelerator IPs from untrusted third parties. A Border Control hardware is proposed in [35] to prevent memory corruption due to unhealthy memory accesses by accelerators. In [36, 37], Dynamic Information Flow Tracking (DIFT) based security measures are introduced to track any spurious information flow during application execution and prevent security violations. All these works primarily focus on mitigating security attacks causing information leakage and data corruption, which compromise the integrity and confidentiality of an accelerator-rich system. DoS attacks which manipulate the availability of shared resources are also explored in the context of NoC-based multi-core systems [38, 39, 40, 41]. In [41], design guidelines were given for mesh-based multi-core systems to mitigate the effects of DoS attacks, whereas [42] presents security verification methods for NoC micro-architectures. The authors in [38] rely on using extra packets to perform security checks, which requires multiple windows of checking and packet injection, further delaying network communication. A central monitoring unit is proposed in [39] to detect unnatural traffic conditions based on average bandwidth violations which are computed empirically at the design time. However, such empirical-based threshold setting is not suitable for accelerator-rich SoCs due to its dynamic traffic variations. Similarly, in [40], a DoS attack detection method is proposed by statically profiling a defined network traffic to distinguish between an attack and non-attack scenario. Although lightweight, such a method will result in inefficient detection due to the challenges involved in setting an upper bound for non-attack traffic behaviour. Recently, the machine learning techniques have also been incorporated in malware [43, 44] and network intrusion detection [45]. The approaches presented in the literature primarily rely on the hardware performance counters (HPCs) available in modern processors for attack detection. In [44],

a malware detector is presented where ML classifiers are trained on HPC values collected from Android ARM and Intel X86 platforms. Similarly, in [43], a hardware-assisted malware detection framework is presented that uses HPC values to classify memory access patterns based on control-flow or data structure modifications. In [45], an ML-based method for intrusion detection in networks is proposed to mitigate Distributed DoS (DDoS) attacks. The ML classifier in [45] is also trained on HPC values along with a number of concurrent active users present in the network. Although working, all the proposed approaches employ HPCs as the primary metric for attack detection, which poses multiple concerns on the reliability and feasibility of such detection frameworks. As discussed in [46], the non-deterministic HPC values and their lack of portability are the two major concerns for employing HPCs in security domains. The non-determinism in HPC events is mostly due to operating system behaviour and the concurrent applications running on the system. HPCs may also overcount certain events [47], which will train the ML classifiers erroneously to detect an attack scenario. Finally, the variations in micro-architectural events across different architectures limit the portability of HPC-based detection framework. HPC events present in one architecture may not be available in another, or may be implemented in completely different ways across the architectures. In this thesis, we propose an attack detection and localization framework which employs an ML-based approach for dynamic tuning of feature metrics. The feature metrics selected to train the classifiers are only based on the network and system state characteristics without relying on HPC events.

Multiple research works have also proposed to optimize the design space of a specific hardware accelerator which in turn optimizes the power and performance efficiency of the heterogeneous SoCs. The CNN accelerator is one of the widely studied accelerator architecture that provides significant performance improvement for the machine learning algorithms. The existing CNN accelerators are mostly designed with traditional interconnect platforms with little effort in optimizing the communication infrastructure. CNAPS [48] is a neuro-computer that uses a broadcast bus for communication with multiple PEs. It's design was developed for supporting smaller networks where there is limited communication with the global memory and becomes obsolete with the increasing network sizes. Truenorth [49] is based on a crossbar structure for intra-core communication and a mesh-based structure for inter-core communication. Du et al. proposed ShiDianNao [50] that exploits the data locality of the 2D feature maps while eliminating the off-chip memory access using an on-chip buffer which is suitable for embedded systems. But, their design depends on traditional bus interconnects for weight broadcasting that will suffer in larger system sizes. While [51] shows potential usage for broadcast-based traffic scenarios, by allowing design for one-to-many and many-to-one bus communication. Other approaches includes NeuFlow [52], a systolic architecture proposed by Farabet et al. and mobile coprocessors by Gokhale et al. [53] which supports accelerating the CNN computations. Chen et al. [18] showcased a CNN accelerator with a data flow mechanism that reduces off-chip memory accesses by data-reuse at multiple levels but depends

on bus-based system and multicast controllers for populating the input data. A theoretical study was done in [54] that compares different interconnection networks in neural network accelerators. Kwon et al.[55] proposed a microswitch based NoC for CNN to lower the area and power of traditional router-based NoCs, enabling addition of more PEs to the network. Choi et.al. [56] discussed about enhancing the on-chip communication network for training deep CNNs on a generic heterogeneous system consisting of multiple CPU and GPU cores. Although their work provides evidence of performance improvement through efficient interconnection but did not account the data-flow mechanism of specialized hardware accelerators. In this thesis, we address the communication bottlenecks of conventional accelerator designs considering their data-flow patterns.

Chapter 3

Design Space Optimization of Shared Memory Architecture in Accelerator-rich Systems

To simplify the IP design and integration in an SoC, the hardware accelerators are developed standalone with each accelerator building block optimized for a specific application. These fixed-function accelerators are typically memory-bound, designed to perform simpler tasks on a high volume of data, and it is observed that memory elements consume an appreciable amount of the total accelerator area. Hence, estimating the best-suited accelerator memory subsystem earlier in the design phase would save a significant amount of chip area bringing down the design cost or freeing space for integrating more logic elements. In this chapter, we first discuss the advantages and challenges in optimizing the memory subsystems of accelerator-rich heterogeneous SoCs. We then present our proposed solution to overcome the challenges and the corresponding evaluation results.

3.1 Memory Resource Optimization Challenges in Accelerator-rich Systems

We demonstrate the memory resource optimization problem using an example application running on an accelerator-rich heterogeneous SoC. A surveillance application is considered as an example case of edge computing systems employed in various scenarios like banking sectors, parking lots, etc. We consider a real-time object detection pipeline (shown in Figure 3.1.a), where pre-processing of input images is carried out at the edge devices and then transferred to cloud for further processing. At the edge device, the input images need to be compressed and encrypted before sending them through the communication network. Since compression

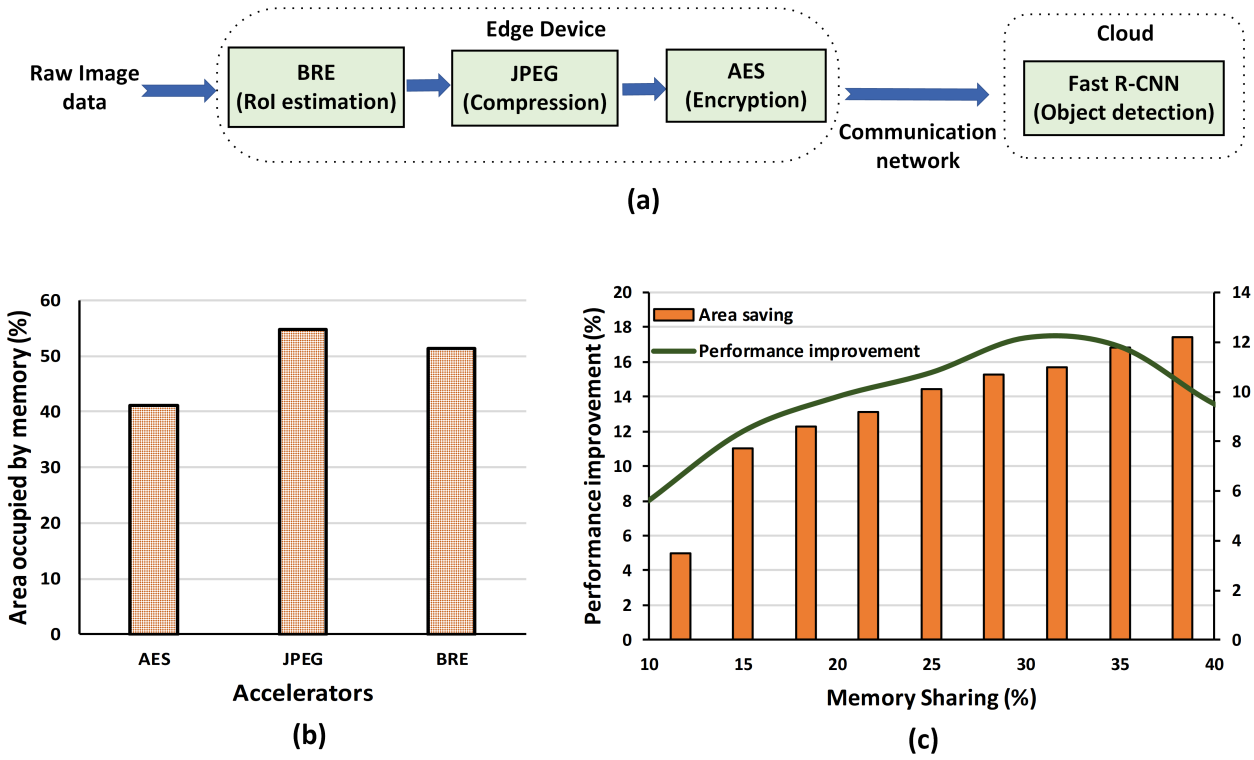


Figure 3.1: Motivating example. (a) Application pipeline; (b) Area occupied by memory subsystem in each accelerator; (c) Performance improvement and area savings by sharing accelerator memory over baseline.

algorithms like JPEG may lose important information of the Region-of-Interests (RoIs), the images are first passed through a binarized normed gradient (BING)-based RoI extraction method called BRE. To achieve high performance and energy efficiency, we choose to employ multiple copies of BRE, JPEG and AES accelerators working parallelly on different input segments. All the accelerator kernels are generated by applying common accelerator optimization techniques through the gem5-Aladdin framework [57].

Although standalone accelerator optimizations reduce design complexity, it may limit the overall system resource optimization when integrated together on the same SoC. Since all the fixed-function accelerators on a single SoC work collaboratively on different application kernels, it provide opportunities to share the system resources between them. Figure 3.1.b shows that the memory system occupies 41% to 54% area in each accelerator in our example scenario. As memory constitutes a significant part of the accelerator area, it is imperative to share the memory blocks in a multi-accelerator system. Small private memories along with a shared memory for all the accelerators optimize the entire system both in terms of area and performance. Figure 3.1.c shows the performance improvement and memory area savings achieved by our example application as the fraction of shared memory increases. The results are obtained for a private/shared memory system against a private-memory-only baseline. By bringing more data on the chip, the shared memory system helps in reducing costly off-chip memory accesses, thus increasing application performance. It also helps in alleviating

replication of shared data between different accelerator kernels, further reducing frequent off-chip memory accesses. Since our example application also comprises of many producer-consumer communication, it is able to utilize the shared memory efficiently by reading and writing data to same memory blocks. As shown in Figure 3.1.c, as the amount of memory sharing increases, we observe a significant reduction in the total on-chip memory area. However, increasing the amount of shared memory beyond a threshold will increase the waiting latency to service the shared data requests coming from different accelerators. This is due to the contention on the on-chip network as well as at the shared memory caused by a high volume of memory requests from each accelerator. As a result, the application performance is affected and might start having a negative effect due to memory sharing. Figure 3.1.c shows a similar trend where the application performance starts to decline after increasing shared memory fraction beyond 35%. Additionally, as more accelerators are integrated on chip, the stringent area budget of embedded systems become a challenge. The imposed area constraint for on-chip memory decreases the private memory at each accelerator as the fraction of shared memory increases. It results in a reduction of the amount of accelerator-specific data stored on the chip and consequently increases off-chip memory accesses for this private data.

In light of the above discussions, it is clear that although a shared memory architecture provides an improvement in terms of performance and area savings, spontaneous memory sharing may degrade the overall system performance. Different applications may perform differently under various shared-memory configurations depending upon their communication and memory access behaviour. Conventionally, chip designers rely on software simulation models to test different design configurations and meet performance requirements. However, with the increasing number of on-chip accelerators, it becomes infeasible to simulate the vast design space for optimization to its entirety while meeting the Time-to-Market (TTM) demand. As a result, the chip designers end up exploring only a part of the design space, which may lead to sub-optimal designs. Therefore, it is necessary to have a methodology to determine the best-suited memory configuration for a target application that i) gives utmost benefit from a shared memory system in terms of performance, energy and area savings under given resource constraints; ii) reduces the computational and human-resource efforts while meeting the TTM demand.

The remainder of this chapter is organized as follows: Section 3.2 introduces our proposed resource optimization framework. Section 3.3 presents the evaluation results and Section 3.4 provides a summary of proposed work and conclusions.

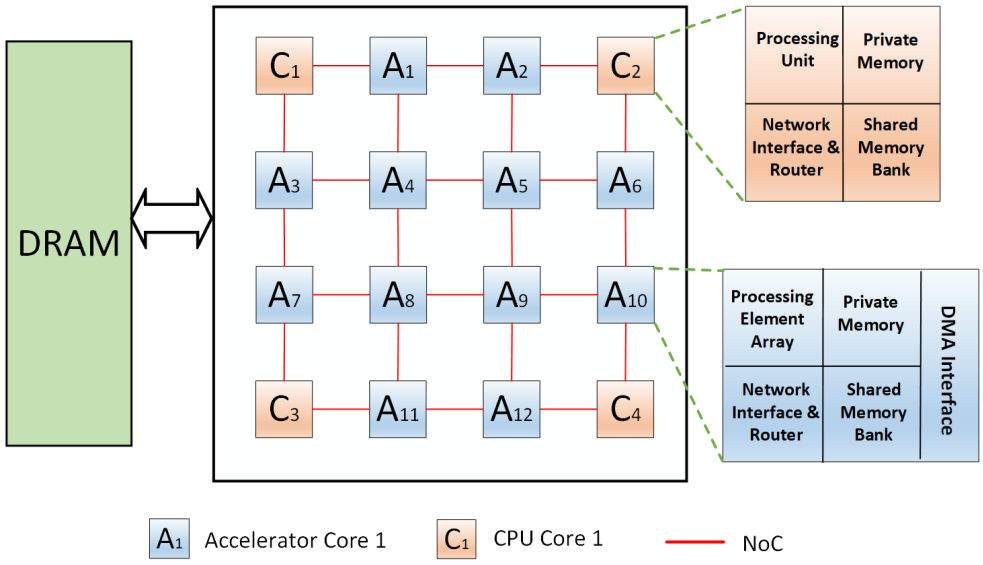


Figure 3.2: A Multi-accelerator heterogeneous system.

3.2 Accelerator Shared Memory Framework (*ASM*)

This Section describes our system architecture, objectives and the proposed *ASM* framework for efficient resource utilization in multi-accelerator systems.

3.2.1 System Architecture

We consider a heterogeneous SoC platform, as shown in Figure 3.2 which comprises of multiple fixed-function LCAs with private SPM, a distributed shared memory and a few general purpose cores connected by an interconnection network. We consider SRAM-based memory design in this work. Since applications running on the LCAs are time-critical, we choose to employ NoC as the communication network due to its ability to support high bandwidth and low latency communication for large system sizes [5, 58]. Given any domain-specific application running on this SoC, the general-purpose cores offload their compute-intensive kernels on a set of hardware accelerators A_1, A_2,.....A_N, where N is the total number of accelerators invoked by the application during its run-time. Let pr_i be the private memory of each accelerator and sh_i denotes the shared memory bank associated with each accelerator A_i. In this work, we are assuming no memory sharing between the accelerators and the general-purpose cores. The total shared memory available for all the accelerators, s_{tot} is divided into N banks and is physically distributed such that each bank is integrated with one on-chip accelerator as shown in Figure 3.2. R_i corresponds to the network router associated with an accelerator A_i or a core C_i tile, that facilitates sending and receiving of network packets between the on-chip modules. To access a remote shared memory bank, a request from a source accelerator A_i traverses through a deterministic network path (here, we

assume the *XY* routing algorithm) to reach the shared memory bank with requested data by passing through a set of network routers R_1, R_2, R_3.....,R_Z. Since we are considering a scratchpad-based memory subsystem, the off-chip DRAM accesses are typically carried out by DMA invocations with the help of custom compiler instructions.

In this work, the system architecture shown in Figure 3.2 is one of the widely-used implementations of a multi-accelerator system which is commonly accepted as a suitable design for many domain-specific applications [27, 11]. However, *ASM* can be easily deployed for other multi-accelerator architectures by providing appropriate input parameters for the given system.

3.2.2 Objective

The time taken to execute an application on the system presented in Section 3.2.1 is given by the combination of its computation time and memory access time. We assume that all the critical kernel computations of an application are carried out on the fixed-function accelerators. Hence, the overall execution time of the application is dominated by the time required by the accelerators to execute the application kernels. The time taken to execute the complete task assigned to an accelerator can be broadly defined as the combination of two components , i.e., T_{comp_i} and $T_{mem\_access\_i}$, representing the computation and memory access time of an accelerator A_i, respectively. The total application execution time is represented by the sum of the execution time of all accelerator kernels invoked by the application, as shown in equation (3.1).

$$\sum_{i=1}^{N} T_{comp\_i} + T_{mem\_access\_i} \tag{3.1}$$

As discussed earlier, due to the stringent design constraints of embedded systems, the available on-chip resources are limited and must be utilized efficiently. The work presented in this chapter addresses the memory subsystem design problem, and hence, the resulting memory hierarchy and allocation must achieve the desired application performance, while adhering to the resource constraints of the system being designed. These design constraints for the memory subsystem are primarily in the form of area as (i) hardware accelerators have huge requirements on available memory for higher performance and (ii) even in existing systems, a considerable fraction of total chip area is dedicated to the memory subsystem, which is limited as more and more computational elements are integrated into a single chip. Owing to these constraints, we define our objective as *"Given a memory area budget constraint for a multi-accelerator system, realize a hierarchy of private and shared memory configurations meeting the budget constraints and that minimizes the overall application execution time"*,

and is shown in equation (3.2).

$$min \sum_{i=1}^{N} T_{comp\_i} + T_{mem\_access\_i} \tag{3.2}$$
$$s.t, B_{resultant} \leq B,$$

where $B_{resultant}$ is the total memory area obtained in the process of achieving our objective function and B is the given memory area constraint.

In this work, our proposed framework investigates the impact of memory hierarchy and associated resource constraints on the performance of an application running on the multi-accelerator system. Since the accelerator compute logic or pipeline remain unmodified, the ideal accelerator computation time when data is readily available remains the same. However, the memory access latency becomes a major performance bottleneck as the accelerator compute units need to stall until the data is fetched from memory. As compared to the ideal computation time of accelerators, the extended time due to memory access latency will increase the application's actual computation time. Therefore, the primary objective of this work is redefined as minimizing the memory access time of all the accelerators, and is represented by equation (3.3). There is however an impact on actual accelerator computation due to memory hierarchy design in the form of compiler induced DMA instructions for bringing in required data from off-chip memory to on-chip SPM. We account for this additional computation time in our framework while computing the overall memory access time, which is discussed further in Section 3.2.3.

$$min \sum_{i=1}^{N} T_{mem\_access\_i} \tag{3.3}$$
$$s.t, B_{resultant} \leq B$$

3.2.3 Design Details of *ASM* Framework

In this Section, we illustrate our proposed framework to achieve the objective function defined in equation (3.3), which aims to minimize the memory access time of all the accelerators and provide utmost performance in an accelerator-rich system. The memory access time of accelerators depends upon the availability of accelerator-specific data on chip and proximity of the on-chip memory containing the data from the accelerator. Hence, we broadly divide the *ASM* framework into two stages: (1) Design Space Optimization (DSO_{Mem}) and (2) Performance-aware Data Allocation (*PDA*). Figure 3.3 shows the overview of *ASM* framework including the two stages. The DSO_{Mem} stage explores the design space of memory subsystem configuration to provide an optimal private/shared memory configuration for an accelerator-rich system under a given memory budget. It is guided by an objective function which aims to minimize the overall memory access time of accelerators, thus increasing application performance. By providing the best-suited private memory configuration for each

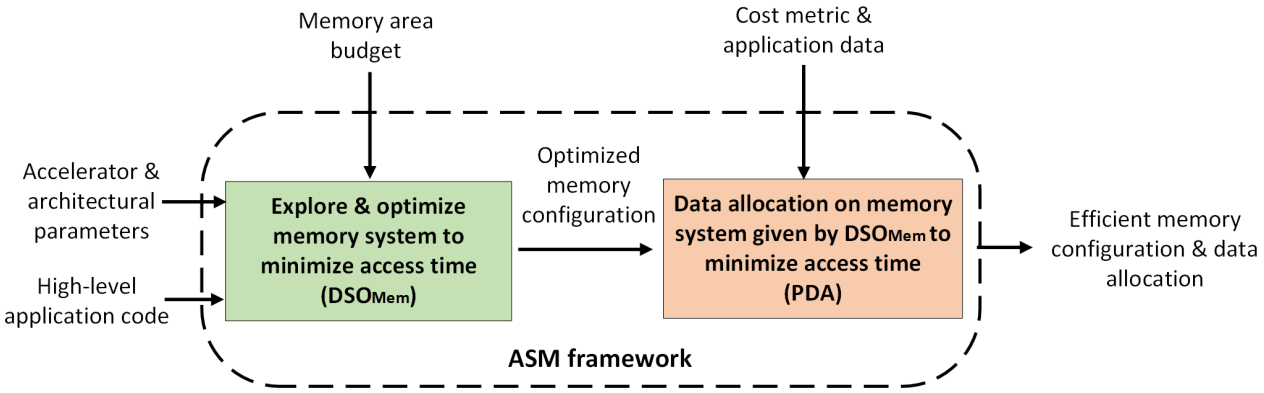


Figure 3.3: Overview of the proposed *ASM* framework.

accelerator, DSO_{Mem} stage minimizes the off-chip memory accesses for accelerator-specific data. Also, an appropriate on-chip shared memory for all the accelerators helps in inter-accelerator communication and data sharing, thus further reducing off-chip memory accesses while providing more space for accelerator data on chip. The second stage of *ASM*, i.e., the *PDA* stage, takes the memory hierarchy obtained from DSO_{Mem} stage and provides an efficient data allocation over the shared memory blocks. The performance-aware placement of data among the distributed shared memory reduces the time taken for the accelerators to access data from remote shared banks through the on-chip network. By performing an effective data allocation, the *PDA* stage minimizes contention at the shared memory and curtails data thrashing, thus minimizing the overall memory access time of the accelerators. Therefore, by providing best-suited memory configurations and efficient data allocation strategy, the DSO_{Mem} and *PDA* stages of *ASM* framework focus on minimizing accelerator memory access time and achieve our objective function.

3.2.3.1 Design Space Optimization (DSO_{Mem})

The design space optimization stage presents a comprehensive model that captures various aspects of the system that affect memory access time of the accelerators within an SoC. It models all phases of memory access requests. We broadly classify the system aspects captured by *ASM* framework as data abstraction level, network level and memory access level, which influences the SoC communication. At the higher data abstraction level, DSO_{Mem} stage identifies different memory access patterns of each accelerator invoked during the execution of an application on a multi-accelerator system. We run and profile a given application with representative input data to collect the memory access traces and identify shared and private accesses at each accelerator. These private and shared data accesses at each accelerator are identified by D_{pri} and D_{shi} respectively and provide memory access behaviour of the application. This application profiling information is then used to model our resource sharing problem. At the communication network level, DSO_{Mem} effectively models the shared

Table 3.1: Different parameters and their definitions used to represent the DSO_{Mem} stage of the *ASM* framework.

Parameters	Definitions	Parameters	Definitions
A_i	Accelerator number i	N	Total number of accelerators
pr_i	Private memory of A_i	sh_i	Shared memory bank of A_i
s_{tot}	Total shared memory available for accelerators	R_i	Network router at node i
T_{comp_i}	Computation time of A_i	$T_{mem\_access\_i}$	Memory access time of A_i
D_{pri}	Total private data accesses of A_i	D_{shi}	Total shared data accesses ofA_i
Lat_{pr}	Latency to access private data	Lat_{sh}	Latency to access shared data
$Access\_lat_{pr}$	Latency to only access pr_i	$Access\_lat_{sh}$	Latency to only access sh_i
$Port\_lat_{pr}$	Port contention latency at pr_i	$Port\_lat_{sh}$	Port contention latency at sh_i
DMA_lat_{pr}	DMA transfer latency for pr_i	DMA_lat_{sh}	DMA transfer latency for sh_i
$Dma\_instr\_lat_{pr}$	Extra compiler instruction execution latency for pr_i	$Dma\_instr\_lat_{sh}$	Extra compiler instruction execution latency for sh_i
NoC_lat	On-chip network latency	lat_{const}	DMA base latency
lat_{slope}	Rate of increase in latency w.r.t. DMA transfer size	hop_{lat}	Latency to traverse between adjacent nodes
t_{exec}	Packet service time for a node	H	Average hop count
pr_{min}	Minimum pr_i	W	Average packet waiting time
ρ	Utilization value	λ	Packet arrival rate
μ	Packet service rate	B	On-chip memory budget
Req_i	Minimal memory requirement of an accelerator	s_i	Memory occupied by shared data of A_i

memory access characteristics and DMA calls. Finally, at the memory access level, different parameters are introduced to capture the characteristics of accessing the memory blocks itself. Table 3.1 summarizes all the variables and parameters of the proposed DSO_{Mem} framework.

Below, we discuss the objective function and constraints of the proposed resource sharing problem:

Memory access time: The performance of hardware accelerators, in general, is highly dependent on the data transfer and memory access time as they perform specific operations on a huge volume of data at any given instance, unlike general-purpose cores [11]. Hence, it is necessary to minimize the memory access time of each accelerator to increase the overall system performance. The average memory access time of an accelerator, A_i is defined as shown in equation (3.4).

$$T_{mem\_access\_i} = \frac{(D_{pri} * Lat_{pri}) + (D_{shi} * Lat_{sh})}{D_{pri} + D_{shi}}, \tag{3.4}$$

where D_{pri} and D_{shi} correspond to the number of private and shared memory accesses of an accelerator A_i respectively; Lat_{pri} and Lat_{sh} represent the latency incurred to access data from accelerator's private memory and any shared memory respectively. The total latency at any memory is comprised of the actual memory access latency, any waiting and contention latency to access the memory and network latency to reach the desired memory. Without loss of generality, we assume that on average, the latency to access any shared bank from any accelerator is similar at all accelerators in context of the design optimization problem.

Private memory access latency: The private memory access latency is defined as the sum of the latency incurred to only access an accelerator's private memory, port contention latency, DMA latency and latency due to additional compiler instructions, and is presented in equation (3.5).

$$Lat_{pr} = Access\_lat_{pr} + Port\_lat_{pr} + Dma\_lat_{pr} + Dma\_instr\_lat_{pr} \tag{3.5}$$

The $Access\_lat_{pr}$ in equation (3.5) is the amount of time required to read/write data from/to an accelerator's private memory once a request is granted access to the memory. Since SPM is software managed and does not have complex control hardware associated with caches, $Access\_lat_{pr}$ for SPMs is small and is in the order of a few clock cycles. In our framework, $Access\_lat_{pr}$ is assumed to be the same for all private memories and is solely dependent on SPM design.

Memory modules in current-day systems and especially for hardware accelerators are typically designed with multiple read/write ports to service multiple memory requests at a time [59] and reduce the waiting latency to access data. While a large number of ports reduce memory access latency, they also increase the area footprint of a memory module [60]. As a result, the number of ports permitted on a memory module is limited and is a function of the memory size [60]. When an accelerator issues multiple memory requests, the total number of requests that can be serviced at a time is limited by the number of ports available. Hence, a large number of requests result in contention at the memory ports and increases the overall memory access latency. This waiting latency is represented by port contention latency ($Port\_lat_{pr}$) in our framework and is inversely related to the private memory size, as shown in equation (3.6).

$$Port\_lat_{pr} = \alpha(pr)^{\beta}, \tag{3.6}$$

where α and β are constants whose values are determined empirically and $\beta < 0$.

As discussed earlier, the SPMM is responsible to periodically bring the required data from the off-chip memory into the on-chip SPMs and vice versa. This is accomplished through compiler induced DMA calls, which is co-dependent to the available memory size and consequently affects the overall access latency. This co-dependency of DMA calls and memory size affecting the access latency is accounted for in our framework through Dma_lat_{pr} and $Dma\_instr\_lat_{pr}$ parameters, shown in equation (3.5). Dma_lat_{pr} is the actual time spent in moving the data between on-chip SPM and off-chip memory and is dependent on the granularity of the DMA requests introduced by the compiler which in turn is proportional to SPM stack size as discussed earlier. Hence, the data movement latency between off-chip memory and SPM, Dma_lat_{pr}, is defined as a function of

private memory size as shown in equation (3.7).

$$Dma\_lat_{pr} = lat_{const} + lat_{slope} * pr, \tag{3.7}$$

where lat_{const} denotes the constant base time required for any DMA transfer and the lat_{slope} is the rate of increase of latency with respect to the increase in DMA transfer size or private memory size in this case.

The $Dma\_instr\_lat_{pr}$ in equation (3.5) represents the time taken to execute the compiler instructions inserted within the original application code to facilitate DMA transfers. The total number of additional compiler instructions inserted depends on the granularity of the DMA. A very fine-grained DMA, that reduces Dma_lat_{pr}, however, results in a large number of compiler instructions to be inserted and increases overall execution time to complete these additional instructions. On the other hand, a highly coarse DMA considerably increases the DMA latency for each set of compiler instructions inserted. Similar to Dma_lat_{pr}, DMA granularity varies with SPM stack size and as the SPM size decreases, the number of additional compiler instructions to bring in the program data increases. This is represented in our framework by $Dma\_instr\_lat_{pr}$ as shown in equation (3.8).

$$Dma\_instr\_lat_{pr} = \delta(pr)^{\gamma}, \tag{3.8}$$

where δ and γ are constants whose values are determined empirically and $\gamma < 0$.

Shared memory access latency: The shared memory access latency represents the time required by an accelerator to access required data from any shared memory bank in the system. It is defined as the sum of memory access-only latency, port contention latency, DMA latency, latency due to extra compiler instructions and latency to access a remote shared bank through the NoC, and is presented in equation (3.9).

$$Lat_{sh} = Access\_lat_{sh} + Port\_lat_{sh} + Dma\_lat_{sh} + Dma\_instr\_lat_{sh} + NoC\_lat \tag{3.9}$$

The $Access\_lat_{sh}$ in equation (3.9) denotes the access-only time for a shared memory resource. The $Port\_lat_{sh}$ and the Dma_lat_{sh} represent the latency incurred due to the port contention at the shared memory resource and the DMA transfer latency for shared data, respectively. The time required to process the extra compiler instructions to enable the DMA transfers is denoted by the $Dma\_instr\_lat_{sh}$. These parameters follow similar trends as discussed in case of private memory access latency, albeit with some differences, and are represented as functions of shared memory size shown in equations (3.10), (3.11), and (3.12).

$$Port\_lat_{sh} = \tau\left(\frac{s_{tot}}{N}\right)^{\eta}, \tag{3.10}$$

$$Dma\_lat_{sh} = lat_{const} + lat_{slope} * \left(\frac{s_{tot}}{N}\right), \tag{3.11}$$

$$Dma\_instr\_lat_{sh} = \sigma\left(\frac{s_{tot}}{N}\right)^{\varphi}, \tag{3.12}$$

where τ, η, σ and, φ are constants, such that $\eta < 0$ and $\varphi < 0$. The values of these constants are determined empirically. The lat_{const} and lat_{slope} denote the base DMA latency and the rate of change in DMA transfer latency with respect to the DMA transfer size, respectively. The latency parameters for shared memory data differ from private memory data as,

- As the total shared memory is distributed physically, the DMA granularity is determined by the size of the smallest individual bank. Since we consider uniformly distributed banks in our work, the Dma_lat_{sh} varies as a function of $(\frac{s_{tot}}{N})$.
- The DMA transfer latency in the case of shared data is incurred only once when the shared data is brought into one of the on-chip shared memory banks for the first time. However, lat_{const} and lat_{slope} remain the same for both private and shared data accesses as these are properties of memory controllers and off-chip interconnection.
- Similarly, the execution time for extra compiler instructions is only incurred at the first accelerator requesting for the shared data. The constants σ and, φ account for this.

In addition to the latencies discussed before, access to a shared memory incurs additional latency in the form of on-chip communication to the memory bank, not associated with the requesting accelerator. The latency to access any remote shared memory bank significantly increases the overall shared memory access time and is the major limiting factor for indiscreetly increasing shared memory size. Hence, it is important to take the NoC latency into consideration while computing the shared memory access latency. Without loss of generality, we consider the average network latency to access any shared bank from an accelerator at design optimization stage. It is to be noted that the average network latency encompasses the overall network behaviour of a given heterogeneous accelerator-rich SoC architecture. In this work, we consider the interactions of different on-chip modules in our system architecture including all the fixed-function accelerators, host CPU processors and memory modules. We make the following assumptions to estimate the average NoC latency:

- The system architecture comprises of nodes connected by the NoC and each node is provided with a network router to send/receive messages as packets from other network nodes.
- The time taken to process a packet at each network router is deterministic.
- The routing algorithm used to route packets through the network nodes is deterministic.

- The nature of packet flow between the nodes resembles a Poisson process where the average arrival time between the packets is known while the exact packet arrival time is random.

Under the above assumptions, the system architecture is represented as an interactive platform of nodes acting as queues and we leverage Queueing theory [61] to determine the average packet waiting time at a node. As the packet traverses through multiple network routers to reach the destination memory bank, the assumption of packet flow as a Poisson process remains intact as shown by Kleinrock independence approximation theory [62]. It states that randomly merging several packet flows into a single packet flow will restore the independence of packet arrival times. Using this, the system is modelled as an M/D/1 queuing system, where the average waiting time (W) for a packet at any node is given by equation (3.13).

$$W = \frac{\rho}{2\mu(1-\rho)}, \tag{3.13}$$

where the utilization value, ρ is given by λ/μ. λ is the arrival rate of packets at a node and μ denotes the service rate of the packet at a node. Packet traversal in NoC is represented in terms of the number of hops and a packet traversing from source *A* to destination *B* through *n* intermediate routers will have *n + 1* hops. To compute the average access time to any shared memory bank, the average hop count of the network is considered and the network communication latency for computing total shared memory latency is given by equation (3.14) as follows.

$$NoC\_lat = H * (W + hop_{lat} + t_{exec}), \tag{3.14}$$

where *H* is the average hop count for the given network, hop_{lat} is the time taken to traverse the link between any two adjacent nodes and t_{exec} is the time taken by a node to service a packet.

Constraints: Equations (3.5) - (3.14) represent the total memory access time at an accelerator, which represents the performance objective function of the proposed *ASM* framework. This memory access performance is constrained by the available memory budget on the multi-accelerator system, accelerator design specifications and memory requirements, and are discussed as follows:

Budget Constraint: The available memory budget on the chip forms the major constraint limiting the achievable performance of the multi-accelerator system. The sum of all the accelerators' private memory and the total shared memory must be less than or equal to the given memory budget, *B*. We represent this constraint by equation (3.15) that poses a strict restriction on the total amount of on-chip memory available for the accelerators while designing an SoC.

$$\sum_{i=1}^{N} pr_i + s_{tot} <= B \tag{3.15}$$

Requirement Constraint: While the total on-chip memory is limited by the chip area budget, each accelerator, depending on its design requires minimum memory available for itself, below which its performance falls steeply. This minimum memory requirement, Req_i, is calculated for each accelerator by offline profiling with representative test input sizes to maintain its performance. While private-memory-only accelerator systems find it infeasible to meet this minimum memory constraint, shared memory architecture of *ASM* can satisfy this constraint in a performance efficient manner as each accelerator has access to both its private memory and entirety of the shared memory. To ensure that the obtained memory hierarchy satisfies Req_i for each accelerator A_i, the sum of each accelerator's private memory and total shared memory is constrained to be greater than or equal to Req_i as shown in the equation (3.16). This constraint will ensure that the total memory available from an individual accelerator's perspective is sufficient to maintain a fair performance improvement through hardware acceleration.

$$\forall_{i|i=1...N}, \quad pr_i + s_{tot} >= Req_i \tag{3.16}$$

Shared Constraint: Since in the shared memory multi-accelerator system, all shared data of each individual accelerator across different applications is stored on shared memory, there is a possibility of severe thrashing and performance degradation with small-sized shared memory. To minimize thrashing in shared memory, the total shared memory is constrained to be at least equal to the memory occupied by all the shared data, s_i, of an individual accelerator A_i. We present this constraint as equation (3.17), which will ensure that the size of shared memory is not negligible due to allocating more private memory to each accelerator. It also ensures that at any given point of time, the shared memory can hold at least all the shared data of any particular accelerator.

$$\forall_{i|i=1...N}, \quad s_{tot} >= s_i \tag{3.17}$$

Private Constraint: Finally, similar to the constraint on shared memory, to ensure that the private memory of each accelerator is sufficiently large so as to not severely impact its performance, a lower limit is imposed on all private memories as shown in equation (3.18). In this work, we consider pr_{min} to be 50% of the total private data size of each accelerator. This prevents the model from severely handicapping accelerators' private memory and performance to meet the budget requirements.

$$\forall_{i|i=1...N}, \quad pr_i >= pr_{min} \tag{3.18}$$

Resource optimization problem of *ASM* framework: Combining equations (3.5) to (3.14) into equation (3.4), the resource optimization problem of the proposed *ASM* framework is formally defined as shown in

equation (3.19).

$$min \sum_{i=1}^{N} T_{mem\_access\_i} \tag{3.19}$$

$$subject\ \ to,$$

$$\sum_{i=1}^{N} pr_i + s_{tot} <= B$$

$$\forall_{i|i=1...N},\ pr_i + s_{tot} >= Req_i$$

$$\forall_{i|i=1...N},\ s_{tot} >= s_i$$

$$\forall_{i|i=1...N},\ pr_i >= pr_{min}$$

where,

$$T_{mem\_access\_i} = \frac{(D_{pri} * Lat_{pri}) + (D_{shi} * Lat_{sh})}{D_{pri} + D_{shi}}$$

$$Lat_{pri} = Access\_lat_{pr} + \alpha(pr_i)^{\beta} + lat_{const} + lat_{slope} * pr_i + \delta(pr_i)^{\gamma}$$

$$Lat_{sh} = Access\_lat_{sh} + \tau\left(\frac{s_{tot}}{N}\right)^{\eta} + lat_{const} + lat_{slope} * \left(\frac{s_{tot}}{N}\right) + \sigma\left(\frac{s_{tot}}{N}\right)^{\varphi}$$
$$+ H * (\frac{\rho}{2\mu(1-\rho)} + hop_{lat} + t_{exec})$$

The resource optimization problem given in equation (3.19) yields a convex optimization problem. In our work, we solve it using a convex solver provided by MATLAB optimization toolbox. It uses the interior point algorithm [63, 64, 65] which is suitable for large-scale convex optimization problems with different design variables. The algorithm first starts with an initial feasible memory system configuration which meets all the constraints given in equation (3.19). It then checks for convergence and outputs the optimal solution if the configuration in-hand meets the tolerance level set for the constraints. If the conditions are not satisfied, then a search direction is computed and a backtracking line search is started with a decreasing step size depending on the decrease in merit function. Finally, after several iterations, the algorithm converges to a global solution which gives the best-suited memory system configuration that minimizes our objective function and meets all the constraints.

3.2.3.2 Performance-aware Data Allocation (*PDA*)

This Section describes the second stage of our proposed *ASM* framework that further optimizes our objective function.

The performance-aware data allocation stage takes the output of the DSO_{Mem} stage, specifically the resultant shared memory configuration, and efficiently allocate the application data to the distributed shared memory banks. Unlike an accelerator's private memory, data allocation on the distributed shared memory banks needs to consider the interference caused by various interactive parts of an application running on different accelerators. For instance, when data variables required by two different accelerators executing simultaneously are mapped to the same shared memory area, it will result in unwanted thrashing. On the other hand, two accelerators that execute independently over time can share the same memory area using overlay. Hence, by employing an efficient method for shared data allocation, we can achieve better space utilization and reduce thrashing.

Recall that, the execution time of an application in a multi-accelerator system is dominated by the memory access time. Hence, in this stage, we wish to minimize the shared memory access time by carefully scheduling the placement of shared data at compile time on the distributed shared memory system. We define our shared data allocation problem as follows.

Given a set of data items, V, and a set of shared memory banks, U, with,

- c_{pq} = *cost of placing a data item q into a shared memory bank p,*
- w_{pq} = *memory occupied by a data item q when placed in a shared memory bank p,*
- cap_p = *capacity of shared memory bank p,*

assign each data item to exactly one shared memory bank so as to minimize the total cost incurred due to the shared memory access latency, without assigning to any shared memory bank more data variables than its capacity, i.e.

$$min \sum_{p=1}^{u} \sum_{q=1}^{v} c_{pq} x_{pq} \tag{3.20}$$

$$\begin{aligned} \textit{subject to}, \quad & \sum_{q=1}^{v} w_{pq} x_{pq} \leq cap_p, && p \,\epsilon\, U = \{1, 2, ..., u\}, \\ & \sum_{p=1}^{u} x_{pq} = 1, && q \,\epsilon\, V = \{1, 2, ..., v\}, \\ & x_{pq} = 0 \textit{ or } 1, && p \,\epsilon\, U, q \,\epsilon\, V, \end{aligned}$$

where,

$$x_{pq} = \begin{cases} 1 & \textit{if data item q is placed in memory bank p;} \\ 0 & \textit{otherwise.} \end{cases}$$

The algorithm for efficient data allocation on the distributed shared memory banks is presented in *Algorithm 1*. The aim of the algorithm is to output a combination of data to shared bank mapping such that it minimizes our objective function presented in equation (3.20). We calculate the cost, c_{pq}, in terms of the latency incurred by the accelerators to access the data q when placed in shared memory bank p, by using equation (3.9). Here, we consider the highest latency among all the accelerators accessing the same data variable.

3.3 Experimental Results

In this section, we present the experimental setup and evaluation results of our proposed *ASM* framework.

3.3.1 Experimental Setup

An overview of the evaluation setup for the *ASM* framework is shown in Figure. 3.4. We extended gem5-Aladdin [57] and Noxim [66] simulators to incorporate fixed-function accelerators along with CPU cores and support memory sharing between the accelerators. gem5-Aladdin applies various optimization techniques on high-level application codes to generate highly optimized fixed-function accelerators [57]. These optimized accelerator designs provide orders of magnitude of increase in performance as compared to their plain software counterparts. We leverage the existing gem5-Aladdin platform to generate all the fixed-function accelerators employed in our experiments. We perform offline profiling of all the accelerator kernels to understand their inherent characteristics and memory access behaviours. Representative input data sizes are considered while performing offline profiling of multi-accelerator benchmark kernels as employed in evaluating optimization frameworks for accelerator-rich systems [29, 67]. The characteristics of the accelerator kernels are then fed to the DSO_{Mem} stage of *ASM* to provide an optimized configuration for accelerator memory subsystems; followed by the *PDA* stage for efficient data allocation which is implemented as a compile-time algorithm within the framework. Then, the optimization toolbox of MATLAB is used to obtain the desired solution. In order to accurately model the communication latency incurred from on-chip network contention, we employ Noxim that provides a cycle-accurate network simulation for on-chip traffic. Hence, the application memory access traces are collected from gem5-Aladdin and given as input to Noxim, as shown in Figure. 3.4. Such trace-based simulation environments have been widely used to capture the impact of on-chip networks on system performance [30, 20].

Table 3.2 presents the system configurations used in our experimental evaluation. Along with the fixed-function accelerators, we simulate two x86 CPU cores to manage the accelerator cores, as implemented in [11]. The memory controllers are equipped to manage all the accelerator memory accesses and a small TLB is used for virtual to physical address translation. To maintain synchronization between multiple accelerator accesses

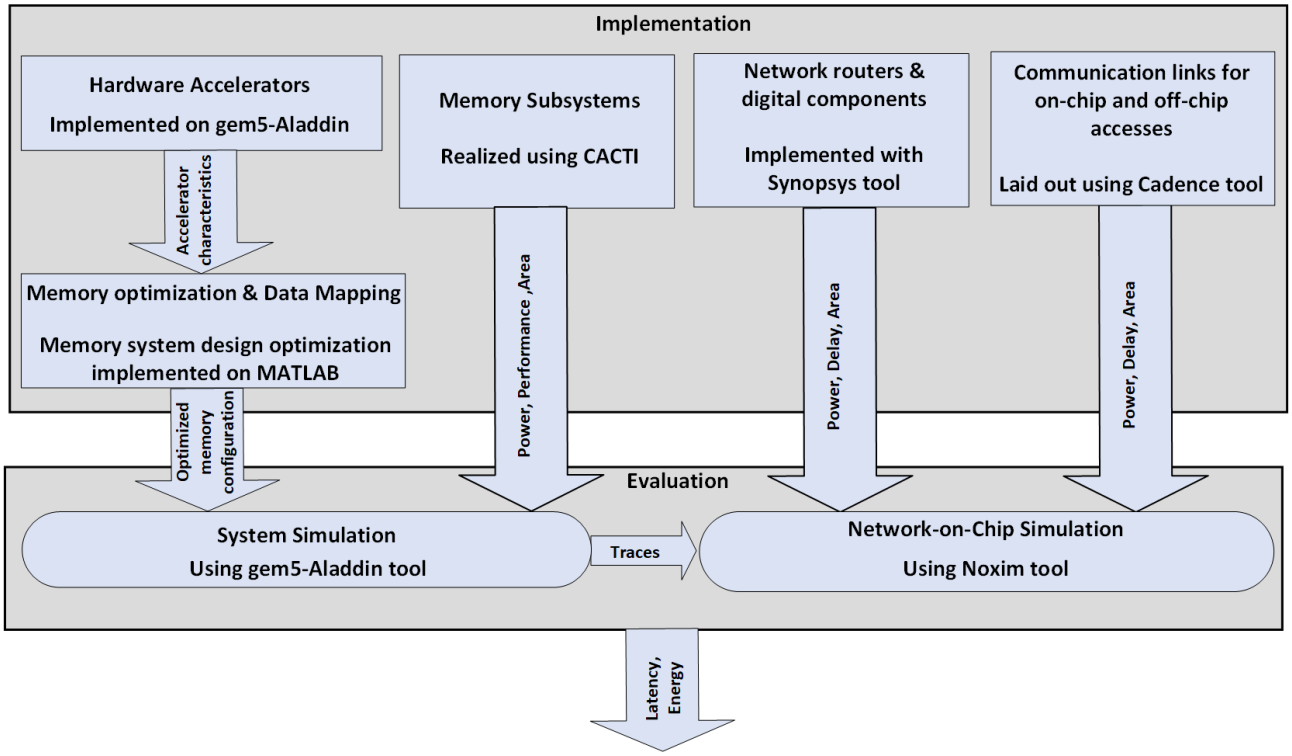


Figure 3.4: Experimental setup for *ASM* framework.

and proper computation flow, read-enable and write-enable flags are used at the shared memory blocks. For instance, when accelerator B produces an output which becomes the input to accelerator A, A needs to stall until the write-enable flag on the shared block is reset and accelerator B has finished writing data to the shared space. The latencies and energy consumptions of the on-chip SPMs and off-chip DRAM were attained from Cacti [68]. The 2D Mesh-based network uses the XY routing algorithm to facilitate communication between the on-chip modules. We have adopted wormhole switching and 3 stage pipeline for the network routers. Synopsys Design Compiler is used to synthesize the network switches present at each network node. The latencies and energy dissipation on communication links are obtained through Cadence simulations. All the latency and power numbers along with the optimized memory system configurations are back-annotated into the tool-chain, as shown in Figure 3.4, to obtain the overall performance and energy efficiency of *ASM* framework.

We evaluate *ASM* in terms of application performance and energy consumption. Since *ASM* does not affect computation structure of accelerators, we consider the performance evaluation with regard to communication

Table 3.2: System Configurations for ASM framework evaluation

Components	Configurations
Processing Cores	2 x86 host cores; Customized fixed-function accelerator cores
Memory Systems	On-chip SPMs; 2 GB Off-chip DRAM
On-chip Network	2D Mesh topology, XY routing

latency incurred by the applications and is driven by equation (3.21).

$$Latency = N_{pr} * L_{pr} + N_{sh} * L_{sh} + N_D * L_D, \tag{3.21}$$

where, N_{pr}, N_{sh} and N_D denotes the number of private memory accesses, shared memory accesses and DRAM accesses respectively. Similarly, L_{pr}, L_{sh} and L_D are the latency of accessing private memory, shared memory and DRAM respectively. The latency of accessing private and shared memories are calculated using equations (3.5) and (3.9) respectively.

The total energy consumption is calculated using equation (3.22), where E_{pr}, E_{sh} and E_D denotes the energy use of private memory, shared memory and DRAM respectively.

$$Energy = N_{pr} * E_{pr} + N_{sh} * E_{sh} + N_D * E_D \tag{3.22}$$

In case of on-chip accesses to remote shared nodes, the total energy is equal to the energy dissipated to reach the remote node through the on-chip network and the shared memory resource itself. The energy consumption to access *i*th remote node is given by equation (3.23).

$$E_{sh\_i} = (L_i - L_{wi}) * E_{buf} + h_i * E_{wire} + E_{shr} \tag{3.23}$$

where, L_i is the latency of *i*th access, L_{wi} denotes the latency of the packet traversal in the wired links from source to destination (without considering the buffer time), h_i is the number of hops in the path of the access, E_{buf} is the energy dissipation of a network packet in the NoC switch buffers per cycle, E_{wire} is the energy consumption of one wireline hop between any two nodes and E_{shr} denotes the energy dissipation of the shared memory itself.

3.3.2 Benchmark Application Kernels

The high-level characteristics of the accelerator kernels used in our experimental evaluation are presented in Table 3.3. We characterized a total of 19 accelerator kernels. *AES*, *Viterbi*, *CONV*, and *FFT* kernels are incorporated from the accelerator benchmark suite MachSuite [69]. A few application kernels like $Susan_{corners}$, $Susan_{edges}$, $Susan_{smoothing}$, and *JPEG* from MiBench suite [70] are modeled as fixed-function accelerator kernels on the gem5-Aladdin platform. We also include a set of representative image processing kernels and optimize the application code in the gem5-Aladdin platform to model them as hardware accelerators. Table 3 also shows the memory footprint of all the accelerator kernels after loop tiling, a common practice for

Table 3.3: Benchmark application kernels.

Accelerator Kernels	Application Domain/Description	Total instructions	Loads/Stores	Memory footprint(KB)*
AES	Encryption(256−bit keys and blocks)	3073577	1455345	185
Susan_corners(SC)	Image recognition (corner detection)	106142911	49122353	365
Susan_edges(SE)	Image recognition(edge detection)	106242341	49314275	365
Susan_smoothing(SS)	Image recognition (brightness/spatial control)	305981901	68998973	360
FFT	Digital signal processing	5262329	2026351	220
Denoise	Image processing (noise reduction)	360129019	78110898	342.12
Color correction(CC)	Image processing (visual tone change)	310872200	76009866	338.22
CONV-I	Image recognition(neural networks)	426249941	237874982	406
Viterbi	Speech recognition	660241220	295060305	450
White balancing(WB)	Image processing (color shift rectification)	344667333	74695505	312.79
Auto levelling(AL)	Image processing (corrects exposure)	340852015	72530843	302.08
Color interpolation(CI)	Image processing (missing data generation)	355858910	71189768	330.23
JPEG	Image compression	4910871	2093182	347
Color-space converter(CSC)	Image processing (color-space transformation)	370188911	75610765	315.19
Tone reproduction(TR)	Image processing (brightness reproduction)	320779034	71015692	344.33
Color enhancement(CE)	Image processing(retrieve finer useful details)	353482025	77802713	342.88
Gamma correction(GC)	Image processing (compensating non-linearities)	354548994	61783374	340.04
CONV-II	Speech recognition (neural network)	428743451	85699801	380
Edge Enhancement(EE)	Image Processing (Finer edge details)	374836422	52384599	352.55

* Memory footprint represents the working set sizes of each accelerator kernel, while the global data sizes are in order of mega-bytes.

scratchpad-based many-core systems [67]. The memory footprint of these accelerator kernels varies from 185KB to 450KB. The total number of instructions varies from 3 million for *AES* to 660 million for *Viterbi*, where load/store instructions range from 1 million to 295 million across the accelerator kernels. As accelerator-rich architectures are most suitable for domain-specific computing, we assume that designers will fabricate a multi-accelerator chip according to the applications that might run on the designed platform. Hence, we use four case studies that incorporate the characterized fixed-function accelerators for domain-specific applications to test the efficacy of our framework. Table 3.4 presents the set of accelerator kernels employed in each of the case studies. The first case study (CS1) is a representative image processing application employed in the digital imaging systems following a similar Image Processing Pipeline (IPP) presented in [71]. The IPP, as shown in Table 3.4 comprises of 14 different kernel functions that we choose to run on hardware accelerators to improve the application performance. Although all the accelerators work in a sequential manner(following the same order as in Table 3.4) on same input segment, we were able to tile input data and pipeline the computations; thus allowing all the accelerators to be active through most of the runtime. The case study 2 (CS2)

Table 3.4: Accelerator kernels in various case studies

Case Study	Accelerator kernels
CS1	Auto levelling (AL), White Balancing (WB), Color Interpolation (CI), $Susan_{edges}$ (SE), Denoise, Color Correction (CC), Color space converter (CSC), Edge Enhancement (EE), Color Saturation Enhancement (CE), Tone Reproduction (TR), Gamma Correction (GC), JPEG
CS2	JPEG (6), AES(6)
CS3	FFT, Viterbi, CONV-I,$Susan_{corners}$ (SC), $Susan_{edges}$ (SE), $Susan_{smoothing}$ (SS), CONV-II, Denoise
CS4	JPEG(20), AES(20)

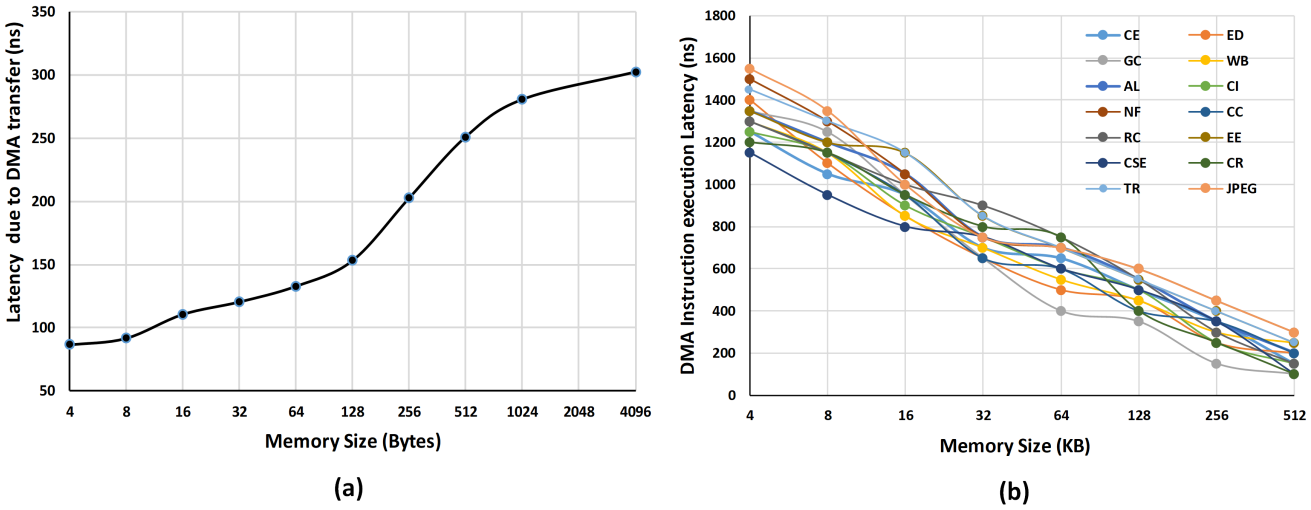


Figure 3.5: Latency vs memory size. (a) The change in DMA transfer latency with respect to the increase in DMA transfer size as a function of memory size; (b) The change in latency due to execution of the extra compiler instructions for DMA transfer (for case study 1) with increasing memory size.

represents a distributed processing system at the edge devices that are highly parallelized as compared to the deep serial pipeline of a digital imaging system discussed in CS1. The system comprises of 6 copies of each JPEG and AES accelerators (shown as AES(6) in Table 3.4) working on different input images in parallel. Such a system is suitable for scenarios where the edge device is responsible for capturing continuous image frames, compress using JPEG accelerators and encrypt using AES accelerators to upload them on the cloud for processing. The case study 3 (CS3) represents modern mobile SoCs that are equipped with various processing kernels supporting multiple parallel operations. The CS3 comprises 16 nodes which includes accelerator kernels like *FFT*, *Viterbi* and *Neural Engines* to support faster and accurate speech recognition. It also includes image enhancement kernels ($Susan_{corners}$, $Susan_{edges}$, $Susan_{smoothing}$) that parallelly processes the incoming image frames from the camera sensors along with convolution kernels to facilitate image classification and recognition. To test the robustness of our framework, we scale CS2 by incorporating 20 copies of each JPEG and AES accelerators in case study 4 (CS4).

3.3.3 *ASM* Evaluation Results

In this Section, we evaluate the performance improvement achieved by our proposed *ASM* framework.

3.3.3.1 Memory Access Characterization

We evaluate the sensitivity of application runtime to the change in DMA transfer message size. As discussed, large DMA bursts avoid the overheads due to a large number of small DMA requests. Hence, data movement

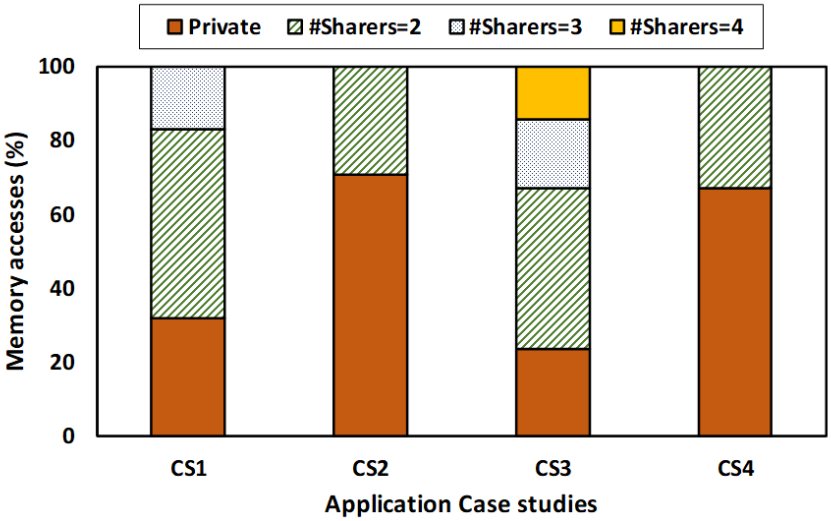


Figure 3.6: Distribution of memory accesses across the case studies.

from off-chip to on-chip memory and vice versa are carried out at whole memory stack space granularity. As a result, we have a correlation between the DMA transfer sizes and the on-chip SPM sizes. Figure 3.5.a shows the trend of DMA operation latency with respect to the increase in the DMA transfer size as a function of memory size. As the memory size increases, DMA burst size increases and more DMA transfer latency is incurred. As shown in Figure 3.5.a, for a memory size of 4KB, the DMA operational latency is 300 ns whereas a small memory size of 4B incurs 90 ns of overhead. Hence, the DMA transfer latency significantly contributes to the application runtime, as the accelerators have to stall for the required data to start its execution, and this trend is captured by equations (3.7) and (3.11) in our framework.

To initiate the DMA transfers between off-chip and on-chip memory subsystems, the original application code is transformed and custom compiler instructions are added in between the original code. As discussed earlier, the latency incurred to execute these extra compiler instructions add to the overall application execution time. We present the evaluation for CS1 to understand the application characteristics with respect to these additional instructions. Figure 3.5.b shows the trend of latency incurred due to extra compiler instructions with increasing memory size, which is accounted by equations (3.8) and (3.12) in our framework. Here, we show the variation of the incurred latency for all the accelerator kernels incorporated in CS1. As seen in Figure 3.5.b, as memory size increases the additional latency incurred decreases. This trend is due to the fact that a larger memory size will force a large DMA burst which will reduce the number of extra compiler instructions to be executed to facilitate DMA transfers. We perform a similar analysis for the port contention latency for all the applications.

We characterize the memory accesses of all the case studies and present their sharing degree in Figure 3.6. The Y-axis represents the distribution of memory accesses to data variables with varying degrees of sharing. For

instance, the brown segment of the bar graph constitutes the percentage of private memory accesses; whereas, the green segment shows the percentage of accesses to variables with two sharers. As evident from the figure, CS2 and CS4 have a significant amount of accesses to private data with remaining accesses to memory space shared between two accelerators. In the case of CS1, we observe at most three sharers across all data variables. For CS3, the memory accesses comprises of varying degrees of sharer cores and the maximum degree of sharing across all the data variables is observed to be four. Hence, the case studies selected for our experimental evaluation caters from low to high degree of data sharing between different accelerator cores.

3.3.3.2 Baseline Architecture

To understand the benefits of shared memory architectures for accelerator-rich systems, we compare *ASM* with a private-memory-only baseline design. The private-memory-only baseline architecture represents a heterogeneous SoC comprised of general-purpose cores and multiple fixed-function accelerators. All the on-chip modules are connected by an NoC. Each accelerator has its own dedicated memory space to load data from off-chip memory. In our experiments, we uniformly divide the total accelerator on-chip memory budget among all the accelerator nodes. An application is run on the general-purpose core that invokes multiple accelerators during its execution. Whenever the application needs to offload computation on the hardware accelerators, it generates a request, and the software manager is responsible for initializing the particular accelerator. After a set of data is loaded onto the accelerator's memory space, it can start its execution. The resultant data are then stored back to the main memory through off-chip DMA transfer. If any intermediate data need to be shared between any two accelerators, they have to accomplish it via off-chip memory. For instance, if accelerator *B* needs an input that is produced by accelerator *A*, *B* have to stall until *A* finishes its store operation on the off-chip memory. Only after *A* has successfully written onto the main memory, *B* can issue an off-chip DMA load operation.

3.3.3.3 Application Runtime

Sharing on-chip memory subsystem among the accelerators leads to better application performance as compared to a private-memory-only baseline architecture. Figure 3.7a shows the improvement in application runtime of *ASM* framework with respect to the private-memory-only baseline architecture. Across all the case studies, *ASM* provides an average performance improvement of 25.42%. The application runtime achieves the speedup due to the reduced off-chip memory accesses in the presence of a shared memory architecture. A maximum improvement of 34.35% is observed for CS3 due to the inherent nature of the application where more than 70% of the memory accesses are to the shared variables. Similarly, in CS1, majority of the inter-accelerator

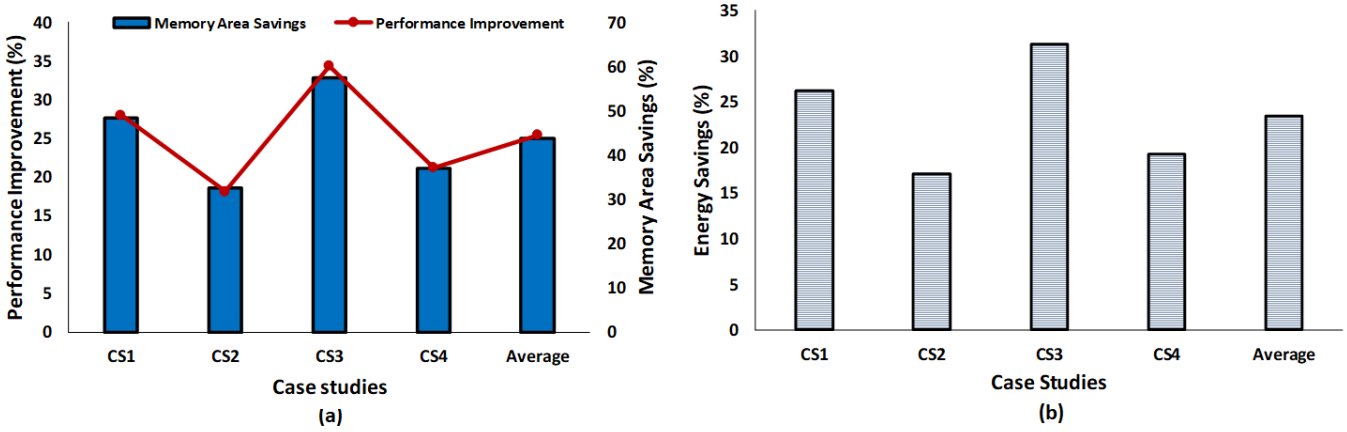


Figure 3.7: *ASM* evaluation against private-memory-only baseline for different case studies; a) Performance improvement and area savings; b) Energy Savings.

communications are dominated by producer-consumer type due to the sequential nature of IPP. By mapping the shared variables for producer-consumer accesses on the shared memory space, *ASM* reduces shared-data replication and thus the off-chip memory accesses. Further, the imposed memory budget constraint results in smaller accelerator memories, which increases off-chip memory accesses of the intermediate accelerators and highly affect application performance in a private-memory-only architecture. Whereas *ASM* (guided by the constraints in equations (3.15) to (3.18)) performs a meticulous selection of private/shared memory configurations to minimize the overall execution time. Since most of the intermediate accelerators are able to take advantage of the shared memory space, the intermediate data transfer cost is reduced. In case of CS2 and CS4, where there is relatively less scope of memory sharing, we still observe runtime improvement of 18.15% and 21.23% respectively with the memory subsystem and data allocations suggested by *ASM* over the baseline.

We evaluate the performance of *ASM* in the worst-case scenario, where the system encounters applications with no data sharing among the accelerators and the on-chip memory is insufficient to maintain accelerator performance. We create a modified version of all the case studies presented in Section 3.3.2. We run all the accelerators individually in each case study with no inter-accelerator communication. Hence, there is no need for data sharing between the accelerators. However, along with their private memory space, the shared memory space is time-shared and used to hold accelerator-specific data. Figure 3.8 shows the performance improvement (application runtime) of *ASM* against the baseline in the presence and absence of shared data among the accelerators. As evident from the figure, the presence of shared data provides relatively more performance improvement across all the case studies. This is due to the fact that the accelerators are able to efficiently utilize the shared memory space and decrease costly off-chip memory accesses. However, it is to be noted that, even in the worst-case scenario where there is no shared data, the *ASM* framework still provides an average performance improvement of 12.12% over the baseline private-memory-only system. This

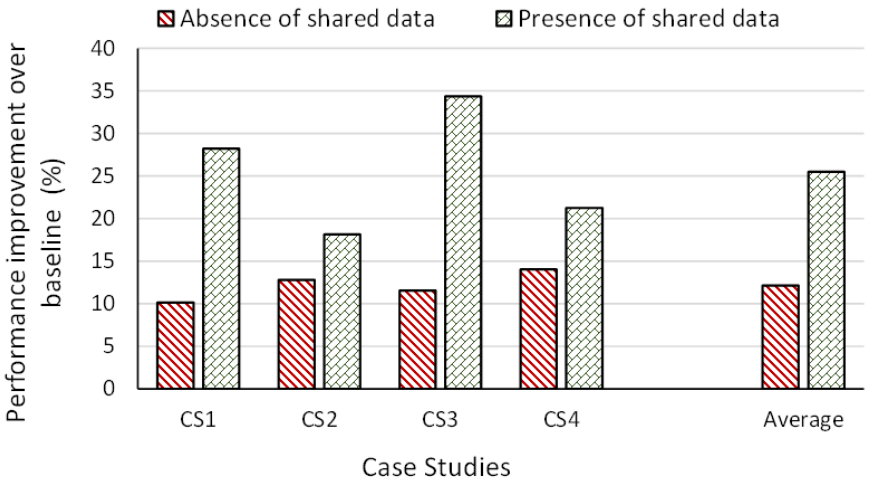


Figure 3.8: *ASM* evaluation against private-memory-only baseline for different case studies in the presence and absence of data sharing among the accelerators.

performance improvement is because of the private/shared configuration along with the efficient data allocation method provided by *ASM*. Due to the shared memory, each accelerator perceives more on-chip memory space and can utilize this space to store individual data whenever available. Furthermore, the data allocation strategy of *ASM* makes optimal use of the available on-chip memory space through time-sharing and latency-aware mapping, thus reducing thrashing and minimizing performance overhead.

3.3.3.4 Area Savings

To evaluate the effect of sharing memory on the chip area, we gradually increase the memory budget of private-memory-only baseline to match the application performance achieved using private/shared memory design suggested by *ASM*. As evident from Figure 3.7.a, sharing memory across the accelerators translates to saving a significant amount of memory area on chip. The area savings range from 32.51% to 57.32% across all the case studies. *ASM* makes an informed decision for sharing memory by analyzing the memory access characteristics of each case study, saving up to 57.32% in CS3, which has the maximum number of accesses to the shared variables. On average, *ASM* is able to save 43.77% of on-chip memory area for all the case studies considered in our experimental evaluation.

3.3.3.5 Energy Savings

Figure 3.7.b presents the energy savings achieved due to private/shared memory configurations and data allocations suggested by *ASM* over the private-memory-only baseline. Here, the total energy is calculated by equation (3.23) presented in Section 3.3.1. As depicted in Figure 3.7.b, *ASM* reduces energy consumption by 23.48%

on average over the baseline. The energy-saving is due to a decrease in costly off-chip memory accesses in the presence of the shared memory space. It is also attributed to the lower router buffer energy and wire energy as a result of an overall smaller active period of the network from reduced runtime. The energy savings range from 17.15% to 31.34% across all the case studies.

3.3.4 Comparison with Existing works

In this Section, we compare the *ASM* framework with representative accelerator-rich systems that provide a platform for shared memory architecture.

3.3.4.1 Representative Accelerator-rich Systems

In our experimental evaluation, a constant area budget constraint is considered for all the designs to make a fair comparison.

AS Store Baseline Architecture. AS provides a platform to allow dynamic memory sharing between multiple accelerators [28]. It comprises of a few CPU cores, multiple accelerator cores with private memories and a collection of SRAM banks that facilitate sharing among the accelerators. All the accelerator cores allocate their shared buffers on the SRAM banks if available. Allocation of accelerator buffers on the shared memory is managed through handle-IDs and a priority table. The accelerators thus use the handle-IDs to communicate among themselves through the shared memory. In our experiments, we partition the AS shared memory into multiple banks distributed across a 2D Mesh-based on-chip network. This increases the number of AS channels or ports available to access the shared memory.

BiN Baseline Architecture. BiN dynamically allocate buffers of co-existing accelerators in a non-uniform cache architecture (NUCA) [11]. It uses a dynamic interval-based global allocation technique to facilitate accelerator buffer allocation. To reduce the effect of buffer fragmentation, BiN also proposes a flexible paged buffer allocation method. Unlike the AS architecture, BiN proposes a unified NUCA and shared buffer architecture, so that the same space can be utilized as a shared buffer or a cache depending upon the number of active accelerators. In their experimental evaluation, best application performance was observed by limiting the floating boundary between cache and accelerator buffers to half of the NUCA size. Hence, in our experiments, we also set an upper bound for the shared buffer size by restricting buffer allocation to at most half of the NUCA size.

3.3.4.2 Application Runtime

Figure 3.9.a shows performance improvement of *ASM* with respect to the representative accelerator-rich baseline designs. We also compare *ASM* with a system combining the memory configurations suggested by *ASM's*

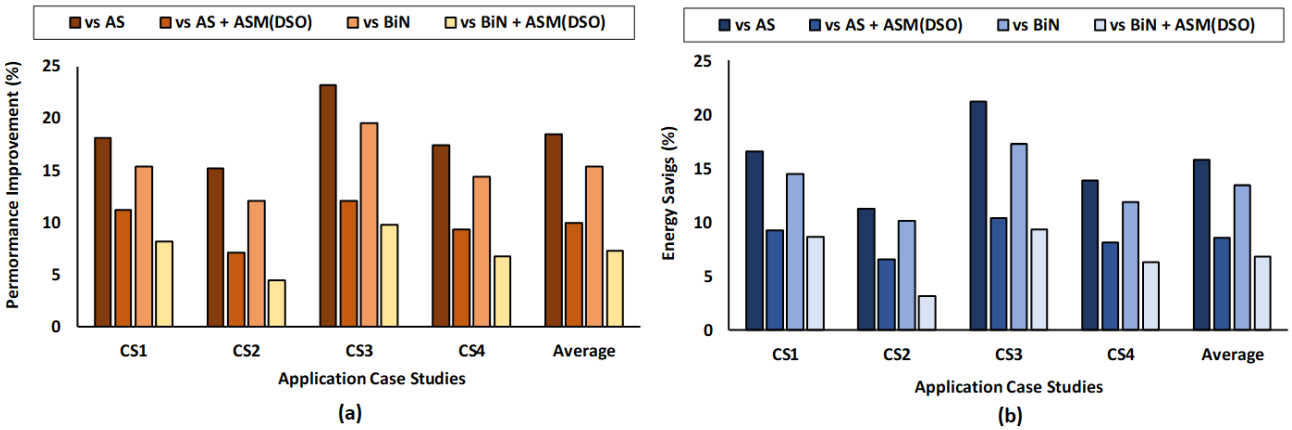


Figure 3.9: *ASM* evaluation against representative accelerator-rich baseline architectures for different case studies; a) Performance improvement; b) Energy Savings.

DSO_{Mem} stage and the allocation strategies presented in the baselines. As shown in Figure 3.9.a, the results depict up to 18.51% average performance improvement by *ASM* across all the case studies against the baseline designs. The performance improvement is achieved due to the reduction in costly off-chip DMA transfers as well as the contention aware memory design and data allocation provided by *ASM*. Although the *AS* baseline architecture provides shared SRAM banks for accelerator buffer allocation, it does not consider the communication latency involved in accessing remote shared banks. In case of a significant amount of shared data accesses from multiple accelerators, the on-chip network contention becomes a challenge which degrades application performance. By carefully selecting the best-suited memory configurations and data allocations under NoC contention and memory area constraints, *ASM* is able to provide up to 18.51% average performance improvement for all the case studies. As shown in Figure 3.9.a, application runtime improvement of 7.15% to 12.11% is observed by *ASM* against the combination of DSO_{Mem} stage and *AS*'s shared memory framework. Similarly, we also observe an average performance improvement of 15.36% against the *BiN* baseline architecture and up to 9.78% improvement while coupling *BiN*'s accelerator buffer allocation method with DSO_{Mem} stage of *ASM* framework. Hence, we identify that the contention-aware memory system configurations and data-allocations recommended by *ASM* enhance the efficiency of shared memory-based accelerator systems as compared to the *AS* and BiN baseline designs.

3.3.4.3 Energy Savings

Figure 3.9.b shows the energy savings achieved by the proposed *ASM* framework over the representative accelerator-rich baseline architectures. The average energy savings achieved by the *ASM* framework across all the case studies against the baselines is observed to be up to 15.72%. Besides the energy consumption due to off-chip memory accesses, the on-chip network communication also contributes to the overall system energy consumption. By considering on-chip communication latency while generating the memory system configura-

tion and data allocations, *ASM* is able to reduce contention at the shared memory and on-chip network. The reduced network contention leads to reduced overall network active time, which translates to low buffer and wire energy dissipation. As a result, we observe up to 16.55% of energy savings by *ASM* when compared to *AS* and *BiN* baseline architectures.

3.3.5 Validation of the proposed framework

In this Section, we investigate the efficacy of our proposed *ASM* framework to provide the best-suited memory configurations for a multi-accelerator heterogeneous system. We perform a pareto-optimal analysis to compare the performance improvement and energy savings of the memory configuration obtained by *ASM* to other possible private/shared memory configurations. We sweep through all the possible private/shared memory configurations for the different accelerators employed in our case studies by keeping a constant memory budget. The optimal memory configurations suggested by *ASM* is presented in Table 3.5. As each accelerator kernel has different memory requirements and sharing degrees, the memory space allocation varies accordingly. *ASM* provides the private/shared memory subsystem configurations depending on the amount of shared data available between the accelerators. For the workloads that require the least communication between the accelerators, *ASM* will provide mostly private-memory based system with a suitable shared memory size. On the contrary, with a large amount of shared data among accelerators, *ASM* will allocate a substantial part of the memory budget as a shared memory. Hence, *ASM* provides best-suited memory system configurations and data allocation depending upon the workload in-hand. Figure 3.10.a shows the performance-energy pareto-optimal graph of different possible memory hierarchies of a multi-accelerator system for CS1. Each small red rhombus in Figure 3.10.a corresponds to different performance and energy values for the application at various memory subsystem configurations. For brevity, we only show a few design points. The performance and energy values of all the memory configurations are normalized to the private-memory-only baseline which is shown by the small circle in Figure 3.10.a. The bold black area of the curve in the figure shows the pareto-optimal frontier which represents a set of memory configurations with the best trade-off between performance improvement and energy savings over the baseline. We consider all the design points at the pareto-optimal frontier and calculate the loss between the *ASM* generated configuration, shown by a small square in Figure 3.10.a, and the average of pareto-optimal configurations. Figure 3.10.b shows the loss in performance and energy savings of the suggested configuration by *ASM* as compared to the average pareto-optimal configurations with 4.12% and 4.77% average loss in performance and energy, respectively. Furthermore, the loss in performance and energy savings in case study 4 is found to be 5.18% and 5.77% for performance and energy, respectively, while compared to the average pareto-optimal configurations, which demonstrates the scalability of our framework. From the above discussions, we have the following observations and conclusions.

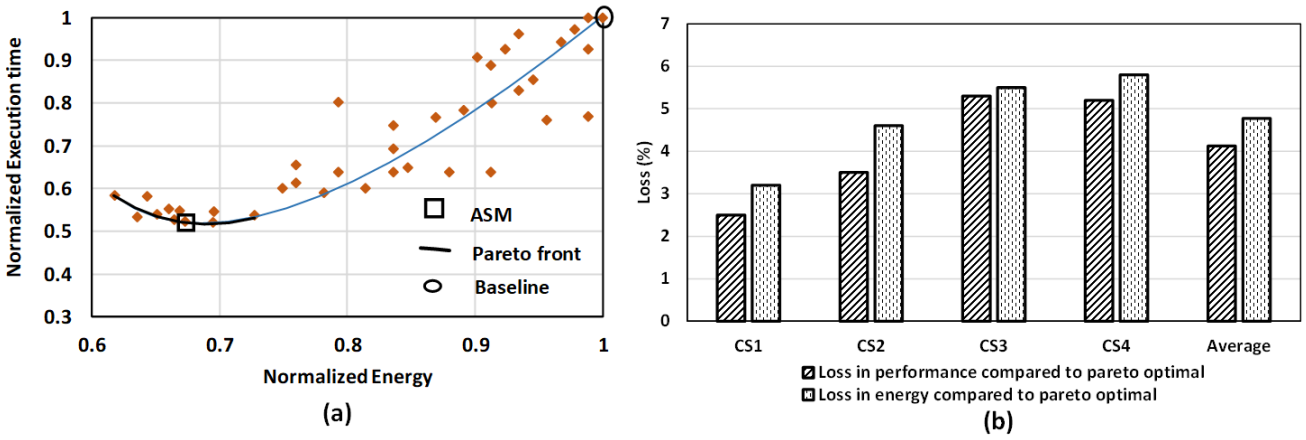


Figure 3.10: Framework validation.(a) Pareto analysis for execution time and energy consumption of *ASM* memory configuration and other possible configurations normalized to private-memory-only baseline for case study 1; (b) Loss in performance and energy of the suggested configuration by *ASM* framework as compared to the average pareto-optimal configurations.

Table 3.5: Memory distribution by *ASM* among all the accelerators for different case studies.

Case Study	Accelerator Private Memory																			Shared Memory	Total Budget
	AES	SC	SE	SS	FFT	Denoise	CC	CONV-I	Viterbi	WB	AL	CI	JPEG	CSC	TR	CE	GC	CONV-II	EE		
CS1	–	–	68	–	–	56	52	–	–	52	64	58	64	50	64	60	58	–	64	1.16MB	2MB
CS2	64	–	–	–	–	–	–	–	–	–	–	–	86	–	–	–	–	–	–	590kB	1.5MB
CS3	–	64	64	60	32	54	–	68	78	–	–	–	–	–	–	–	–	58	–	1.03MB	2MB
CS4	48	–	–	–	–	–	–	–	–	–	–	–	64	–	–	–	–	–	–	752kB	3MB

- Our proposed *ASM* framework produces the best-suited memory configurations for a multi-accelerator system under a tolerable error range against pareto-optimal configurations, with respect to performance improvement and energy efficiency.
- *ASM* delivers a solution to our optimal memory system design problem without going through the exhaustive simulation process to perform pareto-optimal analysis. As a result, it provides a best-effort method to meet the TTM demands of the embedded systems.

3.4 Conclusion

In this chapter, a design optimization framework, ASM framework, is proposed to realize an optimized memory system configuration for accelerator-rich platforms. We perform an extensive analysis of memory access behaviour of all the accelerators invoked by an application during its run-time. The ASM framework achieves an efficient on-chip memory utilization by taking into account the accelerator's private and shared data information and their effect on the on-chip network contention. Our experimental study shows up to 34.35% performance improvement and 31.34% energy savings as compared to the baseline architectures. The framework is also validated against the pareto-optimal frontier of other possible memory configurations and on average,

ASM achieves less than 4.12% and 4.77% loss in performance and energy savings respectively.

Chapter 4

Securing an Accelerator-rich System from Flooding-based Denial-of-Service Attacks

The growing system sizes and Time-To-Market (TTM) pressure of Accelerator-rich SoCs (ArSoCs) compel the chip designers to analyze only part of the design for optimization, forcing the designers to source highly optimized custom-designed accelerator Intellectual Properties (IPs) from third party vendors. The third-party accelerators integrated together with other system modules can lead to various security threats. The lack of access to the design details of such third-party accelerator blocks makes it difficult to validate them and ensure safety. Even with the design details, it is infeasible to perform exhaustive explorations of millions of logic elements to detect a possible security threat due to design flaws or any malicious modifications like Hardware Trojans (HT) [6]. In this chapter, we address the security threat on accelerator-rich SoCs that utilizes the shared interconnect network to mount an attack and degrade the overall system performance.

4.1 DoS Attack on Accelerator-rich Heterogeneous Systems

In an ArSoC, where all the on-chip modules communicate with each other through the Network-on-Chip (NoC), a Malicious Third-party Accelerator (M3PA) can initiate an attack on other legitimate on-chip modules through the commonly shared communication platform. An M3PA can trigger a Denial-of-Service (DoS) attack by continuously injecting useless packets on the NoC and flooding the network. This results in high network congestion and leads to communication latency. A flooding attack by an M3PA resulting in DoS will severely degrade the overall system performance defeating the purpose of using accelerators in the first place. Therefore, it is of utmost importance to efficiently detect flooding attacks in an ArSoC at an early stage and meet the performance requirement of the applications.

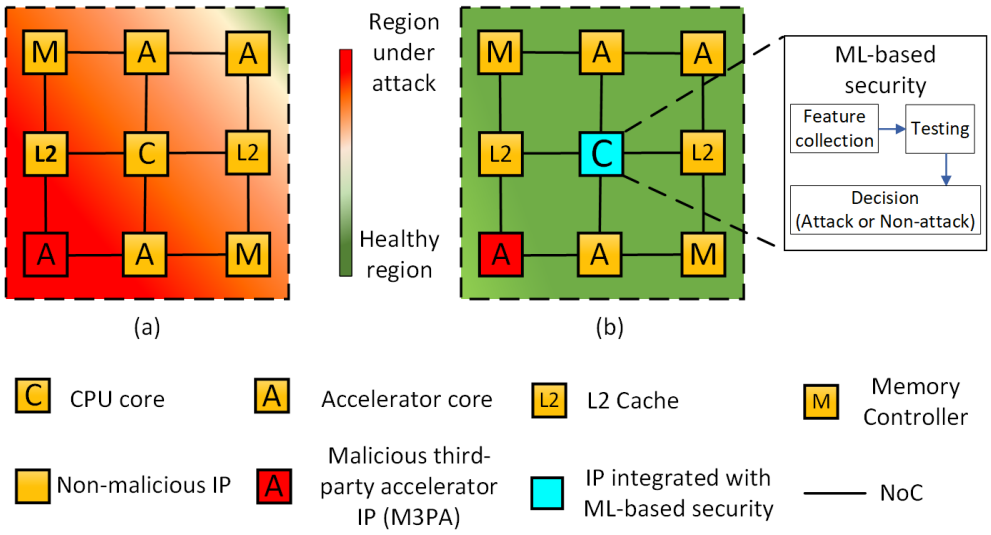


Figure 4.1: An accelerator-rich heterogeneous system where an M3PA creates a DoS attack. a) System under attack with no security mechanism, b) Restored system with proposed ML-based security mechanism.

Although significant work is carried out to address the design challenges for ArSoCs [72], there are very few works addressing the security threats caused by the integration of M3PAs in an ArSoC. Authors in [36] and [37] provide methods to mitigate attacks which exploit information flow causing data leakage, while in [35], a security mechanism is presented to prevent the system from memory corruption. However, some elusive HTs can escape the verification process and create a DoS attack which might not be captured by the above methods. Multiple research works [73, 74] have shown the importance of addressing the security threats while designing the NoC-based SoCs. Other prior works addressing the specific attack like DoS attacks in the NoC based multi-core systems proposed packet latency tracking for attack detection [38] and security verification methods [42], which impose communication and hardware overheads. In [39], a security monitoring system was presented to detect DoS attacks based on bandwidth deviations, whereas [40] used statically tuned network parameters to distinguish between attack and non-attack network traffic. Such attack detection techniques manually set thresholds for the chosen parameters based on empirical observations [39, 40]. However, during real-time operations, different applications co-exist at different times over the entire execution period. To take advantage of such varied workload condition, a number of runtime optimization methods have been proposed like efficient thread-to-core mapping [30, 75], DVFS [30, 76], power gating[28] etc., which provide utmost system performance while staying within power budgets. As a result of these optimizations, a wide variation is observed in system state and network-level activities of an ArSoC during its runtime. Hence, under such a dynamic environment, manual threshold setting of metrics to distinguish between attack and non-attack scenarios will result in inefficient attack detection.

In this work, we propose a Machine Learning (ML) based framework which incorporates a set of carefully

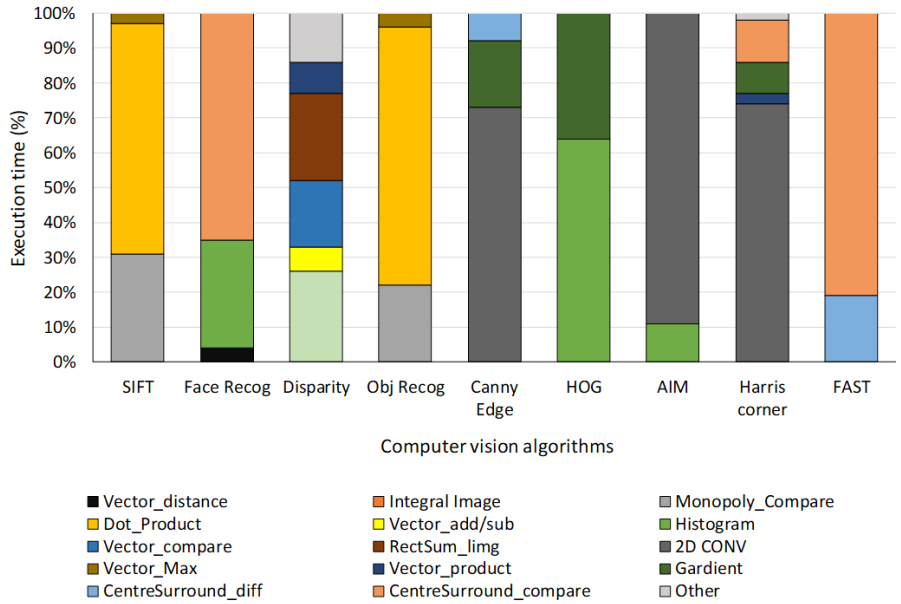


Figure 4.2: Percentage usage of common computation kernels in computer vision algorithms.

selected features and tunes them to appropriate thresholds for accurate attack detection. Figure 4.1 shows an accelerator-rich heterogeneous system under a flooding-based DoS attack by a malicious third-party accelerator (M3PA) IP with, a) no security mechanism and b) the proposed ML-based security mechanism integrated within the system. Our ML-based framework collects the feature data at runtime and performs online testing to flag the system as an attack or non-attack scenario. As evident from Figure 4.1.a, in the absence of the security mechanism, the communication network exhibits a high traffic rate, with relatively more dense traffic around the M3PA node. This results in high network bandwidth reduction and degrades the overall system performance. On the contrary, in the presence of our proposed security mechanism, as shown in Figure 4.1.b, the ML-based detection model is able to accurately detect a flooding attack. After an attack is detected, the operating system shuts down the M3PA to prevent any further attack. As a result, the system gets restored and allows the legitimate IPs to undergo regular network activities. Therefore, by efficiently detecting a flooding attack, our framework provides a scalable solution suitable for accelerator-rich systems.

4.1.1 Motivation

ArSoCs are custom-designed platforms that support a specific computing domain (like computer vision, graphics, medical image processing, etc.) and achieve significant improvement in performance/power efficiency. All the accelerator cores in an ArSoC are specialized to accelerate the common computing kernels employed by the applications in that domain. We explored multiple algorithms from computer vision domain to have an overview of their basic operations or computing kernels. Figure 4.2 presents the percentage usage of regular

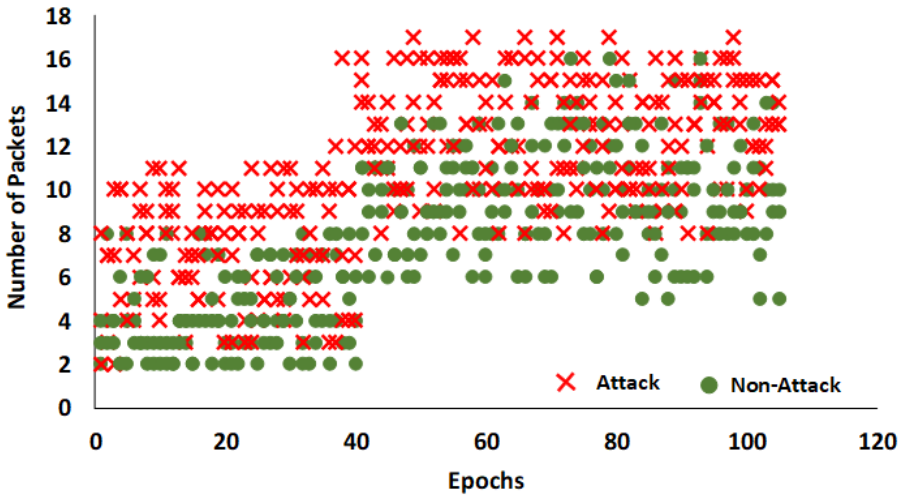


Figure 4.3: Data samples for attack and non-attack scenarios overlap, which makes an appropriate manual threshold setting infeasible.

computing kernels or basic operations by the vision algorithms throughout their execution time. An ArSoC designed for computer vision domain will comprise of multiple copies of accelerator cores specifically designed to enhance the performance of these common computing kernels. We derive two major observations from the figure: i) Although a number of applications use multiple common kernels, each of them will not invoke all the accelerator cores available on the ArSoC. ii) Each application will invoke different accelerator cores at different times, with a few cores used for a large part of its execution time. As a result, at any given time, only a part of on-chip resources like compute engines, memory, network links etc. are exploited, while others remain unutilized. Also, different applications appear or co-exist at different times, invoking various accelerator cores located on different parts of the SoC platform. Numerous research works [76, 30, 75] have been proposed for runtime resource optimization that allow all the concurrent applications to make use of the compute engines in the most efficient way. The optimization methods include efficient thread-to-core mapping against performance/power constraints [75, 30], DVFS to reduce dynamic power consumption [75, 76] and power gating to minimize leakage power [28]. These runtime optimizations influence the system behaviour and create a wide variation of network latency, system performance and power profiles throughout the execution period.

An ArSoC's system/network level activities can also be affected by an M3PA through creating a flooding-based DoS attack. Such an attack is accomplished by injecting a frequent and large number of packets into the interconnection network. As a result, it makes the on-chip resources unavailable for legitimate users and significantly degrades the overall system performance. Multiple works have been proposed to detect such flooding attacks in NoC-based multi-core systems [38, 42, 39, 40]. A common approach in the literature to distinguish between attack and non-attack scenarios is to manually set thresholds for selected parameters based

on empirical observations [39, 40]. However, due to the inherent dynamic behaviour of ArSoCs, it is difficult to differentiate the changes in system load due to regular activities from those caused by flooding attacks.

In order to highlight the complexity of attack detection, we design an experiment to simulate an ArSoC using a cycle-accurate network simulator, Noxim [66]. The system consists of 4 X86 CPU cores, 2D mesh network and multiple copies of accelerator cores custom-designed for computing kernels used in vision algorithms (refer to Section 4.4.1 for detailed simulation method and specifications). We simulate an application mix to represent an example scenario where different vision applications co-exist at different times. As in [40], we consider the number of packets arriving at a particular router at any given time interval as a metric to distinguish between attack and non-attack traffic. We first collect the data samples at each network router in a non-attack scenario. We then simulate a flooding attack where an accelerator core starts injecting a large number of packets into the network and collect the attack data samples. Figure 4.3 shows the data samples collected in the attack and non-attack scenarios for a single network router. Here, the network router is chosen with significant traffic load across the application mix such that it represents a generic behaviour of the network-level activities of an ArSoC. As shown in Figure 4.3, it is observed that the attack and non-attack data samples are significantly overlapping, which makes it difficult to define a threshold for non-malicious data values. Hence, manually setting a threshold for the network parameters to differentiate attack and non-attack samples will result in erroneous attack detection.

Although using multiple parameters for attack detection have been proposed to enhance accuracy in different scenarios [77], it will substantially increase the complexity of the entire process. Firstly, there are a number of system parameters that can become a potential candidate for attack detection. One needs to meticulously study the system behaviour and come up with a set of parameters that will most accurately detect an attack scenario. Secondly, while manual tuning of a single parameter will result in error-prone detection, considering multiple parameters all together to achieve a single goal will drastically increase the complexity and make manual threshold setting infeasible. To address these problems, we explore ML algorithms which are capable of solving multi-dimensional classification problems. By dynamically tuning multiple parameters to appropriate thresholds, the ML classifiers provide an accurate decision about the presence or absence of a flooding attack in an ArSoC. Finally, along with accuracy, a good security system should also provide early attack detection with minimal overheads. Though ML classifiers are able to provide accuracy, frequent invocations or data collection increases computation and communication overheads. This may affect the real-time applications running on ArSoCs and diminish the performance improvement achieved by the accelerator cores. Therefore, we present a two-level security platform, where the ML classifier is triggered only when the first level raises a flag to indicate possible security violations (details given in Section 4.3.1). A coarse-grained security check at the first level helps to prompt attack detection at an early stage while eliminating the need for constant invocation of

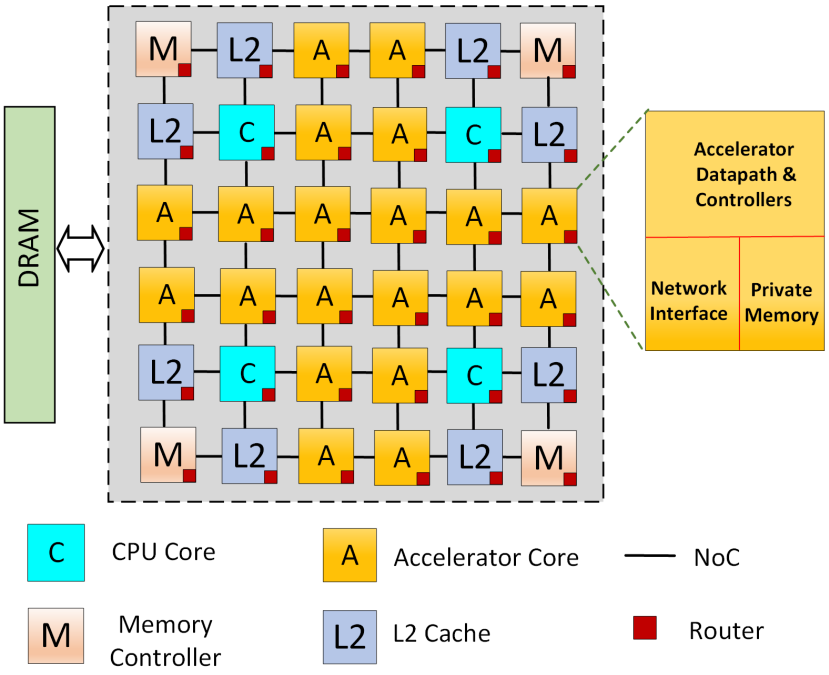


Figure 4.4: System architecture of an accelerator-rich system-on-chip.

ML classifiers. Hence, our proposed framework is able to provide accurate attack detection while minimizing computation and communication overheads.

4.2 System Architecture and Threat Model

In this Section, first we present an overview of the system architecture of an accelerator-rich SoC. We then discuss the assumptions and threat model considered in this work.

4.2.1 System Architecture

An ArSoC comprises of a few general-purpose CPU cores, a number of specialized hardware accelerators, and private and shared memory subsystems, all connected together by an interconnection network. Figure 4.4 shows a typical accelerator-rich SoC connected by an NoC. In this work, we consider a 2D-Mesh based NoC as the interconnection backbone due to its ability to support high bandwidth and low latency [5]. We consider a Loosely Coupled Accelerator-based (LCA) architecture where the accelerators are placed independently from CPU cores, which allows them to have complex datapaths without affecting the processor core design [11, 28]. Each LCA can access its own private memory or a shared memory chunk located remotely at other network nodes. Such a heterogeneous system allows to turn on the most efficient set of accelerators available and power off the rest of the inactive accelerators to reduce power consumption [28]. The compute-intensive part of an

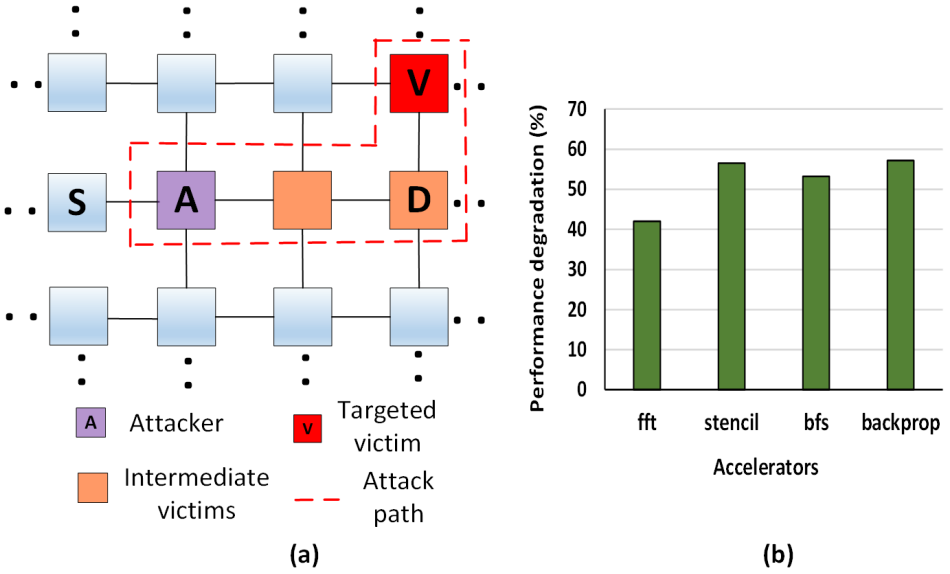


Figure 4.5: (a) An example attack scenario. (b) Performance degradation of benign accelerators due to flooding-based DoS attack by an M3PA.

application is offloaded onto an accelerator while the rest of the application runs on the CPU cores. A CPU core can initiate multiple accelerators depending upon their availability. It is the responsibility of CPU cores to prepare the accelerator-specific data in the memory and initiate their computation through system calls (e.g., ioctl calls). An accelerator starts its execution by accessing the specified memory space through the NoC. After completion of its execution, the accelerator raises an interrupt so that the corresponding CPU core can resume its software execution or call other subsequent accelerators. We leverage this interrupt generation of accelerators and formulate our first level sanity check to indicate a possible security breach in the system.

4.2.2 Assumptions and Threat Model

We make the following assumptions regarding our underlying hardware and software platforms. We assume that the CPU cores, memory modules and on-chip network are trusted. The operating system and software applications are also considered to be secure, preventing any software-based attacks. The third-party accelerator IPs are assumed to be vulnerable and can cause severe security attacks. Without the loss of generality, we consider the vulnerabilities in accelerators that lead to a flooding-based DoS attack.

In this work, we consider flooding-based DoS attacks triggered by an M3PA. In an ArSoC, an M3PA triggers a flooding attack by injecting continuous random packets into the network resulting in higher communication latency. As a result, it manipulates the perceived availability of on-chip resources by other legitimate users and creates a DoS attack. For instance, an M3PA can generate a large number of useless read/write re-

quests to the shared memory banks located at distant nodes. This will thwart the network traffic and degrade the performance of other benign accelerators cores. Figure 4.5.a shows an example attack scenario, where an M3PA node A creates a flooding attack by sending continuous read request packets to the shared memory bank at victim node V. Along with the targeted victim node V, other intermediate nodes are also congested as they fall on the attack path represented in the figure by the dotted red area. Consequently, when a source node S tries to access the destination node D, it will face higher latency due to congestion on its communication path. As a result, the affected accelerator cores starve for memory accesses and thus degrades the overall application performance. However, it must be noted that our proposed framework is generic and oblivious to the type of flooding packet. It can successfully detect a flooding attack whenever an M3PA generates a sufficient amount of useless packets to thwart the network bandwidth, which results in a DoS attack.

To evaluate application performance degradation due to such flooding-based DoS attacks, we simulate an attack scenario for a 64-node ArSoC. We use a network simulator Noxim [66] along with application traces of accelerator benchmarks in MachSuite [69] collected from gem5-Aladdin [57] tool. Figure 4.5.b shows accelerator performance degradation caused by an M3PA with respect to a baseline non-attack scenario. Here, we consider five active accelerators, one of which is malicious and floods the NoC to create a DoS attack. In our experimental evaluation, we found that an average of 40% increase in flooding packets from the number of packets generated in the baseline non-attack scenario is able to thwart the network and create a DoS attack. As shown in Figure 4.5.b, performance degradation of 42.05% to 57.20% is observed among the benign accelerators due to the presence of a malicious M3PA. Given this extreme threat potency from an M3PA, we explore possible solutions to detect flooding-based DoS attacks in an ArSoC.

4.3 Proposed Detection Framework

Figure 4.6 shows the flow of our framework comprising of a two-step method to detect a flooding-based DoS attack. In the first step, we perform a first level sanity check to be carried out by the CPU cores to flag a possible flooding attack based on pre-computed accelerator execution time (AET) and interrupt generation. To accurately differentiate between an attack and a non-attack scenario, we employ a Machine Learning (ML) based method as a second step of detection. Once a flag is raised in the first step, the ML classifier is triggered to perform binary classification of flooding-based DoS attack by inspecting the current system behaviour. The rationale behind employing a two-step framework is to reduce runtime computation overhead by avoiding frequent ML invocation. In the following Section, we describe our attack detection framework in detail.

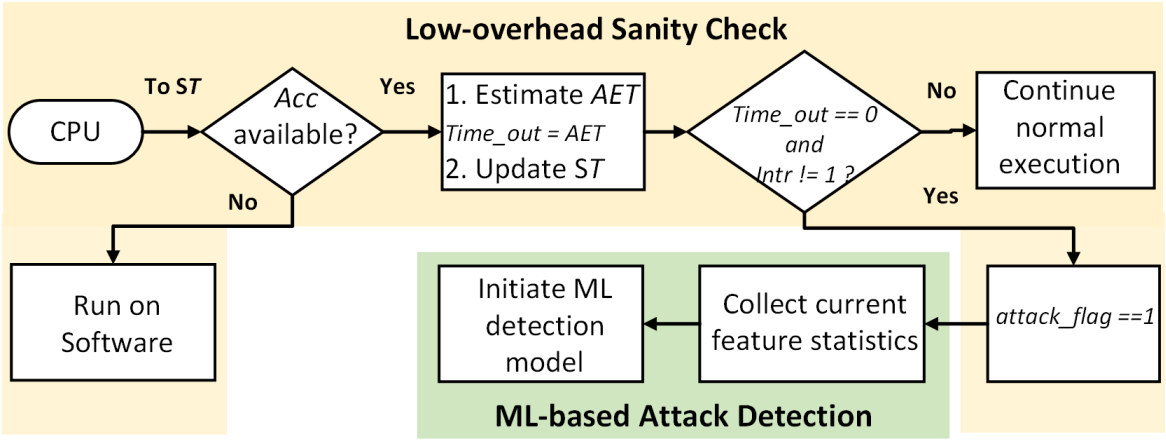


Figure 4.6: Flow of the proposed framework. Here, *Acc*=Accelerator; *ST*=Status table; *Intr*=Interrupt; AET=Accelerator execution time.

4.3.1 First Level Sanity Check

To facilitate the first level sanity check, the CPU cores compute AET of the target accelerator before offloading an application kernel. Here, AET comprises of i) an accelerator's computation time and ii) average network latency. An accelerator's computation time is data-dependent and is estimated using linear regression as a function of input data size. Whereas, the average network latency is estimated using an analytical model given in [78]. Both the metrics are calculated at the design time with representative input data sizes and benchmark applications. All the estimated data are stored in a Status Table (ST) residing in an on-chip shared memory. At runtime, whenever a CPU core initiates an accelerator, it stores the accelerator's ID and corresponding AET back in the ST. Consequently, the ST holds information about the availability of an accelerator at any point of time.

As shown in Figure 4.6, a CPU first enquires the ST regarding the availability of an accelerator. If available, CPU sets a counter, *Time out*, to the estimated AET for each accelerator invoked. Then, it updates ST, prepares input data and starts execution of the accelerator while decrementing the *Time out* counter at each clock cycle. In case of unavailability of an accelerator, the application is executed locally in the CPU core. As discussed earlier in Section 4.2.1, after completing its task, an accelerator must generate an interrupt to update the CPU, which then subsequently updates the ST. If the CPU does not receive an interrupt signal from the accelerator within the estimated execution time (i.e. *intr! = 1* and *Time out == 0*), it raises a flag to indicate a possible attack scenario. The rationale for CPU flag generation is that an accelerator must be starving for data accesses due to network congestion, which delayed its interrupt generation. Once a flag is raised, our second-level check is executed that employs an ML model for accurate attack detection.

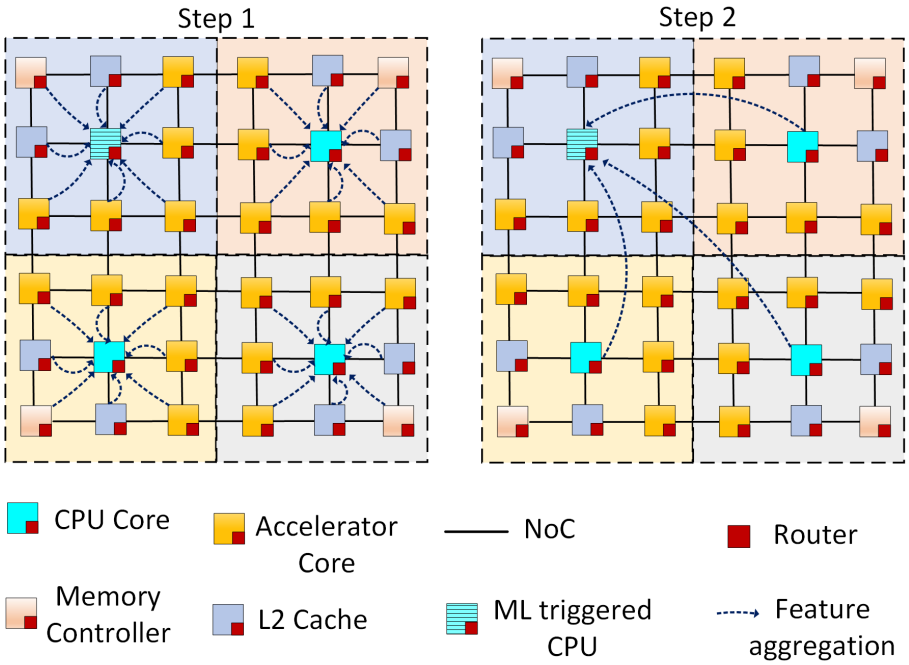


Figure 4.7: Steps for aggregating feature data from every network node.

4.3.2 Machine Learning for Attack Detection

The fundamental notion behind ML-based attack detection is to accurately distinguish the system behaviour in an attack and non-attack scenario. To achieve this, we explore the system state and network statistics to be used as features for training an ML classifier. Although deploying an ML-based solution may lead to accurate attack detection, an inefficient design may introduce computational and performance overheads. One design approach is to have a centralized ML model in which feature data need to be collected from all the network nodes. The other approach is to have a decentralized method where the ML model resides in every network node and takes the decision locally based on the feature data of that node. Such a decentralized approach results in huge computational and/or hardware overheads due to multiple copies of ML classifiers running simultaneously. On the other hand, a centralized approach might incur performance overhead due to feature data collection from each network node.

We adopt a centralized ML-based approach to provide an accurate and low-cost solution while using a hierarchical data collection method to address the performance overheads. We logically divide the entire network into multiple clusters, where a CPU acts as a cluster head and facilitates feature data collection. The ML classifier runs on a CPU and is triggered only when there is a flag raised by it in the first level sanity check, indicating possible security violations. After raising a flag, the CPU sends a broadcast message instructing the cluster heads to start feature data aggregation. Figure 4.7 shows the steps involved in aggregating feature data

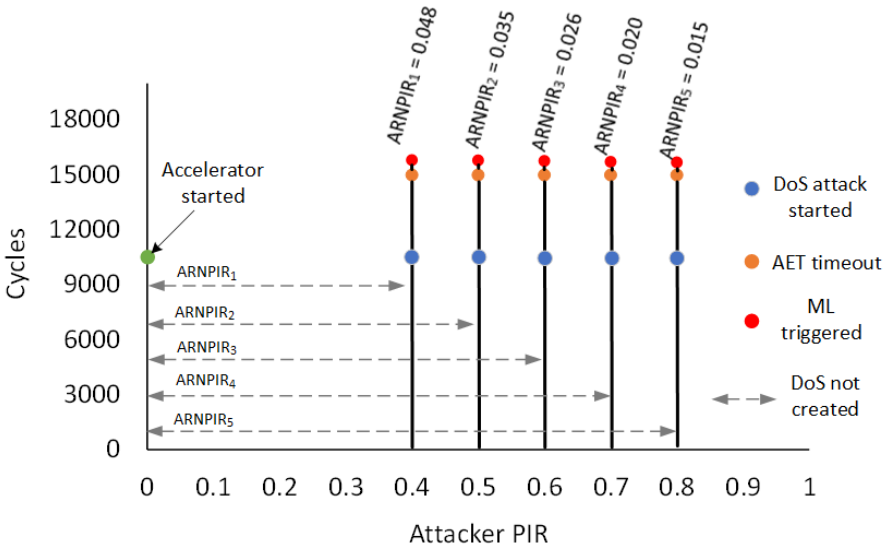


Figure 4.8: The packet injection rate (PIR) of an M3PA at which it creates a DoS attack under different average remaining network PIR (ARNPIR).

from all the network nodes. Each cluster head collects feature data from every node belonging to that particular cluster. Once the data is collected, average feature values are calculated at each cluster head and transferred to the CPU core running the ML classifier. Finally, after receiving data from all the network clusters, the average feature values are calculated for the entire network and fed to the ML model running in the CPU core. To facilitate steady forwarding of feature data through the likely-compromised network, each node groups them into one feature packet and prioritize those packets to avoid any unwanted delay. A priority bit is employed to distinguish a feature packet from the regular network packets. We utilize an unused bit available in the packet header flit as a priority bit which is set to 1 to indicate that it is a feature packet. Whenever a network router encounters a packet whose priority bit is set to *1*, it will immediately forward the packet, promoting an early attack detection. Therefore, our proposed approach is able to i) significantly reduce computational overheads by eliminating the need of concurrently running multiple ML classifiers, ii) minimize performance overhead incurred in feature data aggregation by using a hierarchical approach, and iii) facilitate early detection by prioritizing feature packets over the regular network packets. To this end, we describe our data collection methodology, the selected features and different ML classifiers considered in this work.

4.3.2.1 Data Collection

We employed Noxim [66], a cycle-accurate network simulator to collect data samples for training the ML classifiers. To collect the training data, we operate the simulator in two modes, namely, attack and non-attack modes. In non-attack mode, the system exhibits regular network-level activities. Whereas, the attack mode is

simulated by tampering the packet generation of one or more accelerator nodes within the system. An M3PA exhibits a high packet injection rate (PIR) as compared to other benign nodes which creates a flooding attack on the network. The PIR value at which the M3PA is able to create a DoS attack depends on the state of the system at that point of time. This is because different accelerators have different PIRs and start their execution at varied points of application runtime. Hence, the attacker PIR largely depends on the Average Remaining Network PIR (ARNPIR), which is demonstrated in Figure 4.8. Moreover, due to different invocation times of various accelerators, the AET timeout for first level sanity check also varies. At any given instance, the time at which the ML model is triggered depends on the accelerator with the least remaining time for its AET completion. Here, for clear visualization, we reduce this variability by considering a single active benign accelerator and gradually increase the M3PA's PIR at a constant rate. Also, we consider that an M3PA initiates an attack as soon as the accelerator starts its execution. The system configuration given in Table 5.1 is considered for the experimental setup. As evident from Figure 4.8, to successfully create a DoS attack, the M3PA needs to increase its PIR and inject more number of packets into the network as the ARNPIR decreases. The dotted double arrows in the figure represent the part where there is no DoS attack created for a given attacker PIR and ARNPIR pair. After the M3PA creates a DoS attack, the AET timeout occurs, which leads to feature data collection and finally, the ML model is triggered to accurately detect the attack. To include the effect of a system's current state as a feature, we collected data samples with different numbers and combinations of accelerator cores. In our work, the ML models are trained offline on a system with Intel i5-7200U quad-core processors running at 2.50 GHz frequency and 8 GB RAM. We consider a 64-node system size for our experimental evaluation, while the model can be further trained for larger network sizes. We conducted multiple runs, each for 1000000 cycles, and collected both attack and non-attack data samples. A total of 10000 data samples were collected, 5000 for attack and 5000 for non-attack traffic.

4.3.2.2 Features

We explore two classes of features to train the ML classifiers for attack detection: *Network State Features* and *System State Feature*. The *Network State Features* capture the variation in on-chip communication behaviour over the execution period. These features require collecting statistics from all the network nodes, which is done by our hierarchical feature data aggregation method discussed above. The *System State Feature* captures the system's current state to understand the expected network load at any given time. The *System State Feature* is collected from the ST which keeps track of the number of active accelerator cores in the system. To determine the significance of the selected feature metrics, we performed chi-squared test, a commonly used measure in machine learning to test the relevance of features to the outcome to be predicted. The detection accuracy results with an ML classifier is presented in Section 4.4.3 to highlight the significance of the selected feature metrics.

1) Network State Features:

Inter-packet Interval (*IPI*). In a non-attack scenario, there is an appreciable time between any two consecutive packet reception at a network node. On the contrary, the Inter-packet-Interval (*IPI*) time becomes very small in case of a flooding-based DoS attack. As a result, *IPI* is a potential feature candidate to distinguish between attack and non-attack traffic.

Router-buffer Waiting Time (*RWT*). It refers to the time interval for which a packet waits in the buffer of a router due to the unavailability of its output ports or congestion at the input port of the downstream router. This waiting time significantly increases in an attack scenario and serves as a good metric for attack detection. Similar to *IPI* calculation along with a timestamp-counter, we compute *RWT* for incoming packets at each network router.

Packet Delivery Ratio (*PDR*). *PDR* refers to the ratio of packets received by a destination to those generated by a source node. In comparison to a non-attack scenario, *PDR* value will drastically decrease during a flooding attack as more number of packets will get dropped before reaching the destination nodes. To compute *PDR*, each source node router holds the information of the number of packets generated along with their targeted destination nodes. With that knowledge, and based on the number of re-transmission requests, *PDR* is determined for each node.

Global Average Delay (*GAD*). It represents the average latency incurred by the entire on-chip communication network. In an attack scenario, the network exhibits high congestion as compared to a non-attack scenario. This results in extended communication latency during a flooding attack and hence drastically increase the Global Average Delay (GAD). This signifies the use of GAD as an effective feature to distinguish between attack and non-attack traffic. GAD is readily available from the simulation tool, Noxim[66] employed in our experimental evaluation.

2) System State Feature:

Number of Active Accelerators (*#AAcc*). As discussed earlier, in an ArSoC, the CPU cores request from a pool of accelerators to offload the entire or part of its application. While multiple accelerators are assigned various tasks, the rest of the inactive accelerators are turned off to reduce power consumption. The number of ingress or egress packets generated in the network vastly depends upon the number of accelerators active at a given point of time. This influences all the *Network State Features* due to the increased packet density on the network. Considering the number of active accelerators as a feature will drive the ML classifier to set appropriate thresholds leading to better prediction of attack scenarios. As a result, the classifier will

significantly reduce the number of false positives. This quantifies the importance of including *#AAcc* as a feature to accurately detect an ongoing flooding attack. The information regarding the number of currently active accelerators is collected from the ST and fed to the ML classifier for detection.

4.3.2.3 ML Classifiers

In machine learning, classifiers provide a vector as an output which indicates the probability of a data item belonging to each class in a given set of classes. In our problem scenario, a classifier will yield a vector of two probabilities that determine the likelihood of a data item belonging to an attack class or a non-attack class. There are a variety of ML classifiers available which can be broadly divided into linear and non-linear classifiers. Since no single classifier gives the best performance for every problem, we investigated multiple ML classifiers to evaluate the efficiency of the proposed attack detection framework. We perform our experiments with both linear and non-linear classifiers, with more emphasis on the non-linear classifiers as we anticipate that our data samples are more likely to be linearly inseparable.

To evaluate the efficacy of a linear classifier in our problem scenario, we choose Support Vector Machine (SVM) that creates a line or a hyper-plane to divide the data set into attack and non-attack classes. Then it finds the data samples (support vectors) from both the classes which are closest to that line. Finally, it tries to maximize the distance between the support vectors and the separating line. We explore multiple non-linear classifiers like K-Nearest Neighbour (KNN), Decision Tree (DT), Random Forest (RF) and Artificial Neural Network (ANN). KNN predicts values of new data samples based on feature similarity with the existing data samples in the training dataset. We use the Euclidean distance function to calculate the distance between the data samples and consider $k = 3$ to maintain a good trade-off between accuracy and computational & memory overhead. A DT forms a tree structure where each internal node is a test on an input feature, the arcs coming out of the nodes are the outcome of the tests and each leaf node is labeled with a class name. RF is an ensemble method that uses multiple DT classifiers on sub-samples of the dataset and improves the prediction accuracy by voting out the best solution. Finally, we explore attack classification with ANN which comprises of multiple connected input-output network layers. Each connection is associated with weights which are updated iteratively to improve the performance of the network to classify data samples into attack and non-attack classes. We use scikit-learn library [79] for implementing all the classifiers and perform offline training to reduce runtime computational overhead.

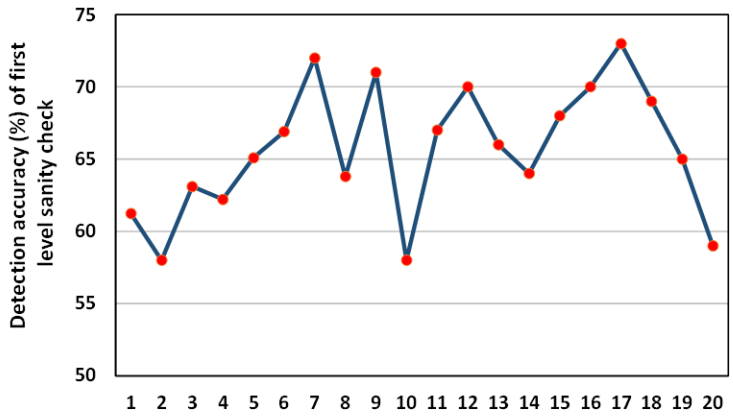


Figure 4.9: Detection accuracy of the first level sanity check.

4.3.3 Discussion

To highlight the importance of the two-level approach adopted in our framework, we evaluate the performance of the first level sanity check in correctly triggering the attack detection process. We consider 20 test cases across accelerator benchmarks (given in Table 4.1), where we set the attack start time randomly for each test case. Figure 4.9 shows the accuracy of the first-level sanity check in correctly triggering a flag to indicate a possible attack. As shown in the figure, the detection accuracy ranges between 58% to 73% across all the test cases, with an average of 65.61%. This is because, the first level sanity check depends on the AET timeout of one of the active accelerators, which is defined statically at the design time. Although computation time of accelerators remains the same, the actual runtime AET may vary due to varied memory access time induced by dynamic network characteristics of multi-accelerator systems. Hence, the first level sanity check may generate multiple false positives and will fall short in providing an accurate indication of the presence or absence of a flooding attack in few cases. However, in the absence of the first-level sanity check, the ML model needs to be frequently invoked to detect an attack scenario. This results in frequent feature data aggregation from all the network nodes which in turn leads to significant performance overheads. The comparison of performance overheads in the presence and absence of the first level sanity check is discussed later in Section 4.4.5.

In a multi-accelerator system, the accelerators are invoked by different applications at different times during their entire execution period. Hence, when an M3PA starts injecting flooding packets into the network, the time remaining to complete the AET period is different for each accelerator at any given time instance. As a result, the first-level sanity check generally gets triggered by the accelerator with the least remaining time for AET completion and is highly affected by the flooding attack. However, in the worst case, the AET timeout may happen after the entire execution of the accelerator. In such a scenario, the first level sanity check will be triggered much later after the attack is initiated, which in turn will extend the total attack detection time. To address this issue, we perform the first level sanity check for multiple intervals by sweeping through accelerator

Table 4.1: Trade-off analysis for granularity of first level sanity check

Granularity	**Decrease in attack detection time (%)**	**Accuracy of first-level sanity check (%)**	**Increase in traffic overhead (%)**
AET/10	78.29	29.22	6.10
AET/5	72.32	38.12	3.24
AET/4	61.10	54.35	2.41
AET/2	42.23	59.53	1.15

start time till its AET completion. This reduces the additional detection time as the second level ML detection does not have to wait for the entire AET completion. However, performing the first level sanity check at earlier stages would lead to increased false positives due to the dynamic traffic, leading to decreased detection accuracy. Moreover, multiple ML invocation will increase the network traffic which in turn will result in performance overheads. We provide a trade-off analysis in Table 4.1 to investigate the effect of multiple first level sanity checks throughout the accelerator execution period. As shown in the table, different timeout granularity provides varied accuracy, traffic overhead and reduction in detection time. One can choose the granularity of first level sanity check depending on their priority metric.

4.4 Experimental Evaluation

In this Section, we present the system configuration and evaluation of the proposed attack-detection framework.

4.4.1 System Configuration

We extend gem5-Aladdin [57] and Noxim [66] simulators to evaluate our proposed attack detection framework. gem5-Aladdin provides a platform to incorporate multiple fixed-function accelerators, CPU cores and memory elements. It applies various optimization techniques on high-level application codes to generate the accelerators which provide 100× improvement in performance/energy efficiency. To implement a 2D mesh network as a communication backbone, we use Noxim which is a cycle-accurate network simulator designed for performance/power analysis of NoC architectures. The proposed detection framework is incorporated in Noxim, which is able to accurately detect a flooding attack based on the feature metrics. Table 4.2 presents the system configurations used in our experimental evaluation.

The application-level traffic/traces are obtained from gem5-Aladdin by simulating applications invoking multiple fixed-function accelerators. The application traces are then fed as an input to Noxim which emulates the network communication behaviour of the ArSoC. Such trace-based simulation environments have been widely used to study the on-chip communication networks and evaluate system performance [30, 20]. A modern

mobile ArSoC is considered for evaluating the efficacy of the proposed framework. The system is equipped with multiple copies of various accelerators to support different concurrent applications like image processing, speech recognition, object classification, compression and encryption. Most of the accelerator kernels are incorporated from the accelerator benchmark suite MachSuite [69]. A few application kernels are obtained from MiBench[70] benchmark suite and modeled as fixed-function accelerators by utilizing the gem5-Aladdin platform. All the accelerator kernels employed in our evaluation are presented in Table 4.3. We evaluate the proposed framework in terms of attack detection accuracy, scalability and its impact on the performance of the applications running on an ArSoC. We first present the detection results for offline analysis by using k-fold cross validation. We then also show on-the-fly detection accuracy achieved by integrating the framework within the simulator using all the ML classifiers. The significance of the considered feature metrics is also shown with respect to the obtained detection accuracy.

4.4.2 Detection Accuracy

In this Section, we evaluate the offline detection accuracy achieved by the ML classifiers on the dataset collected using the method described in Section 4.3.2.1. The detection results are presented by averaging the outcomes of 5-fold cross validation. Table 4.4 presents the offline accuracy results of the ML classifiers used in our framework to detect flooding-based DoS attacks. It also shows the robustness of the classifiers with regard to precision, recall and F1-score metrics. We observed the attack detection accuracy to be ranging from 0.63 to 0.97 for all the classifiers considered in our experiments. A higher value of accuracy and robustness metric represents higher performance. As evident from Table 4.4, the linear SVM classifier performed badly as compared to other classifiers employed in our framework. A relatively low accuracy of 0.63 given by the SVM classifier suggests that the data-space considered for training the classifier is not linearly separable. However, the KNN model performed fairly well with an accuracy of 0.85. This indicates that the classifier was able to cluster the data into attack and non-attack classes with regard to the considered feature space. In the case of neural network (ANN), a detection accuracy of 0.89 was reported. DT and RF classifiers performed extremely

Table 4.2: Simulation setup for evaluating proposed attack detection framework

Component	**Configuration**
Processing Cores	4 x86 CPU cores; Customized fixed-function accelerator cores, 500MHz frequency
Memory	32KB private and 2MB shared on-chip memory; 4GB off-chip DRAM
NoC	8X8 2D mesh, XY routing, wormhole switching, 1-cycle link latency, 3-cycle router latency
ML Models	KNN, RF, SVM, DT and ANN

Table 4.3: Accelerator kernels

Accelerator kernel	Description	Benchmark Suite
aes	Encryption(256-bit keys and blocks	MachSuite
backprop	Image recognition (neural network)	MachSuite
gemm	Matrix multiplication	MachSuite
fft	Digital signal processing	MachSuite
spmv	Image processing	MachSuite
viterbi	Speech recognition	MachSuite
sort	Map reduce	MachSuite
susan(edge)	Image processing	MiBench
susan(corner)	Image processing	MiBench
susan(smooth)	Image processing	MiBench
jpeg	Image compression	MiBench

well with an attack detection accuracy of 0.92 and 0.97 respectively. This is due to the non-linear nature of DT and RF classifiers that allow them to achieve higher accuracy by constructing multiple linear boundaries for detection. To reduce overfitting, the maximum depth of our DT model was limited to 4. The RF classifier performs even better by considering a pool of decision tree outcomes and reducing noise. Since the RF classifier performed best among all the learning models, we explore its performance with varying feature space in the following section.

4.4.3 Feature Importance

Figure 4.10 shows the importance of all the features considered in the evaluation of our framework in terms of prediction accuracy. Here, we take each feature in isolation as well as all possible combination of features and run the RF classifier that reported the best binary-classification accuracy in our experimental evaluation. A prediction accuracy ranging from 45.25% to 76.58% is observed while considering each feature in isolation. This reflects the discussions presented in Section 4.3.2.2 where we studied how the selected feature metrics influenced the system characteristics. Although having relatively less accuracy in isolation, a combination of multiple feature metrics to detect an attack scenario significantly increased the overall prediction accuracy.

Table 4.4: Detection Performance of Classifiers

ML Models	Accuracy	F1 Score	Precision	Recall
KNN	0.855	0.855	0.856	0.856
RF	0.974	0.970	0.970	0.971
SVM	0.638	0.621	0.631	0.630
DT	0.929	0.928	0.928	0.928
ANN	0.891	0.890	0.890	0.890

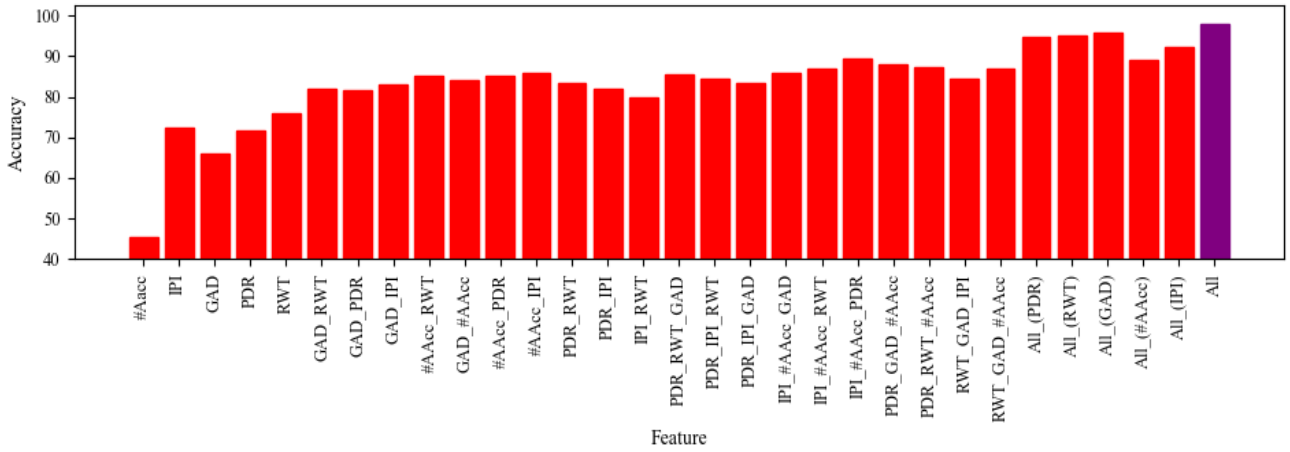


Figure 4.10: Feature importance (accuracy of RF for binary classification with each feature in isolation and in combination). We also plot the accuracy with all features at the right-most bar of the figure (labeled as 'All'). Here, "$All\_(Feature\_name)$" denotes all features except *"Feature_name"*.

Table 4.5: On-the-fly Detection Performance of Classifiers

ML Models	Accuracy	F1 Score	Precision	Recall
KNN	0.832	0.832	0.831	0.831
RF	0.965	0.965	0.965	0.964
SVM	0.598	0.591	0.590	0.590
DT	0.909	0.908	0.909	0.9098
ANN	0.861	0.862	0.862	0.862

4.4.4 On-the-fly Detection Accuracy

The evaluation results provided so far have been obtained offline by applying our detection algorithms on collected data samples. Now, we integrate the ML classifiers within our simulation environment. The flooding attack is simulated by increasing packet injection from a particular accelerator, as discussed in Section 4.3.2.1. Whenever the flag is raised in the first-level sanity check, the ML model is triggered which resides within the corresponding CPU node. In this case, the classifiers are trained offline on the collected data samples while the inference is carried out at the runtime. As a result, the ML classifiers encounter a variation in runtime network behaviour, which helps in evaluating their effectiveness to make accurate attack detection for dynamic runtime network traffic.

The on-the-fly detection accuracy of different ML classifiers are presented in Table 4.5. All the classifiers show similar results as compared to the off-line detection accuracy presented in Table 4.4. Among all the classifiers considered for evaluation, RF classifier provides the best performance in terms of detection accuracy. From these observations, we conclude that our ML-based attack detection framework is suitable for an ArSoC environment where the network traffic exhibits dynamic load behaviour.

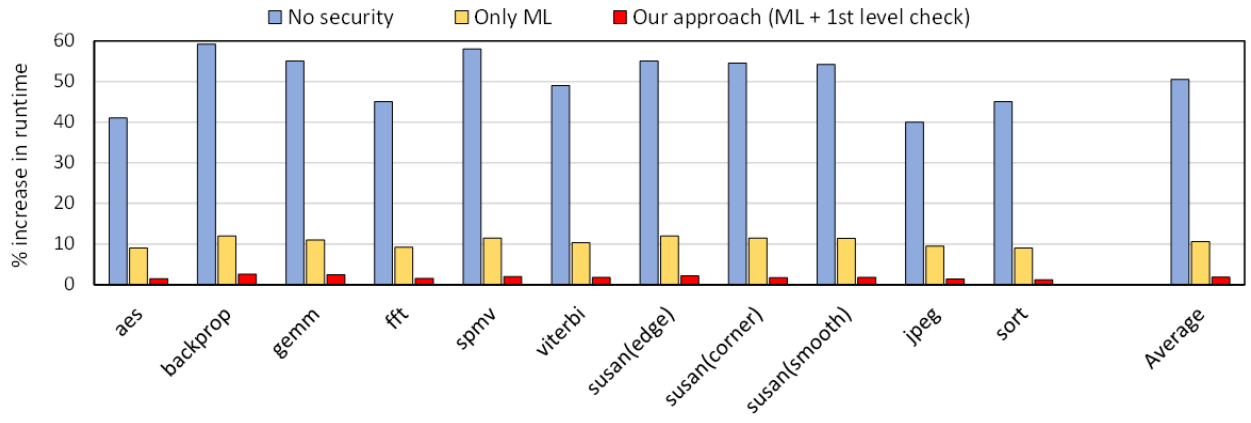


Figure 4.11: Accelerator runtime in three scenarios: i) absence of any security mechanism, ii) ML-only based detection and iii) our approach (two-step ML-based detection: first-level sanity check and second-level ML detection).

4.4.5 Performance Evaluation

In this section, we evaluate the impact of our proposed attack detection framework on the system performance. While our framework will be able to curtail the performance degradation of the accelerators, it might also incur traffic overhead under non-attack scenarios.

We create a flooding attack by triggering an accelerator core to generate a large number of packets for a significant application execution time. We then observe the performance degradation (i.e., increase in runtime) of all the accelerator cores in three scenarios: i) absence of any security mechanism, ii) ML-only based detection and iii) our approach (two-step ML-based detection: first-level sanity check and second-level ML detection). In the first scenario, the system has no built-in security measures whereas in other scenarios, there is a security mechanism to detect a flooding attack. In the second scenario, the ML classifier is assumed to be active after every fixed time interval throughout the application execution time. As a result, feature data is continuously collected at each interval from all the network nodes and fed to the ML classifier. Figure 4.11 shows the increase in runtime of the accelerators in all the three scenarios as compared to a baseline non-attack scenario. As shown in the figure, the absence of any security mechanism significantly affects application performance leading to an increase in accelerator runtime of up to 59.21% as compared to the baseline. The ML-only based method is able to accurately detect a flooding attack and prevent the accelerators from experiencing significant performance degradation. However, it incurs an average runtime overhead of 10.58% due to the additional on-chip communication involved in regularly collecting feature data from all the network nodes. On the contrary, our proposed detection framework incurs a minimal runtime overhead of 1.84% on average, as compared to the baseline. This is because the first-level sanity check introduced in our framework eliminates the need for invoking the ML classifier at regular intervals. As a result, the communication overhead incurred in collecting

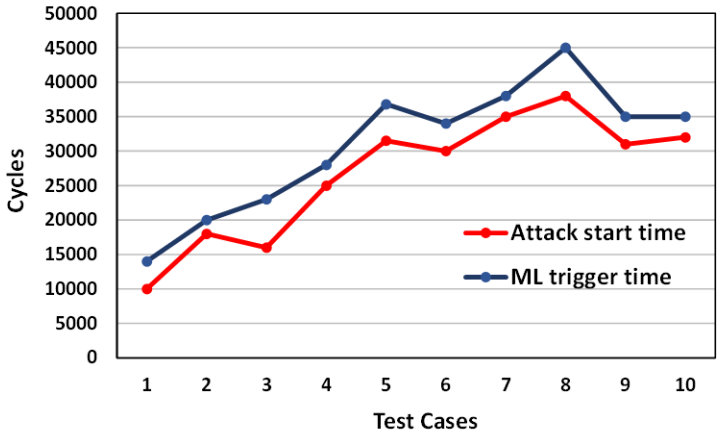


Figure 4.12: Time taken from attack initiation till the triggering of ML model across different test cases.

the feature data for every ML invocation is drastically reduced. Finally, our proposed hierarchical feature data collection further minimizes the overall runtime overhead by allowing cluster-based data aggregation as compared to collecting data from every node as done in an ML-only scenario.

The first level sanity check of our framework prevents frequent ML invocations and reduces the system performance overhead. However, as discussed in Section 4.3.3, the first level sanity check may introduce a few false positives, which will invoke the second level ML detection in the non-attack scenarios. This will result in unnecessary feature data collection and incur traffic overheads in such cases. In our experimental evaluation, a traffic overhead of 0.31% is observed due to this unwanted feature data collection.

4.4.6 Attack detection time

The attack detection time of our framework depends on the time taken by the first level sanity check to trigger an attack flag from the time of attack initiation, the time taken to collect feature data from all the network nodes and the execution time of the ML model. Equation (4.1) presents a formal definition for measuring the attack detection time.

$$Detection\_time = T_{AET\_timeout} + T_{feature} + T_{ML} \tag{4.1}$$

The AET timeout for the first level sanity check ($T_{AET\_timeout}$) depends on the granularity of checking and the attack initiation time. Here, we consider AET/4 as the granularity level depending on the trade-off analysis performed in Section 4.3.3. The feature data collection is performed using a hierarchical method and utilizes a priority-based forwarding, thus reducing the data aggregation time ($T_{feature}$). While we optimized the classifiers to reduce the training and testing time, it can be argued that their software implementations can be further tuned to significantly reduce the execution time. Moreover, implementing dedicated hardware for

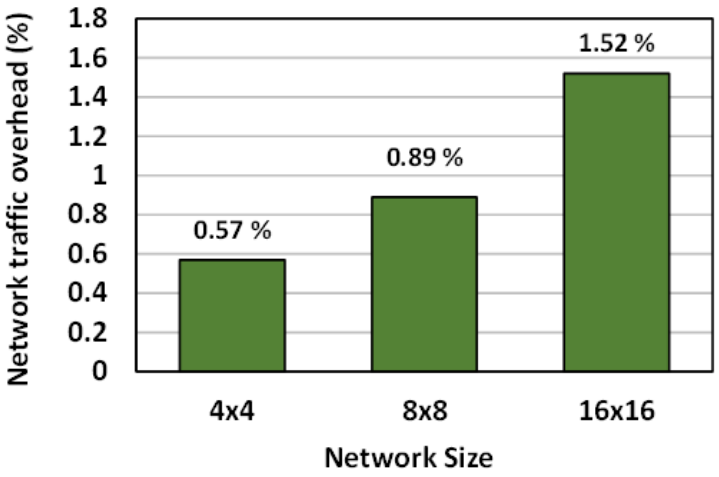


Figure 4.13: Scalability analysis of proposed attack detection framework.

the ML classifier would be extremely efficient [80], which will significantly bring down the attack detection time. The authors in [80] demonstrated a multi-precision hardware implementation of the RF classifier, which performed testing of a data instance in 23 clock cycles at 25 MHz frequency, thus providing faster online testing time. However, optimizing the implementation of the ML classifiers is not the focus of our work. Rather, we focus on providing a framework that analyzes the characteristics of multi-accelerator systems and leverages the state-of-the-art ML models for accurate attack detection. While the time taken for running the ML model is constant, the AET timeout and feature data aggregation may vary depending on the attack start time and the state of the system at that particular instance. To show the influence of AET time out and feature data aggregation on the attack detection time, we analyze the time required for triggering the ML model once an attack is initiated. We consider multiple test cases, where we set the attack start time randomly for each test case. Figure 4.12 shows the time taken to trigger the ML model after attack initiation at different instances of time for each test case. This shows that our proposed framework is able to provide accurate attack detection within an acceptable detection time.

4.4.7 Scalability

As a large number of accelerators are envisioned to be integrated into future ArSoCs [81], the current detection frameworks must also be scalable to larger system sizes. We evaluate the scalability of our proposed framework in terms of the network traffic overhead, i.e., additional network latency, incurred due to feature data aggregation. The feature data packets need to traverse through the on-chip network from remote nodes to be fed to the ML classifier that is triggered for attack detection. This results in increased network traffic and affects the overall system performance. To evaluate the scalability of our framework, we analyze this increased network traffic overhead across various system sizes. We consider 16-node, 64-node and 256-node systems for our ex-

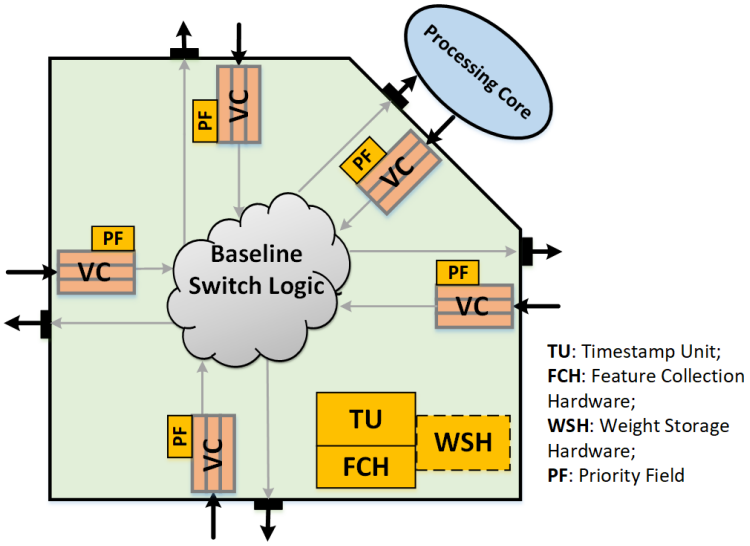


Figure 4.14: Modified Router Architecture.

perimental evaluation. Figure 4.13 shows the network traffic overhead incurred due to feature data aggregation for different network sizes. As shown in the figure, the proposed method incurs minimal overhead ranging from 0.57% in a 16-node system to 1.52% in a 256-node system. This is because our proposed hierarchical data collection method aggregates the feature data from every network node of a cluster, averages them and forwards a single feature packet from each network cluster. Once all the cluster averages are collected at the node running the ML model, they are averaged and fed to the classifier for attack detection. Therefore, even for larger system sizes, the amount of on-chip network communication is significantly reduced, resulting in minimal traffic overhead.

4.4.8 Area and Power

In this Section, we evaluate the area and power overheads incurred due to our proposed attack detection framework. These overheads comprise of implementation cost of ST and Attack Detection Hardware (ADH). A 2Kbits of ST is considered in our framework, which is sufficient to store the accelerator statistics. The ADH is implemented in each router for feature collection, ML weight storage, and feature data prioritization. The microarchitecture of the modified router is shown in Figure 4.14, where the ADH is highlighted. The Feature Collection Hardware (FCH) is capable of collecting and averaging all the network state features,i.e., IPI, RWT, PDR, and GAD. An 8-bit Timestamp Unit (TU) is included in each router that would provide the Arrival Time (AT) of the packets entering the input ports of a router node. The AT information can be stored in the unused bits of the packet head flit. IPI is computed from the difference of the consecutive packets' AT values. RWT is calculated by subtracting the AT value of the packet from the current timestamp value. For the computation of PDR feature, each router node is additionally equipped with two 8-bit counters to keep track of the total number

of packets getting generated from it and the total number of packet re-transmission requests coming to it. For the computation of GAD, each router node contains *n* number of 8-bit packet latency registers. The *n* value depends upon the available area budget, and is considered as 10 in our simulation setup. Packet latency refers to the time taken by the packet to travel from its source to destination. These *n* registers store *n* consecutive packets' latency for which the node is the destination. These *n* values get averaged and is stored in another 8-bit average delay register in the same router. The same operation is repeated for the next *n* packets and the cumulative average with the previously stored value is stored in the average delay register. The Weight Storage Hardware (WSH) is augmented only to the router nodes connected to the CPUs running the ML model. The WSH includes five 8-bit registers to store pre-trained feature weights and the bias values. The Data Prioritization Hardware (DPH) assigns network resources with higher probability to the feature packets moving on the network during the second level attack detection phase. The head flits of feature packets hold a priority bit that distinguishes them from the normal data packets in the network. In *Wormwhole* switching (the flow control mechanism adopted in our framework), only the head flit competes and reserves the VC. Each VC of the router is augmented with an additional 1-bit Priority Field (PF) as DPH, which is updated according to the priority bit of the head flit when the head flit reserves the VC. This priority field is utilized by the body flits during the switch arbitration for the prioritized movement of the feature packets. To compute the area and power overhead of the ADH, the modified router is synthesized in Synopsys design compiler for 32nm technology node. It is observed that the ADH incurs 0.002 mm^2 area and 0.055 mW power, which is 9.53% and 6.5% of area and power overhead respectively as compared to the baseline router architecture. In our system configuration, we observed the runtime power consumption of the adopted ML model to be 25 mW.

4.4.9 Comparison with related works

In this Section, we compare our proposed attack detection framework with the existing DoS attack detection methods for NoC-based heterogeneous architectures.

In [38], the attack detection method relies on generating multiple extra packets to compare the latencies between original packets and the suspected attack packets. This leads to increased network congestion and further disrupts regular communication. Moreover, to achieve an acceptable detection accuracy, a large window of statistics collection is required, which will significantly affect the runtime performance (5.57% on average). In our case, the ML-based model provides an accurate attack detection and does not need multiple data collection windows, thus limiting the performance overheads (0.89% on average). The hardware overheads of our framework are also relatively low as compared to the overheads reported in [38] for their modified network interface.

Table 4.6: Summary of comparison with the related works

	[38]	[39]	[40]	[Our work]
Key characteristics in Attack detection	Generate multiple extra packets & gathers large runtime network statistics	Based on empirically set threshold violation	Detect traffic violations based on statically profiled pattern	1st-level sanity check & 2nd-level ML-based detection
Detection metrics	Packet latency	Average bandwidth	Maximum Packet arrivals	IPI, RWT, PDR, GAD, #Acc
System considered	Heterogeneous system	Generic multi-core system	Heterogeneous system with defined traffic pattern	Multi-accelerator heterogeneous system with varying traffic pattern
Performance Overhead	5.57% on average	Data collection & forwarding to security manager	Depends on implementation of parameter monitoring hardware	0.89% on average
Hardware Overhead	Area: 12.73%, Power: 9.84%	Area: 34.7%	Area: 5.93%, Power: 3.87%	Area: 9.53%, Power: 6.5%

Authors in [39] present a central security manager which takes decisions based on the data collected from various probes distributed in the network. Their method flags any unnatural traffic, which exceeds the average network bandwidth determined at design time. Relying on a single metric with an empirically set threshold makes their method less efficient for detecting DoS attacks in varying network conditions. In comparison, our framework uses multiple parameters and tunes them to appropriate thresholds by using state-of-the-art ML models, hence suitable for the dynamic nature of multi-accelerator systems. Moreover, the hardware overheads presented in [39] is significantly high, around 34.7% of the baseline, as compared to 9.53% in our case.

The DoS attack detection method presented in [40] also considers a statically profiled network parameter to detect an attack scenario. They assume that the system would have a defined traffic pattern and calculates an upper bound for their detection metric to determine the threshold for violation. However, the multi-accelerator systems will exhibit varying traffic patterns due to the dynamic usage of accelerators, making the statically set thresholds unreliable for such systems. On the contrary, our framework provides a strong measure for attack detection by considering multiple network and system characteristics that capture the system variations and gives accurate attack detection. Although the hardware overhead of our framework is slightly higher than [40], the accurate detection provided by our framework is crucial for the time-critical applications running in the multi-accelerator systems. A summary of comparison with the related works is given in Table 4.6.

4.5 Conclusion

In this work, we present a machine learning-based flooding attack detection framework for accelerator-rich SoCs. We analyze the system behaviour to identify the most important features for distinguishing between an attack and a non-attack network traffic. We demonstrate the effectiveness of our proposed approach using real accelerator benchmarks tested against various ML classifiers. In our experimental results, all the ML classifiers performed fairly well, with the highest accuracy of 97% achieved by the RF classifier. As an extension of this work, novel localization techniques can be incorporated to localize the attacker node.

Chapter 5

Sniffer: A Machine Learning Approach for DoS Attack Localization in NoC-based SoCs

As discussed in Chapter 4, a flooding-based Denial-of-Service (DoS) attack attempts to exhaust the network bandwidth and diminishes a system's ability to provide appropriate services. It results in unavailability of the Network-on-Chip (NoC) for other legitimate on-chip modules and degrades the entire system performance. Finding the location of the Malicious IP (MIP) is crucial to restore regular network operations and curtail system performance degradation. In this chapter, we propose Sniffer, an efficient MIP localization framework which employs a low-overhead machine learning approach to accurately trace the attack path and take a collective decision to locate the MIPs.

5.1 Challenges in flooding-based DoS attack localization

The flooding-based DoS attack attempts to degrade the system performance by manipulating the availability of the on-chip network to the legitimate IP blocks. Figure 5.1.b shows the percentage of benign packet loss due to a flooding-based DoS attack on a 64-node heterogeneous SoC with 2D Mesh NoC. The details of the system configuration are provided in Table 5.1. Here, the attack is simulated by configuring an MIP node to send a new packet every cycle to a VIP node for an interval of 20,000 cycles. As shown in the figure, the number of benign packets transferred is reduced by almost half as compared to a baseline non-attack scenario, which directly translates to application performance degradation. Hence, it is of utmost importance to address the security threats from MIPs and prevent extreme performance degradation of the applications running on the SoCs.

The profound security threats from MIPs have triggered multiple works to address flooding-based DoS attacks like monitoring traffic anomalies [39, 42], checking bandwidth violations [38] and introducing verifi-

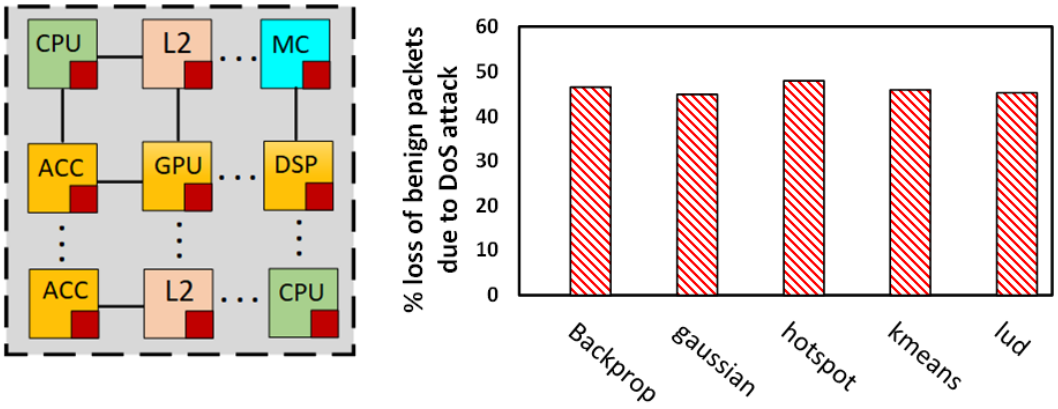


Figure 5.1: (a) Heterogeneous SoC with CPU, GPU, DSP, ACC(accelerator), L2(last level cache) and MC(memory controller) nodes with routers, connected by NoC. (b) Percentage of benign packet loss due to flooding-based DoS attack.

cation guidelines [42]. However, localization of the source of attack (MIPs) has not been properly explored in the context of NoC-based architectures. Once an attack is detected, it is imperative to localize the MIPs and take necessary actions to prevent any further system degradation. But, the inherent nature of NoCs, allowing multiple parallel communication and resource sharing makes it extremely challenging to locate the source of the attack. The presence of multiple MIPs makes it even more onerous and signifies the need for a robust MIP localization framework. An effort is made in [82] and [83] to localize MIPs in NoC-based systems. However, their approaches need multiple checking and communication between the network nodes, which will further increase the complexity of MIP localization. Moreover, the methods proposed in [82] and [83] relies on a single network parameter, whose threshold is set through static profiling, to identify an MIP within the system. However, in real scenarios, a varied range of network traffic is observed due to the dynamic nature of applications running on complex SoCs. Hence, tuning a network parameter through static profiling might set it to an erroneous threshold and result in an inefficient method generating numerous false predictions. While including multiple parameters may decrease the number of false predictions, it will substantially increase the challenge of setting those parameters to appropriate thresholds. To address these issues, we present an efficient localization framework called Sniffer, which employs a low-overhead ML-based approach that tunes multiple network parameters to appropriate thresholds and helps in accurate MIP localization. Moreover, we propose a collective decision-making strategy by considering the neighbours' opinions at every intermediate node during the localization process to increase the accuracy of localization. The ML-based approach, along with collective decision-making strategy enable Sniffer to successfully localize single or multiple MIPs with least communication disruption.

5.2 Threat Model

In this work, we consider a flooding-based DoS attack initiated by one or more MIPs. The MIPs create a flooding attack by injecting frequent and useless packets into the NoC, leading to higher network latency. As a result, it manipulates the perceived availability of on-chip network bandwidth by other benign users and successfully executes a DoS attack. An example scenario is presented in Figure 5.2.a, where an MIP creates such an attack by injecting continuous random packets to a VIP through the NoC. In addition to the target VIP, routers of intermediate nodes on the attack path (shown by the blue arrow) also gets congested and undergo severe latency elongation, resulting in system performance degradation.

We consider a comprehensive threat model that encompasses different scenarios of a flooding-based DoS attack as follows:

Number of MIPs/VIPs. A single MIP can initiate an attack on a single VIP (Figure 5.2.a) or multiple VIPs (Figure 5.2.b). Multiple MIPs can also initiate attacks on a single VIP or multiple VIPs (Figure 5.2.c and 5.2.d, respectively).

Placement of MIPs/VIPs. Different placements of MIPs/VIPs can create different scenarios, where the attack paths can partially or completely overlap with each other, or form a loop. Figure 5.2 shows a few illustrative scenarios.

Coordinated and uncoordinated attack. Multiple MIPs can cooperate and mount a coordinated attack. For instance, in Figure 5.2.e, a coordinated attack is created by exchanging frequent packets between two MIPs. In an uncoordinated attack, the MIPs target VIPs independently without any cooperation.

Sniffer is able to localize MIPs in all the scenarios. In the following section, we explain its working in detail.

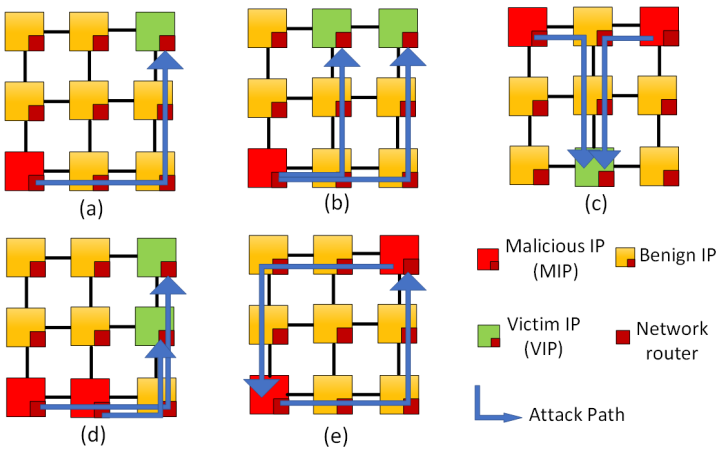


Figure 5.2: Example scenarios of multiple MIP/VIP placement.

5.3 Proposed MIP Localization Framework

An overview of our proposed MIP localization framework, Sniffer, is presented in Figure 5.3. Once a VIP detects an attack, it initiates the localization process by inspecting the congestion status of its router's incoming ports. The notion behind this analysis is that the router port falling on the attack path will exhibit high congestion in an attack scenario as compared to a non-attack scenario. Sniffer employs a machine learning approach to accurately distinguish the congestion status of a given router port in attack and non-attack scenarios. If the ML model flags the status as an attack, the VIP creates a probing packet and forwards it to the neighbour node residing in the direction of the suspected port. Consequently, each router will inspect its incoming ports and forward the probing packet accordingly. The local IP port at each node is also inspected, and the node ID is stored as an MIP if the model flags its status as an attack. In addition, Sniffer also checks if a node is traversed before and stops the process to indicate the presence of a loop. Hence, by traversing back in the opposite direction of the attack path, Sniffer successfully localizes all the MIPs.

The remainder of the section is organized as follows. The employed machine learning approach is discussed in Section 5.3.1. Section 5.3.2 elaborates our proposed framework.

5.3.1 Machine Learning for Localization

A fundamental step in Sniffer is to accurately determine the congestion status of a given router port. If a router port's congestion status is falsely classified as an attack scenario, it might increase localization time and generate unnecessary probing packets into the network. Hence, it is crucial to accurately distinguish the network traffic behaviour for attack and non-attack scenarios. A common approach used in the literature, [82] and [83], is to employ a single network parameter, whose threshold is set by static profiling, to flag anomalies in traffic behaviour. However, the heterogeneous SoCs experience varied network-level activities throughout their exe-

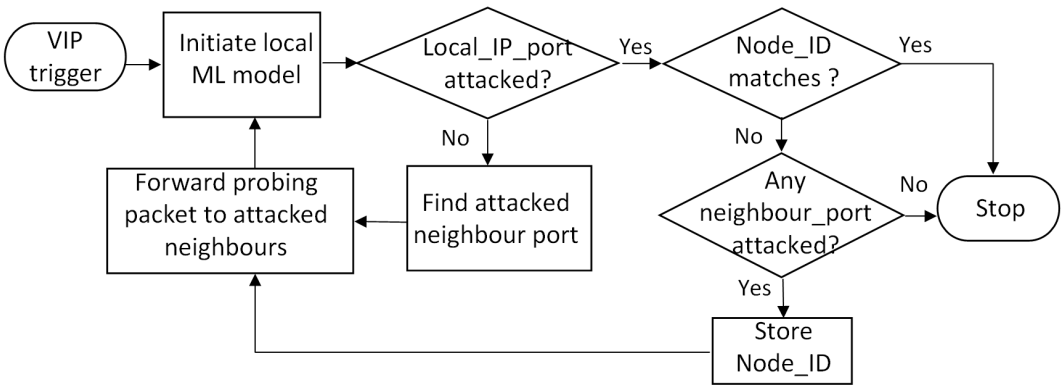


Figure 5.3: Overview of proposed MIP localization framework.

cution due to various runtime optimizations [75, 76, 30] and co-existence of different applications at different times. As a result, setting the threshold of a network parameter through static profiling for determining a port's congestion status will lead to inefficient classification. Although, using multiple parameters has been shown to increase the accuracy in various scenarios [77], it will significantly increase the complexity of the entire process. Firstly, there can be a number of network parameters that have the potential to help in distinguishing the congestion status of a router's port. One needs to meticulously explore the system or network characteristics and boil down to a set of parameters which can most accurately check the congestion status. Secondly, while tuning a single parameter through static profiling can be error-prone, tuning multiple parameters to achieve a single goal would dramatically increase the complexity and make static profiling infeasible.

In our work, we employ an ML-based approach to accurately differentiate the congestion status of a router port as attack and non-attack scenarios. An ML model is able to tune multiple parameters to appropriate thresholds for congestion detection. Therefore, it eliminates the need for onerous and error-prone tuning of parameters by static profiling, and provides a suitable solution for varied network traffic of heterogeneous SoCs. We describe the selected feature metrics, data collection and training process in the following sections.

5.3.1.1 Features

We meticulously select multiple network features to train the ML model to provide accurate congestion status. These features capture the variation in network-level activities throughout the applications' execution period.

Buffer Waiting Time (BWT). BWT is defined as the time interval for which a flit waits in the buffer of a network router. A flit will wait in the router buffer if its output port is unavailable or the input port of the downstream router is congested. Since the routers face high congestion in an attack scenario, a flit will be waiting for a significant time interval as compared to a non-attack scenario. As a result, BWT is a potential feature candidate to accurately detect the congestion status of a router port.

Inter-Flit-Interval (IFI). In a non-attack scenario, a router port observes an appreciable time period between reception of any two consecutive network flits. However, in a flooding-based attack scenario, a router port residing on the attack path observes a large number of incoming flits within a given time interval and IFI becomes significantly small. While BWT gives a good indication for an attack scenario, IFI further helps to capture the cases where, a large number of packets are destined for the same output port, resulting in higher BWT in non-attack scenarios. Hence, IFI serves as a good metric for detecting the congestion status of a given router port.

Virtual Channel Occupancy (VCO). VCO refers to the amount of Virtual Channel (VC) space occupied by the incoming flits at each network router. In case of an attack scenario, the VCs of a router on the attack

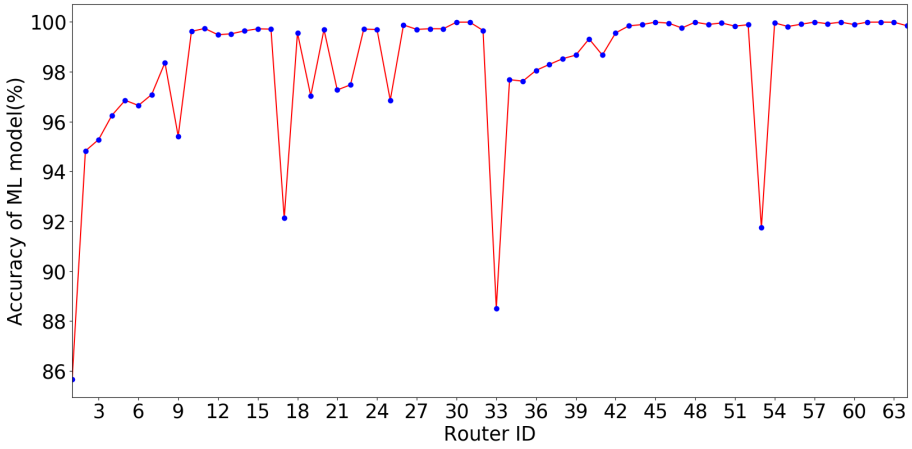


Figure 5.4: Accuracy of the ML model at each router node.

path will be heavily filled due to a large number of incoming flits. This signifies the use of VCO as an effective metric to distinguish between attack and non-attack scenarios.

5.3.1.2 Data Collection

To train the ML model, we collect data samples for attack and non-attack scenarios. Noxim [66], a cycle-accurate network simulator, is used to facilitate data collection in our experimental evaluation. We operate the simulator in two modes, namely, attack and non-attack modes. In non-attack mode, there are no MIPs, and the system experiences regular network-level activities. While in the attack mode, the MIPs target one or more VIPs and inject a large number of frequent packets into the network. The higher packet generation rate of the MIPs affect the network bandwidth and increases congestion, resulting in a flooding-based DoS attack. Since, in a heterogeneous system, the traffic density varies depending on the placement and type of IPs, we train the ML model separately for each network node to precisely capture its communication behaviour. The models are trained offline on the collected data samples to reduce computational overheads. In our experiments, we evaluate our framework on a 64-node system, while the ML models can be further trained for localizing MIPs in larger system sizes. We considered multiple runs of different benchmarks, each for 20,000 cycles during data collection (details of system configurations and benchmarks are given in Section 5.4). We collected a total of 80,640 data samples, 40,320 for attack and 40,320 for non-attack scenarios.

Discussion. While employing ML to check the congestion status enhances MIP localization; it might give false predictions in a few cases. Figure 5.4 shows the accuracy of the ML models residing at each network node. Here, we simulate a heterogeneous SoC using Noxim [66] with application traces collected from gem5-gpu [84]. We consider a 64-node system with 2D mesh NoC (system configurations and considered applications are given in Section 5.4). As shown in the figure, the ML models residing at most of the network nodes provide

high accuracy in checking the congestion status of a router port. However, relatively lower accuracies are obtained in a few of the network nodes. This variation is observed due to the heterogeneity of the system, which introduces relatively high traffic in a few nodes even in the non-attack scenario. This makes the training process of such nodes difficult as the attack and non-attack data samples can overlap.

As a probing packet moves along an attack path, the probabilities of congestion prediction at each node get multiplied. Hence, the effective probability of correctly localizing an MIP is the multiplication of these individual probabilities (equation (5.1)).

$$P(loc) = \prod_{n \epsilon N} P(E_n) \tag{5.1}$$

where N is the set of nodes on a localization path and E_n is the event of the model at node n predicting correctly. Even a small decrease in the individual model's probability reduces the effective probability by a large factor. An incorrect prediction of a single ML model in the localization path can break the chain and mislead the probing packet in a wrong direction. Hence, it is crucial to reinforce the predictions of the ML models to increase accuracy and improve localization efficiency. To achieve this, we have incorporated various design considerations into Sniffer, which are elaborated in the following section. Moreover, we also present the flow of our proposed framework in detail.

5.3.2 Efficient MIP Localization

The primary objective of Sniffer is to localize the MIPs with high accuracy in a timely manner with least disruption to regular network activities. The various design considerations introduced in Sniffer to achieve our objective are as follows:

Choice and placement of ML model: Although an ML-based solution provides an accurate congestion status, an inefficient design may introduce significant communication and/or hardware overhead. A centralized approach, where each node needs to communicate with a central ML model to determine the congestion status, will require frequent invocation and increase network traffic, resulting in communication overhead. Furthermore, multiple contenders for same ML hardware and accessing it from a distant network node will significantly increase MIP localization time. On the contrary, a distributed approach, where an ML model resides at each node, will result in faster MIP localization but would introduce additional hardware overheads. To maintain this trade-off, we choose to employ a lightweight single perceptron-based ML model at each network node. The perceptron architecture is presented in Section 5.3.3. It takes feature values and pre-trained weights as inputs, and determines the congestion status of a given router port. The perceptron-based model provides i) congestion status by tuning multiple parameters to appropriate thresholds with high accuracy and ii) simple hardware de-

sign resulting in low overhead. Hence, we consider a distributed approach where a perceptron hardware resides at each network node.

Granularity of training: In our distributed approach, a perceptron can be broadly trained at two levels of granularity: node-level and port-level. In the node-level, the perceptrons will be trained for all the router ports combined for any given network node. Whereas, in port-level, the perceptrons will be trained separately for each individual router port. Since a perceptron will learn the behaviour of an individual router port, it will result in higher accuracy for classifying congestion status as compared to node-level training. However, the percentage of perceptron related data (feature weights) to be stored on-chip grows significantly in port-level training and increases area/power overheads. To maintain this trade-off between accuracy and overhead, we use a hybrid approach where, all the incoming ports are trained together and all outgoing ports are trained together. Hence, two sets of feature weights, each for incoming and outgoing ports need to be stored on-chip. This significantly reduces the incurred overheads (almost 80%) while giving better accuracy, as compared to node-level training. We train the outgoing router ports to strengthen the localization process, which is explained in the following subsection.

Collective decision-making strategy: As indicated in Figure 5.4, few of the trained perceptron-based ML models might have less accurate predictions. This will result in increased false-positives and false-negatives. To address this issue, we propose to take a collective decision at different levels while localizing the MIPs. A collective decision is taken by the current node and its neighbour nodes before declaring a router port as under attack. While the current node checks its incoming ports, each neighbour node also uses its local perceptron to check the congestion status of its outgoing port in the direction of the current node. Each perceptron pairs' decision on the same link are fed to a probabilistic model which makes the final judgement on the congestion status, by either performing an *AND* or an *OR* operation for every pair of neighbour nodes. In the case of *AND*, the probabilistic model will predict the congestion status as an attack if both the nodes predict an attack scenario. Whereas, the congestion status in the case of *OR* operation is predicted as attack, if any of the two nodes predict an attack scenario.

The probabilistic model chooses the operation to be performed on every pair of neighbour nodes as follows. For any given pair of nodes, the probabilistic model calculates the probability of correct prediction for the *AND* and the *OR* operations based on the false-positive and false-negative rates of the individual nodes. The false-positive and false-negative rates for each router node are obtained offline by running the benchmark applications on our experimental setup (details of system configurations and applications are given in Section 5.4). If the nodes have a higher false-positive rate, i.e. they more eagerly predict attack scenarios, then the probabilistic model chooses an *AND* operation to ensure a lower rate of false-positives. Similarly, if the nodes have a higher false-negative rate, i.e. they conservatively predict attack scenarios, then the *OR* operation is chosen to ensure

a lower rate of false-negatives. Hence, the probabilistic model assigns the operation which gives the higher probability of correct prediction. As a result, it helps to reinforce the congestion check for each router port and improve prediction accuracy. The congestion checks at neighbour nodes are pipelined with the current node and do not yield any additional cycle in the localization process. Finally, along with the collective approach at the port level, all the network nodes on the attack path collectively participate in paving the way to localize the MIPs.

In the following subsection, we first present the algorithm employed by Sniffer for MIP localization. We then provide an illustrative walk-through example to demonstrate the working of Sniffer in detail.

5.3.2.1 Algorithm for MIP localization

Algorithm 1 describes the localization process of Sniffer. During the localization process, one of the three scenarios is encountered at each node as follows. In the case of a non-malicious node, only an incoming neighbour port (here, incoming neighbour port refers to the current node's incoming ports connected to the neighbour router nodes) will be congested, which signifies that the node is on an attack path. Whereas an MIP will experience only a congested incoming local IP port and other incoming neighbour ports will undergo regular network activities. However, in the case of multiple MIPs, there may be scenarios where one of the MIPs resides on the attack path of the other (Figure 5.2.d) or a loop is formed due to a coordinated attack (Figure 5.2.e). In such scenarios, the MIPs will experience congestion in both incoming local IP port and incoming neighbour port. Upon VIP triggering, the localization process starts, and final decisions are taken at each node based on the three scenarios. We explain the steps involved in our MIP localization algorithm as follows.

- At a current node, its local perceptron checks the congestion status of all the incoming router port (step 4 to 7). If the incoming neighbour port is congested ($in\_port == attack$), it is marked as under attack (step 6 and 7). At the same time, a signal is sent to the neighbour nodes to check the congestion status of their outgoing port towards the current node.

- All the neighbour nodes check the congestion status on the corresponding output port by using their locally residing perceptron hardware. If any neighbour node finds its outgoing port congested ($out\_port == attack$), it marks it as under attack and communicates this information back to the current node (step 9 to 12). The perceptrons' decisions of both current and neighbour nodes for each direction are fed to the probabilistic model, which finally declares a router port as under attack or non-attack scenario (step 13 and 14).

Algorithm 1: MIP Localization Algorithm

```
1  MIP[ ]=Null; port_attacked[ ]=Null
2  Upon VIP trigger:
3  do in pipeline
4      foreach in_port from N,E,S,W direction do
5          /*Perceptron check congestion in current node*/
6          if in_port == attack then
7              Mark in_port as attack
8  do in pipeline
9      foreach neighbour in N,E,S,W direction do
10         /*Perceptron check congestion for its outgoing port towards current node*/
11         if out_port == attack then
12             Mark out_port as attack
13         if prob_model(in_port, out_port)==attack then
14             port_attacked += direction
15 if local_IP_port ≠ attack then
16     Forward probing packet
17 else
18     if port_attacked == Null then
19         MIP += Node_ID
20         Stop & return MIP
21     else
22         if Node_ID not traversed then
23             MIP += Node_ID
24             Forward probing packet
25         else
26             Stop & return MIP
27 Reset port_attacked
28 Goto next node with probing packet & Repeat from step 3
```

N=North, E=East, S=South, W=West

- If the incoming local IP port of current node undergoes regular network activities ($local\_IP\_port \neq attack$), it indicates that the node is on the attack path and not the attacker. In such scenarios, a probing packet is forwarded to the neighbour node in the direction of the port under attack (step 15 and 16).

- If the incoming local IP port is under attack and incoming neighbour ports undergo regular activities, then the localization process stops and current $Node\_ID$ is flagged as the MIP (step 18 to 20).

- If both the incoming local IP port and neighbour ports are under attack, our framework first checks if the current node is already traversed. If the current node is not traversed before, the Node ID is stored in a probing packet and forwarded to the neighbour node in the direction of the port under attack (step 21 to

24). If the current node is traversed before, it implies that Sniffer has encountered a complete loop due to a coordinated attack by multiple MIPs. As a result, the localization process stops and all the $Node\_IDs$ stored in the probing packet are flagged as MIPs (step 26).

In the case of a single MIP or multiple MIPs with no closed loop, Sniffer stops as soon as the probing packets flag the attackers. However, in the case where a closed loop is encountered or the attack paths overlap, Sniffer extracts all the Node IDs inserted in the probing packet and flags them as MIPs which were creating a coordinated attack. In scenarios where one MIP targets multiple VIPs, the MIP will be localized by the probing packet that takes minimum time to reach the MIP node. When the MIP gets localized, the probing packets generated from other VIPs do not find any router port under attack and get dropped. Once localized, the operating system shuts down all the MIPs to prevent any further attacks and the system is restored to allow the legitimate IPs to undergo regular network activities.

5.3.2.2 Walk-through example

We explain the MIP localization process of Sniffer with single/multiple MIPs/VIPs which include different scenarios as shown in Figure 5.2. We first demonstrate the localization process using a walk-through example for a single MIP and VIP scenario. Without the loss of generality, we present a transaction-level figure (Figure 5.5) for the walk-through example with a 16-node system. Later, we describe how our proposed framework works in the presence of multiple MIPs/VIPs. Sniffer is able to work in conjunction with any existing DoS attack detection methods where a system is flagged when experiencing a flooding attack. To illustrate our localization process, we consider an example state-of-the-art flooding-based DoS attack detection method presented in [40]. After a flooding attack is mounted on the system, one or more VIPs raise a flag indicating that the system is under attack. Whenever an attack flag is raised, Sniffer starts the MIP localization process from the VIPs which generate the attack flag.

Single MIP and VIP. Figure 5.5.a shows the attack path through which an MIP (Node 12) creates a flooding attack targeting a VIP node (Node 6). Node 6 flags the system as under flooding attack and the localization process is initiated.

- In Step 1 (Figure 5.5.b), Node 6 checks if its incoming port in the North direction is congested or not using its locally residing perceptron model. At the same time, it also sends a signal (chk_signal) to its immediate neighbour nodes (2, 7, 10 and 3) to start the collective decision-making on the congestion status at each port.

- In Step 2 (Figure 5.5.c), after completing the congestion check at North direction, the perceptron model

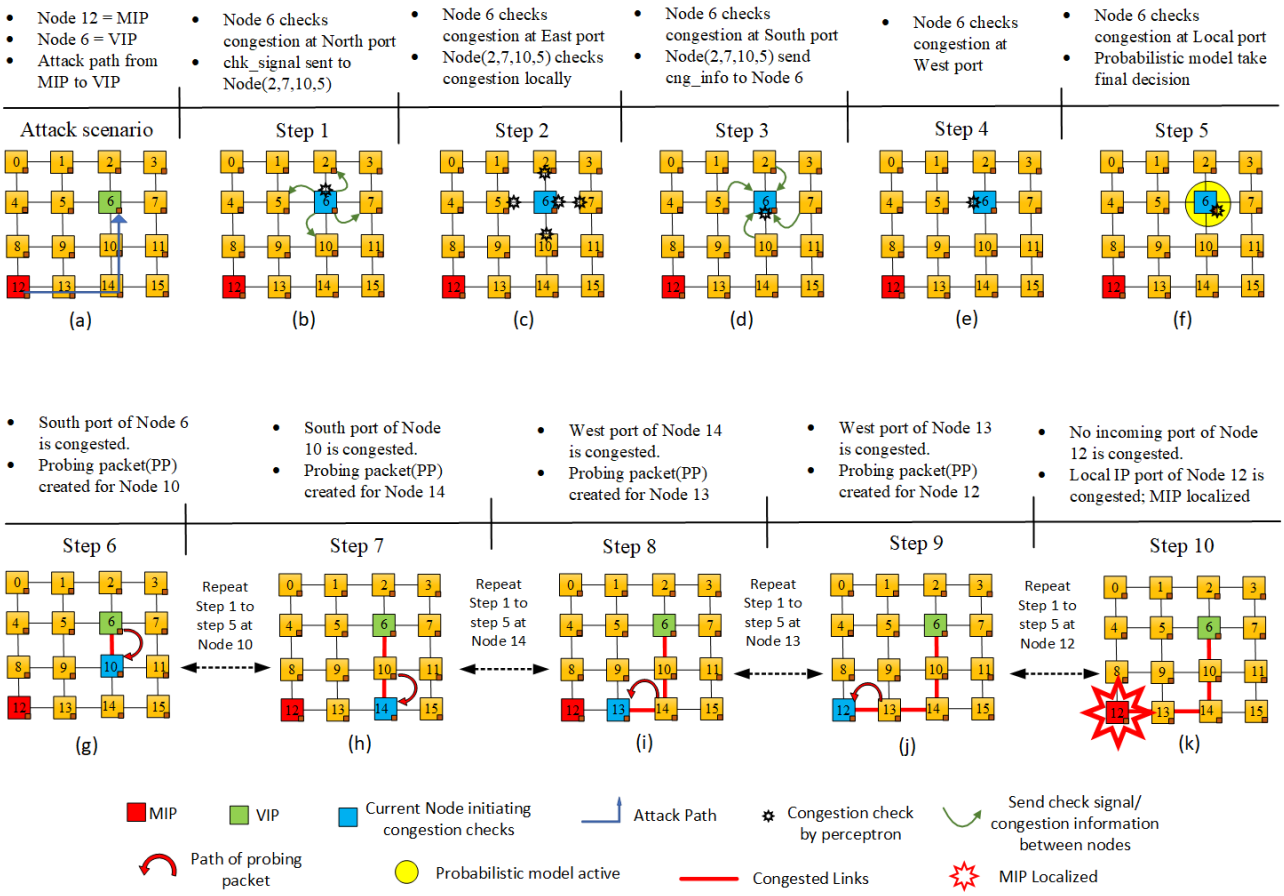


Figure 5.5: A walk-through example for MIP localization by Sniffer with an example attack scenario (Node 12= MIP, Node 6 = VIP).

at Node 6 checks its incoming port at the East direction for congestion. The local perceptrons at Node 2, Node 7, Node 10 and Node 3 also checks the congestion status of their corresponding output ports in the direction of the received chk_signal.

- In Step 3 (Figure 5.5.d), Node 6 subsequently makes a congestion check at its incoming South port, while Node 2, Node 7, Node 10 and Node 3 send their computed congestion information (cng_info) back to Node 6.

- In Step 4 (Figure 5.5.e), Node 6 checks its incoming port in the West direction for determining its congestion status. It is to be noted that Sniffer employs a single perceptron hardware at each router node for low overhead implementation. So, the perceptron at the current node starts the checking of the congestion status of the incoming ports one after another. There is no specific rule for the order in which the ports need to be checked. For this example case, we have considered the checks on incoming neighbour ports in N-E-S-W order.

- In Step 5 (Figure 5.5.f), Node 6 makes a congestion check at its incoming Local IP port to determine if Node 6 is an attacker. After receiving all the congestion check results from its local perceptron and neighbour nodes' perceptrons, the probabilistic models at Node 6 take the final decision for congestion at all its incoming ports.

- If any of the incoming ports is marked as under attack, a probing packet is generated towards the node residing in the corresponding direction. In our example scenario, Node 6 generates a probing packet for Node 10 as shown in Figure 5.5.g. Once Node 10 receives the probing packet, it performs all the steps done at Node 6 (Step 1 to Step 5) to check the congestion status of its incoming ports. Eventually, the incoming South port of Node 10 is found to be congested and the probing packet is forwarded towards Node 14.

- Similarly, Step 1 to Step 5 are repeated at Node 14 and subsequently at Node 13 to find the direction of the probing packet, as shown in Figure 5.5.h and 5.5.i.

- As Node 12 receives the probing packet, it starts its congestion checks by following Step 1 to Step 5, as shown in Figure 5.5.j. Since Node 12 is the only attacker node in this example case, all its incoming neighbour ports will undergo regular network activities, and its incoming local IP port will be congested. Hence, no probing packet is generated, Sniffer flags Node 12 as an MIP node and the localization process stops (as indicated in Figure 5.5.k).

Multiple MIPs/VIPs. Figure 5.2.b to 5.2.e show scenarios where multiple MIPs/VIPs co-exist within the system. Under such scenarios, multiple MIPs can target the same or different VIPs to create a flooding-based DoS attack. They can also mount an effective coordinated attack where multiple MIPs forms a closed loop in the network.

In the case where multiple MIPs follow different attack paths to target different VIPs, Sniffer parallelly starts the localization process from the corresponding VIP nodes. A probing packet starts from each VIP node and follows all the steps described in the walk-through example (Figure 5.5). Each probing packet reaches the corresponding MIP node by backtracking and traversing through all the nodes in the attack path. In the case where the attack paths overlap (for e.g., Figure 5.2.c and Figure 5.2.d), the probing packets are merged and forwarded towards the corresponding node residing at the direction of the congested port. In the case of a closed loop formed by multiple MIPs, (Figure 5.2.e), whenever any one of the nodes flags the presence of an attack, Sniffer starts the localization process and follows the steps shown in the walk-through example. However, the probing packets might keep on traversing through the nodes in a cycle as all the nodes will always find one of their incoming ports under attack. To overcome this situation, Sniffer stores the Node IDs of the attacker

nodes in the probing packets. Whenever it encounters a node which is traversed before, Sniffer identifies a loop and stops the localization process. It then marks all the Node IDs stored in the probing packets as the MIPs. Therefore, Sniffer is able to successfully localize multiple MIPs under different MIP/VIP placements and attack scenarios.

In the above example scenario, we assume that the probing packets will always find free network resources to traverse through the localization path, which is typically taken care of by timely flag generation in robust attack detection algorithms. However, in the worst-case scenario, the network resources may be blocked due to severe and prolonged attacks. To handle such a rare case, the network router will drop packets residing in a VC of the corresponding port which is requested by the probing packet. This will allow a steady forwarding of probing packets and provide efficient MIP localization.

5.3.3 MIP localization Hardware

In this Section, we present the hardware infrastructure required by Sniffer to perform MIP localization. Each network node is augmented with a localization hardware primarily comprising of the perceptron hardware, along with the hardware to facilitate perceptron's weight storage and feature data collection. Figure 5.6 shows the architecture of the modified router and perceptron hardware employed in our proposed framework. The Feature Collection Hardware (FCH) is capable of capturing three different features, i.e., BWT, IFI, and VCO. An 8-bit Timestamp Unit (TU) is implemented in each router, which provides the Arrival Time (AT) of the packets entering the input ports of a router node. Whenever a packet reaches a router's input port, the AT value indicated by TU is stored in the unused bits of its header flit. The individual packet waiting time in the buffer is calculated by subtracting the packet arrival time from the current timestamp value. Average of all the packet waiting times of a particular port is considered as the BWT for that port. Similarly, the arrival time difference between consecutive packets are derived, and an average value per port is computed to provide individual port's IFI metric. Another 3-bit counter is implemented to keep track of the VC occupancy. The Weight Storage Hardware (WSH) uses seven 8-bit registers to store the pre-trained feature weights; three each for storing input and output port feature weights, and one for the bias. A Data Prioritization Hardware (DPH) is also implemented in each network router, which assigns network resources with the highest priority to the probing packets. This allows a smooth transition of probing packets between the network nodes and provide efficient MIP localization. A priority bit is included in the head flit of each probing packet, that distinguishes them from regular packets in the network. In the case of *Wormwhole switching* (the flow control method adopted in our framework), only the head flit of each packet competes and reserves the VC. An additional 1-bit Priority Field (PF) is included in each VC of the router as DPH. Once the head flit of a packet reserves a VC, the PF is

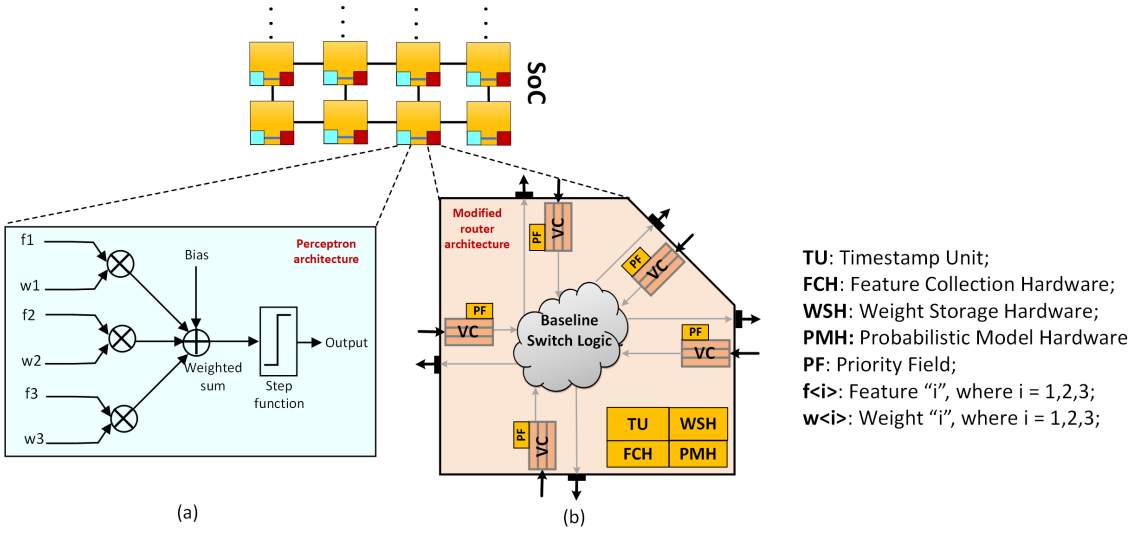


Figure 5.6: Hardware infrastructure required by Sniffer for MIP localization. (a) Perceptron architecture, (b) Modified router architecture.

updated according to the priority bit of the head flit (PF stores *1* for a probing packet and *0* for a regular packet). The subsequent body flits of the packet use this PF value during the switch arbitration to provide prioritized movement of probing packets. The perceptron hardware takes the feature data collected by FCH as inputs, along with the pre-trained weights and bias stored in the WSH structures. As shown in Figure 5.6, the perceptron architecture comprises of multipliers, which multiplies the input features with their respective weights. The resultant products are then added together along with the bias to generate a weighted sum. This weighted sum is then passed through an activation function, which returns a final output. We consider the activation function as a simple step function which outputs either *0* or *1*, based on the set threshold. An output of *0* denotes a non-attack scenario, whereas an output of *1* denotes an attack scenario. In our case, three multipliers and one adder are implemented to generate the weighted sum for the three input features, along with a comparator for the step function. A Vedic multiplier proposed in [85] is implemented within the perceptron to save area and power. In addition, the Probabilistic Model Hardware (PMH) consists of four 2-input AND/OR logic gates, which facilitates taking the final decision on the congestion status of the incoming neighbour ports. The area and power overheads of the MIP localization hardware are presented in Section 5.4.4.

5.4 Experimental Evaluation

In this Section, we discuss the experimental setup and evaluation of the proposed MIP localization framework.

5.4.1 Experimental Setup

To evaluate the performance of Sniffer, we use a cycle-accurate network simulator Noxim [66]. The MIP localization algorithm is implemented in Noxim, which is able to localize the MIPs with high accuracy, after a VIP triggers a flag to indicate an attack scenario. In our experiments, the DoS attack is created by allowing an MIP to send a new packet at every cycle to a targeted VIP. The extra packets are generated on top of a node's regular packet generation rate for an interval of 20,000 cycles. The data collected from the experiments is split into a ratio of 70:30 for training and testing purposes.

We test the system using real-world heterogeneous benchmarks from *Rodinia* [86] benchmark suite. All the application-level traces are obtained from gem5-gpu [84] and fed to Noxim, which emulates the network communication behaviour. We evaluate our proposed framework in terms of accuracy of the perceptron models, the performance of MIP localization, the time taken to localize the MIPs, benign packet loss and incurred hardware overheads. Table 5.1 summarizes our simulation setup and system configurations.

5.4.2 Performance of Perceptron Model

The performance of perceptrons in checking the port congestion status is evaluated in terms of True Positives (TP) and True negatives (TN), i.e., correct classification of congestion status as attack and non-attack respectively, and false-positives (FP) and false-negatives (FN), i.e., incorrect classification of congestion status as attack and non-attack respectively. A higher value of TP and TN indicates that the perceptrons can more accurately classify the congestion status. Whereas, a lower FP and FN represents the robustness of the classification results. We consider several MIP and VIP pairs with different placements for each benchmark application. Table 5.2 presents the perceptron accuracy results (average accuracy among all the network nodes) for different benchmark applications. As shown in the Table, our employed perceptron model is able to provide high accuracy with an average of 98.35% and 98.30% for TP and TN values, respectively. The FP and FN values are also

Table 5.1: Simulation Setup for evaluating Sniffer

Component	**Configuration**
Topology	8X8 2D mesh, XY routing, 1-cycle link latency
Router	5 I/O ports, 1 VCs per port, 4 flit VC buffer, 32 bit flits, wormhole switching, 3-stage router, 2 flit packet
Processing Cores	8 x86 OOO CPU cores; 32 GPU cores
Memory	32KB private L1 cache, 256 KB L2 cache, 8 L2 caches, 2MB L3 cache; 4GB DRAM
Workloads	Rodinia benchmark suite: Backprop, Gaussian, Hotspot, Kmeans, Lud, Mummergpu

Table 5.2: Performance of perceptron model

Benchmarks	TP (%)	FP (%)	TN (%)	FN (%)
Backprop	98.33	1.71	98.28	1.66
Gaussian	98.32	1.73	98.26	1.68
Hotspot	98.35	1.69	98.30	1.65
Kmeans	98.36	1.69	98.31	1.63
Lud	98.34	1.72	98.27	1.65
Mummergpu	98.35	1.66	98.33	1.64
Average	98.35	1.70	98.30	1.65

very less with an average of 1.70% and 1.65% respectively across all benchmarks, signifying the robustness of the model. Hence, the perceptron models trained with carefully selected features provide high accuracy for localizing the MIPs.

5.4.3 Efficiency of proposed localization algorithm

To demonstrate the efficiency of our proposed framework, we first show the performance of Sniffer in localizing the MIPs. Then, we present the time taken for MIP localization across different MIP and VIP pairs. We also assess the time taken for localization in the presence of multiple MIPs. Finally, we analyze the improvement in system performance due to localization under a flooding-based DoS attack.

5.4.3.1 Performance of MIP localization

In this Section, we evaluate the performance of Sniffer in correctly localizing MIPs within the system. We run real-world heterogeneous benchmarks from Rodinia benchmark suite [86] with multiple test cases. All the test cases comprise of different MIP and VIP pairs along with various placements of MIP and VIP nodes across the network. We include multiple design considerations in our framework such that it provides a stable performance across different benchmarks. The node-level training, where each perceptron at a router node is trained on data collected from different heterogeneous benchmarks, ensures that each perceptron model gets a holistic view of various possible scenarios. Along with a carefully designed training method, the probabilistic model for collective-decision making strategy enhances the congestion checks at each router node, which further improves the performance of Sniffer in MIP localization. Table 5.3 shows the MIP localization accuracy of Sniffer for different benchmark applications. As given in the table, Sniffer provides high localization accuracy of 96.754% on average. It is also evident from the table that Sniffer is able to provide a consistent performance across the benchmarks, which makes our proposed framework robust across different applications. To demonstrate the importance of the collective-decision making strategy of Sniffer in localizing MIPs, we evaluate the

Table 5.3: Performance of MIP localization

Benchmarks	Localization Accuracy
Backprop	96.739 %
Gaussian	96.691 %
Hotspot	96.773 %
Kmeans	96.789 %
Lud	96.722 %
Mummergpu	96.808 %
Average	96.754 %

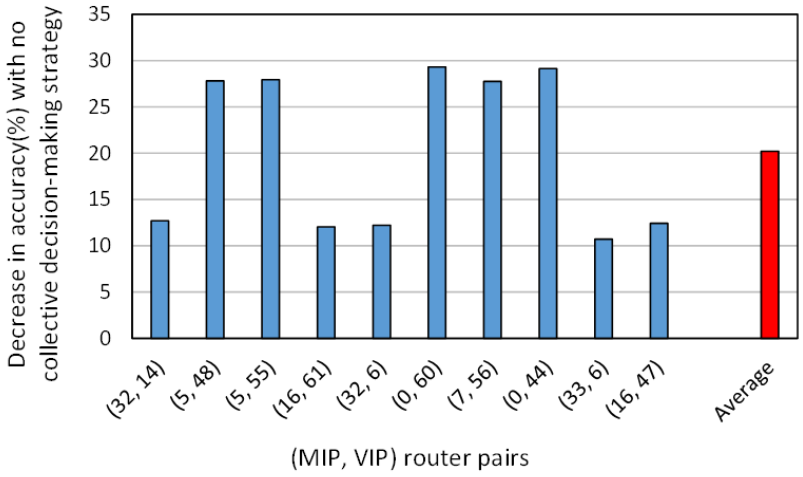


Figure 5.7: Decrease in MIP localization accuracy in the absence of collective decision-making strategy of Sniffer.

performance of Sniffer in the absence of the probabilistic model. Here, we consider that a perceptron is present at each network node and we rely only on its decision for congestion check at the incoming ports of the current node. Based on the congestion decision of the locally residing perceptron, a node creates probing packets which traverse through multiple nodes to localize the MIPs. For our experimental evaluation, we consider a few test cases, where all the test cases comprise of different MIP and VIP pairs. Figure 5.7 demonstrates that an average decrease of 20.19% is observed in Sniffer's localization accuracy in the absence of the collective decision-making strategy. This is because of the following reason. Although most of the individual perceptrons show high accuracy for congestion detection, a relatively lower accuracy of one or more perceptrons on the localization path would reduce the effective probability by a large factor (as explained in Section 5.3.1). This in turn, will mislead the probing packets in wrong directions and falsely localize the MIPs. By incorporating the collective decision-making strategy, Sniffer is able to reinforce the congestion checks at each router port and allow the probing packets to localize the MIPs with high accuracy.

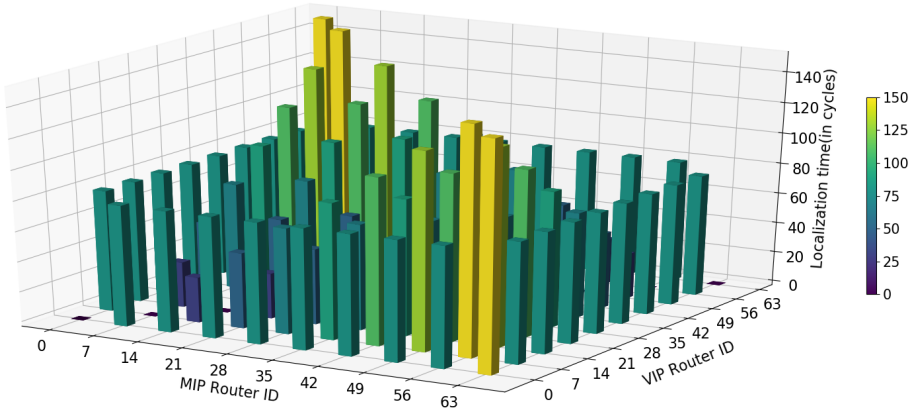


Figure 5.8: MIP Localization time.

5.4.3.2 MIP localization time

The MIP localization time of Sniffer depends on the time taken by the probing packet to traverse through the network nodes, starting from the node at which the localization process is initiated (VIP) to the node at which it is stopped (MIP), and the time taken for congestion checks at each node on the localization path. Figure 5.8 shows the localization time of Sniffer across various test cases, where each test case consists of a single MIP and VIP pair. As evident from the figure, Sniffer is able to localize the MIPs in a timely manner across all the test cases. The worst-case localization time for a 64-node system is found to be as low as 150 cycles in our experimental evaluation. The efficiency of Sniffer in localizing the MIPs in a timely manner is achieved primarily by the collective decision of perceptrons and prioritized probing packet transmission. In large system sizes, the search space for attack localization is vast, as there are a number of possible network nodes which can be a potential attacker. The collective decision of perceptrons significantly decreases the search space by allowing Sniffer to make congestion checks with high accuracy at each step of localization. Hence, Sniffer needs to traverse only through the nodes residing on the attack path, and localize the MIPs in a timely manner. Furthermore, Sniffer forwards the probing packets with higher priority through the attack path, which leads to a faster MIP localization. From Figure 5.8, it is observed that the time required to localize an MIP varies across different (MIP, VIP) pairs. The variations in the localization time are primarily due to the varied distance between each MIP and VIP pair. The localization time increases with the increase in hop distance between the MIP and VIP nodes, as shown in Figure 5.8.

We also analyze the localization time of Sniffer in the presence of multiple MIPs and/or VIPs. In the case, where multiple MIPs target different VIPs present in the system, the time taken to localize all the MIPs is the maximum localization time across all the MIP and VIP pairs. In the case of a coordinated attack mounted by multiple MIPs, which forms a closed loop within the network, the localization time will depend on the time taken to complete one iteration of the loop and the congestion checks at the corresponding network nodes. If

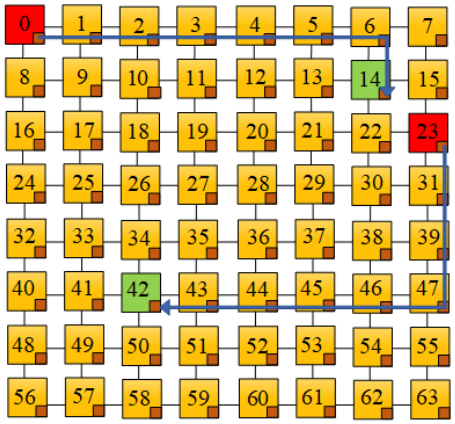


Figure 5.9: A 64-node system showing example attack paths (blue arrow) for (0, 14), (23, 42) router pairs of first test case given in Table 5.4.

Table 5.4: Localization time in the presence of multiple MIPs

Test cases	**Router pairs (MIP, VIP)**	**Localization time for each (MIP, VIP) pair (cycles)**	**Final localization time (cycles)**
2 MIPs & 2 VIPs	(0, 14), (23, 42)	70, 80	80
2 MIPs & 1 VIP	(24, 7), (56, 7)	100, 140	140
3 MIPs & 1 VIP	(29, 22), (40, 22), (62, 22)	20, 50, 70	70
1 MIP & 2 VIPs	(23, 6), (23, 48)	30, dropped	30
2 MIPs (coordinated attack)	(39, 1) coordinates with (1, 39)	(50, 100)	100

[1] Please refer to Section 4.3.2.2.

the system contains only one MIP that targets different VIPs, the total localization time will correspond to the minimum time taken by any one of the probing packets generated from all the VIPs to reach the MIP. This is because, once the MIP is localized and the system is restored, the probing packets generated from other VIPs do not find any congestion in the router ports and get dropped. Table 5.4 shows the localization time for a (MIP, VIP) pair for each test case and the final localization time in the presence of multiple MIPs/VIPs. Here, we show a few illustrative test cases, where each test case comprises of different MIP and VIP pairs. For better visualization, we present Figure 5.9, which shows a 64-node system, highlighting the Node IDs. Moreover, for ease of understanding, we have shown the attack paths for the first test case from Table 5.4.

As evident from the table, in the case of multiple MIPs targeting different VIPs, the final localization time corresponds to the maximum of the individual localization time of the (MIP, VIP) pairs in each test case. Whereas, in a single MIP and multiple VIP case, the probing packet with minimum hop distance localizes the MIP, and the probing packets generated from other VIPs get dropped. Lastly, under a coordinated attack[1], the final localization time of all the MIPs is found to be the time required to check one iteration of the closed loop

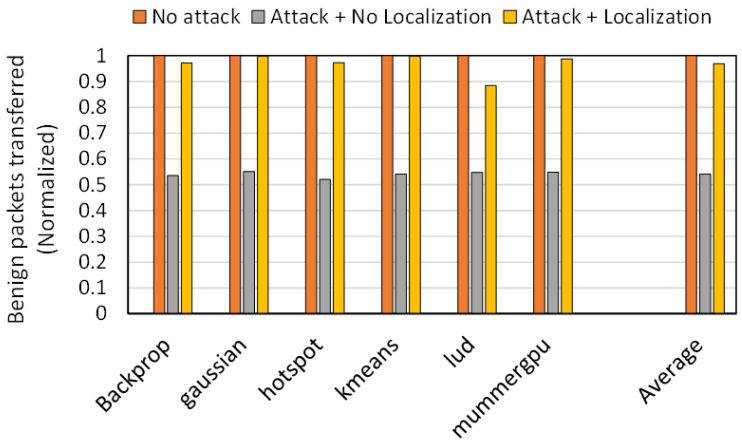


Figure 5.10: Benign packets transferred in different scenarios.

under attack. For instance, a test case for 2 MIPs, where Node 39 and Node 1 create a coordinated attack, is shown in Table 5.4. In this case, Node 1 generates a probing packet, which traverses through the localization path to reach Node 39, and then back to Node 1. When the probing packet reaches Node 1, one iteration of the loop is completed and both the MIPs get localized. Hence, the final localization time is the time taken to complete one iteration of the loop, which is found to be 100 cycles in this case. In this scenario, Node 39 will also generate a probing packet, apart from Node 1. However, the probing packet that will complete the loop first will localize both the MIPs.

5.4.3.3 Impact on application performance

We analyze the network traffic to understand the impact of Sniffer's localization process on applications running on the SoC. We observe the number of benign packets successfully transmitted to their destination nodes in three scenarios: i) baseline system with no attack ii) system under attack with no MIP localization method, and iii) system under attack with Sniffer deployed. To make a fair comparison, we evaluate all the three scenarios for the same time interval across all benchmarks. In an attack scenario with no MIP localization, as the network gets flooded with malicious packets, the applications experience performance degradation due to significant packet loss. This is also evident from Figure 5.10, where the number of successfully transferred benign packets reduced to almost half as compared to the non-attack baseline scenario. However, in the third case, due to the presence of Sniffer, the system under attack gets restored timely, and prevent further packet loss. Hence, Sniffer reduces the effect of flooding attacks and curtails system performance degradation.

5.4.4 Area and Power of Localization hardware

Sniffer provides an ML-based accurate MIP localization framework with low hardware overhead. The MIP localization hardware presented in Section 5.3.3 is synthesized in Synopsys DC compiler for 32nm technology, and is observed to occupy a total area of $0.0018mm^2$. Whereas, the baseline router synthesized for same technology node occupies $0.053mm^2$ of on-chip area. This implies that the area overhead of the proposed hardware is as low as 3.3% of the baseline architecture. The perceptron module is activated only during MIP localization, and is power-gated during regular system operation. Hence, the localization hardware consumes $18\mu W$ during regular operation, and $123\mu W$ during MIP localization, per router.

5.5 Conclusion

In this work, we present Sniffer, an efficient framework for localizing one or more MIPs creating a flooding-based DoS attack on heterogeneous NoC-based SoCs. Sniffer employs machine learning approach along with collective decision-making strategy to trace back the attack path with high accuracy and promptly localize the MIPs. Our experimental results with real-world heterogeneous benchmarks show an average of 96.754% of accuracy in detecting the MIPs in a timely manner with least traffic disruption.

Chapter 6

Data-flow Aware CNN Accelerator with Hybrid Wireless Interconnection

In this chapter, we explore the design-space of the Convolutional Neural Network (CNN) accelerators. We first give a brief description of the state-of-the-art accelerator designs for the CNN applications. We discuss the typical accelerator architectures, their communication infrastructures and the challenges involved in designing such systems. We then present our proposed data-flow-aware design for CNN accelerator in detail.

6.1 Convolutional Neural Network Accelerators

Hardware accelerators have emerged as an indigenous part of heterogeneous computing platforms as the demand for computation power is ever increasing at an exponential rate. The performance and energy efficiency of application specific designs have presented hardware accelerators as an effective alternative to general purpose CPUs and GPUs for certain applications [87]. Since application specific designs have trade-offs between their performance & energy efficiency and generality & cost, their use is well suited to certain class of computations. Machine learning is one such domain which gains significant performance benefits on accelerators. Furthermore, a single machine learning approach spans a wide range of applications, making accelerators well suited to them regarding generality in use [13, 14, 15].

To obtain high computational parallelism, most of the CNN hardware accelerator designs employ multiple PEs with local memory to store or reuse the partially computed data. However, as the number of PEs increases, the system performance does not scale at the same rate due to the overhead of inter PE and off-chip memory communication. A few accelerator architectures overcome this through efficient design of memory systems, reusing data once brought on-chip or storing partially computed data in local scratchpad memory for

future computations [50, 18]. Although the communication backbone of high performance computing systems, on-chip interconnection in existing accelerators has not been aptly addressed to reduce the communication overheads. They tend to employ traditional interconnect systems like bus, mesh, crossbars etc. In this work, we present an efficient interconnect design to address the communication bottlenecks in hardware accelerators for CNN algorithms. We perform an extensive analysis of communication patterns of CNN applications in different accelerator architectures. Using this data-flow profiling, we propose a fused interconnect design with wired and mm-wave wireless links between PEs and on-chip memory elements. The low latency, broadcast capable wireless links are used for weight transmission to all PEs, while computed results are propagated between PEs for further reuse or finally to off-chip memory using wired links. This fused interconnection provides a framework to tackle the communication bottlenecks of traditional interconnects allowing integration of more number of PEs in future accelerator designs. The following sections briefly describes the state-of-the-art accelerator architectures for CNN applications and their data-flow patterns. We then discuss the challenges of communication infrastructures in accelerators and elucidate the possibilities of designing a better interconnect to enhance performance.

6.1.1 Accelerator Architectures and Communication Data-flow

A generic hardware accelerator design consists of a host processor, accelerator cores or PEs, memory system and communication infrastructure. The host processor is responsible for generating macro instructions for the PEs. All components are interconnected by a communication infrastructure that facilitates data movement between the on-chip modules. A CNN hardware accelerator primarily deals with two types of input data: input neurons or activations and kernel weights. Each PE performs a convolution operation on the two inputs resulting in an output neuron or a partial sum for generating the output neuron. Different accelerator architectures takes various approaches to provide these inputs to PEs resulting in varied on-chip communication data-flow. It is critical to understand this data-flow pattern in hardware accelerators to design efficient interconnection architectures for improving performance. Hence, we have done a meticulous study of the communication patterns in accelerators and have divided them into categories based on on-chip data-flow.

6.1.1.1 Broadcast or multicast weights and input neurons

CNN accelerators achieve high parallel computation by making data available to each PE at every clock cycle. Also, the inherent data-sharing nature of CNN algorithms allow the accelerators to reuse input data or partially computed results among multiple PEs. Hence, to meet the high-data demand the accelerators tend to adopt various data-flow patterns which involve either broadcasting or multicasting the input data. To reduce the costly

memory accesses, some accelerator designs broadcast the same kernel weights to a group of PEs leveraging the kernel-sharing nature of the algorithms. The input activations are muticasted to the PE array and reused in subsequent cycles. Each PE is responsible for either generating a single output neuron or a partially computed output neuron. Finally, all the output neurons are sent back to the global memory through the on-chip network that becomes the input for next CNN layer.

6.1.1.2 Broadcast input activations with pre-stored weight

In this data-flow mechanism, the kernel weights are periodically pre-loaded from the DRAM to PE's local memory. The input activations are broadcasted or multicasted from the global on-chip memory to the PE array. Although, such a design reduces the overhead of continuously accessing the weights from global memory, it significantly increases the required on-chip storage.

6.1.2 Interconnect Infrastructures for hardware accelerators

Interconnection architectures are well studied for general purpose processors based on their topologies or architectures [88]. In this section, we focus on conventional communication infrastructures employed in an accelerator-rich environment. We analyze different interconnect paradigms for accelerators and study the potential of incorporating emerging interconnects like mm-wave wireless links to enhance the performance.

6.1.2.1 Broadcast-Bus

In a broadcast-bus based system, PEs are connected to a multi-bus system [48, 18], which is responsible for transmitting input and output data between PE's local memory and global memory. This type of infrastructure allows ease of implementation and has minimal packaging costs. However, buses become severely inefficient as the number of PEs increases. This is primarily due to physical limitations of the wires and contention for the shared interconnection resource. Hence, a bus-based architecture is highly unscalable and is acceptable only in presence of infrequent memory accesses at small system sizes.

6.1.2.2 Crossbar

A crossbar system comprises n-input and m-output switches connecting different on-chip modules. Unlike CPUs, accelerators run 100x faster and performs several load/store operations per clock cycle. Crossbars facilitate conflict-free data accesses by allowing simultaneous communication between accelerator ports and multiple memory banks. However, with an increase in the number of nodes, scalability suffers as the cost of

crossbar implementation increases significantly in terms of area and power.

6.1.2.3 Two-Dimensional (2D) Mesh

A typical 2D mesh comprises of PEs communicating with each other through routers and inter-router links forming a 2D structure. It's 2D structure is suitable for processing images and is found to be acceptable for accelerator designs[54]. Although it has gained popularity due to its simplicity and scalability, the performance of mesh starts getting adverse for very large systems. This performance limitation arises from planar metal interconnect-based multihop communications, wherein the data transfer between two distant PEs results in high latency and power consumption. As system size increases, so does the distance between distant nodes, resulting in further delay and energy consumption.

6.1.2.4 On-chip wireless interconnects

Several novel approaches [89, 90] have been explored to address the limitations of traditional multi-hop wired NoCs, of which mm-wave wireless interconnects offer the most feasible solution because of their compatibility with current fabrication methods. Wireless interconnect provides performance enhancement through optimally placed long range, low power and low latency wireless links in large system sizes. It significantly reduces the long distance communication bottlenecks of many core architectures, resulting in huge performance gains. As system size increases, WNoC provides considerable performance gain by reducing the long distance bottlenecks of the wired network. Furthermore, wireless links have inherent capability of broadcasting, which is especially useful for CNN accelerators that have several data accesses shared across multiple PEs in the system.

The findings of our study on CNN accelerator data-flow and communication infrastructures boils down to the following conclusions:

- The CNN accelerators build upon traditional communication infrastructures like Bus, Mesh or Crossbars cannot achieve its peak performance when the system is comprised of a significant number of computing elements and memory modules.

- To achieve the utmost possible advantages from optimization techniques in data-flow and memory hierarchies, it is important to rethink on employing emerging interconnections within the system. The mm-wave wireless links turns out to be an effective communication alternative. The low-latency and less energy consumption of on-chip wireless links provides an attractive solution over the primitive interconnection designs.

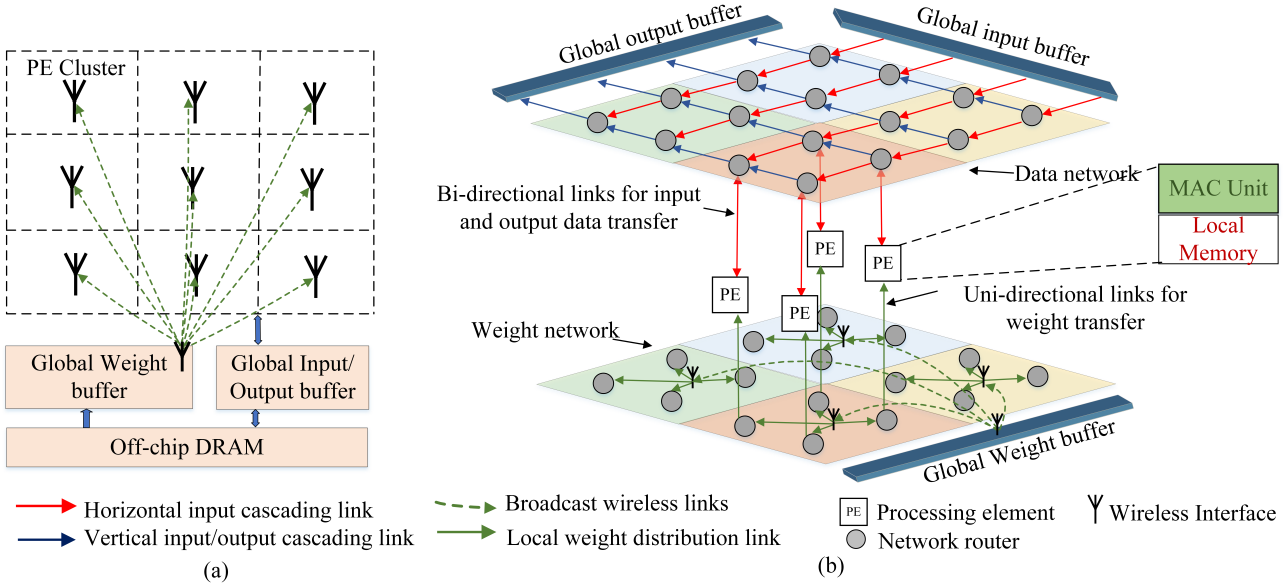


Figure 6.1: Proposed Hybrid Interconnection: a) Overview of system architecture; b) Data-flow of weight and data networks

6.2 Hybrid Interconnection for CNN Accelerators

In this section, we describe the proposed data-flow aware hybrid interconnection architecture for CNN accelerators. We analyze the proposed architecture with a work-through example to support the data-flow pattern discussed before.

6.2.1 System Architecture

We first present a brief overview of the system architecture before describing the wireless interconnection in detail. Most conventional accelerator architectures for CNNs use multiple networks to transfer the inputs and collect the output data and the proposed system architecture follows the same. We implement a dual layered network, taking into account the movement of data among PEs, to design a network that provides optimal use or reuse of existing infrastructure combined with emerging interconnections. Fig. 6.1 shows the architecture of the proposed hybrid interconnection design. The data network, responsible for propagating input and output neurons, communicates over a mesh design, while weight network makes use of wireless interconnection for efficiently transmitting the kernel weights. As shown in the Fig. 6.1.b, each PE connects to both network layers through its network interface. At each clock cycle, an input weight and an input activation are available to every PE, that performs a convolution operation and stores the result in its local memory.

6.2.2 Wireless Based Weight Network

One of the challenges for deep learning accelerators is the requirement to make data available to all PEs immediately to prevent them from starvation. In CNNs, as same weights are shared across multiple output neuron computations, they are broadcasted or multicasted to the PE array leading to high communication latency. This high latency incurred due to long distant broadcast or multicast in CNN accelerators can be addressed by incorporating a small-world topology [91]. In a small-world topology, the entire network is divided into multiple smaller clusters and communication shortcuts are established between the clusters to decrease the transfer overhead across distant nodes. In the proposed architecture, we design a similar interconnection by incorporating mm-wave wireless links as shortcuts for long distance communication. The effectiveness of mm-wave wireless links for long-distance communication has been well studied in [91, 92, 89]. Fig. 6.1 shows the proposed topology, where the PE array is logically clustered into smaller subnets. The subnets communicate with the lower memory modules through the wireless network. Each subnet is provided with a wireless hub and wireless interface (WI), comprised of serializer/deserializer (SERDES), transceiver circuit and transmitter/ receiver antennas [91]. The wireless hub is a multi-ported switch (along with port for WI), which is responsible to ensure that data is available to all PEs within the subnet. The low-latency wireless network facilitates the broadcast and/or multicast communications within the accelerator. Unlike the conventional interconnect infrastructures, the small-world topology with WIs provides a low-latency and energy efficient communication network for large system sizes. The effectiveness of incorporating the wireless links in accelerator architectures can be summarized as follows:

- **Quality of Service(QoS):** The low latency communication provided by the mm-wave wireless links preserves the QoS of CNN applications. For instance, in case of image recognition, the input data must be available to the accelerators for generating immediate and accurate results. In absence of an efficient communication system, the frequent DRAM accesses will have a huge impact on the QoS of on-line image recognition. The integration of a low latency connection help in significantly reducing the communication delay and ensures proper execution.

- **Reduced on-chip Storage:** To amortize the frequent DRAM accesses, some accelerator designs chose to have a large on-chip buffer. Although it tends to relax the off-chip memory access to some extent, incorporating a large on-chip storage becomes significantly expensive. The wireless links provide high bandwidth interconnection to overcome the limitation imposed by off-chip DRAM accesses. Thus, high-bandwidth wireless communication alleviates the need of having a large on-chip storage.

- **Energy efficiency and Scalability:** Accelerators incorporate a significant number of PEs that performs

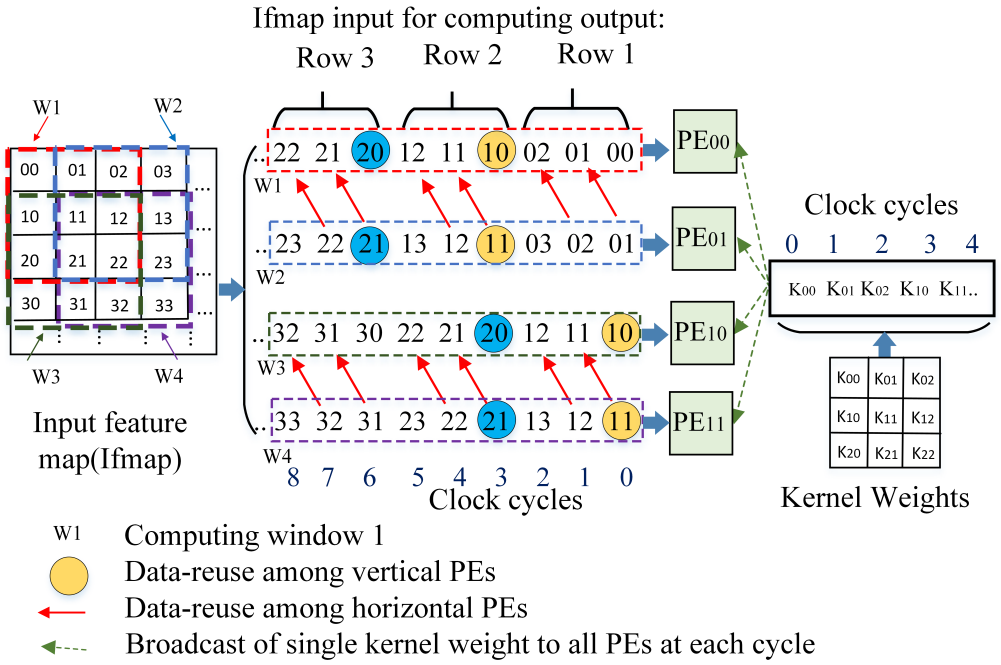


Figure 6.2: PE-Input mapping and data-reuse

concurrent execution on same or different input data. In scenarios that require the input data to be propagated across long on-chip distances, traditional wired links consume high energy. On-chip wireless links exhibit very low per bit energy over long distance, thereby providing an inexpensive alternative for long distance on-chip communication.

6.2.3 Data Network Data-Flow Scheduling

The data network in the proposed interconnection is responsible for transferring the input and output neurons between PEs and global memory. In the data network as shown in Fig. 6.1.b, the PEs are connected in a modified wired mesh topology to facilitate (i) receiving of input activations from global input buffer or neighboring PE and (ii) transmission of output neurons to global output memory. For simplicity, only one PE cluster is shown in Fig.6.1.b with a uni-directional link from weight-network router (for incoming weight input) and bi-directional link from data-network router (for incoming activations and outgoing resultant neuron). All the links within the mesh topology are unidirectional and hence the network follows fixed data flow pattern. The horizontal links (Fig. 6.1.b) in the mesh allow the input network to read data from global buffers and transfer along correspondingly. The output network through vertical links is used to collect the processed outputs to be re-used or written to global buffers.

Fig. 6.2 shows the mapping of input data to the corresponding PEs with 3x3 kernel weight matrix. Each PE is responsible for computing one output neuron of the next CNN layer and thus works on each kernel window

at a time. As shown in the figure, the data access pattern of the PEs allows sharing the input data among each other in successive cycles. Hence, each PE has buffers to hold the input data to be used in later cycles to exploit data reuse within the PE cluster. Initially, each PE reads its corresponding input activation through the mesh network, which we denote as the warm-up time. While the PEs read input data, a single weight input is broadcasted over the wireless weight network and stored in PE's buffer to maintain the network flow-control. Based on the data-reuse pattern of the PEs, we present a data-flow scheduling algorithm which is complemented by a light-weight NoC router.

6.2.3.1 Data-Flow Scheduling

The data-flow and re-configurable scheduling algorithm to efficiently transfer activation/neuron data in the mesh network is shown in Algorithm 2. The scheduling algorithm implements a fixed data-flow pattern that ensures the availability of data to each PE at every clock cycle and proper transfer of computed output neurons to the global memory. At the launch of the application, the network receives user information regarding the kernel size (row, P and column, Q in Algorithm 2) and initiates the scheduling accordingly. The light-weight router

Algorithm 2: Scheduling Algorithm

```
1  Get <P, Q>
2  At any NoC router:
3  while T < (P * Q) do
4      while (P != 0 ) do
5          At East-port:
6          Set counter = Q − 1;
7          while (counter != 0 ) do
8              Activate east-west link;
9              Link traversal through the west-port
10             counter −−;
11         end
12         At North-port:
13         if data available at SIB then
14             Give transmission priority;
15             Transmit data through south-north link;
16         else
17             if data available at LIB then
18                 Transmit data through local-north link;
19             end
20         end
21         P−−;
22     end
23     T−−;
24 end
```

P:Kernel Row ; Q:Kernel Column ; SIB:South-port Input Buffer; LIB:Local Link Input Buffer

associated with each PE maintains a counter at the east input port which is set equal to the "kernel column size - 1". As data passes through the east-west link, the counter is decremented by 1 and continues until it reaches 0, which indicates end of a kernel row computation. At next clock cycle, the vertical links get activated, PEs cascade their stored input data to the corresponding vertical PEs and mark the beginning of next kernel row computation. The above steps are repeated for each row of a computing kernel and the partial results are stored in the PE's local memory. After complete computation, the PEs transmit the output neurons through the vertical links of data network in parallel to reading input activation through the horizontal links. At any time if data is available at the input buffer of south port as well as the local link buffer, the former is given the priority since in the current scenario the read operation is more valuable than the write operation. Thus, we save the cost of having a separate output network and reuse the existing links efficiently with minimal overhead.

6.2.3.2 Lightweight NoC Router

The mesh topology for data network in proposed interconnection is highly data flow dependent and traditional NoC routers incur high area and power overheads for such an implementation. They employ fully connected crossbars to allow communication from any input port to any output port of the router, that is known to be costly in terms of area and power consumption. The fixed data-flow pattern allows us to employ a lightweight router with minimum overhead. The scheduling controller is integrated into the light weight router that monitors data movement through a stripped-down crossbar. The crossbar in light weight router interconnects east-west ports and north-south ports, keeping in accordance with the data-flow in proposed interconnection. The deterministic routing allows to incorporate a single-stage router without the need of having all the traditional router pipeline stages (RC, VC, SA and ST). Hence, the data flow dependent, unidirectional wired mesh network with a light weight, low overhead router architecture efficiently transfers the input/output data, while wireless interconnection of weight network ensures availability of kernel weights.

6.2.4 Walkthrough Example

We explain the proposed method by a walk-through example, considering a 16-node system, 3 x 3 kernel weight matrix and one-stride convolution operation as shown in Fig. 6.3. The walk-through example assumes that the warm-up stage is complete and each PE has at least one input activation in its local memory. The cycle $\mathbf{t}_0$ is next clock cycle after the completion of warm up stage. For simplicity, only the internal communication structure of the system is shown in the figure with interaction to on-chip memory modules.

- $\mathbf{t}_0$: All PEs read corresponding input activations from the global input buffer(GIB). The weight broadcasting from global weight buffer is initiated in the warm-up stage. Thus, at time $\mathbf{t}_0$, one weight and one

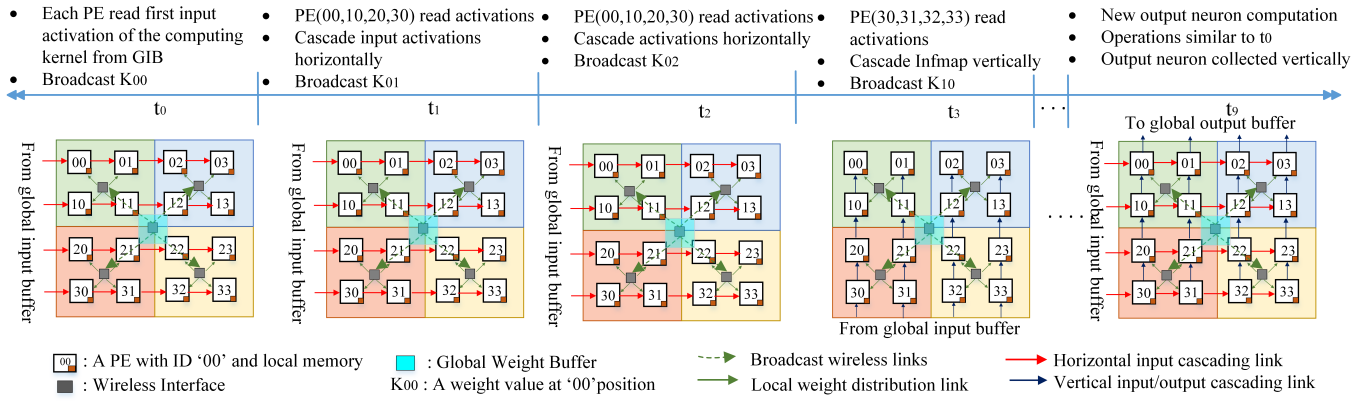


Figure 6.3: Example scenario at different time period illustrating the proposed architecture.

input activation are available to each PE to perform the convolution operation. Each PE also stores the input activation in its local memory for forwarding it to the neighbouring PE in next cycle.

- $\mathbf{t}_1$: The outermost PEs on the left (00, 10, 20, and 30) read corresponding input activations from the GIB. The other PEs receive input from their horizontal neighbours and the weight input is read from local FIFO. While the convolution operation is carried out, next weight input is broadcasted through the wireless links. Hence the data transfer and computation run in parallel.

- $\mathbf{t}_2$: Following the data-flow pattern, corresponding inputs are again read from global input-buffer, cascaded to neighbouring horizontal PEs and next weight data is broadcasted. At the end of $\mathbf{t}_2$, one row of an output neuron is calculated and the partial sums are stored in PE's local memory.

- $\mathbf{t}_3$: The time $\mathbf{t}_3$ marks the beginning of computation of the next row of kernel matrix. The outermost bottom PEs (30, 31, 32, 33) read the input activations from the global input buffer. The input activation for other PEs is now available from their bottom neighbours.

- $\mathbf{t}_9$: Since we have considered a 3x3 kernel with one stride convolutions, an output neuron will be generated by each PE after every 9 clock cycles. Each PE completes the execution of an output neuron and propagates the result through the vertical wired network. Whenever the input data needs hold of the vertical wired network, it is given the priority and the transfer of output neuron is stalled. The PEs also read inputs for the next kernel computation in parallel to output data transfer.

6.3 Experimental Setup And Results

In this section we discuss the experimental setup and evaluate our proposed design over the baseline architectures.

6.3.1 Experimental Setup

To evaluate our proposed hybrid-interconnect design, we use Multi2Sim[93] which is a cycle accurate simulator and provides the flexibility to design custom communication networks. The traditional topologies of hardware accelerators (Broadcast Bus, Mesh and Crossbar) are also implemented in Multi2sim and evaluated against our proposed design. The baseline topologies use only wired links, whereas the proposed design is a combination of wired links and long-range wireless links with configurations as shown in Table. 6.1. We consider varying system sizes (16, 32 and 64 PE-system) for our experiments to observe the performance improvement, energy savings and scalability of our approach. The size of each cluster is fixed to four PEs, connected in a star topology (in weight network), for all system sizes. The cluster size is kept small, since a relatively larger PE cluster connected to single wireless hub in star topology increases the intra-cluster latency. The wireless links are implemented using on/off keying (OOK) modulation, adopted from [91]. The evaluation is carried out using AlexNet [14], which is a popular and widely accepted CNN model.

6.3.2 Performance Evaluation

We evaluate the performance of proposed design in terms of average network latency & peak bandwidth and compare it against baseline topologies at different system sizes. These baseline topologies are based on the construct of the interconnect infrastructures discussed in Section 6.1.2, which represent common existing CNN accelerator architectures presented in Section 2.2.

6.3.2.1 Latency

Fig. 6.4 shows the network performance in terms of latency with varying system sizes. It depicts the decrease in average network latency achieved by our proposed design over the baseline topologies. A Mesh-based architecture is not suitable for a broadcast traffic, eventually results in serialized transmission and suffers from

Table 6.1: System configurations for evaluating proposed CNN accelerator

Components	**Configurations**
Topology	Input Activations: 2D modified-Mesh Weight network: Wireless-star for the global network Wired-star for the clusters
PE Cluster	4 PEs and 1 wireless receiver each cluster
Routers	5 ports,single-stage
Wireless link	60 GHz carrier, 16 Gbps bandwidth, 1 cycle latency
Application	Alexnet[14]

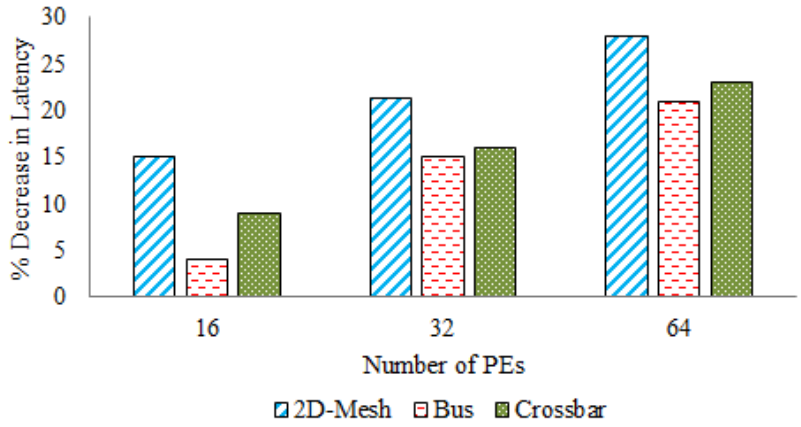


Figure 6.4: Decrease in Latency using the proposed topology over baseline topologies.

data-transfer latency, thus increasing the overall network latency. Our proposed topology take advantage of the low-latency broadcast communication through the wireless links and achieves 28% decrease in latency as compared to a Mesh topology in a 64-PE system. The Bus-based architectures are able to take advantage of the data communication pattern of CNN algorithms. Since, the CNN algorithms need multiple data to be broadcasted to a set of processing elements, a multi-cast Bus-based architecture gives more benefit that a typical mesh architecture. While the Bus-based architecture is generally considered for broadcast or multicast traffic, it tends to effect the network performance significantly when the system size increases. Hence, our approach achieved a significant improvement in network performance by bringing down the network latency by 28%, 21% and 23% as compared to Mesh, Bus and crossbar networks (in 64-PE system).

6.3.2.2 Bandwidth

We compare our proposed architecture with the baseline designs to analyze the bandwidth improvement provided by our approach. Fig. 6.5 shows the normalized bandwidth improvements where, our proposed design is found to provide approximately 1.9x, 1.2x and 1.3x more peak bandwidth than the baseline (Mesh, Crossbar and Broadcast-bus respectively) topologies in a 64-PE system.

6.3.3 Energy Savings

The average packet energy savings in the proposed topology over the baselines are shown in Fig. 6.6. It is found that our approach saves about 21% to 35% of average network energy as compared to the base architectures with different system sizes. The reduction in energy consumption in our proposed design as compared to the baseline topologies are mainly achieved by the optimized cluster size and ideal use of interconnect resources focusing the application data-flow. Also, the light-weight NoC router brings down the overall network energy

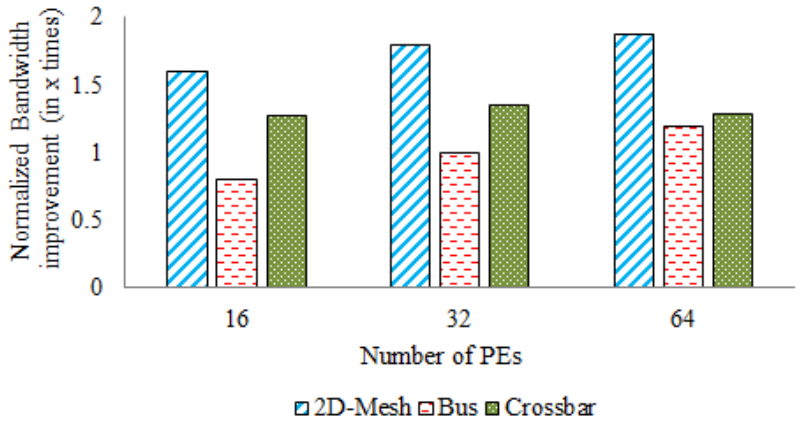


Figure 6.5: Peak network bandwidth improvement with proposed design over baseline designs.

as compared to the traditional NoC designs. Although a bus-based structure works for a small system size, it becomes inefficient when the load on the bus architecture increases significantly. Hence, large number of PEs connected to a broadcast-bus exacerbate the energy consumption of the network. The improvement is more significant as the number of PE increases.

6.3.4 Scalability

The idea of using a small-world network for transferring data over long on-chip distance, makes our proposed architecture scalable. In case of the baseline infrastructures, the broadcasting network is usually a broadcast bus that suffers severely if the number of PEs attached to the bus increases significantly. Since, the accelerators are gradually employing large number of PEs to increase the computation capability, the broadcast based bus will not be able to render the desired performance. Thus, our approach turns out to be appropriate in such scenarios

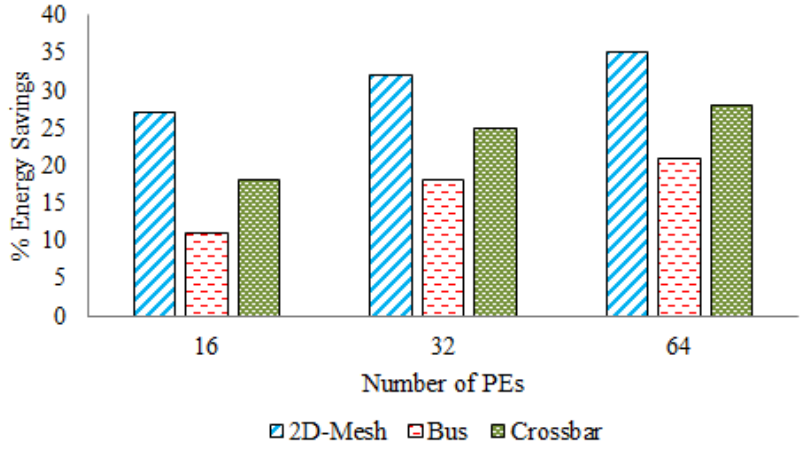


Figure 6.6: Average packet energy savings using proposed approach over baseline designs.

with minimal overhead where the number of cores are increasing rapidly. Also, with the integration of more number of PEs and support of energy-efficient & broadcast-enabled wireless links, the application performance can be further enhanced by enabling the PE clusters to operate on different inputs concurrently (for instance, operating multiple kernels on same input feature map in different PE clusters).

6.3.5 Overheads

The transceiver circuit including OOK modulation, LNA and PA requires a total area of $0.3mm^2$ per WI [91]. The metal zigzag antenna is chosen for transmitter/receiver antennas which provides best power gain with smallest area overhead. It has a length of $0.38mm$ and $58\mu m$ width [91]. The truncated NoC router consumes an area of $4292.281\mu m^2$ which is around 18% lesser than traditional NoC routers synthesized in Synopsis Design Compiler using $32nm$ CMOS technology. The transceiver circuit consumes a power of $36.7mW$ [94]. The power consumption due to our NoC router is $98.04\mu W$ which is 36% lesser as compared to traditional routers.

6.4 Conclusion

A data-flow-aware accelerator design is presented targeting the deep CNN algorithms. The proposed topology takes into account the underlying communication pattern and efficiently place the interconnect resources to achieve maximum performance and minimal network energy consumption. The deployment of low-latency and energy-efficient wireless links along with the data-flow mechanism resulted in up to 28% decrease in latency, 1.9x bandwidth improvement and 35% energy savings over baseline architectures. Hence, a scalable framework is proposed for designing energy efficient accelerator architectures.

Chapter 7

Conclusions and Future Directions

In this work, we focus on addressing the design space optimization problem and security vulnerabilities of emerging heterogeneous SoCs to improve the performance of applications running on the underlying system. The major contributions of this thesis are:

1. ASM, an accelerator-shared memory framework is proposed in Chapter 3. The ASM framework studies the behaviour of a given set of accelerator cores integrated into the same SoC in terms of data communication and sharing among themselves. By gathering the insights from the memory access profile of the applications running on the accelerator-rich system, the framework uses a mathematical formulation to provide the best-suited memory subsystem configuration for the given set of accelerators. To further improve the application performance, the ASM framework carries out an efficient shared data allocation on this optimized memory system configuration. Besides improving application performance, it significantly reduces the on-chip area requirement and allows to meet the stringent budget constraints of embedded systems.

2. In Chapter 4, we highlight the vulnerabilities induced by the integration of third-party IP blocks in the heterogeneous SoCs. We focused on the security threats that aim to degrade the system performance and, introduced the flooding-based DoS attacks that exhaust the on-chip network resource and manipulate the perceived availability of the NoC to the legitimate network nodes. A machine learning-based framework is proposed to detect such an attack in case of accelerator-rich heterogeneous systems. Our framework uses carefully selected network and system parameters to understand the behaviour of the network in case of attack and non-attack scenarios. By training the model with these parameters, our framework is able to accurately detect a flooding-based DoS attack and minimize the performance degradation of the overall system.

3. In Chapter 5, we propose Sniffer, an attack localization framework for emerging heterogeneous systems. Sniffer uses perceptron models at each network node to identify the port which is under attack. Each perceptron are trained separately for each router node using network features that help in distinguishing the port congestion in an attack and non-attack scenario. Sniffer also uses a collaborative approach by considering the decisions of the adjacent nodes in declaring a port under attack. A probing packet is used to trace back the attack path by backtracking through the nodes under attack and localize the malicious IP nodes. Sniffer is able to detect one or multiple attacker nodes and provides an efficient way of accurately finding the attacker nodes.

4. In Chapter 6, we propose an efficient architecture for Convolutional neural network (CNN) accelerators. We address the communication bottlenecks of these accelerators by extensively studying the application data-flow and computation patterns. The proposed design uses a fused wired and wireless interconnection topology. It employs broadcast enabled low latency wireless links along with traditional wired links to efficiently support the data-flow of accelerators and achieve high communication performance.

We present below some of the possible future research directions to extend our work.

- The ASM framework presented in Chapter 3 explore the design-space for memory subsystem of accelerator-rich heterogeneous SoCs. By sharing memory between the accelerators and taking the shared NoC under consideration, the ASM framework proposes to enhance application performance. However, sharing the on-chip resources between the accelerators might make the system vulnerable and increase the security threats. The shared memory can become a source of vulnerability, where a malicious accelerator could try to inject wrong or tampered information, which is read by a benign accelerator core. In such cases, the objective of the malicious accelerator core is to disrupt the regular working of the legitimate accelerator core and produce unwanted results or behaviour. The malicious accelerator can also create a Denial-of-Service attack by consuming the shared medium (memory, interconnect etc.) and denying access to other benign accelerators. Such an attack can lead to performance degradation as the benign accelerators will starve for the shared resources and will not be able to complete their respective tasks within the deadline. In some cases, the malicious accelerator can attack the integrity of benign accelerators. The critical information can be collected through the shared mediums and the malicious accelerator can leak the information to unwanted sources. This type of attack does not have a direct impact on the performance of the applications running on the SoC and therefore creates difficulty in attack detection. At its current construct, the ASM framework does not consider the security vulnerabilities while performing design-space optimization. The framework can be further extended by considering the security aspects of the underlying system. The optimization model can take care of the security threats that might increase

due to the resource sharing and provide a comprehensive framework that aims to improve performance of the applications while also minimize the security vulnerabilities of the systems.

- As one of the future work, we will look into designing models for design-space optimization of different on-chip resources altogether. In the current proposed solutions, we have primarily focused on optimizing the memory subsystem of the accelerator-rich heterogeneous SoCs. Although while optimizing the memory architecture we consider the interconnection latency into the model, there is still a large scope of improving the design process. For instance, the on-chip interconnection can be optimized to the needs of the given set of applications. Different on-chip modules can have different communication patterns. Some communications may need more bandwidth, while others might be highly sensitive to communication latency. So, it is of utmost importance to study the behaviour of data transfer between different on-chip modules and make a design choice for employing different types of interconnection medium. Furthermore, the interconnection topology is also another important characteristic that impacts the performance efficiency of a given system. As discussed before, in this case as well, we need to understand the need of the system and make a design choice regarding the topology for the underlying interconnection network. Hence, the on-chip interconnection can be redesigned and further optimized along with the on-chip memory system to provide a holistic resource optimization framework for a given system. Such frameworks can formulate multi-objective optimization problems including different resource budget constraints.

- Besides the flooding attack, the on-chip networks are also vulnerable to various attack scenarios that may arise due to integration of malicious IPs. One such attack that also targets the performance of the system is the packet tampering attack. Such attacks are mounted by a malicious construct within the SoC which results in tampering with the content of a network packet. Such tampering can be in the form of changing the source, destination or payload information carried by the packet. By tampering the payload of a network packet, the malicious IP can make the destination node to consume incorrect information. The destination node will process the incorrect information and will generate unexpected results. On the other hand, when the malicious IP changes the source or destinations of a network packet, it results in deflecting the legitimate network packets towards incorrect destination nodes. In the case where the destination ID is tampered, the packets do not reach their intended destinations, leading to packet re-transmission which finally degrades the system performance. Whereas, in the case where the source is tampered, it creates problem in passing acknowledgement messages from the destination nodes. The acknowledgement messages will be passed to wrong source nodes, while the legitimate source nodes will be waiting for the acknowledgement signals. This can disrupt the proper functionality of the network

and degrade system performance. Therefore, it is of the utmost importance to develop frameworks that can address such attacks and minimize application performance degradation.

Bibliography

[1] Y. S. Shao, S. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Toward cache-friendly hardware accelerators," in *HPCA Sensors and Cloud Architectures Workshop (SCAW)*, 2015, pp. 1–6.

[2] B. Barry, C. Brick, F. Connor, D. Donohoe, D. Moloney, R. Richmond, M. O'Riordan, and V. Toma, "Always-on vision processing unit for mobile applications," *IEEE Micro*, vol. 35, no. 2, pp. 56–66, 2015.

[3] S. Tan, F. Qiao, B. Xia, H. Yang, and H. Wang, "A functional model of systemc-based mpeg-2 decoder with heterogeneous multi-ip-cores and hybrid-interconnections architecture," in *2009 2nd International Congress on Image and Signal Processing*. IEEE, 2009, pp. 1–5.

[4] ITRS. (2007) system drivers. [Online]. Available: http://www.itrs.net/.

[5] L. Benini and G. De Micheli, "Networks on chips: A new soc paradigm," *computer*, vol. 35, no. 1, pp. 70–78, 2002.

[6] Y. Huang, S. Bhunia, and P. Mishra, "Scalable test generation for trojan detection using side channel analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 11, pp. 2746–2760, 2018.

[7] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementation—a performance view," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 559–572, 2007.

[8] L. Gwennap, "Adapteva: More flops, less watts," *Microprocessor Report*, vol. 6, no. 13, pp. 11–02, 2011.

[9] J. Lu, K. Bai, and A. Shrivastava, "Ssdm: smart stack data management for software managed multicores (smms)," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 149.

[10] A. Kannan, A. Shrivastava, A. Pabalkar, and J.-e. Lee, "A software solution for dynamic stack management on scratch pad memory," in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*. IEEE Press, 2009, pp. 612–617.

[11] J. Cong, M. A. Ghodrat, M. Gill, C. Liu, and G. Reinman, "Bin: a buffer-in-nuca scheme for accelerator-rich cmps," in *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*. ACM, 2012, pp. 225–230.

[12] L. E. Olson, S. Sethumadhavan, and M. D. Hill, "Security implications of third-party accelerators," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 50–53, 2015.

[13] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[15] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[16] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee, 2013, pp. 6645–6649.

[17] C. Teuscher, "Nature-inspired interconnects for self-assembled large-scale network-on-chip designs," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 17, no. 2, p. 026106, 2007.

[18] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[19] J. Li, G. Yan, W. Lu, S. Gong, S. Jiang, J. Wu, and X. Li, "Synergyflow: An elastic accelerator architecture supporting batch processing of large-scale deep neural networks," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 24, no. 1, pp. 1–27, 2018.

[20] M. Sinha, S. H. Gade, W. Singh, and S. Deb, "Data-flow aware cnn accelerator with hybrid wireless interconnection," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2018, pp. 1–4.

[21] S. Murray, W. Floyd-Jones, Y. Qi, G. Konidaris, and D. J. Sorin, "The microarchitecture of a real-time robot motion planning accelerator," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 45.

[22] X. Liu and Y. Deng, "Fast radix: A scalable hardware accelerator for parallel radix sort," in *2014 12th International Conference on Frontiers of Information Technology*. IEEE, 2014, pp. 214–219.

[23] W. Ahmed, M. Shafique, L. Bauer, and J. Henkel, "mrts: Run-time system for reconfigurable processors with multi-grained instruction-set extensions," in *2011 Design, Automation & Test in Europe*. IEEE, 2011, pp. 1–6.

[24] R. Koenig, L. Bauer, T. Stripf, M. Shafique, W. Ahmed, J. Becker, and J. Henkel, "Kahrisma: a novel hypermorphic reconfigurable-instruction-set multi-grained-array architecture," in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, 2010, pp. 819–824.

[25] J. Henkel and R. Ernst, "An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 9, no. 2, pp. 273–289, 2001.

[26] A. Prakash, C. T. Clarke, S.-K. Lam, and T. Srikanthan, "Rapid memory-aware selection of hardware accelerators in programmable soc design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 3, pp. 445–456, 2017.

[27] J. Cong, M. A. Ghodrat, M. Gill, C. Liu, G. Reinman, and Y. Zou, "Axr-cmp: Architecture support in accelerator-rich cmps," in *2nd Workshop on SoC Architecture, Accelerators and Workloads*, 2011.

[28] M. J. Lyons, M. Hempstead, G.-Y. Wei, and D. Brooks, "The accelerator store: A shared memory framework for accelerator-based systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 48, 2012.

[29] C. Pilato, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, "System-level optimization of accelerator local memory for heterogeneous systems-on-chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 435–448, 2016.

[30] B. K. Reddy, A. K. Singh, D. Biswas, G. V. Merrett, and B. M. Al-Hashimi, "Inter-cluster thread-to-core mapping and dvfs on heterogeneous multi-cores," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, no. 3, pp. 369–382, 2017.

[31] R. Garibotti, L. Ost, A. Butko, R. Reis, A. Gamatié, and G. Sassatelli, "Exploiting memory allocations in clusterised many-core architectures," *IET Computers & Digital Techniques*, 2019.

[32] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, "Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings," *IEEE Micro*, vol. 40, no. 3, pp. 20–29, 2020.

[33] Synopsis Platform Architect. http://www.eigen.in/pdf/Platform_Architect.pdf.

[34] Synopsis Platform Architect Ultra. https://www.synopsys.com/verification/virtual-prototyping/platform-architect.html.

[35] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, "Border control: Sandboxing accelerators," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2015, pp. 470–481.

[36] L. Piccolboni, G. Di Guglielmo, and L. P. Carloni, "Pagurus: Low-overhead dynamic information flow tracking on loosely coupled accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2685–2696, 2018.

[37] C. Pilato, K. Wu, S. Garg, R. Karri, and F. Regazzoni, "Tainthls: High-level synthesis for dynamic information flow tracking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 798–808, 2018.

[38] R. JS, D. M. Ancajas, K. Chakraborty, and S. Roy, "Runtime detection of a bandwidth denial attack from a rogue network-on-chip," in *Proceedings of the 9th International Symposium on Networks-on-Chip*, 2015, pp. 1–8.

[39] L. Fiorin, G. Palermo, and C. Silvano, "A security monitoring service for nocs," in *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, 2008, pp. 197–202.

[40] S. Charles, Y. Lyu, and P. Mishra, "Real-time detection and localization of dos attacks in noc based socs," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1160–1165.

[41] D. Fang, H. Li, J. Han, and X. Zeng, "Robustness analysis of mesh-based network-on-chip architecture under flooding-based denial of service attacks," in *2013 IEEE Eighth International Conference on Networking, Architecture and Storage*. IEEE, 2013, pp. 178–186.

[42] T. Boraten, D. DiTomaso, and A. K. Kodi, "Secure model checkers for network-on-chip (noc) architectures," in *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*. IEEE, 2016, pp. 45–50.

[43] Z. Xu, S. Ray, P. Subramanyan, and S. Malik, "Malware detection using machine learning based analysis of virtual memory access patterns," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 169–174.

[44] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 559–570, 2013.

[45] V. Jyothi, X. Wang, S. K. Addepalli, and R. Karri, "Brain: Behavior based adaptive intrusion detection in networks: Using hardware performance counters to detect ddos attacks," in *2016 29th international conference on VLSI design and 2016 15th international conference on embedded systems (VLSID)*. IEEE, 2016, pp. 587–588.

[46] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "Sok: The challenges, pitfalls, and perils of using hardware performance counters for security," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 20–38.

[47] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013, pp. 215–224.

[48] D. Hammerstrom, "A highly parallel digital architecture for neural network emulation," in *VLSI for Artificial Intelligence and Neural Networks*. Springer, 1991, pp. 357–366.

[49] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam *et al.*, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 34, no. 10, pp. 1537–1557, 2015.

[50] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 92–104.

[51] T. Krishna, L.-S. Peh, B. M. Beckmann, and S. K. Reinhardt, "Towards the ideal on-chip fabric for 1-to-many and many-to-1 communication," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 71–82.

[52] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Cvpr 2011 Workshops*. IEEE, 2011, pp. 109–116.

[53] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014, pp. 682–687.

[54] D. Vainbrand and R. Ginosar, "Network-on-chip architectures for neural networks," in *2010 Fourth ACM/IEEE International Symposium on Networks-on-Chip*. IEEE, 2010, pp. 135–144.

[55] H. Kwon, A. Samajdar, and T. Krishna, "Rethinking nocs for spatial neural network accelerators," in *2017 Eleventh IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*. IEEE, 2017, pp. 1–8.

[56] W. Choi, K. Duraisamy, R. G. Kim, J. R. Doppa, P. P. Pande, D. Marculescu, and R. Marculescu, "On-chip communication network for efficient training of deep convolutional networks on heterogeneous manycore systems," *IEEE Transactions on Computers*, vol. 67, no. 5, pp. 672–686, 2017.

[57] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and soc interfaces using gem5-aladdin," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[58] H. G. Lee, N. Chang, U. Y. Ogras, and R. Marculescu, "On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 12, no. 3, pp. 1–20, 2008.

[59] Y.-T. Chen and J. Cong, "Interconnect synthesis of heterogeneous accelerators in a shared memory architecture," in *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2015, pp. 359–364.

[60] Y. Tatsumi and H. Mattausch, "Fast quadratic increase of multiport-storage-cell area with port number," *Electronics Letters*, vol. 35, no. 25, pp. 2185–2187, 1999.

[61] V. Sundarapandian, *Probability, statistics and queuing theory*. PHI Learning Pvt. Ltd., 2009.

[62] L. Kleinrock, *Communication nets: Stochastic message flow and delay*. Courier Corporation, 2007.

[63] R. H. Byrd, M. E. Hribar, and J. Nocedal, "An interior point algorithm for large-scale nonlinear programming," *SIAM Journal on Optimization*, vol. 9, no. 4, pp. 877–900, 1999.

[64] R. H. Byrd, J. C. Gilbert, and J. Nocedal, "A trust region method based on interior point techniques for nonlinear programming," *Mathematical programming*, vol. 89, no. 1, pp. 149–185, 2000.

[65] R. A. Waltz, J. L. Morales, J. Nocedal, and D. Orban, "An interior algorithm for nonlinear optimization that combines line search and trust region steps," *Mathematical programming*, vol. 107, no. 3, pp. 391–408, 2006.

[66] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti, "Noxim: An open, extensible and cycle-accurate network on chip simulator," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2015, pp. 162–163.

[67] V. Venkataramani, M. C. Chan, and T. Mitra, "Scratchpad-memory management for multi-threaded applications on many-core architectures," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 1, pp. 1–28, 2019.

[68] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, vol. 27, p. 28, 2009.

[69] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 110–119.

[70] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 2001, pp. 3–14.

[71] W.-C. Kao, S.-H. Wang, L.-Y. Chen, and S.-Y. Lin, "Design considerations of color image processing pipeline for digital cameras," *IEEE Transactions on Consumer Electronics*, vol. 52, no. 4, pp. 1144–1152, 2006.

[72] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman, "Accelerator-rich architectures: Opportunities and progresses," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2014, pp. 1–6.

[73] S. Evain and J.-P. Diguet, "From noc security analysis to design solutions," in *IEEE Workshop on Signal Processing Systems Design and Implementation, 2005.* IEEE, 2005, pp. 166–171.

[74] J.-P. Diguet, S. Evain, R. Vaslin, G. Gogniat, and E. Juin, "Noc-centric security of reconfigurable soc," in *First International Symposium on Networks-on-Chip (NOCS'07)*. IEEE, 2007, pp. 223–232.

[75] V. Iskandar, C. Salama, and M. Taher, "Dynamic thread mapping for maximizing performance in power-efficient multi-core systems," in *2018 13th International Conference on Computer Engineering and Systems (ICCES)*. IEEE, 2018, pp. 230–235.

[76] K. M. Attia, M. A. El-Hosseini, and H. A. Ali, "Dynamic power management techniques in multi-core architectures: A survey study," *Ain Shams Engineering Journal*, vol. 8, no. 3, pp. 445–456, 2017.

[77] W. Xu, W. Trappe, Y. Zhang, and T. Wood, "The feasibility of launching and detecting jamming attacks in wireless networks," in *Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing*, 2005, pp. 46–57.

[78] Z. Qian, D.-C. Juan, P. Bogdan, C.-Y. Tsui, D. Marculescu, and R. Marculescu, "A comprehensive and accurate latency model for network-on-chip performance analysis," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2014, pp. 323–328.

[79] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[80] A. Bermak and D. Martinez, "A compact 3d vlsi classifier using bagging threshold network ensembles," *IEEE Transactions on Neural Networks*, vol. 14, no. 5, pp. 1097–1109, 2003.

[81] (2007) Itrs. system drivers. [Online]. Available: http://www.itrs.net/

[82] C. G. Chaves, S. P. Azad, T. Hollstein, and J. Sepúlveda, "Dos attack detection and path collision localization in noc-based mpsoc architectures," *Journal of Low Power Electronics and Applications*, vol. 9, no. 1, p. 7, 2019.

[83] S. Charles, Y. Lyu, and P. Mishra, "Real-time detection and localization of distributed dos attacks in noc-based socs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4510–4523, 2020.

[84] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous cpu-gpu simulator," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 34–36, 2014.

[85] M. Shoba and R. Nakkeeran, "Energy and area efficient hierarchy multiplier architecture based on vedic mathematics and gdi logic," *Engineering Science and Technology, an International Journal*, vol. 20, no. 1, pp. 321–331, 2017.

[86] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[87] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 37–47.

[88] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection networks*. Morgan Kaufmann, 2003.

[89] M.-C. F. Chang, E. Socher, S.-W. Tam, J. Cong, and G. Reinman, "Rf interconnects for communications on-chip," in *Proceedings of the 2008 international symposium on Physical design*, 2008, pp. 78–83.

[90] A. Shacham, K. Bergman, and L. P. Carloni, "Photonic networks-on-chip for future generations of chip multiprocessors," *IEEE Transactions on Computers*, vol. 57, no. 9, pp. 1246–1260, 2008.

[91] S. Deb, K. Chang, X. Yu, S. P. Sah, M. Cosic, A. Ganguly, P. P. Pande, B. Belzer, and D. Heo, "Design of an energy-efficient cmos-compatible noc architecture with millimeter-wave wireless interconnects," *IEEE Transactions on Computers*, vol. 62, no. 12, pp. 2382–2396, 2012.

[92] S. H. Gade and S. Deb, "Hywin: Hybrid wireless noc with sandboxed sub-networks for cpu/gpu architectures," *IEEE Transactions on computers*, vol. 66, no. 7, pp. 1145–1158, 2016.

[93] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: A simulation framework for cpu-gpu computing," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2012, pp. 335–344.

[94] S. H. Gade and S. Deb, "Achievable performance enhancements with mm-wave wireless interconnects in noc," in *Proceedings of the 9th International Symposium on Networks-on-Chip*, 2015, pp. 1–2.