Domain-Specific AI Hardware: Strategies for Design-time and Runtime Optimization

By

Ahmet Alper Goksoy

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2024

Date of final oral examination: 08/05/2024

The dissertation is approved by the following members of the Final Oral Committee:

Umit Y. Ogras, Associate Professor, Electrical and Computer Engineering, University of Wisconsin-Madison

Yu Hen Hu, Professor, Electrical and Computer Engineering, University of Wisconsin-Madison

Karthikeyan Sankaralingam, Professor, Computer Sciences, University of Wisconsin-Madison

Chaitali Chakrabarti, Professor, Electrical, Computer and Energy Engineering, Arizona State University

© Copyright by Ahmet Alper Goksoy 2024
All Rights Reserved

Dedicated to my parents Ayşe Gülnihal and Ismail Hakkı,
my brother Osman Gökalp, my wife Betül,
and my son Alparslan Fatih.

ACKNOWLEDGMENTS

First and foremost, I would like to express my profound gratitude to my advisor, Prof. Umit Y. Ogras, for his outstanding guidance, counsel, and support throughout my Ph.D. journey. His insightful ideas and suggestions have been invaluable in overcoming numerous challenges I encountered during my studies. In particular, his guidance on organizing my thoughts, writing research papers, managing time efficiently, and most importantly, fostering my ability to conduct independent research, will greatly benefit me in my future career. I deeply value the relationship we have built over the years and am confident it will endure beyond my time at the University of Wisconsin-Madison.

I am sincerely grateful to Dr. Yu-Hen Hu, Dr. Karu Sankaralingam, and Dr. Chaitali Chakrabarti for generously dedicating their time to serve on my Ph.D. defense committee. Their constructive feedback and recommendations have been instrumental in enhancing the quality of my research and this dissertation.

I deeply appreciate Dr. Chaitali Chakrabarti, Dr. Yu Cao, Dr. Radu Marculescu, and Dr. Ali Akoglu for our enlightening research discussions over the years. I highly value their collaboration, insights, and feedback.

I am thankful for the stimulating discussions, encouraging words, and enjoyable moments shared with friends and colleagues at eLab: Dr. Samet Arda, Dr. Ganapati Bhat, Dr. Sumit Mandal, Dr. Anish Krishnakumar, Dr. Yigit Tuncel, Dr. Sizhe An, Dr. Toygun Basaklar, Jie Tong, Nuriye Yildirim, Aditya Ukarande, Dr. Shruti Narayana, Lukas Pfromm, Alish Kanani, Jiahao Lin, Mingcong Cao, Xinmiao Xiong, Deepak Vasudevan, Khamida Begaliyeva, Shilpa Murthy, and Conrad Holt. I also extend my gratitude to colleagues and friends at other universities: Dr. Alex Chiriyath, Dr. Arindam Dutta, Dr. Joshua Mack, Dr. Sahil Hassan, Serhan Gener, Umut Suluhan, Ilkin Aliyev, and Xing Chen.

I greatly appreciated the words of encouragement and the lighter moments shared with my friends; at Madison: Gokhan Cakal, Emir Karakaya, Emre Yildizci, Elvan Sahin, Ensar Basakin, Dr. Yunus Alapan, Mazlum Kesen, Ferhat Alkan; and at Arizona: Muslum Emir Avci, Melih Sozmen.

I also extend my gratitude to Defense Advanced Research Projects Agency (Grant FA8650-18-2-7860) for funding the research presented in this dissertation. My heartfelt thanks go to Dr. Gunhan Dundar for inspiring me to pursue a research career during my undergraduate years.

Lastly, this achievement would not have been possible without the support of my parents, Ismail Hakki and Ayse Gulnihal, my brother Osman Gokalp, and my entire family. My son Alparslan Fatih's love and joy have been a tremendous source of strength during the final year of this journey. Finally, I am profoundly grateful to my loving wife Betul, whose unconditional support and sacrifices have made it possible for me to complete this journey smoothly.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Domain-specific AI hardware is designed to meet the unique computational demands of artificial intelligence applications, providing superior performance and efficiency compared to general-purpose processors. As AI workloads become increasingly complex, design-time and runtime optimizations are critical to maximizing hardware capabilities. This dissertation addresses significant challenges in domain-specific AI hardware and other computationally intensive applications through innovative strategies for design-time and runtime optimizations. 2.5D chiplet-based architectures have been proposed to enable cost-efficient implementation of large-scale systems by integrating smaller chips, called chiplets, on a silicon interposer. To this end, we propose a heterogeneous big-little chiplet-based in-memory computing (IMC) architecture, leveraging big and little IMC-based chiplets with an optimal Network-on-Package configuration. This architecture achieves up to 329× improvement in the energy-delay-area product and 2.8× higher IMC utilization than homogeneous chiplet architectures. Conversely, deploying neural networks on edge devices with limited resources poses a challenge due to the significant computational resources required for training. To address this challenge, we propose an energy-efficient on-chip training framework utilizing a resistive RAM-based IMC accelerator to customize mmWave-based human pose estimation models, improving the MPJPE by 23.89% with on-chip training. Furthermore, hardware-software codesign approaches can reduce the complexity of the DNN models. To this end, we propose a communication-aware sparse neural network optimization framework that enables hardware-aware pruning, optimizing the communication with up to 6.8× improvement in the energy-delay product (EDP) without significant accuracy degradation.

Fully utilizing the domain-specific hardware's potential requires optimal task scheduling on the processing elements. To this end, we develop a machine learning (ML)-based dynamic adaptive scheduling framework that combines fast, low-overhead with complex, high-overhead schedulers. This framework achieves up to 1.29× speedup and 45% lower EDP in experiments with five streaming appli-

cations. ML-based task schedulers depend critically on the representativeness of the training data. Hence, their performance may diminish or even fail under unknown workloads, especially new applications. To address this challenge, we propose a runtime monitoring framework for ML-based scheduling algorithms toward robust domain-specific SoCs. It detects workload generalization with up to 98.39% accuracy and enables up to 14$\times$ faster execution times when the scheduler is incrementally trained.

1 INTRODUCTION

Advances in machine learning have driven transformative changes across various industries [7, 8, 9, 10, 11, 12]. This surge in artificial intelligence (AI) applications necessitates the development of custom AI hardware optimized to handle the massive computational demands and efficiency requirements of these complex algorithms. Traditional general-purpose processors, like CPUs, often fail to deliver the performance and energy efficiency needed for AI tasks. Consequently, specialized hardware such as custom AI accelerators have become crucial, enabling faster processing speeds, lower power consumption, and improved scalability, facilitating more effective and widespread deployment of AI technologies.

Domain-specific AI hardware, such as Graphics Processing Units (GPUs), Tensor Processing Units (TPUs), and various custom accelerators, is designed to optimize performance for specific tasks like deep learning, natural language processing, and computer vision. These specialized devices significantly improve computational efficiency, energy consumption, and processing speed compared to general-purpose processors. The growing complexity and scale of AI models necessitate hardware that can handle massive parallelism and high-throughput data processing, driving innovation in designing and implementing domain-specific AI hardware.

As DNN models grow larger and more complex, the communication between different hardware components and across distributed systems becomes a significant bottleneck, requiring communication-aware designs to achieve high performance while maintaining energy efficiency. These designs aim to optimize the

Design-time			Run-time	
Dataflow and Architecture	**Hardware Technologies**	**Packaging and Integration Technologies**	**Model Compression**	**Resource Management**
Dataflow Weight Stationary Input Stationary Output Stationary Row Stationary No Local Reuse **Architecture** In-Memory Computing Systolic Array CGRA FPGA	CMOS **Near-Memory Computing** DRAM **In-Memory Computing** Resistive RAM SRAM PCM STT-RAM CBRAM	Monolithic 2.5D Architectures (Chiplets) 3D Architectures	Quantization Sparsification (Pruning) Compact Model Tensor Decomposition	Task Scheduling Runtime Monitoring Power Management Thermal Management

Figure 1.1: Key research ideas on accelerating AI workloads on domain-specific architectures.

data movement and communication overheads. By addressing these challenges, communication-aware architectures ensure that AI systems can scale effectively and deliver processing capabilities within the constraints essential for applications in edge computing, autonomous systems, and large-scale data centers.

Figure 1.1 outlines the key research ideas on accelerating AI workloads on energy-efficient architectures. These ideas fall into two categories: design-time and runtime optimizations. Design-time optimizations involve architectural enhancements, dataflow architectures, and advanced hardware technologies made during the development phase to ensure the hardware can meet the demands of AI workloads. This includes techniques such as hardware-software co-design, where the hardware and software are developed in tandem to optimize for specific applications, and the use of advanced manufacturing processes to create more efficient and powerful processing units. By incorporating dataflow architectures, designers can improve data movement efficiency within the hardware, reducing bottlenecks and

enhancing overall performance. Additionally, leveraging cutting-edge hardware technologies allows for the creation of more robust and capable AI accelerators.

Runtime optimizations focus on improving performance during the operation of the hardware. This includes dynamic resource management, adaptive power scaling, efficient scheduling algorithms, and model compression techniques that can adjust to the varying demands of AI tasks in real time. Model compression techniques, such as pruning, quantization, sparse training, and knowledge distillation, help reduce the size and complexity of AI models, making them more efficient in running on specialized hardware without sacrificing performance. These runtime optimizations ensure optimal utilization of hardware resources, minimizing latency, and maximizing throughput.

As DNN models increase in size, the need for larger architectures becomes inevitable. However, yield and fabrication costs also increase with the area of monolithic chips. In response, 2.5D chiplet-based architectures have been proposed for implementing large DNN models. These architectures are designed by integrating smaller chips, called chiplets, on a silicon interposer to enable cost-efficient implementation of large-scale systems. This modular approach is essential for handling diverse AI workloads that require intensive data communication and processing. By leveraging chiplets, AI hardware can better accommodate DNN models and utilize communication-aware designs, resulting in enhanced performance, efficiency, and scalability necessary for AI applications.

Existing architectures do not consider the non-uniform distribution of weights and activations within DNNs while designing the chiplet-based architecture. For

example, the initial layers of the ResNet-50 model on the ImageNet dataset have more activations, leading to more on-chip data movement, while fewer weights imply reduced computations. In contrast, the latter layers have more weights and fewer activations, resulting in increased computations and reduced data movement. Hence, the chiplet-based AI hardware should be optimized to match the non-uniform algorithm structure and maximize the efficiency of computation and data movement across the DNN layers. This shows the inefficiency of homogeneous chiplet-based architectures as they fail to exploit the underlying distribution of weights and activations within DNNs. To this end, we developed a heterogeneous chiplet-based in-memory computing (IMC) architecture that integrates big and little-chiplet banks. Additionally, we developed an algorithm to determine the optimal configuration of the big-little IMC chiplet architecture. This algorithm favors mapping the early layers to the little chiplet banks and the subsequent layers to the big chiplet banks, achieving up to 2.8× higher IMC utilization and up to 329× improvement in the energy-delay-area product (EDAP) compared to homogeneous chiplet IMC architectures.

In addition, there is an increasing demand to implement neural networks on mobile edge devices for both on-chip training and inference following recent developments in the Internet of Things (IoT). Nonetheless, due to the significant computational resources required for training, deploying neural networks on edge devices with limited resources poses a challenge. For edge devices, on-chip training removes the need to transfer data between the device and the cloud, leading to quicker and safer training. It is ideal for applications that run on edge devices

like home-based rehabilitation systems. Home-based rehabilitation systems allow patients to perform rehabilitation without going to clinics, thus, reducing the commute and healthcare costs. Human joint estimation allows visualization of body movements required for rehabilitation. However, the estimations can be inaccurate if not customized for the specific patient. For this purpose, we developed PHR, ReRAM-based AI hardware with energy-efficient on-chip training capacity for home-based rehabilitation systems. This system enables real-time processing of RGB image data, fast on-chip training of mmWave radar-based human pose estimation, and energy-efficient model inference for continuous patient usage. It utilizes RGB camera data to generate the joint reference coordinates and customize the initial model for the patient using on-chip training. Once the model that uses mmWave signals is customized with on-chip training, only the mmWave radar is used for inference. Both on-chip training and inference are executed on the ReRAM-based IMC accelerator. The initial model achieves 130.7 mm mean per joint point error (MPJPE) for the test subjects, while the corresponding error for the subjects in the training set is 97.8 mm, which is about 25% lower. These results show that the initial model can achieve very high accuracy for the known subjects, but the accuracy degrades for new users. On average, customization using on-chip training improves the MPJPE by 23.89%. Experiments show that it customizes to new patients successfully and provides energy-efficient estimates of human joint coordinates with 611.1$\times$ lower inference energy and 14.0$\times$ faster training. Both the training and inference of the mmWave model and the inference of the RGB model (9.7ms) can be performed in real time using our IMC-based accelerator.

Recent research has shown that the DNN models and the AI architectures can be codesigned, enabling high accuracy and throughput while minimizing energy and cost [13, 14]. Codesign approaches can reduce the complexity of the DNN models to run efficiently and faster. For example, quantization reduces the precision of calculations, allowing smaller memory, less computation, and communication requirements to run faster and more efficiently on specialized hardware [15, 16, 17]. Furthermore, approaches that aim at reducing the number of operations, such as model compression and pruning, utilize networks' sparsity [18, 19, 20, 21, 22, 23, 24, 25, 26, 27]. Many neural network weights are zero or close to zero, which allows for reduced computational complexity without significantly affecting the models' accuracy. To this end, we propose CANNON, a communication-aware sparse neural network optimization framework. The proposed technique's first step maps the DNN layers on the target hardware resources, e.g., to the processing tiles of an NoC. The second step performs hardware-aware dynamic sparse training. The proposed technique prunes the p−percent of the weights based on the significance of weight columns (called the *z-index*) towards any inference decision per unit communication cost. Finally, we maintain the target sparsity throughout the training process by choosing an equal number of weight columns (p−percent) with the smallest communication cost and adding them back to the network at the end of each epoch. Our hardware-optimized sparse neural networks result in up to 6.8× improvement in the energy-delay product (with respect to the neural networks with pruning and state-of-the-art mapping techniques) without any significant accuracy degradation.

To use the full potential of the domain-specific hardware, optimal scheduling of

applications on the processing elements is crucial for optimizing the performance and efficiency of computationally intensive applications. This process involves strategically allocating and managing tasks across specialized hardware units to ensure optimal resource utilization and minimize execution time. Unlike general-purpose processors, domain-specific hardware such as GPUs, TPUs, and custom accelerators are designed to handle specific types of workloads with high efficiency. Effective task scheduling considers these hardware units' unique characteristics and capabilities, balancing load, reducing bottlenecks, and leveraging parallelism to achieve superior performance.

AI workloads typically represent fixed workloads with well-defined and predictable tasks and resource requirements. This allows for a more straightforward scheduling approach, where resources can be allocated based on the known demands of the AI models. In contrast, other domains often deal with dynamic workloads with varying requirements and real-time changes in processing needs. For instance, communication and radar applications can exhibit significant variability in workload patterns, necessitating adaptive scheduling strategies. Task scheduling for these dynamic workloads must be highly responsive and capable of reallocating resources on the fly to accommodate fluctuating demands. By efficiently managing AI workloads and dynamic workloads, domain-specific hardware can deliver high performance and flexibility, making it an essential component in diverse technological applications. To this end, we propose a dynamic adaptive scheduling (DAS) framework that combines the benefits of a simple scheduler with fast decision-making and a sophisticated scheduler with high-quality decisions

using machine learning techniques. The DAS framework dynamically combines two schedulers and outperforms both of them. It also results in low scheduling overhead with 4.2 nJ energy and six ns runtime for low to medium loads, 27.2 nJ energy, and 65 ns runtime for heavy workloads. Experimental results with five streaming applications show that DAS achieves 1.29× speedup and 45% lower EDP than the sophisticated scheduler at low data rates and 1.28× speedup and 37% lower EDP than the fast scheduler when the workload intensifies.

ML-based task schedulers can make quick, high-quality decisions at runtime. Like any ML model, these offline-trained policies depend critically on how representative the training data is. Hence, their performance may diminish or even catastrophically fail under unknown workloads, especially new applications. Runtime monitoring of machine learning-based task scheduling algorithms adds another dimension of runtime optimizations by continuously assessing and adjusting the scheduling decisions based on real-time performance metrics. By leveraging runtime monitoring, domain-specific hardware can achieve even greater efficiency and responsiveness, further enhancing the capabilities of AI and other high-demand applications. To this end, we propose a novel framework to continuously monitor the system to detect unforeseen scenarios using a gradient-based generalization metric called coherence. The proposed framework accurately determines whether the current policy generalizes to new inputs. If not, it incrementally trains the ML scheduler to ensure the robustness of the task-scheduling decisions. The proposed framework is evaluated thoroughly with a domain-specific SoC and six real-world applications. It can detect whether the trained scheduler generalizes to the current

workload with 88.75% to 98.39% accuracy. Furthermore, it enables 1.1× to 14× faster execution time when the scheduler is incrementally trained.

In summary, this dissertation makes the following contributions:

- A heterogeneous big-little chiplet-based IMC architecture that utilizes a big and little IMC-based chiplet compute structure coupled with an optimal NoP configuration (interposer and bridge) [28],
- An energy-efficient on-chip training framework with A resistive RAM-based in-memory computing accelerator that customizes mmWave-based human pose estimation model for higher accuracy [29],
- A communication-aware sparse neural network optimization framework [24],
- A dynamic adaptive scheduling framework that combines fast, low overhead and complex, high overhead tasks scheduling algorithms [30, 31],
- A runtime monitoring framework for ML-based task scheduling algorithms to enable robust domain-specific SoCs.

The rest of the dissertation is organized as follows. The literature survey is discussed in Chapter 2. Chapter 3 presents big-little chiplets for design-time optimization of AI hardware, and discusses its role in large-scale computing. An energy-efficient on-chip training approach using an IMC AI accelerator for personalized home-based rehabilitation systems is presented in Chapter 4. Communication-aware sparse neural network optimizations are discussed in Chapter 5. Chapter 6

presents a runtime optimization strategy using task scheduling. Runtime monitoring for ML-based task schedulers is discussed in Chapter 7. Finally, Chapter 8 concludes this dissertation.

2 LITERATURE REVIEW

2.1 Chiplet-based Architectures

Chiplet-based architectures are well explored for high-performance computing applications [32, 33, 34, 35, 36, 37, 38]. A co-design flow considering architecture, chip, and package for a chiplet-based system is proposed in [32]. A detailed design space exploration with the proposed co-design flow shows significant improvement in power consumption and area with respect to a monolithic design. Vivet et al. [33] proposed a chiplet-based system with 96 computing cores and a 3D memory are distributed over 6 chiplets. Another recent work proposed a 2,048 chiplet (14,336 cores) wafer-scale processor that utilizes a bridge-based integration [34]. The authors discuss the challenges of designing a wafer-scale processor and provide insights into power delivery, clock routing, and testing.

Chiplet-based architectures have proven to be both more energy-efficient and cost-effective than monolithic architectures for complex DNNs. Several prior studies proposed chiplet-based architectures for DNN acceleration [1, 39, 40]. The authors of [1] proposed a fine-grained 36-chiplet architecture for DNN inference acceleration. Each chiplet utilizes a homogeneous structure with 16 PEs that operate using a weight stationary dataflow. The chiplets are connected by a 6×6 NoP mesh that utilize the ground-referenced signaling technique [41]. The authors of [40] proposed a hierarchical and analytical framework, NN-Baton, to analyze DNN mapping and communication overheads in a chiplet-based DNN accelerator. NN-Baton supports different mapping schemes of DNNs onto the chiplets. Furthermore, an

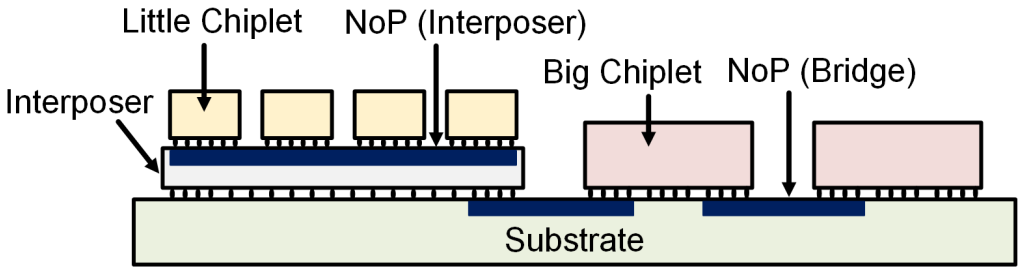

Figure 2.1: Cross-sectional view of the big-little chiplet-based IMC architecture. The architecture consists of a little chiplet bank with little chiplets (connected by an NoP within the interposer and a big chiplet bank with big chiplets connected by a bridge NoP. NoP properties: 1.5–8mm length, 2–4.5µm pitch, and 0.5–2µm width.

analytical model to quantify the communication overhead for NoP is also presented in NN-Baton. A family of chiplet topologies are proposed in [39]. The authors explored different chiplet topologies and compared their performance through an analytical metric which estimates latency. A chiplet-based IMC benchmarking tool for design space exploration, SIAM, is proposed in [4]. SIAM supports different chiplet architectures, IMC crossbar tile structures, NoP, NoC, and DRAM estimation. However, all prior studies assume a homogeneous chiplet structure across all chiplets interconnected by a single NoP. Furthermore, none of the prior works considered the non-uniform distribution of weights and activations in the DNN during the mapping process. Hence, many chiplets remain under-utilized while the large NoP leads to an increased area and energy overhead.

In contrast to prior works, we propose a heterogeneous big-little chiplet-based IMC architecture that combines big chiplet bank with a bridge-based NoP and a little IMC chiplet bank with an interposer-based NoP to enhance the IMC utilization and improve energy efficiency. Furthermore, we propose a customized methodology

that exploits the non-uniform distribution of DNN weights and activations in mapping the DNNs onto the big-little chiplet IMC architecture. To the best of our knowledge, this is the first heterogeneous chiplet-based IMC architecture that leverages different IMC structures collectively with a heterogeneous NoP coupled with a customized DNN mapping.

2.2 Home-based Rehabilitation Systems

Home-based rehabilitation systems draw significant attention, especially during the pandemic era, since they facilitate patient access to rehabilitation exercises at home and reduce in-person physical therapy sessions. To this end, researchers established the relationship between human joint location and rehabilitation movements [42, 43]. Authors in [42] proposed an approach that can get human joint information and face videos to relate the pain to the patient's movement. UI-PRMD [43] dataset provides exercise data using Kinect and motion capture system. To check whether the exercises conform to standards, rehabilitation systems require accurate human pose estimation, usually obtained from RGB images [44, 45]. OpenPose [44] and HRNet [45] are the most representative approaches that achieve fast and accurate human pose estimation from RGB sources.

mmWave radar-based pose estimation addresses privacy concerns and enhances robustness to the environment compared to RGB-based approaches, thus being an emerging solution for rehabilitation systems [46, 47, 48]. These approaches map 3D mmWave point cloud to ground truth human joints using smaller CNNs

than those processing RGB video ones since the mmWave frames are significantly smaller than their RGB counterparts. However, existing techniques focus on offline learning and algorithm design. Since they assume that offline-design CNNs will generalize to arbitrary users, they only consider inference during rehabilitation exercises. Hence, they do not deal with on-device training after a new patient starts using the system. In strong contrast, our proposed framework achieves end-to-end real-time mmWave-based human pose estimation, including training and inference.

The acceleration of both training and inference is critical for the real-time execution of applications. To this end, we utilize a Resistive RAM (ReRAM)-based in-memory computing (IMC) AI accelerator for our framework. IMC-based hardware accelerators perform computation inside memory units to reduce off-chip data communication. ReRAM-based approaches achieve high density and low energy consumption. Therefore, they are widely used for machine learning acceleration [5, 1, 49, 50, 28]. The training of CNN models is vulnerable to gradient precision and the write endurance and nonlinear properties of ReRAM architectures can cause an accuracy loss during training [51, 52]. Authors in [52] and [53] proposed methods to mitigate these problems. By utilizing these methods, on-chip training on ReRAM-based IMC accelerators is possible without seeing a significant accuracy drop.

In contrast to prior work, we propose PHR, a ReRAM-based IMC accelerator with energy-efficient on-chip training capacity for home-based rehabilitation systems. To the best of our knowledge, it is the first system that enables real-time processing of RGB image data, fast on-chip training of mmWave radar-based human pose

estimation, and energy-efficient model inference for continuous patient usage.

2.3 Hardware-Aware Neural Network Optimizations

Network pruning is a widely studied approach to reducing network sizes by eliminating redundant parameters. Optimal Brain Damage [54] and Optimal Brain Surgeon [55] are early works in this domain. Authors in [56] prune the network weights with an absolute value closest to zero and a little to no contribution to the output. More recent pruning methods incorporate training to retain a similar accuracy as the original network [57]. For example, the Lottery Ticket Hypothesis (LTH) finds a subnetwork, referred to as the winning ticket, that can achieve the same test accuracy as the unpruned network if trained in isolation [2]. The authors train a network with randomly initialized weights for a predefined number of epochs. Then, p-percentage of the weights with the smallest absolute value is pruned to obtain the winning ticket. After that, the unpruned weights are reset to their initial values before repeating the training. Then, the pruned network, referred to as the *winning ticket* subnetwork, is trained for the same number of epochs. By iteratively repeating this process, the authors generate winning tickets that have 80%–90% fewer weights while keeping a similar accuracy. Table 1 shows the comparison of CANNON against LTH and other the state-of-the-art pruning and mapping methods.

Similar to network pruning, a sparse evolutionary training approach, called SET, can guarantee built-in sparse structures during training [58]. SET achieves

Table 2.1: Comparison of various mapping and pruning approaches

Method	FC Pruning	Conv Pruning	HW-Aware Training	Mapping
Wu et al. [6]	no	no	no	yes
Mocanu et al. [58]	yes	no	no	no
Frankle & Carbin [2]	yes	yes	no	no
Karimzadeh et al. [59]	yes	yes	no	yes
Chu et al. [60]	yes	yes	no	yes
Meng et al. [25]	yes	yes	no	yes
CANNON	**yes**	**yes**	**yes**	**yes**

higher test accuracies compared to unpruned networks. However, it uses sparse connections only for the fully connected layers, leaving the convolutional layers intact.

None of the mentioned approaches considers the hardware resources and accounts for the communication cost during pruning. Hence, these approaches *cannot* exploit the full potential of the target hardware. Recent studies started considering hardware-aware techniques due to the importance of the target hardware. Wu et al. [6] use Reinforcement Learning (RL) for mapping DNNs to hardware, but they do not perform pruning. Wang et al. [61] quantize DNNs to reduce the model size using a hardware-aware quantization method. A Linear Feedback Shift Register is used to decide which weights to prune (or not) in [59]. Lately, structural pruning gained popularity due to its pruning in regular shapes. Structural pruning prunes the network such that a significant chunk of memory elements can be removed. Structural pruning techniques for CPUs are proposed in [62, 63]. Similarly, the Scalpel technique [23] prunes networks to match the network size to SIMD

units. 3PXNet [64] combines binarization and pruning for edge machine learning. Technique in [65] proposes to remove a complete channel filter of a convolutional network instead of randomly selecting the weights in a channel to prune on embedded GPUs. However, the authors show that existing GPU libraries do not perform channel pruning efficiently. The methods proposed in [25, 66, 60, 67, 68, 69, 70, 71] use resistive-RAM (RRAM) based hardware accelerators for structured pruning and propose crossbar-aware structural pruning to prune the network. Authors in [60, 25, 70, 69, 66] employ crossbar-aware column and row pruning, which necessitates an indexing unit to handle migrating weight columns. This hardware unit results in extra overhead. Authors in [67] do not use indexing units but reduce the number of bits required for the ADC. Authors in [68, 71] use operation units that divide the crossbar arrays into small matrices to utilize each crossbar column with multiple weight columns. This approach utilizes the time division principle (i.e., activates some part of a crossbar column at a particular time instance) for the weight columns, resulting in an extra time overhead. However, any of the mentioned approaches do not consider the overhead of the communication of activations on the system while utilizing structured pruning.

The unique aspects of our work are highlighted in Table 2.1 where we compare CANNON against the state-of-the-art network pruning and mapping methods. In contrast to prior techniques, we account for the communication cost to enable hardware-aware training. We first map the target network onto the target hardware to minimize the communication latency and then perform hardware-aware dynamic sparse training. Our proposed methodology can work with any pruning technique.

We chose SET [58] as the baseline since it guarantees sparse connections between each DNN layers by construction. However, different than SET, our approach also prunes convolutional (in addition to the fully connected) layers. Our systematic approach introduces sparse graphs that can enable structured sparsity where blocks of zeros can be obtained. Hence, our proposed technique complements efficient sparse representations like Compressed Sparse Block (CSB) [72], Compressed Sparse Row (CSR) [73], Compressed Sparse Fiber (CSF) [74] to minimize data communication overheads. To the best of our knowledge, CANNON is the first hardware-aware training technique that combines all the features listed in Table 2.1.

2.4 Task Scheduling Techniques for Heterogeneous Architectures

Schedulers have evolved significantly to adapt to different requirements and optimization objectives. Static [75, 76] and dynamic [77, 78, 79, 80] task scheduling algorithms have been proposed in the literature. Completely Fair Scheduler (CFS) [77] is a dynamic approach that is widely used in Linux-based OS and aims to provide resource fairness to all processes while the static approaches presented in [75, 76] optimize the makespan of applications. CFS [77] was initially developed for homogeneous platforms, but it can also handle heterogeneous architectures (e.g., Arm big.LITTLE). While CFS may be effective for client and small-server systems, high-performance computing (HPC) and high-throughput computing (HTC) necessitate different scheduling policies. These policies, such as Slurm and

HTCondor, are specifically designed to manage a large number of parallel jobs and meet high-throughput requirements [81, 82]. On the other hand, DSSoCs demand a novel suite of efficient schedulers that execute at nanosecond-scale overheads since they deal with scheduling tasks that can execute in the order of nanoseconds.

The scheduling overhead problem and scheduler complexities are discussed in [83, 84, 85, 86, 87, 88, 89, 90]. The authors in [83] propose two dynamic schedulers, calledCATS and CPATH, where CATS detects the longest and CPATH detects the critical paths in the application. CPATH algorithm shows inefficiency in terms of its higher scheduling overhead. Motivated by high scheduling overheads, Anish et al. [91] propose a new scheduler that approximates an expensive heuristic algorithm using imitation learning with low overhead. However, the scheduling overhead is still approximately 1.1 s, making it impractical for DSSoCs with nanosecond-scale task execution. Authors in [90] propose a neural network-based scheduler selector that schedules eight independent tasks to eight cores from three different architectures. However, using a neural network-based architecture for the selector results in high overheads unsuitable for DSSoCs. Energy-aware schedulers for heterogeneous SoCs have limited applicability to DSSoCs because of their complexity and large overheads [92, 93, 94, 95]. There are other papers that discuss the overhead of data offloading through dispatching between accelerators and CPU units using schedulers [96, 97].

Several scheduling algorithms that demonstrate the benefits of using multiple schedulers are proposed in [98, 99, 100]. Specifically, the authors in [98] propose a technique that switches between three schedulers dynamically to adapt to varying

job characteristics. However, the overheads of switching between policies are not considered as part of the scheduling overhead. The approach in [100] integrates static and dynamic schedulers to exploit both design-time and runtime characteristics for homogeneous processors. The hybrid scheduler in [100] uses a heuristic list-based schedule as a starting point and then improves it using genetic algorithms. However, it does not consider the scheduling overhead of the individual schedulers. The authors in [84] discuss the performance comparison of a simple round-robin scheduler and a complex earliest deadline first (EDF) scheduler and their applicability under different system load scenarios.

Using insights from literature, we propose a novel scheduler that combines the benefits of the low scheduling overhead of a simple scheduler and the decision quality of a sophisticated scheduler (described in Section 6.2.3) based on the system workload intensity. To the best of our knowledge, this is the first approach that uses a novel runtime preselection classifier to choose between simple and sophisticated schedulers at runtime to enable scheduling with low energy and nanosecond scale overheads in DSSoCs.

2.5 ML-based Task Schedulers and Runtime Monitoring Techniques for Domain-Specific SoCs

Domain-specific SoCs have gained traction in recent years following the demand for specialized processing and energy-efficient solutions. Recent work discussed these architectures and proposed accelerations frameworks across different application

domains [101, 102, 103, 104]. Modern computing systems, including domain-specific SoCs, often rely on runtime heuristics for task scheduling [77, 78, 105]. Alongside heuristic schedulers, list-based schedulers [75, 88, 89, 94, 106, 76] have been proposed for task scheduling, aiming to optimize performance metrics at design time. However, one limitation of list-based schedulers is their inability to account for scenarios involving multiple streaming applications with varying initialization times. Furthermore, optimization-based schedulers, such as those utilizing integer linear or constraint programming techniques [80, 79, 107], aim for optimal decision-making but suffer from infeasible runtime overheads due to computational complexity.

ML-based task schedulers have recently emerged as alternatives to conventional algorithms and heuristics [108, 109, 91, 110, 111, 112, 113, 114, 115, 116], offering reduced overhead while achieving near-optimal outcomes. They leverage various features, including performance counters, task, and application-related data, to make informed decisions. These features encompass a broad spectrum of metrics, ranging from task execution durations on diverse resources to communication latencies and resource availability, chosen strategically to optimize decision-making. At the same time, a deep neural network or decision tree policy enables predictable execution time and runtime overhead optimization. These schedulers leverage various ML methods such as support vector machine (SVM) [116], imitation learning (IL) [91, 112], and reinforcement learning (RL) [108, 109, 111, 114, 113, 115]. IL models, for instance, emulate the behaviors of complex schedulers impractical for runtime usage, showcasing efficiency by eliminating the need for exhaustive search

or optimization algorithms. However, they are prone to sensitivity toward their training datasets and inherent biases from expert behaviors, rendering them vulnerable to unseen changes and generalization issues. In contrast, RL-based schedulers learn a policy that optimizes a performance metric [111, 109] or multiple metrics [114] by exploration. For instance, Decima [108] specializes in cluster-level scheduling for streaming applications using graph and deep neural networks. These schedulers also allow runtime adaptability, iteratively refining their weights to accommodate changes in response to evolving workload dynamics. They provide a significant advantage over static approaches, such as heuristics, particularly in swiftly changing environments inherent to domain-specific SoCs. Nonetheless, all these methods necessitate a monitoring framework to (1) confirm the generalization of the ML policy to new data encountered at runtime and (2) adapt the policies if needed.

Monitoring frameworks for ML models focus on detecting data and concept drifts. Data drift detection methods [117, 118] typically employ statistical models to assess whether the observed data deviate significantly from a reference distribution. In contrast, concept drift detection methods [119, 120, 121] focus on detecting shifts in the relationship between input and output using statistical and ML-based classifiers. However, these methods often have high computational complexity and execution times in the order of seconds), making them impractical for runtime applications, especially in scenarios with short task durations typical of domain-specific SoCs. Additionally, their ability to adapt to evolving data distributions may be limited due to inherent assumptions. In contrast, our proposed framework takes

a different approach by leveraging changes in gradients to quantify generalization to new data without making assumptions about the input data. Moreover, recent work has explored the robustness of ML models using mixed integer linear programming (ILP), resulting in runtime requirements ranging from seconds to minutes [122]. On the task scheduling problem, researchers discuss the robustness of task scheduling methods [123], using metrics such as expected execution time and missed deadlines, often overlooking considerations related to generalizing to new applications. To the best of our knowledge, our proposed framework is the first runtime monitoring framework tailored for ML-based task scheduling on domain-specific SoCs.

3 BIG-LITTLE CHIPLETS FOR IN-MEMORY ACCELERATION OF DNNS: A SCALABLE HETEROGENEOUS ARCHITECTURE

3.1 Background, Motivation, and Contributions

State-of-the-art deep neural networks (DNNs) have become more complex with deeper, wider, and more branched structures to cater to demanding applications [124, 125]. The growing complexity reduces hardware inference performance due to increased memory accesses and computations [124]. To boost the performance and energy efficiency, in-memory computing (IMC)-based architectures embed the matrix-vector-multiplications within the memory arrays [5, 50, 126, 127, 128]. However, IMC architectures with stationary weights stored on the chip result in significant area overhead and fabrication cost [4, 5]. Hence, 2.5D/3D architectures are adopted to design large-scale DNN accelerators using an array of small chips (i.e. chiplets) connected by a network-on-package (NoP) [34, 129].

Prior studies have demonstrated chiplet-based architectures based on both IMC and conventional multiply-and-accumulate (MAC) engines for DNN acceleration [4, 40, 1, 34, 37, 129, 32, 33, 36, 130, 131, 132, 133]. However, existing schemes do not consider the non-uniform distribution of weights and activations within DNNs while designing the chiplet-based architecture. Figure 3.1(a) and Figure 3.1(b) show the distribution of activations and weights (normalized) across all layers of ResNet-50 on ImageNet and VGG-19 on CIFAR-100. The initial layers have more activations between layers but have fewer weights. A larger number of activations lead

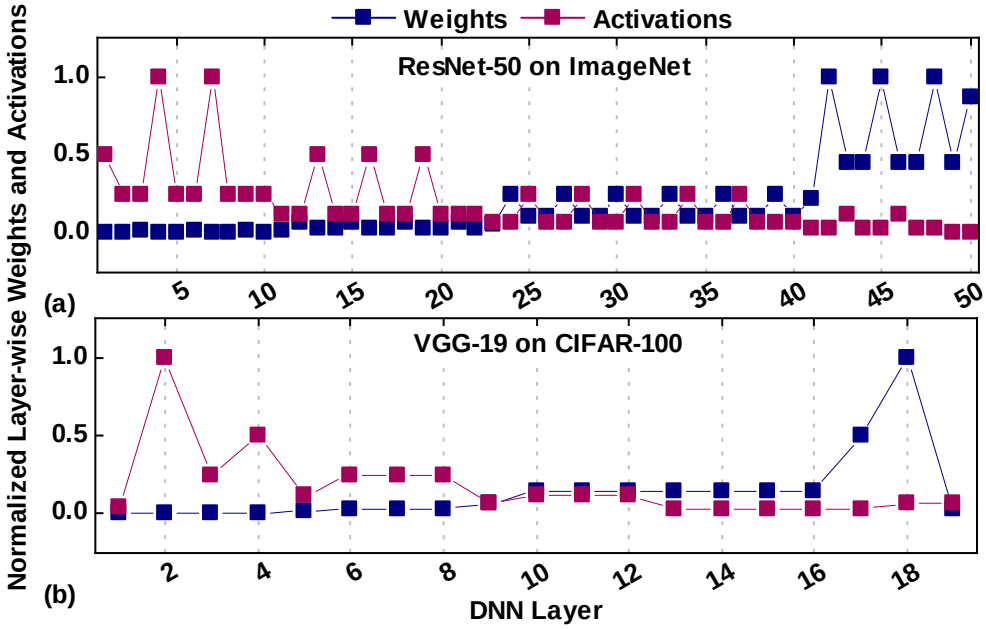

Figure 3.1: Normalized layer-wise activation/weight distribution for (a) ResNet-50 (ImageNet) and (b) VGG-19 (CIFAR-100). Initial/latter layers are activation/weight dominated.

to more on-chip data movement, while fewer weights imply reduced computations. In contrast, the latter layers have more weights and fewer activations, resulting in increased computations and reduced data movement. Hence, the chiplet-based IMC architectures should be optimized to match the non-uniform algorithm structure and maximize the efficiency of computation and data movement across the DNN layers.

Figure 3.2(a) shows the IMC utilization of four different DNNs using a homogeneous chiplet-based RRAM IMC architecture. The architecture utilizes chiplets with 16 tiles, where each tile consists of an array of 16 IMC crossbar arrays of size 256×256 [4]. The chiplets are interconnected by a 32-bit wide NoP operating at

250MHz, having the signaling scheme in [41]. Smaller DNNs like DenseNet-40 on CIFAR-10 have 29% IMC utilization, while larger DNNs like VGG-19 on CIFAR-100 achieve 40% IMC utilization. A lower IMC utilization leads to increased IMC array arrays and in turn, higher energy and latency. Furthermore, a single NoP structure results in significant area overhead due to the large NoP driver and interconnect cost. Figure 3.2(b) shows that for the homogeneous structure, the NoP accounts for 90% and 50% of the total area for VGG-19 on CIFAR-100 and DenseNet-40 on CIFAR-10, respectively. In addition, the increased NoP bus width leads to higher NoP energy with up to 53.75× higher cost relative to an 8-bit multiply-and-accumulate (MAC) operation in 16nm technology node [40].

Optimizing the architecture and the NoP will lead to efficient execution of DNN models. Therefore, this work addresses the inefficiency of homogeneous chiplet-based IMC architectures that fail to exploit the underlying distribution of weights and activations within DNNs. To this end, we propose a heterogeneous chiplet-based IMC architecture that integrates big and little-chiplet banks, as illustrated in Figure 2.1. Specifically, we develop an algorithm to determine the optimal configuration of the big-little IMC chiplet architecture. The little-chiplet bank consists of little chiplets interconnected by an interposer-based NoP (chiplets are placed closed to each other) [41]. Similarly, the big-chiplet bank consists of big chiplets interconnected by a bridge-based NoP [134]. Little chiplets consist of fewer/smaller IMC crossbars or processing element (PE) arrays, while the big chiplets have more/larger IMC crossbars or PE arrays. In addition, each chiplet (big/little) utilizes a local DRAM to store the weights of the DNN.

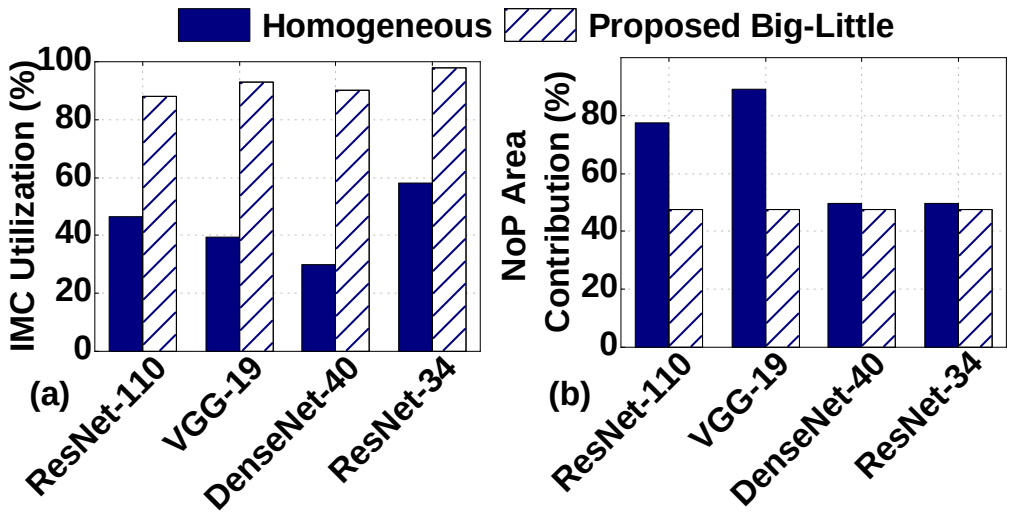

Figure 3.2: IMC utilization for different DNNs using a homogeneous chiplet RRAM IMC architecture [4] and the proposed heterogeneous big-little chiplet architecture. The heterogeneous big-little architecture improves the IMC utilization.

In addition to the hardware architecture, we also propose a new technique to map DNNs onto the big-little chiplet-based IMC architecture. Taking a cue from the non-uniform distribution of the weights and activations within the DNN, we propose to map the early layers within a DNN onto the little chiplet bank and the subsequent layers onto the big chiplet bank. The smaller structure of the weights in the early layers results in higher utilization within the little chiplet bank, while the larger layers towards the end of the DNN achieve high utilization on the big-chiplet bank. To achieve this, we develop a custom mapping algorithm that performs the mapping of the DNN on to the big-little architecture. We note that, the algorithm is universal and applies to the case when the resource in a given big-little chiplet is not enough to store all DNN weights.

We exploit the activation distribution by utilizing an interposer-based NoP

with high bandwidth within the little chiplet bank, which houses the early layers with higher on-chip data movement. Simultaneously, the subsequent layers with lower on-chip data movement (fewer activations) utilize the bridge-based NoP with lower bandwidth within the big chiplet bank. Experimental evaluation of the proposed big-little chiplet-based RRAM IMC architecture on ResNet-50 on ImageNet shows up to 259×, 139×, and 48× improvement in energy-efficiency with lower area compared to Nvidia V100 GPU, Nvidia T4 GPU, and SIMBA [1] architecture, respectively.

The main contributions of this chapter is as follows:

- We propose a heterogeneous big-little chiplet-based IMC architecture that utilizes a big and little IMC-based chiplet compute structure coupled with an optimal NoP configuration (interposer and bridge).
- We present a custom mapping strategy of DNNs onto the big-little chiplet IMC architecture that exploits the non-uniform distribution of weights and activations,
- Our experiments of the proposed big-little chiplet-based RRAM IMC architecture on ResNet-50 on ImageNet achieve up to 259×, 139×, and 48× improvement in energy-efficiency and lower area compared to Nvidia V100 GPU, Nvidia T4 GPU, and SIMBA [1] architecture, respectively.

The rest of this chapter is organized as follows. Section 3.2 provides an overview of architecture. In Section 3.3, we discuss the parameters of the architecture and mapping methodology, followed by relevant experimental results in Section 3.4.

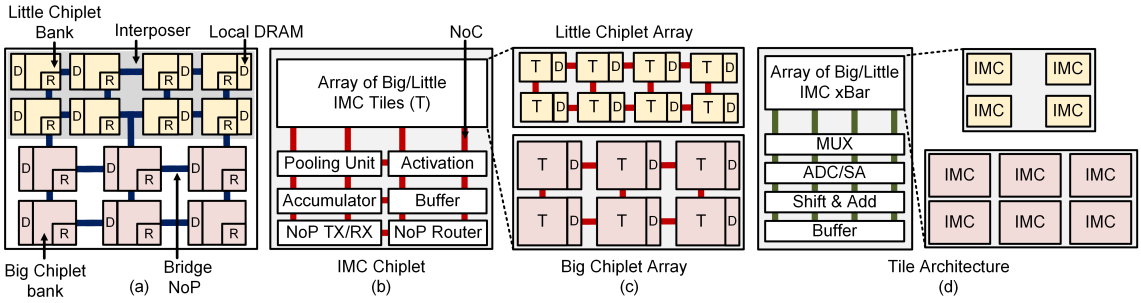

Figure 3.3: (a) Overview of the big-little chiplet IMC architecture. The little chiplet bank utilizes smaller chiplets connected by a interposer-based NoP while the big chiplet bank utilizes bigger chiplets connected by a bridge-based NoP. Each chiplet utilizes a local DRAM, (b) IMC chiplet architecture (big and little). Each chiplet consists of an array of IMC tiles and a dedicated NoP transceiver and router, (c) The little chiplet bank consists of fewer and smaller tiles while the big chiplet bank consists of more bigger tiles. Both chiplet structures utilize a mesh-based NoC for on-chip communication, and (d) Structure of each tile within the big and little chiplet. It consists of an array of IMC crossbar arrays and associated peripheral circuits with an interconnect similar to that in [5]. The little chiplet consists of fewer and smaller IMC crossbars while the big chiplet has larger and more IMC crossbar arrays.

3.2 Overall Architecture

Figure 3.3(a) shows the top-level block diagram of the heterogeneous big-little chiplet IMC architecture. The architecture consists of two banks of IMC chiplets, a little bank (shown in yellow color) and a big bank (shown in light red color). The little IMC chiplet bank consists of chiplets with smaller and fewer IMC crossbar arrays compared to the big chiplets. It is placed on an interposer that houses the NoP. The NoP provides high bandwidth and a compact structure for on-package communication within the little chiplet bank. At the same time, the increased size and count within the big chiplet bank allow for higher computation capability. The

big chiplets are directly connected to the substrate using micro-bumps. A bridge-based NoP is utilized within the big chiplet bank for on-package communication. Long wires of the bridge NoP allow easy integration of the big chiplets. We utilize the Y–X routing methodology for the NoP. Each chiplet (big and little) consists of a local DRAM (DDR4 in this work) that stores the weights required for the IMC crossbar arrays.

Figure 3.3(b) shows the structure of a IMC chiplet. Each chiplet utilizes a hierarchical structure that consists of an array of big (bottom of Figure 3.3(c)) or little IMC tiles (top of Figure 3.3(c)) and each tile consists of an array of IMC crossbars or PEs. In addition, the chiplet contains a pooling unit, non-linear activation unit, accumulator, and buffer. The accumulator is used for the partial sum accumulation across different tiles within the chiplet. Furthermore, the buffers allow for efficient data movement in and out of the chiplet. Each IMC chiplet consists of a dedicated NoP transceiver used for the transmission and reception of packets across the NoP. In this work, we adopt the NoP transceiver from [41]. Each transceiver consists of a local PLL circuit that provides the clock for the transceiver. A five-port router is utilized for routing of the data across the NoP.

Each IMC chiplet utilizes a local DRAM to store the weights. The local DRAM allows for external memory access, thus making our proposed big-little architecture a generic platform. If a DNN does not fit on the entire chip, the DRAM stores all the weights necessary for each chiplet. First, the DRAM loads the necessary weights into the IMC crossbar arrays. Next, while the computation is performed, the DRAM loads the next set of weights of the DNN. The buffer is designed to support a ping-

pong operation [135]. The weights from the DRAM are loaded into the first buffer stage (ping) and then moved to the second buffer stage (pong). Therefore, the big-little IMC chiplet architecture masks the DRAM latency with the computation latency, achieving high throughput.

Finally, Figure 3.3(d) shows the structure of an IMC tile. Each array in the crossbar consists of PEs that perform the computations. In this work, we focus on a resistive random-access-memory (RRAM) based IMC crossbar array due to its superior energy-efficiency [5]. The computations are performed in the analog domain by turning on all wordlines (WL) together and performing accumulation along the bitline (BL). The inputs are given through the WL while the weights are stored within the RRAM cells. Each IMC array consists of specialized peripheral circuitry that assists the computation. The peripheral circuitry includes a column multiplexer (mux), an analog-to-digital converter (ADC), a shift and add circuit, and a buffer. The column mux is used to share the ADC across columns of the IMC array. The ADC converts the MAC output in the analog domain across each column into the digital domain. The big-little IMC architecture does not utilize a digital-to-analog converter (DAC) by employing bit-serial computing. The shift and add circuit handles the positional value of each bit within the multi-bit input activations that are computed using the IMC arrays. The buffers within the tile are utilized for storing the partial sums and the input activations.

The following two sections present the implementation details and experimental evaluations, respectively.

3.3 Parameters of the Big-Little Architecture and Mapping

This section describes the implementation and mapping details of the Big-Little chiplet architecture.

The underlying non-uniform distribution of weights and activations within a DNN results in an increased number of activations in the early layers and larger number of weights in the subsequent layers (Figure 3.1). This non-uniform weight distribution leads to under-utilization of chiplets in the early layers, thus a lower overall IMC utilization. To improve the IMC utilization, crossbar arrays with smaller size (e.g. 32×32 instead of 128×128) can be used everywhere. However, using smaller crossbar arrays also leads to increasing number of chiplets in the system. In turn, larger number of chiplets in the system increases the area as well as energy consumption (due to higher relative area and energy of the peripheral circuits) masking the benefit of using chiplet-based system. Therefore, a balance between crossbar array size and number of chiplets in the system is necessary. To this end, we propose a technique to optimize the big-little chiplet configuration as discussed next.

Algorithm 1 Determining Big-Little Chiplet Configuration

Input: DNN structure, number of chiplets (N_C), set of crossbars sizes for the little chiplets ($\mathcal{X}_\mathcal{L}$) and the big chiplets ($\mathcal{X}_\mathcal{B}$); set of number of tiles in the little chiplets ($\mathcal{T}_\mathcal{L}$) and the big chiplets ($\mathcal{T}_\mathcal{B}$); number of little chiplets ($\mathcal{N}_\mathcal{L}$) and number of big chiplets ($\mathcal{N}_\mathcal{B}$)
Output: Tile utilization for each configuration i (U_i)
$N_{cfg} \leftarrow$ number of configurations in the set containing all possible combinations of the elements in $\mathcal{X}_\mathcal{B}$, $\mathcal{X}_\mathcal{L}$, $\mathcal{T}_\mathcal{L}$, $\mathcal{T}_\mathcal{B}$, $\mathcal{N}_\mathcal{L}$, $\mathcal{N}_\mathcal{B}$
$L \leftarrow$ number of DNN layers
for $i = 1 : N_{cfg}$ **do**
    n_l = Number of little chiplets in Config-i
    n_b = Number of little chiplets in Config-i
    $j \leftarrow 0$ // Number of layers already mapped
    $U \leftarrow 0$ // Sum of utilization
    $n_l^u \leftarrow 0$ // Number of little chiplets used
    while $n_l^u \leqslant n_l$ and $j < L$ **do**
        $j_t \leftarrow$ Number of tiles required for layer-j
        $r_t^l \leftarrow$ Number of remaining tiles in the little chiplet
        if $j_t < r_t^l$ **then**
            Map layer-j to the little chiplet
            $u_j \leftarrow$ Tiles utilization for layer-j (Eq. 3.1)
            $U = U + u_j$;
            $j = j + 1$
        else
            $n_l^u = n_l^u + 1$
        end if
    end while
    $n_b^u \leftarrow 0$ // Number of big chiplets used
    while $n_b^u \leqslant n_b$ and $j < L$ **do**
        $j_t \leftarrow$ Number of tiles required for layer-j
        $r_t^b \leftarrow$ Number of remaining tiles in the big chiplet
        if $j_t < r_t^b$ **then**
            Map layer-j to the big chiplet
            $u_j \leftarrow$ Tiles utilization for layer-j (Eq. 3.1)
            $U = U + u_j$;
            $j = j + 1$
        else
            $n_b^u = n_b^u + 1$
        end if
    end while
    $U_i = \frac{U}{L}$
end for

3.3.1 Configuration of the big-little chiplets

We first determine the configuration of big-little chiplets by computing the tile utilization with different big-little chiplet configurations for a given DNN. Algorithm 1 shows our proposed technique to find the utilization. The inputs to the algorithm are

1. the set of crossbar sizes for the little chiplets ($\mathcal{X}_{\mathcal{L}}$) and the big chiplets ($\mathcal{X}_{\mathcal{B}}$),
2. set of number of tiles in the little chiplets ($\mathcal{T}_{\mathcal{L}}$) and the big chiplets ($\mathcal{T}_{\mathcal{B}}$),
3. number of little chiplets ($\mathcal{N}_{\mathcal{L}}$) and big chiplets ($\mathcal{N}_{\mathcal{B}}$),
4. the DNN structure,
5. the total number of chiplets in the system.

We note that the initial layers of the DNN are mapped on to little chiplets since there are fewer weights in the initial layers. A DNN layer is mapped on to a chiplet when number of tiles required for that layer is less than the number of remaining tiles in the chiplet, i.e., the available resource on the chiplet is sufficient for the layer (as shown in line 13–17 of Algorithm 1). Once a layer (layer-j) is mapped on to a chiplet, the tile utilization is computed as:

$$\begin{aligned} IMC_j &= \left\lceil \frac{K_j^x \times k_j^y \times N_j^{if}}{x} \right\rceil \times \left\lceil \frac{N_j^{of} \times Q}{x} \right\rceil \\ u_j &= 100 \times \frac{K_j^x \times k_j^y \times N_j^{if} \times N_j^{of} \times Q}{IMC_j \times x \times x} \end{aligned} \tag{3.1}$$

where K_j^x and K_j^y are the kernel sizes of layer-j, N_j^{if} and N_j^{of} are the number of i/p and o/p features for layer-j, Q is the quantization precision, IMC_j is the number of IMC crossbars required for layer-j and x is the IMC crossbar size ($x \times x$). Once the resources of a chiplet are exhausted, the next chiplet is considered for mapping. This process continues until no chiplet (little/big) is available.

In the proposed method, for each chiplet configuration, we obtain the average utilization for a particular DNN after each layer is mapped(line 36 of Algorithm 1). Then we sort (in descending order) the configurations based on the utilization and save the top K configurations. The above procedure is repeated for M different DNNs and the configuration with highest utilization which is common for all DNNs is considered as the final configuration for the big-little chiplet system. We note that K and M are user-defined parameters and our proposed technique is independent of these parameters.

3.3.2 Configuration of the big-little NoP

The heterogeneous chiplet configuration (discussed in Section 3.3.1) improves the overall chiplet utilization by using smaller chiplets that match well to the early layers with fewer weights. However, the initial DNN layers produce higher number of activations compared to later layers. Therefore, the volume of traffic between little chiplets (used for initial DNN layers) is higher than the traffic volume between big chiplets (used for later DNN layers). Hence, the network-on-package (NoP) configuration between little chiplets needs to be different than that of the big chiplets. To this end, we propose a technique to determine optimal NoP configuration for a

Algorithm 2 Determining Big-Little NoP Configuration

1: **Input:** DNN structure, number of chiplets (N_C), set of NoP bus widths for the little chiplets ($\mathcal{W}_\mathcal{L}$) and the big chiplets ($\mathcal{W}_\mathcal{B}$); set of NoP frequency for the little chiplets ($\mathcal{F}_\mathcal{L}$) and the big chiplets ($\mathcal{F}_\mathcal{B}$), mapping of layers to the big-little chiplet ($\mathcal{L} \rightarrow \mathcal{C}$)
2: **Output:** NoP EDP for each configuration-i (E_i)
3: $N_{cfg} \leftarrow$ number of configurations in the set containing all possible combinations of the elements in $\mathcal{W}_\mathcal{L}$, $\mathcal{W}_\mathcal{B}$, $\mathcal{F}_\mathcal{L}$, $\mathcal{F}_\mathcal{B}$
4: $L \leftarrow$ number of DNN layers
5: n_l = Number of little chiplets
6: n_b = Number of big chiplets
7: **for** $i = 1 : N_{cfg}$ **do**
8: w_l = Bus-width of little chiplets in Config-i
9: w_b = Bus-width of big chiplets in Config-i
10: f_l = NoP frequency of little chiplets in Config-i
11: f_b = NoP frequency of big chiplets in Config-i
12: $E_i \leftarrow 0$ // Initializing EDP of Config-i
13: **for** $j = 1 : n_l$ **do**
14: Compute edp_j by from Equation 3.2
15: $E_i = E_i + edp_j$ // Communication EDP
16: **end for**
17: **for** $k = 1 : (n_b - 1)$ **do**
18: Compute edp_k from Equation 3.2
19: $E_i = E_i + edp_k$ // Communication EDP
20: **end for**
21: **end for**

system with big-little chiplet targeted for a particular DNN. Algorithm 2 shows the technique to determine NoP configuration for a particular DNN. The inputs to the algorithm are:

1. big-little chiplet configuration obtained from Algorithm 1,

2. set of NoP bus width for the little chiplets ($\mathcal{W}_\mathcal{L}$) and the big chiplets ($\mathcal{W}_\mathcal{B}$),

3. set of NoP frequency for the little chiplets ($\mathcal{F}_{\mathcal{L}}$) and the big chiplets ($\mathcal{F}_{\mathcal{B}}$),

4. the DNN structure.

We evaluate the energy-delay product of communication for each NoP configuration in the set of configurations. An analytical expression based evaluation is incorporated to perform fast exploration in the NoP configuration space. First, we evaluate communication volume of each NoP configuration given a particular DNN. The communication volume is equivalent to the number of packets transferred between two chiplets, and the number of packets (P) is expressed as $P = \frac{b}{w}$, where b is the number of bits to be communicated and w is the NoP bus width. We divide the number of packets by NoP frequency (f) to obtain an approximation of NoP latency $d = \frac{P}{f} = \frac{b}{w \times f}$. Next, we compute NoP power consumption by assuming that it is proportional to cube of NoP frequency [136]; $p = f^3$. Then the approximate energy consumption (e) is computed by multiplying communication latency and communication power; $e = d \times p$. Finally, communication EDP between each pair of chiplet (edp) is computed as:

$$edp = e \times d = d \times p \times d = d^2 \times f^3 = \left(\frac{b}{w \times f}\right)^2 \times f^3 = \frac{b^2 \times f}{w^2} \tag{3.2}$$

The total communication EDP for each NoP configuration for a particular DNN is obtained by adding the communication EDP between each pair of chiplets. A total of K NoP configurations with lower EDP are saved and the above procedure is repeated for M different DNNs. The configuration with lowest cost which is common for all DNNs is considered as the final NoP configuration for the big-little

chiplet system. Similar to the technique of selecting big-little chiplet configuration (described in Section 3.3.1), K and M are the user defined parameter and our proposed technique is independent of these parameters.

3.3.3 Mapping a Previously Unseen DNN to a System on big-little Chiplets

So far, we described our proposed technique to determine the optimal configuration of big-little chiplet and the NoP. The optimal configuration is determined by performing design space exploration with several DNNs. However, an unknown DNN (not seen before) may be encountered at runtime. Moreover, there is no guarantee that all the weights of a given DNN will fit in the on-chiplet resources since the number of DNN parameters seem to be continuously growing. In these cases, we need to divide the entire DNN into multiple parts and load the weights of each part from DRAM before executing. Algorithm 3 shows the DNN partitioning as well as the mapping technique. The input to the algorithm is the DNN structure, big-little chiplet configuration and big-little NoP configuration. First, we compute the number of in-memory computing bits available on the system (S_B). Specifically, for each type (little/big) of chiplets, we multiply the number of available chiplets (n_l/n_b), the number of tiles in each chiplet (t_l/t_b), the number of crossbar array in each tile (16), and the size of IMC crossbar array for big and little chiplets (x_l/x_b). Then we add the product for big and little chiplets to obtain the total number of

in-memory computing bits available on the system (S_B):

$$S_B = (n_l \times t_l \times 16 \times x_l \times x_l) + (n_b \times t_b \times 16 \times x_b \times x_b) \tag{3.3}$$

Next, we compute the number of bits required to store all the weights of the DNN (D_B). Assuming average utilization of $u (0 < u \leqslant 1)$, the total number of partitions (Pr) required for the DNN is computed by taking the ceiling of the quotient obtained by dividing the required number of bits to store all weights (D_B) by the available number of in-memory bits on the system (S_B):

$$Pr = \left\lceil \frac{D_B}{S_B \times u} \right\rceil \tag{3.4}$$

For each partition, first, we compute the utilization of i^{th} layer on a big chiplet (U_B^i) as well as on a little chiplet (U_L^i). We compute U_B^i and U_L^i using Equation 3.1. If the big chiplet utilization ((U_B^i)) is less than the little chiplet utilization (U_L^i) and the little chiplet bank is not exhausted, then the layer is mapped onto a little chiplet, as shown in lines 7–12 of Algorithm 3. Otherwise, we compute the number of big chiplets required (a_B) to map the rest of the layers. If a_B is less than or equal to the number of available big chiplets (A_B), then we map the rest of the layers to the big chiplet bank, else the algorithm throws an error since the resource requirement exceeds the available capacity (shown in line 18–23 of Algorithm 3). Thus, we ensure that the initial layers with fewer weights are mapped into little chiplets and the latter layers with higher number of weights are mapped onto big chiplets with more computation resources. Therefore, our proposed custom mapping of the

Algorithm 3 Mapping DNN Layers to Big-Little Chiplets

Input: DNN layers ($\mathcal{L}$), IMC crossbar size in Big chiplet (x_b), IMC crossbar size in little chiplet (x_l), number of tiles in big chiplets (t_b), number of tiles in little chiplets (t_l), number of available big chiplets (n_b), number of available little chiplets (n_l)
Output: Mapping of layers to of big-little chiplet ($\mathcal{L} \rightarrow \mathcal{C}$)
Compute S_B by following Equation 3.3
Compute Pr by following Equation 3.4
for $j = 1 : Pr$ **do**
    $\mathcal{L}_j \rightarrow$ DNN layers for partition-j; $\mathcal{L}_j \in \mathcal{L}$
    for $i = 1 : |\mathcal{L}_j|$ **do**
        $a_L \rightarrow 1$ // Number of little chiplets used
        Compute utilization of i^{th} (U_B^i) layer on a big chiplet using x_b, t_b
        Compute utilization of i^{th} (U_L^i) layer on a little chiplet using x_l, t_l
        if $((U_B^i < U_L^i) \& (a_L \leqslant A_L))$ **then**
            Map i^{th} layer to little chiplet.
            if Resource in a_L is exhausted **then**
                $a_L \rightarrow a_L + 1$
            end if
        else
            Compute # of big chiplets (a_B) required to map layer-i – layer-$|\mathcal{L}_j|$
            assert($(a_B \leqslant A_B)$, 'Error')
            for $k = i : |\mathcal{L}_j|$ **do**
                Map k^{th} layer to big chiplet.
            end for
            break
        end if
    end for
end for

DNN onto the big-little chiplet architecture ensures high IMC utilization.

3.4 Experimental Evaluation

3.4.1 Experimental Setup

Evaluation platform: To evaluate the proposed heterogeneous big-little IMC chiplet architecture, we use a customized version of the open-sourced tool SIAM [4]. The customization includes the addition of the custom mapping scheme detailed in Section 3.3. In addition, we handle the big-little chiplet IMC architecture by adding the number of each type (big/little) of chiplets, the number of tiles inside big and little chiplets, and the big-little IMC structure. Furthermore, we also assume that each type of chiplet can use different NoP width. The simulator performs the mapping of a given DNN onto the big-little IMC chiplet architecture. The outputs include area, energy, latency, throughput, energy efficiency, and IMC utilization (for all individual components in the architecture). Finally, we add support for intermediate DRAM access (DDR4 [137]) for each chiplet to handle the case where all weights do not fit on the system at once. We plan to open-source the tool and optimization methodology upon acceptance of the paper.

DNN algorithms and architectural parameters: We evaluate the proposed heterogeneous chiplet architecture with DenseNet-40 (0.26M) on CIFAR-10, ResNet-110 (1.7M) on CIFAR-10, VGG-19 (45.6M) on CIFAR-100, ResNet-34 (21.5M) and ResNet-50 (23M) on ImageNet. We utilize an RRAM-based IMC structure for DNN inference with the following parameters: one bit per RRAM cell, a R_{off}/R_{on} ratio of 100, ADC resolution of 4-bits with 8 columns multiplexed, operating frequency of 1GHz [138, 5], and a parallel read-out method. We use 8-bit quantization for

Table 3.1: Set of configurations considered to determine big-little chiplet and NoP structure.

Chiplet Configuration		NoP Configuration	
Parameter	Values in the Set	Parameter	Values in the Set
$\mathcal{X}_{\mathcal{L}}$	{32, 64}	$\mathcal{W}_{\mathcal{L}}$	{16, 32, 64}
$\mathcal{X}_{\mathcal{B}}$	{128, 256, 512}	$\mathcal{W}_{\mathcal{B}}$	{4, 8, 12, 16, 20, 24}
$\mathcal{T}_{\mathcal{L}}$	{9, 16, 25}	$\mathcal{F}_{\mathcal{L}}$	{600, 1000, 1400, 1800} MHz
$\mathcal{T}_{\mathcal{B}}$	{36, 49}	$\mathcal{F}_{\mathcal{B}}$	{600, 800, 1000} MHz

Table 3.2: Performance comparison of each component of a homogeneous (Little only, Big only) chiplet architecture and the heterogeneous Big-Little IMC chiplet architecture for VGG-19 on CIFAR-100.

Configuration	Area					Energy					Latency				
	IMC (%)	NoP (%)	NoC (%)	Total (mm^2)	Normalized to big-little (×)	IMC (%)	NoP (%)	NoC (%)	Total (mJ)	Normalized to big-little (×)	IMC (%)	NoP (%)	NoC (%)	Total (ms)	Normalized to big-little (×)
Little only	11.9	88.0	0.1	952.1	10.9	99.7	0.2	0.1	1.3	4.1	99.7	0.1	0.2	1.6	1.3
Big only	44.0	55.5	0.5	597.2	6.8	78.6	11.0	10.4	0.43	1.3	99.6	0.1	0.3	3.2	2.7
Big-Little (this work)	**52.4**	**47.4**	**0.2**	**87.4**	**1.0**	**99.8**	**0.1**	**0.1**	**0.32**	**1.0**	**99.2**	**0.3**	**0.5**	**1.2**	**1.0**

the weights and activations, and a 32nm CMOS technology node. The chiplets are placed to achieve the least Manhattan distance. The NoP parameters include E_{bit} of 0.54pJ/bit [41], interconnect parameters width, length, and pitch for the interposer-based NoP from [41] and for bridge-based NoP from [139] (Figure 2.1), per lane NoP TX/RX area of 5,304 μm^2, and NoP clocking circuit area of 10,609 μm^2 [140]. In addition, we also model the μbump for both the interposer [141] and bridge-based [142] NoP by utilizing the PTM models [143].

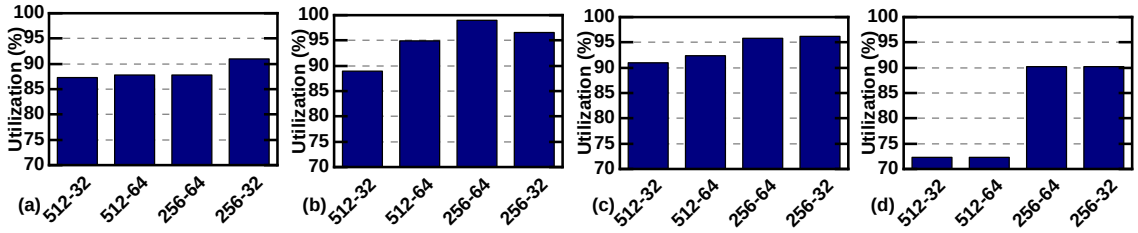

Figure 3.4: IMC utilizations for different DNNs across different big-little chiplet-based RRAM IMC configurations for (a) ResNet-110, (b) ResNet-34, (c) VGG-19, (d) DenseNet-40. Based on the utilization, we choose crossbar size of big chiplet as 256×256 and crossbar size of little chiplet as 64×64 (256–64).

3.4.2 Big-Little IMC Structure and NoP

This section demonstrates the parameters related to big-little IMC structure and big-little NoP. Specifically, we consider four DNNs (mentioned in Section 3.4.1) and execute Algorithm 1 to determine the top 10 (K=10) configurations with highest utilization for each DNN. We consider a system with 36 chiplets to limit the total area and power consumption of the system. Table 3.1 shows the input parameters ($\mathcal{X}_\mathcal{L}, \mathcal{X}_\mathcal{B}, \mathcal{T}_\mathcal{L}, \mathcal{T}_\mathcal{B}$) to the algorithm. We vary the number of little chiplets from 1 to 35 while maintaining the total number of chiplets to be 36. Then, we choose the best configuration which is common for all four DNNs. We observe that a system with *25 little chiplets* with a *64×64 IMC crossbar* and *25 tiles per chiplet*, and *11 big chiplets* with a *256×256* IMC crossbar and *36 tiles per chiplet* provides best utilization across all four DNNs. Figure 3.4 shows the utilization for a system with 25 little and 11 big chiplets with varying size of crossbars (both for big and little chiplet) for all four DNNs. In this case, we also fixed the number of tiles per chiplet to 25 for the little chiplets and 36 for the big chiplets. Figure 3.4 reveals that the configuration where

the crossbar size of the big chiplets is 256×256 and the crossbar size of the little chiplets is 64×64 (256–64, 256 denotes crossbar size of big chiplets and 64 denotes crossbar size of little chiplets) shows higher utilization than other configurations for three out of four DNNs. Only in the case of ResNet-110, the configuration 256–32 shows higher utilization than 256–64. However, we choose 256–64 over 256–32 since it provides more on-chip resources, lower area and energy efficiency for the IMC crossbar array (due to peripheral circuits).

Similarly, we execute Algorithm 2 for four DNNs to obtain the NoP configuration. Table 3.1 shows the set of different NoP parameters ($\mathcal{W}_{\mathcal{L}}, \mathcal{W}_{\mathcal{B}}, \mathcal{F}_{\mathcal{L}}, \mathcal{F}_{\mathcal{B}}$ used as inputs to Algorithm 2). The parameters are adopted from [144]. EDP for NoP is obtained for all NoP configurations for the four DNNs. Then, the NoP configuration having the lowest EDP for all four DNNs is chosen. Based on the EDP results, the big NoP frequency and the little NoP frequency is set to 600 MHz and 1 GHz, respectively; the big NoP bus width and the little NoP bus width is set to 24 and 32, respectively. Figure 3.5 shows the normalized NoP EDP for different combination of bus width for big and little chiplets. For illustration purpose, we show VGG-19 and ResNet-34 since these two DNNs utilize more than 34 out of 36 chiplets. From Figure 3.5, it is observed that the configuration with big NoP bus width of 24 and little NoP bus width of 32 shows the lowest EDP. Since little chiplets produce higher number of activations than the big chiplets, it is intuitive that little NoP are wider (larger bus width) than the big NoP.

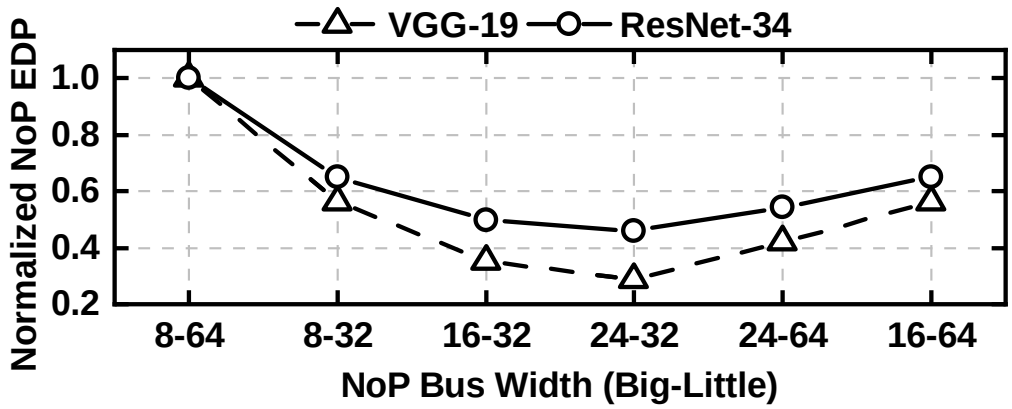

Figure 3.5: Normalized NoP EDP for different bus-widths for VGG-19 and ResNet-34. The NoP with bus width of 24 for big and 32 for little chiplets (24–32) shows lowest EDP.

Table 3.3: Performance comparison of a homogeneous (Little only, Big only) chiplet architecture and the heterogeneous Big-Little IMC chiplet architecture for different DNNs.

Configuration	Utilization (%)				Area (mm^2)				Energy (mJ)				Latency (ms)			
	Res-110	VGG-19	Dense-40	Res-34	Res-110	VGG-19	Dense-40	Res-34	Res-110	VGG-19	Dense-40	Res-34	Res-110	VGG-19	Dense-40	Res-34
Little only	69	92	58	93	171.7	952.1	71.5	657.8	1.4	1.3	0.22	41.1	23.0	1.6	1.6	13.1
Big only	44	59	32	82	220.0	597.2	220.2	595.9	0.28	0.43	0.11	3.7	1.1	3.2	0.02	20.2
Big-Little (this work)	**88**	**93**	**90**	**98**	**87.4**	**87.4**	**87.4**	**87.4**	**0.18**	**0.32**	**0.06**	**8.2**	**1.1**	**1.2**	**0.03**	**48.6**

3.4.3 Comparison with Baseline Architectures with Homogeneous Chiplets

We compare the performance of our proposed big-little chiplet architecture with respect to two baseline architectures with homogeneous chiplets [4]. **1) Little only:** In this configuration, we consider a system where the configuration of all chiplets as well as the NoP is same as that of the little chiplets. **2) Big only:** In this configuration, we consider a system where the configuration of all chiplets as well as the NoP is same as that of the big chiplets. We note that, the total number of

chiplets with 'Little only' and 'Big only' configurations vary for different DNNs.

Table 3.2 shows the performance comparison for 'Little only', 'Big only' and the proposed big-little architectures for VGG-19 on CIFAR-100. In this table, the performance of each component of the architecture, i.e. IMC, NoP and NoC is shown. Our proposed big-little chiplet architecture results in a balanced distribution of the area among the circuit and NoP components, while the NoC accounts for a minimal portion (0.2%) of the total area. In 'Little-only' architecture, NoP becomes the bottleneck for area since the chiplets have smaller size, hence more number of chiplets are required which increases the NoP. In 'Big only' architecture, NoP consumes more energy due to higher volume of data movement between each pair of chiplets. In contrast, the proposed big-little architecture with its high IMC utilization and reduced on-chip communication as well as on-package data movement results in less total energy consumption and less inference latency. Overall, the proposed heterogeneous big-little architecture achieves up to 10.9× lower area, 4.1× lower energy, and 2.7× lower latency than 'Little only' and 'Big only' architectures.

Next, we compare the IMC utilization and the performance (area, energy and latency) for ResNet-110, VGG-19, DenseNet-40, and ResNet-34 against 'little only' and 'big only' architecture. For VGG-19, our proposed big-little architecture achieves the highest IMC utilization of 93% compared to 92% and 59% for 'Little only' and 'Big only', respectively. Similarly, the big-little architecture achieves 88%, 90%, and 98% IMC utilization for ResNet-110, DenseNet-40, and ResNet-34, respectively, up to 2.8× greater than 'Little only' and 'Big only' architectures. We observe that the big-little architecture provides *up to 7.8× improvement in energy and up to 21×*

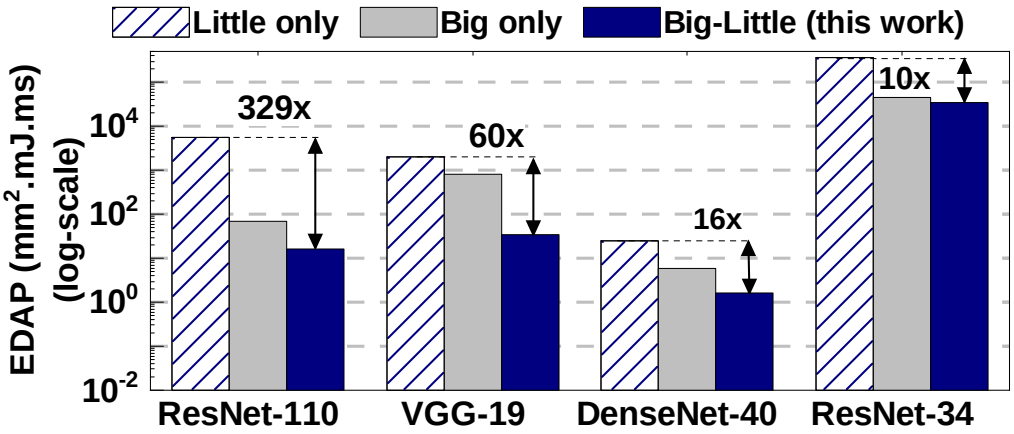

Figure 3.6: EDAP comparison (log-scale) of the big-little chiplet-based RRAM IMC architecture to 'Little only' and 'Big only' chiplet-based RRAM IMC architectures. The big-little architecture achieves up to 329× improvement compared to 'Little only' architecture.

improvement in inference latency with respect to baseline homogeneous architectures. 'Big only' architecture consumes less energy and less latency than big-little architecture for ResNet-34, but in this case, the area of 'Big only' is 6.8× higher than big-little architecture. To better analyze the performance comparison, we plot the energy-delay-area product (EDAP) for all DNNs, as shown in Figure 3.6. The big-little chiplet architecture provides up to 329× lower EDAP than the 'Little only' and 'Big only' architectures across all four DNNs. Although 'Big only' architecture shows improvement in energy consumption and inference latency with respect to big-little for ResNet-34, the EDAP with 'Big only' is 1.3× higher than big-little architecture in this case. Hence, the proposed big-little IMC architecture achieves optimal performance through reduced EDAP at higher IMC utilization across different DNNs.

Table 3.4: Ratio between DRAM energy and compute energy for VGG-16 and VGG-19 with systems having different number of chiplets (**All weights of VGG-19 fit on chip with this configuration, significantly reducing the DRAM energy).

# Chiplets	VGG-16		VGG-19	
	#partitions	Ratio	#partitions	Ratio
36	2	1.1	1	0.08**
25	2	2.1	2	131
16	3	3.6	2	161

3.4.4 Results with DRAM (DDR4)

In this section, we show the performance results when the resource on a big-little chiplet-based system is not sufficient to store all the weights of a given DNN. In that case, the DNN is divided into multiple partitions. One partition is mapped on to the big-little chiplets at a time. While the computations of a partition of the DNN are performed, the weights corresponding to the next partition are loaded from the DRAM into the ping-pong buffer. The additional DRAM accesses result in increased energy. At the same time, the impact on latency is reduced through the ping-pong buffers [135]. Table 3.4 shows the ratio between DRAM energy and compute energy for VGG-16 and VGG-19 with systems having different number of chiplets. We observe that, the ratio of DRAM energy to computation energy increases with reduction in the system sizes for both the DNNs. With decreasing system size, more weights need to be stored and loaded from DRAM, thereby increasing DRAM energy.

Table 3.5: Comparison with other platforms for ResNet-50 on ImageNet (*reported in [1]).

Platform	Area (mm^2)	Energy Efficiency (Images/s/W)
Nvidia V100 GPU*	815	8.3
Nvidia T4 GPU*	525	15.5
SIMBA [1]	215	45
Big-Little (this work)	**85**	**827**

3.4.5 Comparison with State-of-the-art Work

Table 3.5 shows the comparison of the proposed heterogeneous big-little RRAM IMC chiplet architecture with an Nvidia T4 and V100 GPU, and SIMBA [1]. The big-little chiplet architecture achieves a lower area for the architecture due to the custom RRAM-based IMC and the optimized NoP structure. Compared to the Nvidia V100, Nvidia T4, and SIMBA architecture, the big-little IMC architecture achieves 9.6×, 6.2×, and 2.5× area improvement and 99.6×, 53.4×, and 18.4× energy-efficiency improvement, respectively. The improved energy efficiency is attributed to the higher IMC utilization, analog computation within the RRAM-based IMC, reduced NoP data movement and bus width, and the absence of intermediate DRAM transactions for weights and partial sums.

4 ENERGY-EFFICIENT ON-CHIP TRAINING FOR CUSTOMIZED HOME-BASED REHABILITATION SYSTEMS

4.1 Background, Motivation, and Contributions

There is an increasing demand to implement neural networks on mobile edge devices for both on-chip training and inference following recent developments in the internet of things (IoT). Nonetheless, due to the significant computational resources required for training, deploying neural networks on edge devices with limited resources poses a challenge. One such case is home-based rehabilitation systems. Home-based rehabilitation using video cameras and wearable sensors has attracted significant attention due to its potential to help millions of people [42, 43, 46, 145]. For example, a recent study shows more than a two-fold increase in the number of amputations during the COVID-19 pandemic [146]. Remote monitoring and rehabilitation can complement infrequent and prolonged in-person visits to enable early diagnosis and intervention.

Prevalent use of home-based rehabilitation systems requires addressing three critical challenges. *First*, these systems must be sufficiently accurate to detect abnormal behavior and produce actionable data for health professionals. This requirement leads to the *second* challenge: sophisticated algorithms, such as machine learning (ML) and artificial intelligence (AI) techniques. Offloading these algorithms to the cloud is not a desirable solution since sending raw sensor data incurs high communication energy and latency while threatening user privacy. Hence, the

third challenge is accomplishing home-based rehabilitation by running algorithms locally, at the edge.

Recent techniques enable home-based rehabilitation using RGB cameras [42, 43], wearable inertial measurement units (IMU) [145], and millimeter-wave (mmWave) radar sensors [46]. These techniques collect sensor data as the patients perform rehabilitation movements. Then, they process the sensor data, typically using a convolutional neural network (CNN), to produce human joint coordinates. While these approaches show strong potential, they have one fundamental shortcoming. All prior techniques train their inference models with the user data available at design time. Then, they assume that future users, whose number is likely to be much larger than the training set, will use the produced model for inference. Even if the CNNs inference models generalize to arbitrary users, there is no guarantee that their accuracy will remain accurate. Hence, this limitation jeopardizes the first requirement: high accuracy in estimating the joint coordinates. As a result, there is a strong need for approaches that customize the deep learning models to specific users through on-device training in the home environment.

RGB cameras are the most common sources since they offer true-color real-world information. However, *always-on cameras at home* can raise serious privacy concerns such as facial information leakage. In contrast, mmWave radar, an emerging wireless sensing device, can accurately measure objects' moving trends while retaining privacy. In this work, we focus on systems that use mmWave radar inputs since they also have significantly lower processing requirements than RGB camera inputs and do not require users to wear any special sensors. We assume that an inference

model, such as CNN, is trained offline to convert mmWave signals to human joint positions. Then, it is acquired by a new patient for home-based rehabilitation. Since the accuracy of this model is limited by the offline data, the proposed system aims to customize the initial model to the new user, as illustrated in Fig. 4.1. The camera is *activated only during this customization process* to produce the human joint coordinates using the video frames. Then, these joint positions are used as a reference to supervise the incremental training of the inference model that uses the mmWave signals. After the customization, *only the mmWave signals and corresponding inference model* are used during the device lifetime, achieving over 13-fold inference time and 131-fold power consumption savings.

Current processors used for inference at the edge (e.g., at home) have limited processing capability due to their cost and energy constraint. For example, the Nvidia Jetson Xavier NX board can perform inference using mmWave inputs in 149.7 μs per input frame. However, training using RGB camera input reference takes 620 ms per frame on the same device. It can barely achieve a 1.6 frame per second (FPS) operation, which is impractical considering realistic 30 FPS or higher video frame rates. Storing the video frames and performing inference later is also not practical due to excessive memory requirements. Hence, practical solutions require novel AI hardware and methodologies to perform on-device training at the edge. To address this need, we propose an energy-efficient on-device training approach that enables personalized home-based rehabilitation, PHR. The proposed approach first generates the ground truth 3D joint coordinates data using RGB cameras. These coordinates are used to supervise on-device training. Then, we

customize a baseline mmWave human pose estimation model using energy-efficient on-device training. After the customization, our framework uses mmWave radar signals and the customized home-based rehabilitation model.

The main contributions of this chapter is as follows:

- An energy-efficient on-chip training framework that customizes mmWave-based human pose estimation model for higher accuracy.
- A Resistive RAM-based in-memory computing accelerator for on-chip training and inference of mmWave and inference of RGB models.
- Experimental results that demonstrate the practical real-life use of our framework, with a 28.01% lower error, 611.1× lower inference energy, and 14.0× faster training than a baseline model on Nvidia Jetson Xavier NX [3].

The rest of this chapter is organized as follows. Section 4.2 provides an overview of the framework. In Section 4.3, we discuss the experimental results.

4.2 Home-Based Rehabilitation System

This section first overviews the proposed system. Then, Section 4.2.2 presents the proposed hardware platform used for the on-chip training and inference. Section 4.2.3 discusses the on-chip training and ground truth generation. Finally, Section 4.2.4 presents hardware implementation.

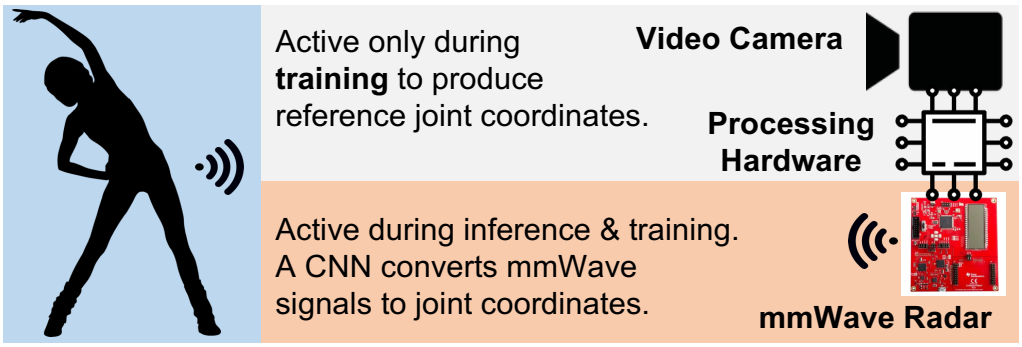

Figure 4.1: Illustration of the target rehabilitation system. The RGB camera is used *only during training* to generate the reference joint coordinates when the initial model is customized to the target user. Once the model that uses mmWave signals is trained, only the mmWave radar is used for inference.

4.2.1 Overview of the Proposed PHR System

Initial Offline Training: The initial inference model, a CNN in this work, is trained offline using a limited set of users. Two video cameras and a mmWave radar detect the patients' movements during training. An *RGB-CNN* [45] model transforms the video frames to 2D joint coordinates. Then, we find the 3D joint coordinates by triangulating 2D joints from two cameras. Finally, these coordinates are used as references for supervised training of a *mmWave-CNN* [46] model that can produce joint coordinates using only the mmWave radar, as shown in Fig. 4.1.

Customization at Home: The initial *mmWave-CNN* can have a poor performance when a new patient starts using it at home, as demonstrated in Section 4.3. Therefore, we personalize it to the new user before continuous execution in three steps. First, we activate RGB cameras for a short duration and generate the 2D joint coordinates by employing the *RGB-CNN* model used in the initial training. Then, the 3D joint coordinates are found in the same way as the initial training. Finally,

we incrementally train the *mmWave-CNN* using these reference coordinates and mmWave signals as inputs. After the customization, the new user uses the product only with mmWave signals and the inference of *mmWave-CNN* model for rehabilitation feedback. Running *RGB-CNN* for RGB image inference and *mmWave-CNN* for both on-chip training and inference are not feasible on a conventional SoC, as demonstrated in Section 4.3. Therefore, we propose an IMC-based hardware accelerator to perform these tasks.

In summary, the proposed PHR pipeline consists of (i) taking RGB images with two cameras, (ii) inferring 2D human key points from the images using *RGB-CNN*, (iii) obtaining ground truth 3D human joints by triangulating two 2D human joints, and (iv) training and inferring the human joints with mmWave signals using *mmWave-CNN*.

4.2.2 In-Memory Computing-based Hardware Acceleration

We employ an in-memory computing (IMC)-based hardware accelerator because it combines highly-dense storage and computation into the same hardware unit. It also alleviates the latency and energy consumption of memory accesses. Energy-efficient DNN accelerators utilizing IMC technology [5, 1, 50] have been proposed in the literature in the last few years. IMC-based architectures integrate multiple processing units, called tiles, into the system, as shown in Fig. 4.2. These tiles are connected via a network-on-chip (NoC) for the data communication of activations, errors, and gradients. Each tile has processing elements along with input/output buffers and an accumulator. Processing elements are composed of crossbar ar-

rays that store the neural network model weights and perform computations and peripheral circuits such as ADCs, adder trees, and buffers. This work uses the mapping methodology described in [53] to map the weights onto the crossbar arrays. We used a ReRAM-based IMC architecture for inference and training. We also include an SRAM-based hardware block for training to avoid excessive writes to the ReRAM crossbar arrays because of the write endurance issues of ReRAM. The IMC architecture also includes activation units, buffers, and accumulators. More detailed discussions about the on-chip training on IMC accelerators and hardware configurations are provided in Section 4.3.

4.2.3 On-Chip Training for Personalization of *mmWave-CNN*

To customize the inference model, *mmWave-CNN*, to a new user, we first need the ground truth joint coordinates. We use two RGB cameras on the edge device to get precise 3D joint coordinates, as described in Section 4.2.1 and Fig. 4.1. The reason for utilizing two RGB cameras is that we can generate the ground truth with minimum error with two RGB images. Two RGB images are then processed by the *RGB-CNN* model. For the *RGB-CNN* model, we followed the structure given in [45], which has 28.5M parameters with a backbone pre-trained on the ImageNet dataset. Then, we generate 3D human joint coordinates by triangulating a pair of 2D human joints. The mmWave radar sensor provides inputs to *mmWave-CNN* model. The *mmWave-CNN* [46] is composed of 3.2M parameters with two convolutional layers followed by two fully connected layers. The radar sensor generates a point cloud which is composed of points reflected from an object in sight. For each point, the radar

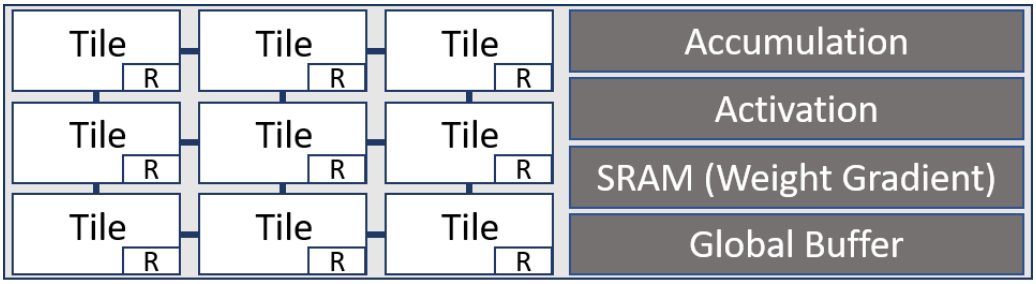

Figure 4.2: The architecture of IMC-based hardware accelerator. Feedforward, error calculation, and weight update stages are performed in the accelerator tiles whereas the weight gradient calculation is executed in the weight gradient block. Tiles are connected via NoC. (R: NoC Router)

sensor generates five features which will be used as inputs for the *mmWave-CNN* model. These are x, y, and z coordinates, the Doppler velocity, and the reflection intensity. Then, using the radar data, we customize the baseline *mmWave-CNN* model to the new user's body pose with the on-chip training on the IMC hardware accelerator. The training consists of four parts: feedforward, error calculation, weight gradient calculation, and weight update. Each step requires DRAM access to store and load the necessary data, as there are thousands of input data. The accuracy and on-chip training results are discussed in Section 4.3.3 and 4.3.5. After the on-chip training, the *mmWave-CNN* is customized for the new user. Finally, the last step in the framework is the customized *mmWave-CNN* model inference that is performed on the IMC accelerator using the tiles of crossbar arrays.

4.2.4 Hardware Implementation and Exploration

The crossbar array sizes in each tile affect the structure and performance of the IMC hardware accelerator. Smaller crossbar arrays increase the number of tiles,

resulting in traffic congestion in the NoC because of high data movement. In contrast, larger array sizes reduce the utilization of the IMC accelerator. Thus, it causes an area overhead compared to small crossbar sizes. Therefore, we explore the hardware parameters to achieve optimal energy and area efficiency, as discussed in Section 4.3.5.

ReRAM-based architectures are vulnerable to the write endurance problem and nonlinear properties. These properties include device-to-device (D2D) and cycle-to-cycle (C2C) variations, long-term potentiation (LTP), and long-term depression (LTD) [51]. Therefore, we analyzed the nonlinear properties of ReRAM-based IMC hardware accelerator design. Detailed explorations of nonlinear properties, variations, and their effects on the accuracy are given in Section 4.3.5. The on-chip training for the customization is an incremental training approach that requires less number of epochs, thus the lower number of writes on the crossbar arrays. Therefore, our framework can achieve better accuracy without assuming limitless writes.

We use the following parameters in our IMC-based accelerator design: an ADC precision of 4 bits, 8-bit quantization for weights and activations, one bit per ReRAM cell, an R_{off}/R_{on} ratio of 100, and 1 GHz operating frequency, and 32 nm technology node [5]. We employ NeuroSim V2.1 [53] to evaluate the performance and area of the proposed ReRAM-based IMC hardware accelerator and data communication between the main memory and the accelerator. NeuroSim V2.1 supports both inference and training on-chip. Only the weight gradient calculation part requires extra hardware (SRAM) for the on-chip training because this stage needs heavy

writing of errors into the arrays along the batch. Feedforward, error calculation, and weight update stages are implemented on the ReRAM tiles. Data movement within the accelerator between tiles via an NoC is evaluated using a customized version of BookSim [4]. In this version, we use a traffic trace-based cycle-accurate execution using a mesh NoC architecture.

4.3 Experimental Results

4.3.1 Experimental setup

We conduct our experiments with the mRI open-source mmWave human pose estimation dataset [145]. mRI offers over 160K synchronized 3D point cloud from mmWave radar (TI IWR1443) [147], and ground truth 2D and 3D human joints from two Kinect V2 cameras [148]. It evaluates ten clinical-suggested rehabilitation movements covering the main parts of the human body, performed by twenty diverse subjects. The dataset benchmarks mmWave-based human pose estimation using HRNet-W32 [45] (images to 2D key points) and MARS [46] (mmWave point cloud to 3D human joints). HRNet-W32 and MARS generate 17 human joint points in 2D and 3D space, respectively. We implemented these CNN-based networks as the *RGB-CNN* and *mmWave-CNN* models using PyTorch. For the initial training, we used an SGD optimizer with momentum ($\beta = 0.9$), and an initial learning rate of 0.001 for 100 epochs with a batch size of 128.

4.3.2 Baseline Accuracy before Customization

We trained the baseline MARS model using half of the user subjects in the dataset. To produce accurate results, we generated three random sets given in Table 4.1. Then, the trained baseline model is tested against the remaining ten subjects for each set. We used *Mean Per Joint Point Error* (MPJPE) and *Procrustes Analysis MPJPE* (PA-MPJPE) metrics, which are widely used for human joint estimation studies [149]. MPJPE calculates the mean Euclidean distance between the reference and the prediction joints. PA-MPJPE uses a similarity transformation for a further rigid alignment.

The *mmWave-CNN* model achieves 130.7 mm MPJPE and 76.4 mm PA-MPJPE for the test subjects in *Set-1*. In contrast, the corresponding errors for the subjects in the training set are 97.8 mm and 57.1 mm, which is about 25% lower. The results are similar for other sets with slightly over 130.7 mm MPJPE and 76.4 mm PA-MPJPE for the test subjects, as shown in Table 4.1. These results show that the initial model can achieve very high accuracy for the known subjects, but the accuracy degrades for new users. Even if one can improve the test accuracy by adding users to the training set, it is impractical to add all potential users who will buy the rehabilitation system.

Table 4.1: Random set configurations for experimental evaluations and training results of the baseline *mmWave-CNN* model.

Sets	Training Subjects	Test Subjects	**MPJPE (mm)**	**PA-MPJPE (mm)**
1	1-3,7,9,14,16,17,19,20	4-6,8,10-13,15,18	130.7 ± 3.4	76.4 ± 1.4
2	1,2,5-7,9,13,17-19	3,4,8,10-12,14-16,20	133.5 ± 2.5	78.4 ± 1.9
3	3,5,6,8,10-12,14,18,20	1,2,4,7,9,13,15-17,19	134.5 ± 1.2	78.5 ± 1.1

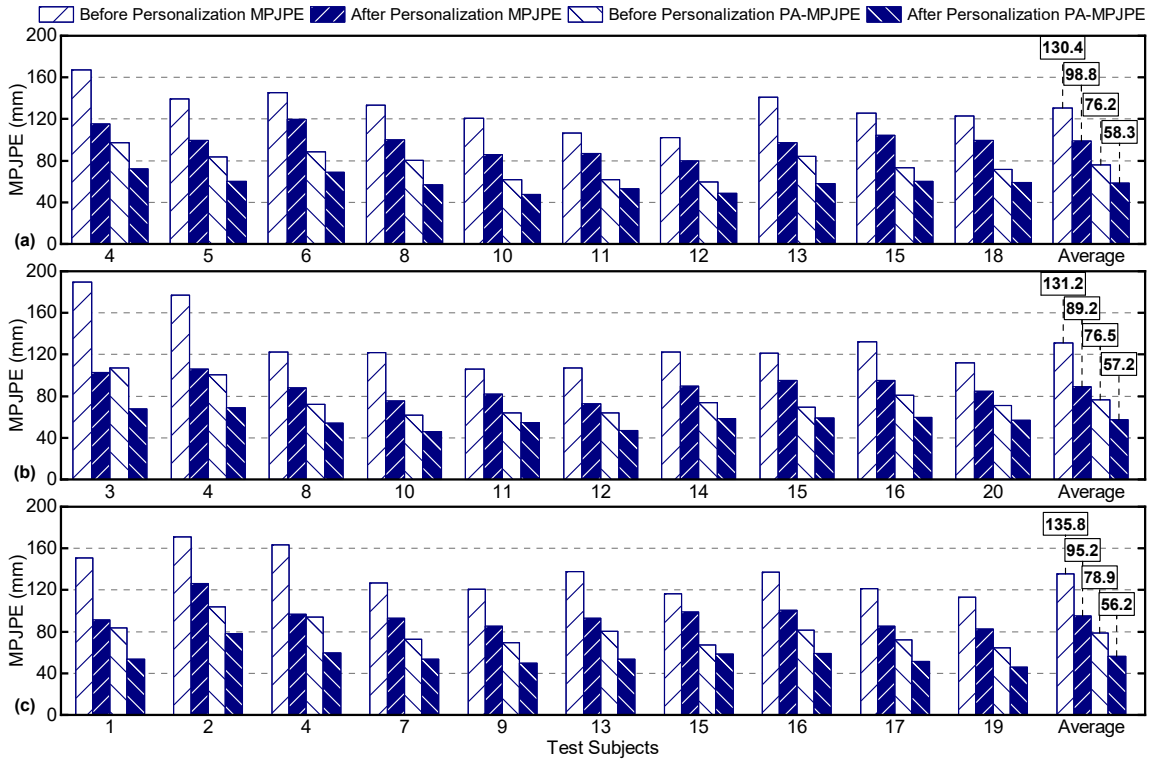

Figure 4.3: MPJPE and PA-MPJPE comparisons for all three random sets. Results show MPJPE and PA MPJPE before customization using 10 subjects for training and after customization which is customized for each test subject separately. Parts (a), (b), and (c) represent *Set-1*, *Set-2*, and *Set-3* results, respectively. As they are randomly split, each plot shows the results for different subjects.

Therefore, *enabling incremental online training at the edge is imperative to adapt the initial model to new users*.

4.3.3 Test Accuracy after Customizing to New Users

On-device training for customization comprises three steps:

1. The video camera is activated for a short period (2.5 minutes, i.e., 4500 frames),

2. The *RGB-CNN* model inference generates the ground truth joint coordinates used for supervision,

3. The baseline *mmWave-CNN* model is incrementally trained using the point cloud data from the mmWave radar sensor as input and the ground truth joint coordinates from step 2 as labels.

We employ the three random sets used for baseline accuracy analysis (Table 4.1) to evaluate the proposed on-device learning technique. Specifically, we customize the baseline model for each subject in the test subjects one-by-one. Fig. 4.3 compares the accuracy of the personalized model to the baseline model. Both MPJPE and PA-MPJPE results improve significantly for all users in *Set-1*, as shown in Fig. 4.3(a). On average, the customization improves the MPJPE and PA-MPJPE by 23.89% and 22.94%, respectively. Most importantly, the average MPJPE and PA-MPJPE reduce to 98.8 mm and 58.3 mm, within only 2 mm of their training accuracy. Fig. 4.3(b) and Fig. 4.3(c) show that these improvements are observed for *Set-2* and *Set-3* as well. The MPJPE improvement ranges from 17.3 mm (14.92%) to 87.1 mm (45.82%), while PA-MPJPE improvement is between 9.0 mm (13.22%) to 39.5 mm (36.83%). In conclusion, the customized model performs consistently and significantly better than the baseline model for all subjects in each random set, enabling accurate feedback necessary for home-based rehabilitation.

Table 4.2: Hardware results for *mmWave-CNN* model inference and training on Jetson Xavier NX with 2 configurations and our framework with 2 configurations and the speedup comparisons. 128×128 and 256×256 represent the crossbar array sizes. (P: PHR, J: Jetson)

	Configurations	Jetson-1 (2 CPUs)	Jetson-2 (6 CPUs)	PHR-1 (128x128)	Improvement (×) P-1 vs J-1	Improvement (×) P-1 vs J-2	PHR-2 (256x256)	Improvement (×) P-2 vs J-1	Improvement (×) P-2 vs J-2
Inference	Power (mW)	3379.0	8724.0	12.1	**277.8**	**717.3**	25.7	131.4	339.2
	Time per frame (µs)	162.2	149.7	10.8	15.0	13.8	4.2	**38.5**	**35.5**
	Energy per frame (µJ)	548.0	1305.9	1.1	488.8	1162.2	0.9	**611.1**	**1452.7**
Training	Power (mW)	9761.4	9986.0	2954.5	**3.3**	**3.4**	4223.3	2.3	2.4
	Time per frame (µs)	306.9	299.4	32.3	9.5	9.2	21.8	**14.0**	**13.7**
	Energy per frame (µJ)	2995.4	2989.6	95.6	31.3	31.2	92.3	**32.4**	**32.3**

4.3.4 Energy and Performance Results

Home-based rehabilitation systems should run on a compact edge device for the practicality of the approach. Therefore, we implemented the baseline model on an Nvidia Jetson Xavier NX [3] board for hardware performance comparisons. The Jetson board has 6 Carmel Arm CPU cores, an Nvidia GPU with 384 CUDA cores, and 48 tensor units. We compared the hardware results of our framework against the Jetson board for both training and inference.

Model Inference: We used two different hardware configurations for performance comparisons. *Jetson-1* and *Jetson-2* shown in Table 4.2 have 2 and 6 active CPU cores (1.9 GHz) and GPU (1.1 GHz) on the board for processing, respectively. The Jetson board's power consumption is on the Watts scale (3.38 and 8.7 W). The latency per frame varies between 149.7 µs and 162.2 µs. The lower energy consumption per frame comes from *Jetson-1* with 548 µJ as it utilizes 2 CPU cores.

Similarly, we evaluated the proposed IMC accelerator using two configurations. The crossbar sizes are selected as 128×128 and 256×256 for *PHR-1*, and *PHR-2*, respectively. The detailed hardware exploration is discussed in Section 4.3.5. Our

accelerator's power consumption is on the scale of milliwatts (12.1 to 25.7 mW) which results in improvements up to 717.3× compared to the Jetson board. The latency per frame are 10.8 µs and 4.2 µs for *PHR-1* and *PHR-2*, respectively. When we compare against Jetson configurations, *PHR-2* shows a higher speedup with 38.5× and 35.5× against *Jetson-1* and *Jetson-2*, respectively. We see considerable improvements in energy consumption as IMC accelerators are highly energy-efficient. The energy consumption per frame of PHR is 1.1 and 0.9 µJ for *PHR-1* and *PHR-2*, respectively. *PHR-2* has a greater improvement in energy consumption with 611.1× and 1452.7× compared against *Jetson-1* and *Jetson-2*, respectively, as it has less number of tiles and less NoC energy.

Model Training: Table 2 also shows the model training results of our proposed framework and Jetson configurations. We include two sets of results, one per epoch and one per frame. The latency per epoch for *Jetson-1* and *Jetson-2* are measured as 1.37 and 1.34 seconds, whereas *PHR-1* and *PHR-2* achieve 0.14 and 0.09 seconds, respectively. *PHR-2* shows a higher speedup of 14.0× and 13.7× against *Jetson-1* and *Jetson-2* configurations. The energy consumption of *Jetson-1* and *Jetson-2* configurations are both 13.4 J. *PHR-1* and *PHR-2* achieve 0.43 J and 0.41 J, respectively, which results in an improvement of 31.3× and 31.2× for *PHR-1* and 32.4× and 32.3× against *Jetson-1* and *Jetson-2*, respectively. The energy consumption reduction of training is lower than that of inference. The reason is the high DRAM access required in the training and the gradient calculation performed in the IMC accelerator's SRAM block. SRAM IMC consumes higher energy than the ReRAM crossbar array but because of the write endurance problem

Table 4.3: Hardware results for *RGB-CNN* inference on Jetson Xavier NX with 6 CPU cores and our framework with 256 $\times$ 256 crossbars.

	Jetson	**PHR**	**Improvement** ($\times$)
Power (W)	14.5	0.4	**34.8**
Time per frame (ms)	622.2	9.7	**64.1**
Energy per frame (J)	9.1	0.05	**782**

seen in ReRAM crossbar arrays, writing too much data on ReRAM is not practical. *PHR-1* achieves an improvement of 3.3$\times$ and 3.4$\times$ against *Jetson-1* and *Jetson-2* configurations, respectively while *PHR-2* achieves an improvement of 32.3$\times$ and 2.4$\times$ against *Jetson-1* and *Jetson-2* configurations, respectively.

Our framework also utilizes HRNet-W32 as the *RGB-CNN* model for the ground truth joint coordinate generation. Table 4.3 compares the results on the Nvidia Jetson board to the proposed framework. The energy consumption per frame is 9.1 J with a latency of 622.2 ms for Jetson, while they are 0.05 J and 9.7 ms for PHR. The energy consumption and latency of *RGB-CNN* inference are higher than *mmWave-CNN* model because HRNet-W32 is a deeper network with densely connected layers. It requires more DRAM accesses for data movement as all the network weights do not fit in the accelerator. Despite this, it can provide sufficient performance to process at runtime with over 30 FPS.

4.3.5 Hardware Architecture Exploration

Crossbar Array Size: We next analyze the effect of the crossbar sizes in each tile. Previous sections considered only 128 $\times$ 128 and 256 $\times$ 256 array sizes. When we reduce the size to 64 $\times$ 64, the inference latency (356.7 μs) becomes higher than

the Jetson board latency, making the configuration impractical. At the same time, an array size larger than 256 $\times$ 256 reduces the utilization to less than 50% and also increases the area overhead. Therefore, we decided to use only *PHR-1* and *PHR-2* configurations which have 128 $\times$ 128 and 256 $\times$ 256 crossbar array sizes in our evaluations.

Nonlinear Properties of ReRAM: Our analysis assumes a C2C variation of 2%, D2D variation of 0.1%, and nonlinearity of 0.5/-0.5 following [49]. An ideal case would have no variations and nonlinearities. Fig. 4.4 compares the PA-MPJPE with the selected nonlinear properties and the ideal case for user *Set-2*. On average, we see an improvement of 24.3% and 29.8% against the baseline model for *Nonlinear* and *Ideal* cases, respectively. The difference between the cases varies between 3.1% and 8.1%, with an average of 5.5% for ten subjects. Note that even with the nonlinear

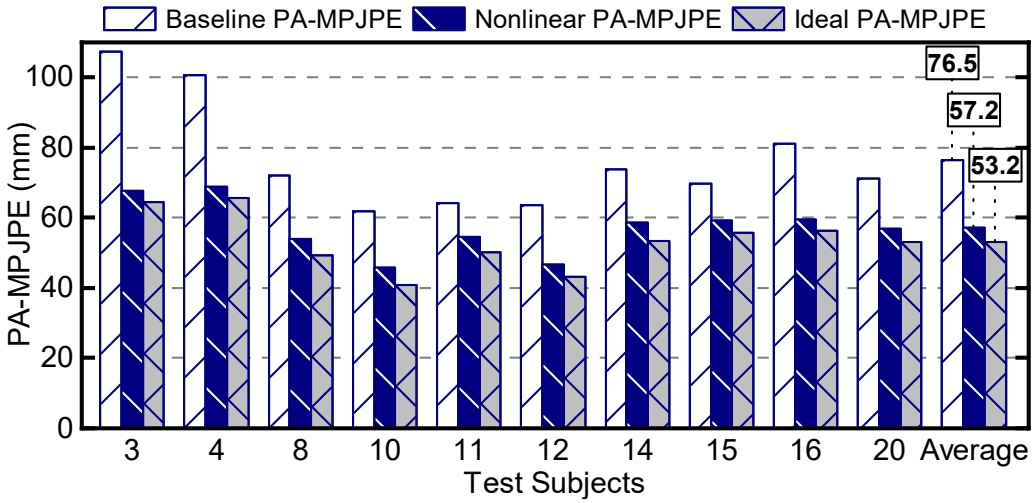

Figure 4.4: PA-MPJPE comparisons for the baseline model (*Baseline*), a customized model with nonlinear properties (*Nonlinear*), and a customized model without nonlinear properties (*Ideal*) for 10 test subjects from *Set-2*.

properties of ReRAM, we can achieve significant improvements compared to the baseline model after the customization.

5 COMMUNICATION-AWARE SPARSE NEURAL NETWORK OPTIMIZATION

5.1 Background, Motivation, and Contributions

Deep neural networks (DNNs) exhibit a high degree of redundancy due to dense interconnections between successive layers. Besides posing overfitting risks, redundant connections increase the communication cost and implementation overhead, thus leading to lower performance and energy efficiency when implemented in hardware. Indeed, many pruning techniques aim at removing DNN connections with minimal impact on their accuracy [150, 65, 68]. Sparse neural networks are preferred since they can enable minimal communication and implementation overhead, thus significantly reducing the computation and memory requirements.

Sparse inter-layer connections enable significantly faster and more energy-efficient DNNs. However, sparsity alone is *not* sufficient since good algorithmic performance *does not* necessarily translate into real performance on hardware. For instance, some inter-layer connections can lead to long paths when mapped on hardware. Consequently, they can undermine the overall hardware performance due to high communication latency and energy costs. For example, the sparse evolutionary training (SET) approach [58] drastically decreases the training time using sparse graphs instead of pruning a trained network; this makes the training scalable, while improving the test accuracy on a wide range of datasets, including multi-layer perceptron (MLP) and convolutional neural networks (CNNs) for

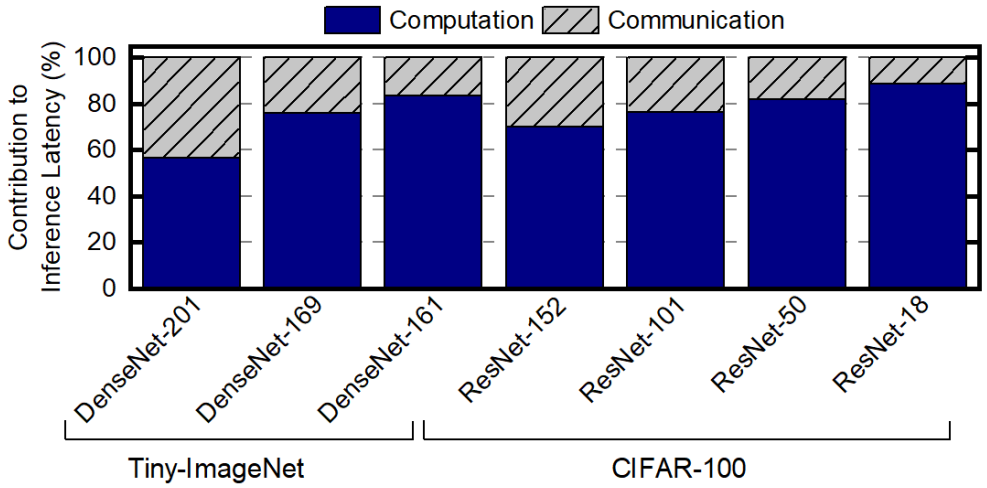

Figure 5.1: Percentage contribution to inference latency for various networks on two datasets. The communication latency can take up to 43% of the total inference latency.

unsupervised and supervised learning.

Although it can achieve higher accuracies, the networks remain oblivious to the real hardware. The performance of DNNs on real hardware is critical since it determines the inference latency and power consumption. For example, a DNN targeting real-time applications, such as autonomous driving, may become impractical if the inference latency violates the timing constraints. To analyze the inference latency, we perform experiments using an in-memory computing (IMC)-based DNN accelerator where the inter-layer communication for activation data movement is implemented via a network-on-chip (NoC). We use a state-of-the-art reinforcement learning based mapping algorithm [6] with unpruned networks using two different datasets. Our evaluations show that the communication between the DNN layers alone can take up to 43% of the total inference latency for a wide range of DNNs,

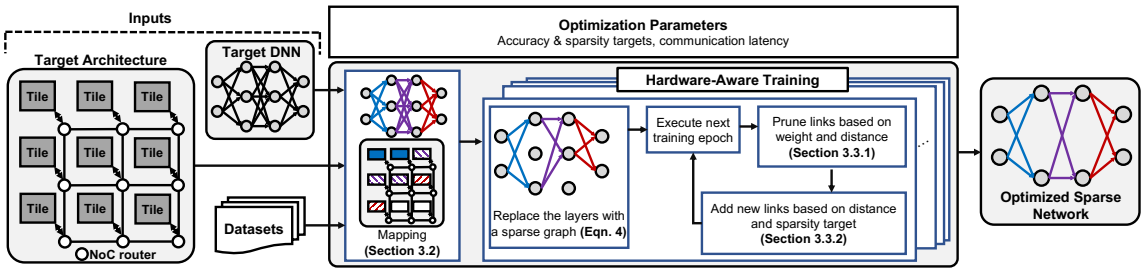

Figure 5.2: Overview of the proposed approach. It consists of mapping the target DNN onto the target architecture using latency-aware mapping and hardware-aware dynamic sparse training. The training process first replaces the DNN layers with sparse graphs; then, at the end of each epoch, employs hardware-aware pruning and link addition. Each circle in the target DNN represents the feature map of DNNs; each link in the target DNN represents the weights of DNNs. The weights are mapped onto the in-memory computing (IMC) tiles with the same color as the corresponding links. The circles and the rectangles in the target architecture denote the NoC routers and IMC tiles, respectively.

as depicted in Figure 5.1.

Although sparse training can achieve a higher accuracy than a network with no pruned links [2], if the network remains oblivious to the target hardware while pruning and adding links, then, this can lead to unacceptable latency and power overheads. Therefore, there is a strong need for *hardware and communication-aware sparse training* methodologies that can lead to shorter communication distances when the DNN layers are mapped onto hardware resources.

Starting from these observations, we present CANNON, a novel **c**ommunication-**a**ware sparse **n**eural **n**etwork **o**ptimizatio**n** technique applicable to both fully connected and convolutional layers. The first step of the proposed technique maps the DNN layers on the target hardware resources, e.g., to the processing tiles of an NoC. Our proposed mapping technique minimizes the distance the packets

between two consecutive DNN layers need to travel in the NoC which helps reducing the overall communication latency. The second step performs hardware-aware dynamic sparse training. Suppose two nodes in the DNN are connected by links with non-zero weights. If these DNN nodes are mapped onto different tiles on the NoC, the activations generated between these nodes will incur communication costs during inference. The proposed technique prunes the p−percent of the weights based on the significance of weight columns towards any inference decision per unit communication cost. Finally, we maintain the target sparsity throughout the training process by choosing an equal number of weight columns (p−percent) with the smallest communication cost and adding them back to the network at the end of each epoch.

We evaluate CANNON exhaustively using well-established simulators (NeuroSim [151], BookSim [152]), popular DNN structures (ResNet [153], DenseNet [154], VGG-16 [155], MLP), and datasets (Tiny-ImageNet [156], CIFAR-100, CIFAR-10 [157], MNIST [158]). The hardware performance of the sparse neural networks with our proposed technique is also compared against state-of-the-art pruning techniques [58, 2]. Our hardware-optimized sparse neural networks result in up to 6.8× improvement in the energy-delay product (with respect to the neural networks with pruning and state-of-the-art mapping techniques), without any significant accuracy degradation.

The major contributions of this chapter are as follows:

- A latency-aware mapping technique which minimizes the distance packets between DNN layers travel between processing elements using an NoC;

- A hardware-aware pruning technique using a newly proposed *z-index* that guarantees sparsity while further reducing the communication latency;
- Extensive experimental evaluations showing 1.6×–3.1× latency and 2.7×–6.8× energy-delay product improvements compared to state-of-the-art pruning techniques without a significant accuracy loss.

The rest of the chapter is organized as follows. The proposed technique is described in detail in Section 5.2. The experimental results are presented in Section 5.3.

5.2 Methodology

In this section, we first overview the proposed approach. Then, we present our latency-aware mapping and hardware-aware dynamic sparse training in detail. Finally, we illustrate the evolution of the *z*-index.

5.2.1 Overview of CANNON

The inputs to our framework are the target DNN, the datasets, and the hardware architecture, as shown in Figure 5.3. Our goal is to prune the DNN to meet a given sparsity target which maintains the accuracy of the DNN, while minimizing communication latency on the target hardware. To this end, we first map the DNN nodes to the target hardware as described in Section 5.2.2. Then, we perform the newly proposed hardware-aware training. At the beginning of training, we replace the layers of the DNN with a sparse graph. Then, we perform hardware-aware

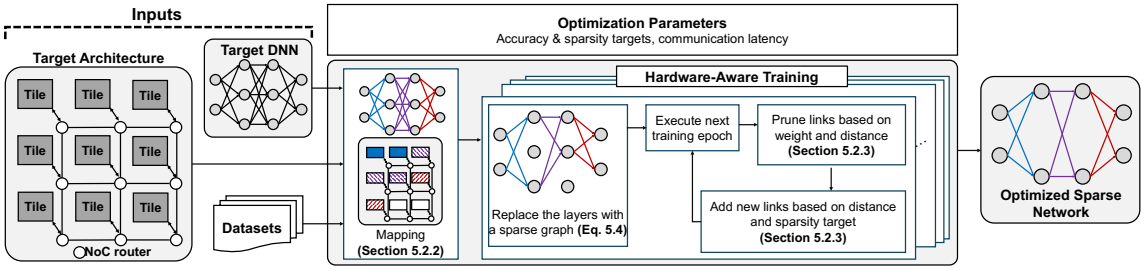

Figure 5.3: Overview of the proposed approach, CANNON. It consists of mapping the target DNN onto the target architecture using the latency-aware mapping and hardware-aware dynamic sparse training. The training process first replaces the DNN layers with sparse graphs; then, at the end of each epoch, employs hardware-aware pruning and link addition. Each circle in the target DNN represents the feature map of DNNs; each link in the target DNN represents the weights of DNNs. The weights are mapped onto the in-memory computing (IMC) tiles with the same color as the corresponding links. The circles and the rectangles in the target architecture denote the NoC routers and IMC tiles, respectively.

pruning, as well as hardware-aware link addition during each epoch as described in sections 5.2.3 and 5.2.3, respectively. At the end of the training process, we obtain a hardware-aware sparse network and its mapping onto a mesh NoC.

5.2.2 Latency-Aware Mapping

The first step of CANNON is to map the nodes of the target DNN onto the target hardware, which we assume is an in-memory computing (IMC)-based DNN accelerator. We consider IMC-based DNN accelerators since they integrate computation with memory, and decrease the latency and energy cost of memory accesses. Several recent research have proposed energy-efficient DNN accelerators with IMC technology [126, 127, 5, 50]. IMC accelerators integrate multiple processing elements (known as tiles) into the system, as shown in Figure 5.4. The number of tiles

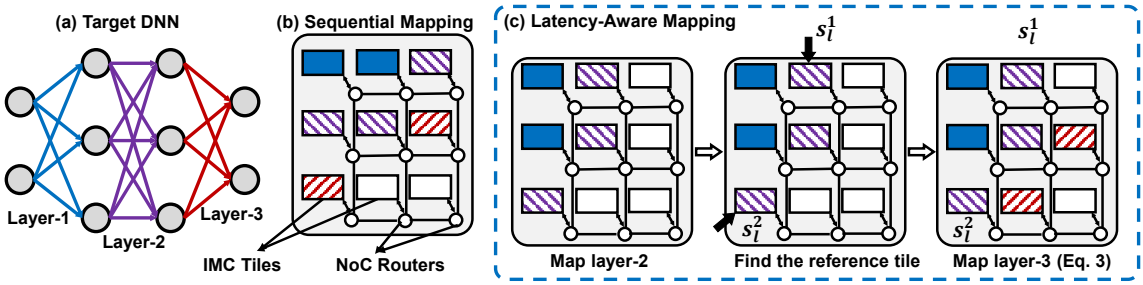

Figure 5.4: Illustration of latency-aware mapping algorithm. (a) the target DNN to be mapped, (b) sequential mapping [5] maps the layers of the target DNN to the tiles of NoC sequentially using the number of tiles required for each layer, (c) latency-aware mapping algorithm in CANNON first finds the reference tiles (s_l^1, s_l^2) of previously mapped layers and then maps the next layer by minimizing the total distance to the reference tiles. **All rectangular boxes in (b) and (c) represent IMC tiles that contain processing elements while the small circles represent NoC routers.**

in the system is a function of neural network parameters, as well as the hardware parameters. In this work, the workload is divided into tiles following the technique described in [151]. Each tile, placed on a grid, consists of memory elements that store the DNN weights and perform computations. Finally, the tiles are connected to NoC routers which facilitate the data exchanges between different neural network layers. The CANNON framework maps the DNN into the IMC accelerator layer-by-layer. To this end, we analyze the traffic between any two consecutive layers of the neural network to map their nodes. We note that the communication latency between two consecutive network layers is mainly determined by the position of tiles the nodes in these layers are mapped to. Figure 5.4 shows an illustrative example of mapping a target DNN (Figure 5.4(a)) onto tiles. The sequential mapping places the DNN nodes onto the IMC tiles in order, from left to right and from top to bottom [5]. With this mapping style, some of the packets may need to travel long

distances. For example, in Figure 5.4(b), there is communication between the tile in the top right corner to the tile in the bottom left corner.

In contrast to prior work [5], we compute the distance between any s (source) and d (destination) tiles before mapping as $M(s, d)$:

$$M(s, d) = |x_s - x_d| + |y_s - y_d| \tag{5.1}$$

where x_s (or x_d) denotes the physical x-coordinate of the tile-s (or tile-d) and y_s (or y_d) denotes the physical y-coordinate of the tile-s (or tile-d). Then, to ensure that the DNN nodes in two consecutive layers are not physically far apart from each other, we minimize the maximum distance between the tiles. Hence, Equation 5.2 represents the objective of our mapping technique:

$$\text{minimize} \quad \max_{i,j} M(s_i, d_j), \;\; 1 \leqslant i \leqslant T_l, 1 \leqslant j \leqslant T_{l+1}, \tag{5.2}$$

where, T_l denotes the number of tiles in the l^{th} layer.

For a system with K tiles, the complexity to solve Equation 5.2 is $O(T_{l+1}(K - T_{l+1}))$. To reduce this complexity, we consider a pair of source tiles (s_l^1, s_l^2) that are physically farthest from each other, instead of considering all the source tiles (tiles corresponding to the l^{th} layer in Equation 5.2).

Figure 5.4(c) illustrates the proposed latency-aware mapping used in CANNON. Assume that there are K IMC tiles, and each tile is connected to an NoC router. At the beginning of the mapping process, we map the first layer of the neural network to the IMC tiles closest to the input of the accelerator (the blue tiles in Figure 5.4(c)).

Since T_l denotes the number of tiles required for l^{th} layer, the remaining layers that are mapped after l^{th} layer are $R_l = K - \sum_{i=l+1}^{L} T_i$, where L is the total number of DNN layers. To map the $(l+1)^{th}$ layer, we calculate the total distance from each of the remaining tiles (R_l) to s_l^1 and s_l^2 (highlighted with solid arrows in Figure 5.4(c)) following Equation 5.1. Then, the tile that minimizes the sum of the distance to the reference tiles is selected. Specifically, the position of this tile is:

$$\underset{j}{argmin}(M(s_l^1, d_j) + M(s_l^2, d_j)), 1 \leqslant j \leqslant R_l \tag{5.3}$$

The above process is repeated for *each* layer until all the layers are mapped. We note that our mapping algorithm always finds a solution if $\sum_{l=1}^{L} T_l \leqslant K$.

We note that our proposed mapping technique is independent of the traffic volume exchanged between different DNN layers. Specifically, we consider only two consecutive layers at a time. We also note that the proposed mapping technique takes care of all existing types of connections in a neural network, namely linear, skip, and dense connections. Therefore, our proposed mapping technique is applicable to *any DNN*, irrespective of traffic volume and connection pattern.

5.2.3 Hardware-Aware Dynamic Sparse Training

Hardware-Aware Pruning

Our hardware-aware pruning approach uses the Sparse Evolutionary Training, i.e., SET, technique as a baseline to guarantee sparse connections between each layer in the training process [58]. It follows a phenomenon observed in complex

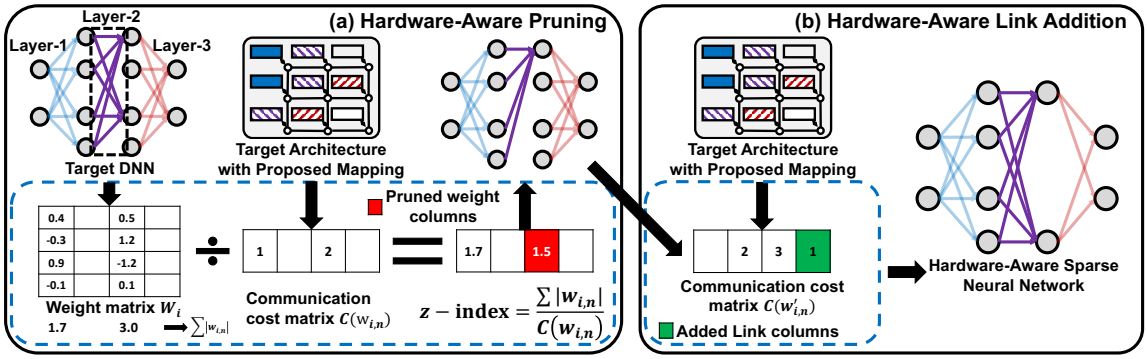

Figure 5.5: Proposed hardware-aware pruning and link-addition. (a) In hardware-aware pruning, z-index of each weight column is computed based on the absolute value and the communication cost. Then, the p-percentage of weight columns with the lowest z-index is removed. (b) The communication cost of already pruned weight columns is calculated in the hardware-aware link addition step. Then, the p-percentage of weight columns with the lowest communication cost is added back. This procedure is repeated every epoch for each layer. The numbers in weight and communication cost matrices are shown for illustration purposes.

real-world networks such as protein interaction networks. This phenomenon shows that starting with an Erdös–Rényi random graph [159] and following its natural evolution, the network reaches a point that has a more structured connectivity resembling scale-free [160] or small-world [161] networks. Inspired by this phenomenon, at the beginning of the training process, the fully connected layers in the DNN are replaced by sparsely connected layers following Erdös–Rényi random sparse graphs as in Equation 5.4:

$$r(\mathcal{W}_i) = \frac{\epsilon(n_i + n_{i-1})}{n_i n_{i-1}} \tag{5.4}$$

where $\mathcal{W}_i$ represents the set of weights in layer-i, where $1 \leqslant i \leqslant L$ and L represents the number of layers in the target DNN. n_i and n_{i-1} represent the number of weights

in layer-i and layer-$(i-1)$, respectively. $\epsilon \in \mathbb{R}^+$ is a tunable parameter used to adjust for the target sparsity level [58], while $r(\mathcal{W}_i)$ represents the probability that each weight in the set is retained. Our hardware-aware pruning approach is very general, as it works for both fully connected (FC) and convolutional (Conv) layers. For layer-i, we define the depth of the input feature channels as D_i, the kernel depth as N_i, and the kernel size as K_i. We can organize the weights of the FC and Conv layers as matrices of sizes $D_i \times N_i$ and $K_i \times K_i \times D_i \times N_i$, respectively. Therefore, we can represent the n^{th} weight column of layer-i as $w_{i,n} \in W_i$ where $1 \leqslant n \leqslant N_i$. Each column of the weight matrix ($D_i \times 1$ for FC, $K_i \times K_i \times D_i \times 1$ for Conv layer) is mapped to a corresponding column of the crossbar array. During execution, each input feature map block (the same size as the weight matrix column) is multiplied with each crossbar column, generating output feature map blocks ($1 \times N_i$). By working at the level of individual columns in the weight matrix, our approach results in pruned columns rather than producing an unstructured sparse weight matrix.

Our goal at the end of each training epoch is to prune the DNN weights that incur a significant communication latency without losing significant accuracy, as illustrated in Figure 5.5. Each weight column of the DNN generates activation(s) communicated between any two DNN layers. Therefore, pruning weights implicitly leads to removing some communication between any two DNN layers.

Communication Cost: The contribution of a weight column n for layer-i ($w_{i,n} \in \mathcal{W}_i$) to the communication can be quantified by the *average Manhattan distance* ($M(s, d)$) between the source tile (s) and the destination tile (d) of the correspond-

ing activations:

$$C(w_{i,n}) = \frac{\sum_{j=1}^{A_{i,n}} M(s_i^j, d_i^j)}{A_{i,n}} \tag{5.5}$$

where $A_{i,n}$ is the number of activations generated by the corresponding weight column $w_{i,n}$ and $M(s, d)$ is given in Equation 5.1. However, the significance of weights also depends on their total absolute value. Thus, pruning weights should be based on both weights' absolute value and corresponding communication cost due to their activation.

z-index: z-index is the ratio between the total absolute value of a weight column and its communication cost:

$$z(w_{i,n}) = \frac{\sum(|w_{i,n}|)}{C(w_{i,n})} \tag{5.6}$$

The z-index reflects the *importance* of a weight column in terms of both absolute value and the communication cost, as defined in Equation 5.6. Note that, a weight column with a higher total absolute value will have a higher z-index, while a weight column with higher communication cost will have a lower z-index. Hence, the z-index can be interpreted as the *weight's significance* per unit of communication cost. Therefore, we use the z-index of each weight column to determine whether to prune it. Specifically, we sort all weight columns in layer-i based on their z-index values and prune the smallest p-percentage of weight columns, where p represents the target pruning ratio.

Hardware-Aware Link Addition

After pruning weights during each epoch, an equal number of weights are added to the DNN to maintain the initial pruning ratio. In SET [58], these weights are added randomly. In contrast, *we add new weights considering the communication cost* ($C(w'_{i,n})$), where $w'_{i,n} \in \mathcal{W}'_{i,n}$ and $\mathcal{W}'_{i,n}$ denotes the set of already pruned weight columns from layer-i. Then, we sort $C(w'_{i,n})$ and add p-percentage of weights in $\mathcal{W}'_{i,n}$ with the lowest communication cost as shown in Figure 5.5(b). This way, the performance gains from latency-aware mapping and hardware-aware pruning are preserved by considering the communication cost in the link addition process. As a result, we retain the same number of weights throughout the training process.

5.2.4 Evolution of the Average z-Index

In this section, we analyze the evolution of the average z-index of all weights in the DNN during training. This analysis provides invaluable insights into how the proposed hardware-aware pruning and hardware-aware link addition techniques work. We use the ResNet-50 network on the CIFAR-100 dataset for illustration, but we note that all models and datasets considered in our experimental evaluations show similar trends.

The accuracy and z-index variations for each epoch in the training are shown in Figure 5.6(a) and Figure 5.6(b), respectively. The average z-index grows similar to the accuracy for two reasons. First, the weights (the numerator of the z-index) that are preserved are larger than those pruned in each epoch. Second, hardware-aware pruning and weight addition favor smaller communication costs (the denominator

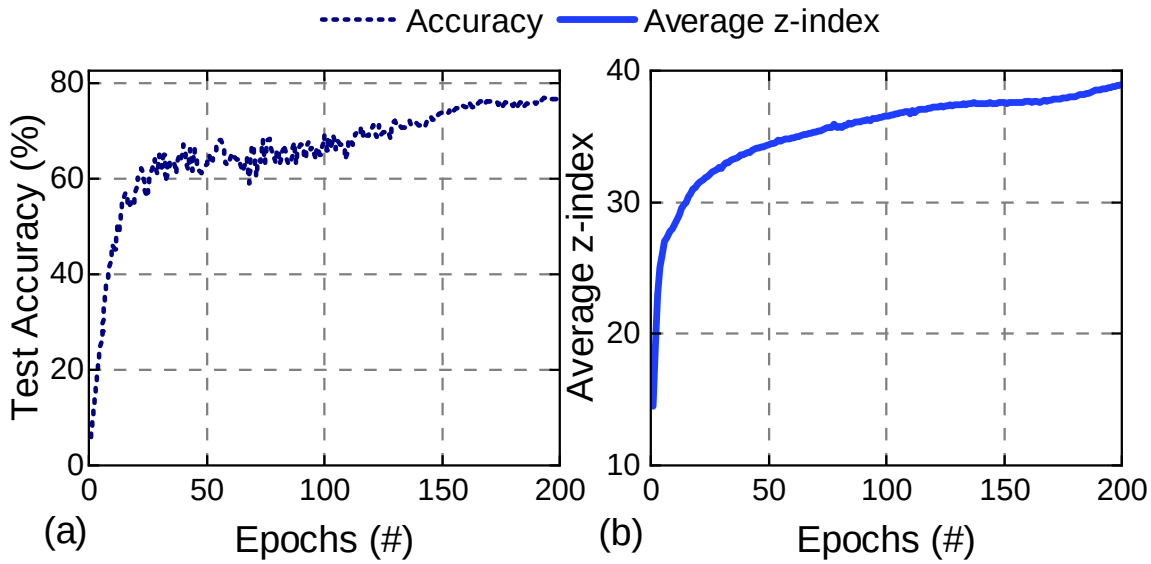

Figure 5.6: Illustration of the evolution of the (a) test accuracy and (b) the average z-index of all weight columns during training for ResNet-50 on the CIFAR-100 dataset. The accuracy, as well as the average z-index, increases throughout the training. Increasing z-index denotes a DNN with larger weights and lower communication latency due to our hardware-aware dynamic sparse training approach.

of the z-index). Indeed, Figure 5.6(b) confirms that the average z-index grows rapidly during training and follows a similar trend to the classification accuracy. Specifically, the average z-index is small at the beginning of the training. Our proposed hardware-aware pruning process removes the weights with lower absolute value and that causes higher communication costs during training. Therefore, the average z-index increases over time. At the end of Epoch-200 (when the accuracy stabilizes), the average z-index is $3.2\times$ larger than the average z-index in the beginning. The growth seen in the average z-index indicates a DNN with lower communication latency. Hence, it improves the hardware performance, as discussed in the following section.

5.3 Experimental Evaluation

This section first introduces our experimental setup to enable reproducible results. Next, we demonstrate the effectiveness of our proposed latency-aware mapping and hardware-aware dynamic sparse training approaches in terms of the number of hops distribution in NoC. Then, we compare the hardware performance of CANNON in terms of latency, energy, energy-delay product (EDP), and accuracy against state-of-the-art techniques. After that, we perform an ablation study of CANNON. Finally, we compare our results against Lottery Ticket Hypothesis using networks from ResNet family on the CIFAR-10 dataset.

5.3.1 Experimental Setup

Network Models and Datasets: We evaluate CANNON with 14 well-known network models and four datasets. Specifically, we use DenseNet-201, DenseNet-169, and DenseNet-161 from the DenseNet family with the Tiny-ImageNet dataset. Similarly, we employ ResNet-152, ResNet-101, ResNet-50, ResNet-18 and VGG-16 with CIFAR-100 and CIFAR-10 datasets. DenseNet, ResNet, and VGG-16 networks consist of convolutional and fully connected layers. We also discuss performance results using an MLP-based network on the MNIST dataset. We use the same MLP-based network structure used in *SET* [58] which consists of three hidden layers with 1000 neurons.

Simulation Setup: The experiments run on a machine that uses 6 Intel Xeon Gold 6242R cores and 1 Nvidia 3090 GPU on Python using the PyTorch library. All CNNs

Table 5.1: Summary of circuit level and NoC parameters

Circuit		**NoC**	
PE array size	256×256	Bus width	32
Cell levels	2 bit/cell	Routing algorithm	X–Y
Flash ADC resolution	8 bits	Number of router ports	5
Technology used	RRAM	Topology	Mesh

are trained for 200 epochs with the SGD optimizer and a momentum of 0.9. We set the initial learning rate as 0.1 and use the Cosine Annealing scheduling as the learning rate scheduler. The communication performance is evaluated using a cycle-accurate NoC simulator, BookSim [152]. To this end, we use a customized version of BookSim that supports simulation with workload traces. As different networks show different structures, we generate traces for each network and dataset pairs. Each trace consists of three entries: source router, destination router, and timestamp for the generated packet. We generate a trace file for each layer and feed this to BookSim to measure the communication performance. The number of IMC tiles required for each layer of the neural network is evaluated through NeuroSim [151]. In the IMC tile structure, adopted from [5], there are 16 PEs; each PE consists of a 256×256 IMC crossbar. We design separate accelerators to show the effectiveness of our hardware-aware mapping and training approaches on each dataset and network model and assume that the accelerator is big enough to accommodate the network as in [67]. In this work, we do not consider a pipelined architecture since such architectures may result in pipeline stalls [162]. Moreover, pipelining incurs an extra area overhead due to the extra control logic required. Therefore,

Table 5.2: Properties of different approaches with respect to the mapping method, Fully Connected (FC) layer pruning, and Convolutional (CONV) layer pruning.

Approach	Mapping	FC Pruning	CONV Pruning
Mapping Only [6]	RL	X	X
SET [58]	sequential [5]	weight-based	X
SET [58] + *Mapping*	latency-aware	weight-based	X
***CANNON* (*FC Layers*)**	latency-aware	hardware-aware	X
***CANNON* (*FC+CONV Layers*)**	latency-aware	hardware-aware	hardware-aware

our experimental evaluations report the overall end-to-end communication latency and energy when there is no layer-to-layer pipelining. Table 5.1 shows different hardware parameters incorporated in our evaluations.

Baseline Approaches: We compare CANNON against multiple baseline approaches summarized in Table 5.2. The first baseline technique is a RL-based mapping algorithm [6] working on an unpruned network (*Mapping Only* in Table 5.2). This baseline is added to the comparison set to assess the impact of the mapping algorithm alone. SET [58] is utilized in two baselines. The first one uses SET with a widely-used mapping algorithm [5] (*SET* in Table 5.2). In contrast, the second one uses SET with our proposed latency-aware mapping to show the performance of our proposed mapping method (*SET + Mapping* in Table 5.2). Two variants of the CANNON framework are evaluated in these experiments. We selected SET technique as the baseline pruning approach in the training. However, we emphasize that CANNON can be used in conjunction with any other pruning technique including state-of-the-art structured pruning techniques [60, 68, 25, 70]. We first show the results of our proposed dynamic sparse training when applied to the fully

connected layers of neural networks (*CANNON* (*FC Layers*) in Table 5.2) since the approach proposed in SET prunes FC layers only.

Finally, we compare all the baselines against CANNON applied to both fully connected and convolutional layers (last row in Table 5.2). The pruning ratios in these experiments are 50% for the convolutional layers and 80% for the fully connected layers for ResNet, DenseNet, and VGG-16 networks selected based on the highest pruning value without seeing an important accuracy drop. For MLP-based networks, we use a pruning ratio of 95% to have a fair comparison against *SET*. The pruning and addition ratios in the hardware-aware dynamic sparse training are selected as 30% of the remaining weights. To quantify the overhead of the z-index calculations during training, we also compare CANNON against a random pruning scheme using identical pruning ratios. Our measurements indicate that CANNON has only 7% higher training time per epoch for the ResNet-18 model on the CIFAR-100 dataset. Once the training completes, CANNON does not introduce any additional inference overhead.

5.3.2 Number of Hops Distribution

CANNON aims at minimizing the communication latency by minimizing the maximum Manhattan distance the packets travel between two consecutive DNN layers (see Eqn. 5.2). Therefore, CANNON decreases the number of hops the packets travel compared to the state-of-the-art mapping technique [6]. Figure 5.7 compares the probability distribution of the number of hops the packets travel in the NoC for the DenseNet-201 model on Tiny-ImageNet in our proposed mapping

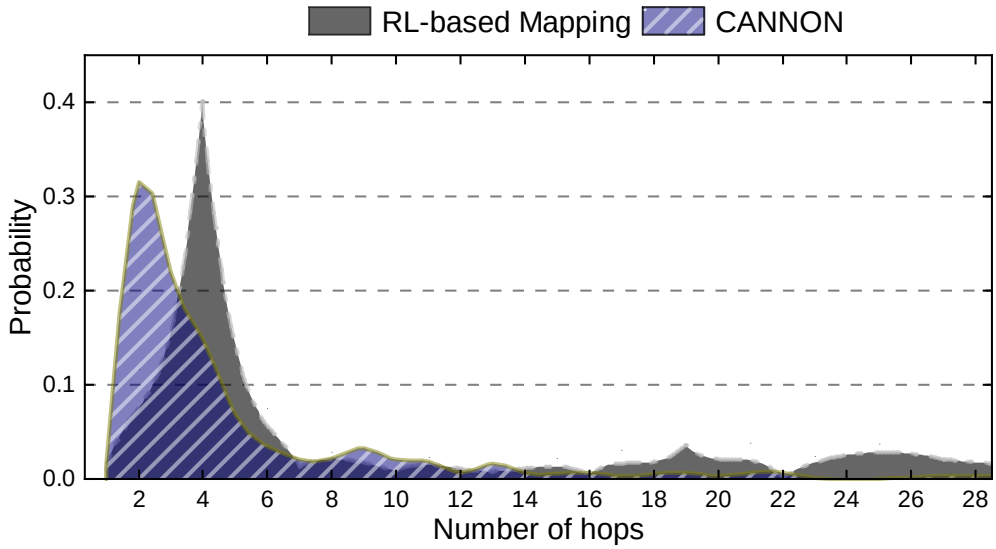

Figure 5.7: Comparison of **hop distribution** between RL-based mapping [6] and CANNON. Latency-aware mapping minimizes number of hops probability toward smaller values.

technique and the technique described in [6]. Using the probability distributions, we observe that 86% of the packets need to traverse more than 3 hops in the NoC with the RL-based mapping approach. In contrast, the portion of the packets with more than 3 hops reduces to 45% with our proposed mapping technique.

The probability distribution for the number of hops for ResNet-152 model on CIFAR-100 and DenseNet-201 model on Tiny-ImageNet are shown in Figure 5.8(a) and Figure 5.8(b), respectively. These figures compare the distributions at the beginning of the training (Epoch-0), at the end of Epoch-100, and at the end of training (Epoch-200). We observe that the probability of achieving a smaller number of hops increases with the number of epochs for the ResNet-152 model on CIFAR-100. Many long-range communications disappear as the training progresses due to our proposed hardware-aware pruning technique. We also analyze the

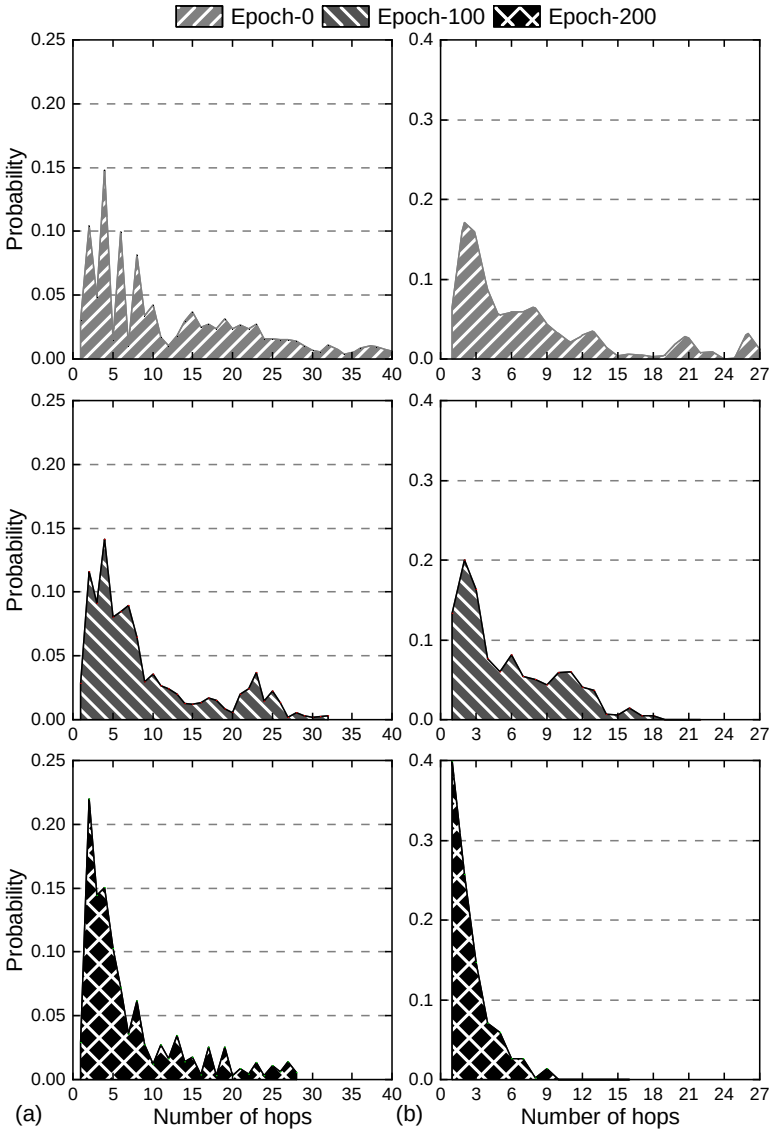

Figure 5.8: Comparison of **hop distribution** for hardware-aware training before Epoch-0, at Epoch-100, and at the end of Epoch-200 for (a) ResNet-152 on the CIFAR-100 and (b) for DenseNet-201 on the Tiny-ImageNet dataset. The distribution of the number of hops shifts toward smaller numbers as we move forward during the training. There are still weights with higher hops at the end of Epoch-200 because the hardware-aware pruning considers communication cost and whether or not the weight is significant.

percentage of weights distributed among different numbers of hops using probability distributions. Specifically, 44% of the packets need to traverse less than six hops at the beginning of Epoch-0; with our proposed hardware-aware dynamic sparse training, at the end of Epoch-200, 71% of the packets need to traverse less than six hops, as shown in Figure 5.8(a). However, we note that some long-range connections still remain because our hardware-aware pruning considers not only the communication cost but also the weight value. If there are weights with large values, we tend not to prune them even if they produce higher communication cost. Overall, the average number of hops decreases by 28% by the end of Epoch-100 and 42% by the end of Epoch-200 compared to Epoch-0. We observe a similar trend for the DenseNet-201 model on Tiny-ImageNet, where the probability of lower number of hops increases as the training progresses, as shown in Figure 5.8(b). The average number of hops decreases by 25% at the end of Epoch-100 and 70% at the end of Epoch-200. This is important because it translates into improved latency and energy efficiency.

5.3.3 Latency Analysis

This section compares the communication latency of CANNON against the four baseline techniques introduced earlier. The properties of each baseline are presented in Table 5.2. We use 14 widely-known network models listed in Section 5.3.1.

Experiments for convolutional neural networks are performed using DenseNet models on the Tiny-ImageNet dataset, ResNet and VGG-16 models using the CIFAR-100 and CIFAR-10 datasets, and an MLP using the MNIST dataset (Figure 5.9).

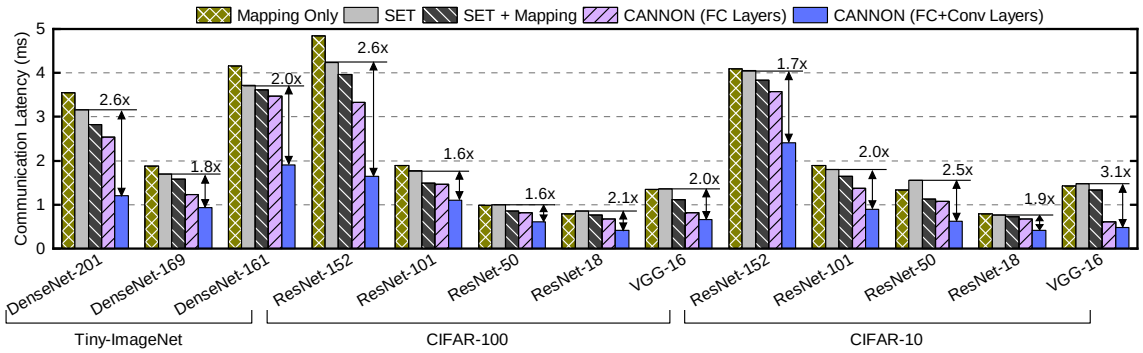

Figure 5.9: Comparison of communication **latency** across different DNNs and datasets. CANNON consistently improves the latency for all network models and datasets.

The plot highlights the improvements of CANNON over the *SET* approach since a comparison against a pruned network is more appropriate. **DenseNet Network Results:** Reinforcement Learning (RL)-based mapping approach using an unpruned DenseNet-201 network model on the Tiny-ImageNet dataset shows 3.55ms communication latency. Using *SET* [58] with the mapping in [5], the communication latency is improved to 3.16ms. The decrease is mainly due to the lack of some weight columns because of high pruning ratio. Our proposed latency-aware mapping employed with *SET* improves the communication latency by 20% compared to the RL mapping approach on an unpruned network for the DenseNet-201 model. The only difference between *SET* and *SET+Mapping* approaches is due to the mapping algorithm. Thus, the direct comparison between them demonstrates the effectiveness of our proposed latency-aware mapping. The improvement of *SET+Mapping* compared to *SET* ranges between 3% to 12% for the DenseNet models. *CANNON* (*FC Layers*) shows a speedup between 1.1×–1.4× compared to *SET*. Finally, *CANNON* (*FC+Conv Layers*) shows a higher speedup varying between 1.8× to 2.6× and

2.0× to 3.0× compared to *SET* and *Mapping Only* approaches for all networks in the DenseNet family, respectively. Considering the savings of computation, the improvements for total inference latency are up to 57% for DenseNet.

ResNet Network Results: The number of output channels of the convolutional layers of ResNet DNNs is higher than that of DenseNet DNNs. Therefore, the communication latency of ResNet DNNs is higher than DenseNet DNNs. The *Mapping Only* approach, which uses RL-based mapping on an unpruned network, shows the highest communication latency in most ResNet networks using the CIFAR-100 and CIFAR-10 datasets. The RL-based mapping performs better than the *SET* approach with ResNet-50 and ResNet-18 networks on CIFAR-100 and ResNet-50 on CIFAR-10 datasets. The *SET + Mapping* approach, which uses our proposed latency-aware mapping but pruning as proposed in SET, consistently outperforms *SET* and *Mapping Only* techniques. *CANNON* (*FC Layers*) further improves the latency between 1.1× to 1.2× compared to the *SET + Mapping* approach on the CIFAR-100 dataset. Finally, our proposed *CANNON* (*FC+Conv Layers*) shows a speedup between 1.6× to 2.6× compared to *SET* on CIFAR-100, as illustrated in Figure 5.9. On the CIFAR-10 dataset, the improvements are in the range of 1.7× to 2.2× compared against the *Mapping Only* approach. Communication latency improvements result in a 23%–51% drop in total inference latency for ResNet DNNs.

VGG-16 and MLP Results: We also show the results with VGG-16 network with CIFAR-100 and CIFAR-10 datasets in Figure 5.9. *CANNON* (*FC+Conv Layers*) shows a speedup of up to 3.1× for VGG-16 with negligible accuracy impact. Finally, experiments with an MLP on MNIST dataset obtain 2.4× lower communication

latency than *SET* (6.6 µs vs 3.7 µs) with similar accuracy. These results are excluded from Figure 5.9 for readability since the latency for MLP is three orders of magnitude smaller than other networks.

5.3.4 Energy and EDP Analysis

In this section, we compare the communication energy consumption of CANNON against four different approaches. The details of the communication energy results for DenseNet, ResNet, and VGG-16 models on the Tiny-ImageNet, CIFAR-100, and CIFAR-10 datasets are given in Figure 5.10. We present detailed energy-delay product (EDP) results in Figure 5.11.

DenseNet Network Results: The *Mapping Only* approach shows the highest communication energy for DenseNet-201 and DenseNet-161 networks. The *SET + Mapping* approach, where our latency-aware mapping is utilized together with the pruning technique in SET, performs better than the *SET* and *Mapping Only*

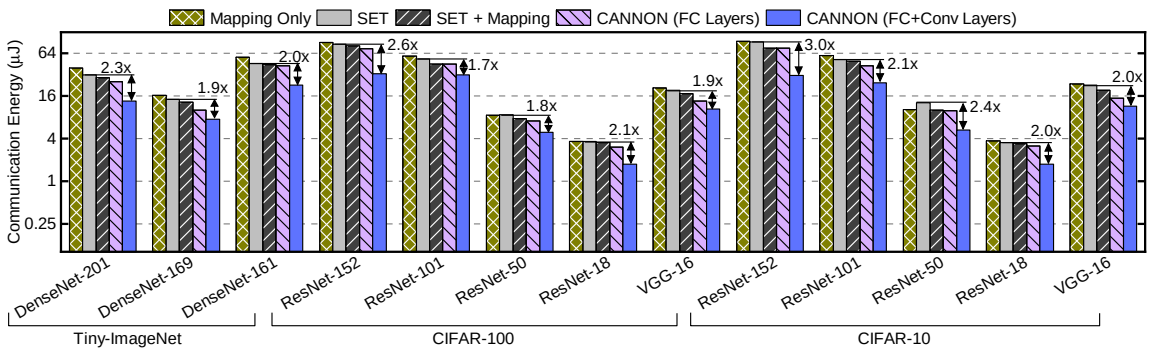

Figure 5.10: Comparison of communication **energy** (log scale) across different DNNs and datasets. CANNON consistently improves the communication energy for all network models and datasets.

approaches. In this case, the gain against the *SET* approach shows that our latency-aware mapping reduces energy consumption. Furthermore, we observe a 10%–22% reduction in communication energy consumption with *CANNON* (*FC Layers*) against *SET* + *Mapping*. Finally, *CANNON* (*FC+Conv Layers*) consistently outperforms all other approaches. Specifically, we observe 48%-58% reduction in communication energy compared to *SET*.

The communication latency and energy reduction also result in significant EDP improvements. Since the *Mapping Only* approach does not consider pruning, it has the highest EDP among all techniques for all DenseNet networks on the Tiny-ImageNet dataset. The *SET* + *Mapping* approach has 7%–20% improvement with respect to the *SET* approach. *CANNON* (*FC Layers*) further improves the EDP by 9%–39%. Furthermore, *CANNON* (*FC+Conv Layers*) outperforms all other approaches consistently and shows an EDP improvement of 3.5×–6.2× compared to *SET* (Figure 5.11).

ResNet Network Results: The ResNet results include four DNNs and two datasets.

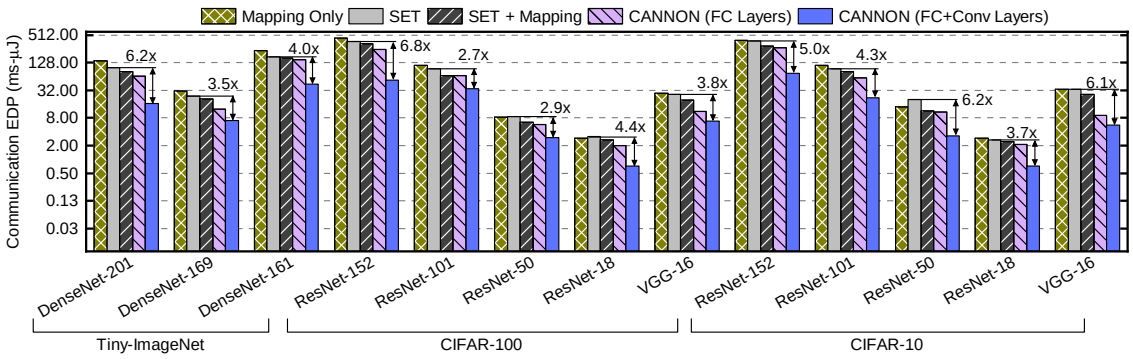

Figure 5.11: Comparison of communication **EDP** (log scale) across different DNNs and datasets. As shown, CANNON consistently improves the communication EDP for all network models and datasets.

The *Mapping Only* approach has the highest communication energy among all techniques except for the ResNet-50 model on both datasets. The improvements of *SET + Mapping* over *Mapping Only* are 4%–21% and 5%–22% on CIFAR-100 and CIFAR-10 datasets, respectively. CANNON consumes the least communication energy compared to all other approaches. While *CANNON* (*FC Layers*) has 1.2×–1.3× improvement over *SET*, *CANNON* (*FC+Conv Layers*) has an improvement of 1.7×–2.6× with respect to *SET* on the CIFAR-100 dataset. On the CIFAR-10 dataset, *CANNON* (*FC+Conv Layers*) further improves the communication energy up to 3.0× compared to *SET*, as shown in Figure 5.10.

Similar to energy consumption, CANNON achieves significant improvements in EDP too with respect to all other approaches. The improvement varies between 2.7×–6.8× and 3.7×–6.2× on CIFAR-100 and CIFAR-10 datasets, respectively, when compared to *SET*, as shown in Figure 5.11.

VGG-16 and MLP Results: We show the energy and EDP results with VGG-16 on CIFAR-100 and CIFAR-10 datasets in Figure 5.10 and Figure 5.11, respectively. *CANNON* (*FC+Conv Layers*) achieves 1.9× and 2.0× communication energy savings with respect to *SET* for VGG-16 on CIFAR-100 and CIFAR-10 datasets, respectively. EDP improvements are even higher with 3.8× and 6.1×. Finally, we performed experiments with MLP using MNIST dataset. *CANNON* (*FC+Conv Layers*) achieves 1.9× lower energy consumption and 4.4× lower EDP than *SET* (105.7 nJ vs 195.8 nJ) without any accuracy impact. MLP results are not shown in Figure 5.10 and 5.11 for readability. These savings offer huge advantages for edge devices as they have limited energy budgets.

Table 5.3: Accuracy (%) comparison across different models and datasets. DN and RN denote DenseNet and ResNet, respectively.

Dataset	**Tiny-ImageNet (%)**			**CIFAR-100 (%)**					**CIFAR-10 (%)**					**MNIST (%)**
Network	**DN201**	**DN169**	**DN161**	**RN152**	**RN101**	**RN50**	**RN18**	**VGG16**	**RN152**	**RN101**	**RN50**	**RN18**	**VGG16**	**MLP**
Mapping Only [6]	60.15	60.45	62.04	80.26	79.37	78.52	77.82	75.12	90.74	90.14	89.87	84.16	90.70	98.22
SET [58]	61.40	61.53	62.71	78.98	80.04	79.17	78.01	75.78	90.52	91.10	90.55	88.94	91.37	98.68
CANNON (FC Layers)	60.84	61.22	62.28	78.83	79.54	77.80	77.65	74.81	89.16	90.32	89.77	88.23	90.58	98.34
CANNON (FC+CONV Layers)	59.50	58.74	60.95	78.78	78.06	76.90	76.39	74.58	88.88	88.96	87.98	85.66	89.31	NA
Accuracy Difference from [6]	**-0.65**	**-1.71**	**-1.09**	**-1.48**	**-1.31**	**-1.62**	**-1.43**	**-0.54**	**-1.86**	**-1.18**	**-1.89**	**1.50**	**-1.39**	**0.02**

5.3.5 Accuracy Analysis

The previous sections show that CANNON significantly improves the hardware performance and energy consumption efficiency compared to the baseline techniques. In this section, we discuss the test accuracy results of CANNON for all DNNs and datasets. All results are shown in Table 5.3.

DenseNet Network Results: The *Mapping Only* approach [6] using an unpruned DenseNet-201 network model on the Tiny-ImageNet dataset achieves 60.15% test accuracy. As shown in Table 5.3, *CANNON* (*FC Layers*) improves the test accuracy by 0.69% with significant hardware efficiency improvements compared to *Mapping Only*, as discussed in previous sections. Moreover, *CANNON* (*FC Layers*) has 0.56% decrease from the *SET* technique [58]. For DenseNet-169 and DenseNet-161, we also achieve up to 0.77% and 0.24% accuracy improvements over *Mapping Only*, respectively. Table 5.3 shows that *CANNON* (*FC Layers*) yields 0.65% decrease from the unpruned network on DenseNet-201. Similarly, we have only 1.71% and 1.09% accuracy loss over the unpruned networks of DenseNet-169 and DenseNet-161, respectively.

ResNet Network Results: *SET* achieves the highest test accuracy in most ResNet-based networks using CIFAR-100 and CIFAR-10 datasets. The *Mapping Only* ap-

proach has an accuracy difference in the range of -1.28% to 0.96% compared to the *SET* technique, except for ResNet-18 on CIFAR-10. The *Mapping Only* approach has 4.78% lower accuracy than *SET* for ResNet-18 on CIFAR-10. *CANNON* (*FC Layers*) has an accuracy difference as minimal as between -1.43% to 0.17% compared to the *Mapping Only* approach on unpruned networks on the CIFAR-100 dataset. The accuracy improvement ranges from -1.58% to 0.18% compared to the *Mapping Only* approach on the CIFAR-10 dataset, except for ResNet-18. For ResNet-18 on CIFAR-10, the accuracy improvement is up to 4.07%. *CANNON* (*FC+Conv Layers*) shows an accuracy difference between -1.89% to 1.50% compared against the *Mapping Only* approach on both datasets.

We remark that CANNON yields a significant hardware efficiency improvement with less than 2% accuracy loss compared to the unpruned networks. It is then possible to use CANNON with lower pruning ratios to compensate for the accuracy loss.

VGG-16 and MLP Results: As shown in Table 5.3, CANNON achieves almost the same accuracy as *SET* for MLP. Moreover, it shows 0.12% accuracy improvement over *Mapping Only*. For VGG-16, CANNON shows 0.54% and 1.39% accuracy loss for CIFAR-100 and CIFAR-10, respectively.

5.3.6 Ablation Study

In this section, we conduct a new ablation study to examine how hardware awareness impacts pruning. We compare two approaches: *CANNON* (*FC+Conv Layers*) and a communication-agnostic *Baseline* (*FC+Conv Layers*). CANNON prunes the

Table 5.4: *CANNON* (*FC+Conv Layers*) comparison with respect to *Baseline* (*FC+Conv Layers*). Model and dataset combinations are ResNet-18 on CIFAR-10, VGG-16 on CIFAR-100, and DenseNet-169 on Tiny-ImageNet.

Model	**ResNet-18**			**VGG-16**			**DenseNet-169**		
Metric	**Acc. (%)**	**Lat. (ms)**	**EDP (msμJ)**	**Acc. (%)**	**Lat. (ms)**	**EDP (msμJ)**	**Acc. (%)**	**Lat. (ms)**	**EDP (msμJ)**
Baseline (*FC+Conv*)	85.42	0.70	2.32	74.88	1.11	19.14	59.21	1.62	22.70
CANNON (*FC+Conv*)	85.66	0.41	0.72	74.58	0.66	6.85	58.74	0.93	7.04
Improv.	**0.28**	**1.70×**	**3.24×**	**-0.40**	**1.68×**	**2.79×**	**-0.80**	**1.74×**	**3.22×**

network by considering both weight magnitudes and communication costs, whereas *Baseline* (*FC+Conv Layers*) prunes solely based on weight magnitudes without considering communication costs. Both approaches use the same pruning ratios for fairness.

Table 5.4 presents that both approaches achieve comparable accuracy. However, *CANNON* (*FC+Conv Layers*) outperforms *Baseline* (*FC+Conv Layers*) by up to 1.74× higher performance, with a timing improvement from 0.7ms to 0.41ms. Additionally, *CANNON* (*FC+Conv Layers*) results in up to a 3.24× reduction in EDP compared to *Baseline* (*FC+Conv Layers*). These results highlight the importance of hardware-aware training for achieving optimal performance.

5.3.7 Comparison to Lottery Ticket Hypothesis

This section compares the accuracy and performance of CANNON with FC and CONV layer pruning against the Lottery Ticket Hypothesis (i.e., LTH) [2] using the official implementation of LTH from the GitHub repository [163]. We use three

Table 5.5: CANNON comparison with respect to lottery ticket hypothesis [2].

Dataset	ResNet-152				ResNet-101				ResNet-50			
Metric	Accuracy (%)	Latency (ms)	Energy (μJ)	EDP (ms-μJ)	Accuracy (%)	Latency (ms)	Energy (μJ)	EDP (ms-μJ)	Accuracy (%)	Latency (ms)	Energy (μJ)	EDP (ms-μJ)
LTH [2]	91.08	3.48	68.09	237.12	91.82	1.31	42.34	55.55	91.57	1.00	9.08	9.08
LTH + Mapping [2]	91.08	3.45	66.87	230.69	91.82	1.23	38.81	47.82	91.57	0.88	8.44	7.42
CANNON	88.88	2.41	31.30	75.38	88.96	0.90	24.76	22.22	87.98	0.62	5.32	3.28
Improvement over LTH [2]	**-2.20%**	**1.45×**	**2.18×**	**3.15×**	**-2.86%**	**1.46×**	**1.71×**	**2.50×**	**-3.59%**	**1.61×**	**1.70×**	**2.77×**

iterations, each pruning 20% of the network during the training process. In the end, the network of the winning ticket is approximately 49% pruned. We use an equal pruning ratio in CANNON to make the comparison fair. We utilize three networks from the ResNet family with 152, 101, and 50 layers on the CIFAR-10 dataset. We incorporate the mapping method used in [5] (LTH) and our proposed latency-aware mapping (LTH + Mapping) to evaluate the hardware performance of LTH pruning. The accuracy, latency, energy, and EDP results are shown in Table 5.5. Since our proposed technique prunes a DNN considering hardware performance, CANNON consistently outperforms LTH in all three hardware performance metrics with slightly lower accuracy. The latency improvements with CANNON are 1.45×–1.61× compared against LTH with the mapping method used in [5]. We observe such an improvement due to our proposed latency-aware mapping and hardware-aware dynamic sparse training methodologies, which reduces the communication cost without pruning the significant weights. We also observe that the improvements in energy consumption vary between 1.70× – 2.18×. The highest improvements are seen in the energy-delay product (EDP). CANNON achieves up to 3.15× improvement in EDP with respect to the Lottery Ticket Hypothesis with the mapping method used in [5].

As a conclusion, we note that the reduction in network size and the increase in

accuracy due to pruning do *not* necessarily translate into better hardware results unless they consider hardware-aware mapping and pruning methods. Overall, CANNON provides excellent results compared to state-of-the-art pruning methods and can work synergistically with any pruning technique.

6 DAS: DYNAMIC ADAPTIVE SCHEDULING FOR ENERGY-EFFICIENT HETEROGENEOUS SOCS

6.1 Background, Motivation, and Contributions

Heterogeneous systems-on-chip (SoCs) combine the energy efficiency and performance of custom designs with the flexibility benefits of general-purpose cores. Domain-specific SoCs (DSSoCs), a subset of heterogeneous SoCs, are emerging examples that integrate general-purpose compute elements and hardware accelerators that target the commonly encountered tasks (i.e., computational kernels) in the target domain [164, 165, 166, 167]. For example, a DSSoC for autonomous driving incorporates computer vision and deep learning accelerators, while a DSSoC for 5G/6G communication domain accelerates signal processing operations, such as fast Fourier transform (FFT). In addition, general-purpose CPU clusters and programmable accelerators, such as coarse-grained reconfigurable architectures (CGRA), offer alternative execution paths for a broader set of tasks, besides improving flexibility [165, 168].

In contrast to fixed-function designs, a critical distinction of DSSoCs is their ability to run multiple applications from the same domain [169]. When multiple applications run concurrently, the number of ready tasks can exceed the capacity of the available accelerators resulting in resource contention. This resource contention leads to a complex runtime scheduling problem since there are many different ways to prioritize and run such tasks. For example, waiting for the most suitable

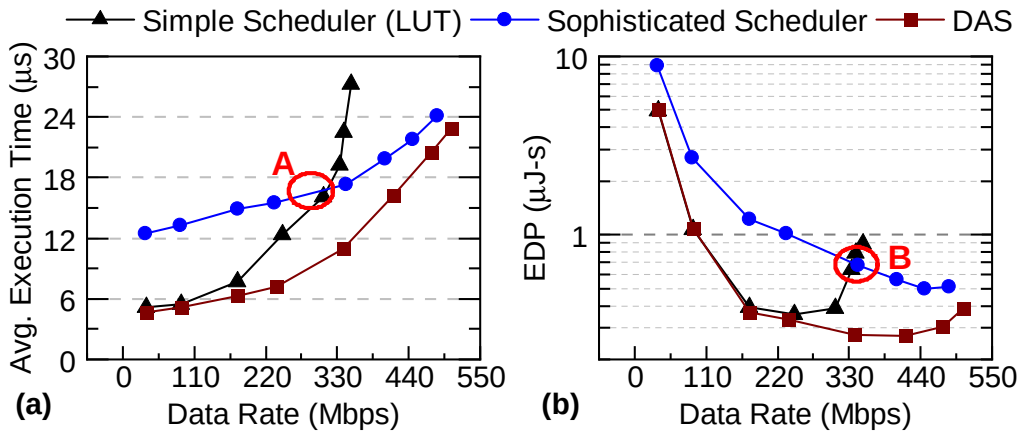

Figure 6.1: An example of the relationship of (**a**) execution time and (**b**) energy-delay product (EDP) between simple low-overhead (lookup table or LUT) and sophisticated high-overhead schedulers.

resource to become available can lead to higher energy efficiency than resorting to an immediately available less suitable resource like a CPU core or a reconfigurable accelerator. Besides the complex runtime scheduling problem, recent studies show that execution times for applications of domain-specific systems are on the nanosecond scale [170, 164]. Therefore, the classical scheduling problem encounters a new challenge in heterogeneous DSSoCs because domain-specific tasks can run in the order of nanoseconds, i.e., at least two or three orders of magnitude faster than general-purpose cores when they are executed on their specialized pipelines. If the scheduler takes a significantly longer amount of time to make a decision, it can undermine the benefits of hardware acceleration. For instance, Linux Completely Fair Scheduler (CFS) takes 1.2µs to make a scheduling decision when running on an Arm Cortex-A53 core [171, 91, 77, 172]. This overhead is clearly unacceptable when there are many tasks with orders of magnitude faster execution times.

DSSoCs require fast scheduling algorithms to keep up with tasks that can run

in the order of nanoseconds and achieve high efficiency. However, poor scheduling decisions of simple low-overhead schedulers can decrease the system performance, especially under heavy workloads. For example, Figure 6.1 shows the average execution time and energy-delay product for a workload mix of a wireless communication system. When the data rate is low (i.e., few frames are present concurrently), there is lower contention for the same SoC resources. Hence, a fast low-overhead scheduler (in this work, a lookup table) outperforms a more sophisticated scheduler (e.g., a complex heuristic or an integer programming-based scheduler) since it can make the same or very similar assignments with a significantly lower overhead. As the data rate increases, the number of concurrent frames, and hence the complexity of scheduling decisions, also grow. At the same time, the waiting times of the tasks in the core queues exceed their actual execution time in the accelerators. Therefore, a sophisticated scheduler outperforms the simple one (point A in Figure 6.1a), and the overhead of making better scheduling decisions pays off. In contrast, when an application involves a high degree of task-level parallelism, we expect more tasks in the ready queue waiting for a scheduling decision. For such applications, the scheduling overhead can dominate the execution time, even for lower data rates (Figure 6.1b). For these cases, using a sophisticated scheduler cannot outperform the scheduling decisions of a simple scheduler. As the data rate increases, a sophisticated scheduler can degrade the system performance because of its dependence on the number of ready tasks. Therefore, a simple scheduler outperforms a sophisticated scheduler under various data rates. Hence, the trade-off between the scheduling decision quality and the scheduling overhead is an opportunity to

exploit.

To exploit the trade-off between the scheduler overhead and decision quality, we propose a *dynamic adaptive scheduling* (DAS) framework that combines the benefits of the sophisticated and simple low-overhead scheduling algorithms using an integrated support mechanism. Making a scheduling decision at the scale of nanoseconds is highly challenging because executing possibly complex decisions and loading the necessary context, such as performance counters, requires extra time. Hence, even the data movement part alone can violate the fast decision target. Moreover, the framework needs to determine at runtime whether the low-overhead or sophisticated scheduler should run. The DAS framework outperforms both underlying schedulers despite these challenges by following these *key observations* in the design process: *First*,, the scheduling will be called with 100% certainty and uses features (a subset of available performance counters). Therefore, prefetching the required features in a background process and writing them to a pre-allocated local memory location can hide the extra overhead. *Second*, the choice of a sophisticated or a simple scheduler can be made by the same prefetching process before the following scheduling process begins. If the simple scheduler is chosen, the only extra overhead on the critical path is the time to access the LUT, which is measured as 6 ns on Arm Cortex-A53. We run the sophisticated scheduler at runtime only if a more complex scheduler is needed.

The main contributions of this chapter are as follows:

- The DAS framework that dynamically combines two schedulers and outperforms each of them with low scheduling overhead;

- Experimental results with five streaming applications and profiling scheduling overheads on Xilinx Zynq ZCU102;
- Integration of the DAS framework with an open-source runtime environment and its training and deployment on Xilinx Zynq ZCU102;
- Extensive performance evaluation in the trade space of execution time, energy, and scheduling overhead over the Xilinx Zynq ZCU102 based on workload scenarios composed of real-life applications;

The rest of the chapter is organized as follows. We describe the proposed DAS framework and algorithms in Section 6.2. Section 6.3 discusses and analyzes the experimental results on a DSSoC simulator with real-world applications, while Section 6.4 presents the training and implementation details of the DAS framework on an FPGA emulation environment using real-world applications.

Table 6.1: Type of performance counters used by DAS framework

Type	**Features**
Task	Task ID, Execution time, Power consumption, Depth of task in DFG, Application ID, Predecessor task ID and cluster IDs, Application type
Processing Element (PE)	Earliest time when PE is ready to execute, Earliest availability time of each cluster, PE utilization, Communication cost
System	Input data rate

6.2 Dynamic Adaptive Scheduling Framework

6.2.1 Overview and Preliminaries

This work considers streaming applications that can be modeled by data flow graphs (DFGs). More precisely, consecutive data frames are pipelined through the tasks in the graph. The current practice of scheduling is limited to a single scheduler policy. On the other hand, DAS allows the OS to choose one scheduling policy $\pi \in \mathbf{\Pi_S} = \{\mathsf{F}, \mathsf{S}\}$, where F and S refer to the *fast* and *slow* (*or sophisticated*) schedulers, respectively. When the task becomes ready (predecessors of the task are completed), the OS can call either a slow ($\pi = \mathsf{S}$) or a fast ($\pi = \mathsf{F}$) scheduler as a function of the workload and system state. The OS collects a set of performance counters during the execution to enable two aspects for the DAS framework: (1) precise assessment of the system state, (2) desirable features for the classifier to switch between the *fast* and *slow* schedulers at runtime.

Table 6.1 shows the various types of performance counters that are collected for the DAS framework. The total number of performance counters is 62 for a DSSoC with 19 PEs. The goal of the slow scheduler S is to handle more complex scenarios when the task wait times dominate the execution times. In contrast, the fast scheduler F aims to approach the theoretically minimum (i.e., zero) scheduling overhead by making decisions in a few cycles with a minimum number of operations. *The DAS framework aims to outperform both underlying schedulers by dynamically switching between them as a function of system state and workload.*

6.2.2 Zero-Delay DAS Preselection Classifier

The first runtime decision of DAS is the selection of the fast or slow scheduler. We should optimize this decision to approach the goal of zero scheduling overhead as it is on the critical path of both schedulers. One of the novel contributions of DAS is that we recognize this selection as a deterministic task that will be eventually executed with a probability of one. Therefore, we prefetch the relevant features that are required for this decision to a pre-allocated local memory. We re-use a subset of the performance counters which are shown in Table 6.1 as desirable features to minimize the scheduling overhead. The relevant subset of features is presented in Section 6.3.2. To reflect the system state at that point in time, the OS periodically refreshes the performance counters. DAS framework runs a light-weight classifier which determines whether the fast or slow scheduler should be used for the next ready task each time the features are refreshed. As it is refreshed with the features that reflect the most recent system state, we note that this decision will always be up to date. Thus, DAS can determine which scheduler should be called even before a task is ready for scheduling. Consequently, the preselection classifier of the DAS framework introduces zero latency and minimal energy overhead. Next, we will discuss the offline preselection classifier and its online use. The latter will cover the overhead involved in switching between policies.

Offline Classifier Design: The first step for the design process of the preselection classifier is to generate the training data based on the domain applications known at design time. Each scenario in the training data consists of concurrent applications and their respective data rates. For example, a combination of WiFi transmitter

and receiver chains, at a specific upload and download speed, could be one such scenario. To this end, we run each scenario *twice* on an instrumented hardware platform or a simulator (see Figure 6.2).

First Execution: The instrumentation enables us to run *both fast and slow schedulers* each time a task scheduling decision is made. We note that the scheduler is invoked whenever a task is to be scheduled. If the decisions of the fast (D_F) and slow (D_S) schedulers for a task T_i are identical, then we label task T_i with F (i.e., the *fast scheduler*) and store a snapshot of the performance counters. The label F implies that the fast scheduler can reach the same scheduling decision as the sophisticated algorithm under the system states captured by the performance counters. If the decisions of the schedulers are different, then the label is left as *pending* and the execution continues by following the decision of the fast scheduler, as described in Figure 6.2. At the end of the execution, the training data contains a mix of both labeled (F) and pending decisions.

Second Execution: The same scenario is executed for the second time. This time the execution always follows the decisions of the slow scheduler. At the end of the execution, we analyze the target metric, such as the average execution time or the total energy consumption. If a better result can be achieved by using the slow scheduler, the pending labels are replaced with S to indicate that the slow scheduler is preferred despite its larger overhead. Otherwise, we conclude that the lower overhead of the fast scheduler pays off, and the pending labels are replaced with F. An alternative approach is to evaluate each pending instance individually; however, this would not offer a scalable solution as scheduling is a sequential

decision-making problem, and a decision at time t_k affects the remaining execution.

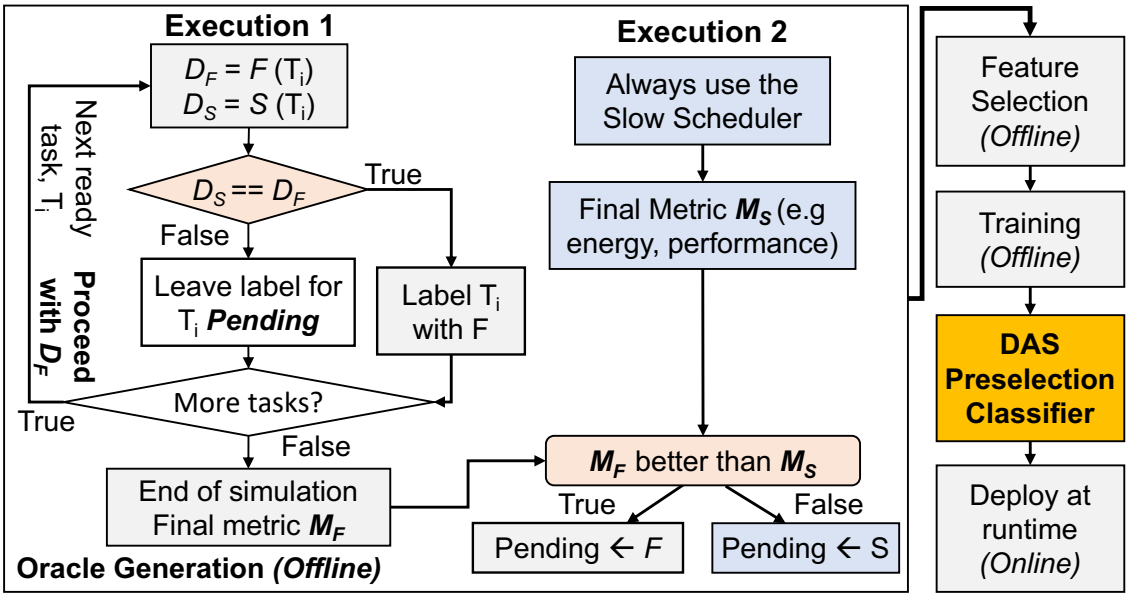

Figure 6.2: Flowchart describing the construction of an Oracle to dynamically choose the best-performing scheduler at runtime.

In this work, the training data is generated using 40 distinct workloads. Each workload is a mix of multiple instances of five applications, consisting of approximately 140,000 tasks in total and executed at 14 different data rates, as detailed in Section 6.3.1. A higher data rate presents a larger number of concurrent applications contending for the same SoC resources. Then, we design a low-overhead classifier using machine learning techniques and feature selection methods [173], as described in Section 6.3.2.

Online Use of the Classifier: The last step for the classifier is the deployment at runtime (last block in Figure 6.2). At runtime, a background process periodically updates a pre-allocated local memory with the subset of performance counters that

the classifier requires for the scheduler selection. After each update, the classifier determines whether the fast *F* or slow *S* scheduler should be used for the next available task. When a new ready task becomes available, the features are already loaded and we know which scheduler is a better choice for the read task. Therefore, DAS framework does not incur any extra delay on the critical path. Moreover, it has a negligible energy overhead, as demonstrated in Section 6.3, which is critical for the performance and applicability of such an algorithm.

6.2.3 Fast & Slow (Sophisticated) (F&S) Schedulers

The proposed DAS framework can work with any choice of fast and slow scheduling algorithms. This work uses a lookup table (LUT) implementation as the fast scheduler as the goal of the fast scheduler is to achieve almost zero overhead. The LUT stores the most energy-efficient processor in the target system as a decision for each known task in the target domain. Unknown tasks are mapped to the next available CPU core. Hence, the LUT access at runtime is the only extra delay on the

Algorithm 4 ETF Scheduler

```
1:  while ready queue 𝒯 is not empty do
2:      for task T_i ∈ 𝒯 do
3:          // 𝒫 = set of PEs
4:          for PE p_j ∈ 𝒫 do
5:              FT_{T_i,p_j} = Compute the finish time of T_i on p_j
6:          end for
7:      end for
8:      (T′, p′) = Find the task & PE pair that has the minimum FT
9:      Assign task T′ to PE p′
10: end while
```

critical path and overhead of the fast scheduler. To profile the scheduling overhead of LUT, we developed an optimized C implementation with inline assembly code. We ran the script 10,000 times and averaged the latency and energy consumption. We used the on-chip TI INA226 power sensors to measure the energy consumption. Our experiments show that *the fast scheduler takes ~7.2 cycles (6 ns on Arm Cortex-A53 at 1.2 GHz) on average, and incurs negligible (2.3 nJ) energy overhead.*

The DAS framework uses a commonly used heuristic, earliest task first (ETF) as the slow scheduler [105]. ETF is chosen since it recursively iterates over the ready queue tasks and processors to provide the schedule that achieves the fastest finish time, as shown in Algorithm 4. It performs a comprehensive search, which can make the best decision when the system is loaded with many tasks. It starts from the first task-T_i in the ready queue-$\mathcal{T}$. It computes the finish time of task-T_i on each PE-p_j. Then, it continues this process for every task-PE pair (T_i-p_j). Then, it selects the task-PE pair (let's say T'-p') that gives the earliest finish time. After assigning task-T' to PE-p', the process repeats with the remaining tasks in the ready queue-$\mathcal{T}$. Hence, its computational complexity is quadratic with respect to the number of ready tasks. ETF's detailed computational and energy overhead are presented in Section 6.3.1.

6.3 Evaluation of DAS Using Simulations

Section 6.3.1 describes the experimental setup used in this work. Section 6.3.2 explores different machine learning methods and features for DAS. The evaluation

Table 6.2: Characteristics of applications from radar processing and wireless communication domains used in this study. (FFT = fast Fourier transform, FEC = forward error correction, FIR = finite impulse response, SAP = systolic array processor).

Application	Number of Tasks	Supported Clusters
Range Detection	7	big, LITTLE, FFT, SAP
Temporal Mitigation	10	big, LITTLE, FIR, SAP
WiFi-TX	27	big, LITTLE, FFT, SAP
WiFi-RX	34	big, LITTLE, FFT, FEC, FIR, SAP
App-1	10	LITTLE, FIR, SAP

and detailed analysis of DAS for different workloads is shown in Section 6.3.3. Finally, we demonstrate the latency and energy consumption overheads of DAS in Section 6.3.4.

6.3.1 Experimental Setup

Domain Applications: The DAS framework is evaluated using five real-world streaming applications: range detection, temporal mitigation, WiFi-transmitter, WiFi-receiver applications and a proprietary industrial application (*App-1*), as summarized in Table 6.2. We then construct 40 different workloads for runtime analysis of the schedulers used in this work. All workloads are run in streaming mode, and for each data point in the figures in Section 6.3.3, approximately 10,000 tasks are scheduled.

Emulation Environment: Conducting a realistic runtime overhead and energy analysis is one of our key goals in this study. For this purpose, we leverage a Compiler-integrated, Extensible DSSoC Runtime (CEDR) framework introduced

by Mack et al. [168]. The CEDR has been validated extensively on x86 and Arm-based platforms. It enables pre-silicon performance evaluations of heterogeneous hardware configurations composed of mixtures of CPU cores and accelerators based on dynamically arriving workload scenarios. Compared to the other emulation frameworks (e.g., ZeBu [174] and Veloce [175]), this portable and open-source environment offers distinct plug-and-play integration points where developers can individually integrate and evaluate their applications, scheduling heuristics, and accelerator IPs in a realistic system.

The emulation framework [176] combines compile-time analysis with the run-time system. The compilation process involves converting each application to LLVM intermediate representation (IR), identifying what section of the code should be labeled as "kernels" (frequently executing IR-level blocks) or "non-kernels", and automatically refactoring the LLVM IR into a directed acyclic graph (DAG), where each node represents a kernel. By abstracting the application with a DAG, compile-time flow generates a flexible binary structure for the run-time system to be able to invoke each function call on all its supported processing elements (PEs) in the target architecture. The runtime system monitors the state of system resources, parses dynamically the arriving applications, generates the queue of tasks that have resolved dependencies (called ready queue), schedules the ready queue tasks based on the user-defined scheduling policy and manages the data transfers to and from the PEs. The run-time system supports scheduling heuristics, such as Round Robin, Earliest Finish Time (EFT), and Earliest Task First (ETF). We selected ETF scheduler as the complex scheduler of our analysis because of its complexity

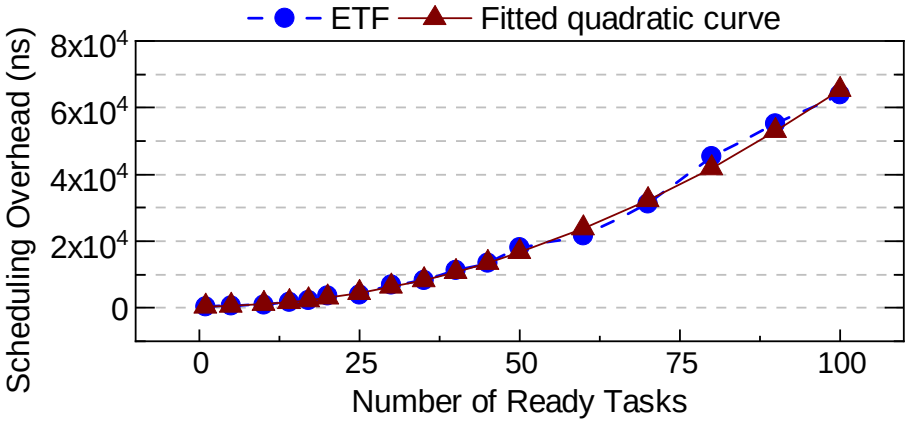

Figure 6.3: ETF scheduling overhead and fitted quadratic curve.

and quality decisions that it provides. As shown in Algorithm 4, ETF recursively iterates over the ready queue tasks and PEs with the objective of finding the task to PE mapping decisions that result with the minimum finish time. It does this process recursively until each ready task is scheduled to a PE. We generate a wide range of workloads—ranging from all application instances belonging to a single application to a uniform distribution from all five applications to evaluate the impact of the scheduling overhead of the ETF scheduler. We present the impact of the number of tasks in the ready queue on the scheduling overhead of ETF in Figure 6.3, as measured on the Xilinx Zynq ZCU102 [177]. The x-axis in the plot shows the number of tasks that are in the ready queue for each workload scenario, and the y-axis shows the scheduling overhead of the ETF scheduler. We generate a quadratic equation to formulate the ETF scheduling overhead observed at runtime based on these measurements. Later, we utilize this equation to evaluate the average execution time and the EDP of the DAS scheduler on the simulator. We note that the DAS framework is by no means limited to the processors, schedulers,

and applications that are used for demonstration purposes.

Simulation Environment: We use DS3 [178], an open-source domain-specific system-on-chip simulation framework, to perform detailed evaluations of our proposed scheduling approach. DS3 is a high-level simulation tool that includes built-in scheduling algorithms, models for PEs, interconnect, and memory systems. The framework has been validated with Xilinx Zynq ZCU102 and Odroid-XU3 (with a Samsung Exynos 5422 SoC) platforms. Therefore, it is a robust and reliable tool to perform detailed exploration and evaluation of our proposed scheduling approach. DS3 supports the execution of streaming applications (represented as directed flow graphs). It allows multiple modes of injecting new applications at run-time, which include a fixed-interval injection and an exponential distribution-based job injection. The inputs to the tool are the execution profiles of the processing elements in the SoC, application DFGs, and the interconnect configuration. At the end of the simulation, DS3 provides various workload statistics, including the injection rate, execution time, throughput, and PE utilization. We use these metrics provided by DS3 in our extensive evaluation to demonstrate the benefits of the DAS scheduling methodology.

DSSoC Configuration: We construct a DSSoC configuration that comprises clusters of general-purpose cores and hardware accelerators. The hardware accelerators include fixed-function designs, and a multi-function systolic array processor (SAP). The application domains used in this study are wireless communications and radar systems, and hence, accelerators that expedite the execution of relevant tasks are included.

Table 6.3: DSSoC configuration used for DAS evaluation

Processing Cluster	No. of Cores	Functionality
LITTLE	4	General-purpose
big	4	General-purpose
FFT	4	Acceleration of FFT
FEC	1	Acceleration of encoding and decoding operations
FIR	4	Acceleration of FIR
SAP	2	Multi-function acceleration
TOTAL	**19**	

The DSSoC used in our experiments includes the Arm big.LITTLE architecture with 4 cores each. We also include dedicated accelerators for fast Fourier transform (FFT), forward error correction (FEC), finite impulse response (FIR), and SAP. We include 4 cores each for the FFT and FIR accelerators, one core for the FEC and two cores of the SAP. The FEC accelerator accelerates the execution of encoder and decoder operations while the SAP accelerator accelerates multiple tasks for the application domain. In total, the DSSoC integrates 19 PEs with a mesh-based network-on-chip (NoC) to enable efficient on-chip data movement. This configuration is summarized in Table 6.3.

DSSoC Simulator Workload Mixes: Figure 6.4 presents the workload mixes used in the DSSoC simulator. Each workload is a mix of multiple instances of five applications, consisting of 100 jobs (approximately 140,000 tasks) in total. Each workload is executed at 14 different data rates. The stacked bar for each workload denotes the number of instances of each type of application. For example, Workload-6 (WKL-6) uses 50 instances of WiFi-Transmitter and 50 instances of App-1, and Workload-31 (WKL-31) uses 20 instances of each type. The distribution is selected

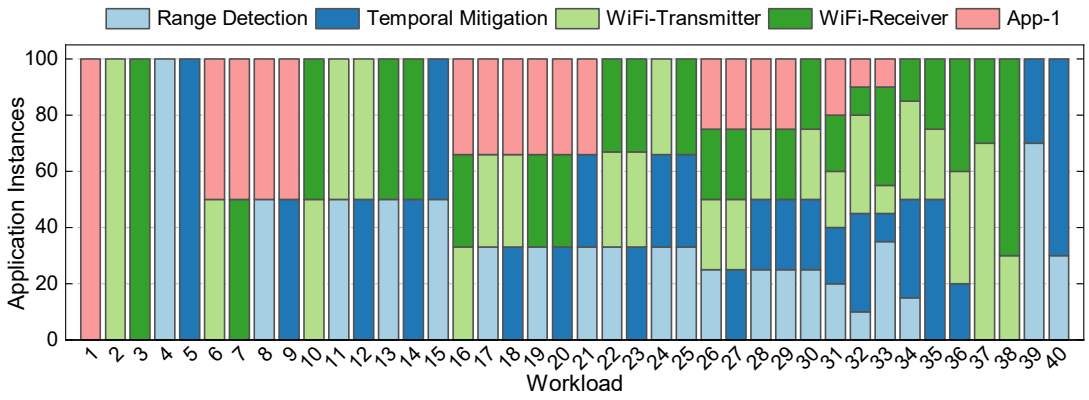

Figure 6.4: The distribution of number and type of application instances in the 40 workloads used for evaluation of the DAS framework on the DSSoC simulator.

to represent a variety of low, medium, and high data rates. These workload mixes are used for all the experiments in Section 6.3.

Performance Metrics: We use average execution time per application instance as our main performance metric. This metric is calculated by dividing the time each application instance is executed to the number of application instances. Another metric for performance comparison is average job slowdown. It is calculated as the average execution time of a scheduler divided by the average execution time of the idealized version of the ETF scheduler to show the distance of scheduler performance's from an ideal scenario. Furthermore, we use the energy-delay product (EDP) metric to analyze the effects on the energy consumption. EDP is calculated as the multiplication of the total execution time (makespan) and energy consumption.

6.3.2 Exploration of ML Techniques and Feature Space for DAS

We train DAS models using various machine learning methods. As we target a model with a very-low scheduling overhead, our analysis on the method selection considers *classification accuracy* and *model size* as main metrics. Specifically, we investigated support vector classifiers (SVC), decision tree (DT), multi-layer perceptron (MLP), and logistic regression (LR). The training process with SVCs with simple kernels exceeded 24 h, rendering it infeasible. The latency and storage requirements of the MLP (one hidden layer and 16 neurons) did not fit the budgets of low-overhead requirements. Therefore, these two techniques are excluded from the rest of the analysis. Table 6.4 summarizes the classification accuracy and storage overheads for the logistic regression and decision tree classifiers as a function of the number of features and depth of tree for the decision trees.

Machine Learning Technique Exploration: DT classifiers achieve similar or higher accuracies compared to LR classifiers with lower storage overheads. While a DT with a depth of 16 which uses all the features achieves the best classification accuracy, there is a significant impact on the storage overhead, which, in turn, affects the latency and energy consumption of the classifier. In comparison, DTs with tree depths of 2 and 4 have negligible storage overheads with competitive accuracies ($> 85\%$). Hence, we adopt the DT classifier with depth 2 for the DAS framework.

Feature Space Exploration: We collect 62 performance counters in our training data. Selecting a subset of these counters as the DAS classifier features is crucial to minimize the energy and performance overheads. A systematic feature space exploration is performed using feature selection and importance methods such as

Table 6.4: Classification accuracies and storage overhead of DAS models with different machine learning classifiers and features

Classifier	Tree Depth	Number of Features	Classification Accuracy (%)	Storage (KB)
LR	-	2	79.23	0.01
LR	-	62	83.1	0.24
DT	2	**1**	**63.66**	0.01
DT	2	**2**	**85.48**	0.01
DT	4	6	85.51	0.03
DT	16	62	91.65	256

analysis of variance (ANOVA), F-value and Chi-squared stats [173]. Among the top six features, growing the feature list from a single feature (*input data rate*) to two features with the addition of *earliest availability time of the Arm big cluster* increases the accuracy from 63.66% to 85.48%. We use an 8-entry$\times$16-bit shift register to track the data rate at runtime. Therefore, we selected the two most important features: data rate and the earliest availability time of the Arm big cluster to design the DAS classifier model with a decision tree of depth 2. We note that these two features don't contain task-related information. Hence, this allows DAS to be compatible with diverse task scenarios without incurring additional overheads since it is not on the critical path.

6.3.3 Performance Analysis for Different Workloads

This section compares the DAS framework with LUT (*fast*), ETF (*sophisticated*), and ETF-ideal schedulers. ETF-ideal is a version of the ETF scheduler which ignores the scheduling overhead component. Therefore, ETF-ideal is an idealized version of a sophisticated scheduler that helps us establish the theoretical upper bound

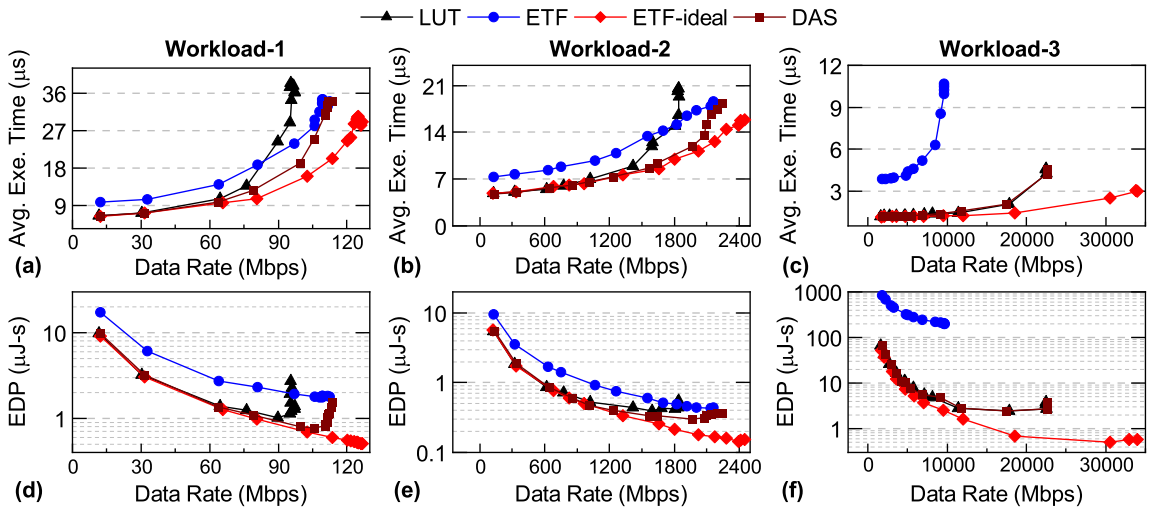

Figure 6.5: Comparison of (a)–(c) average execution time and (d)–(f) EDP between DAS, LUT, ETF, and ETF-ideal for three different workloads.

of the achievable execution time and EDP. Out of the 40 workloads described in Section 6.2.2, three representative workloads are chosen for a detailed analysis of execution time and EDP trends. The three chosen workloads present different data rates, which are a function of the applications in the workload. Workload-1 (Figure 6.5a,d) presents low data rate (complex applications), Workload-2 (Figure 6.5b,e) presents moderate data rates, and Workload-3 (Figure 6.5c,f) represents a high data rate workload (simplest applications).

Figure 6.5a–c compare the execution times of DAS, LUT, ETF, and ETF-ideal schedulers for three different workloads, while Figure 6.5d–f show their corresponding EDP trends. For Workload-1 and Workload-2, the system is not congested at low data rates. Hence, the performance of the DAS is similar to the LUT, as expected. As data rates increase, DAS aptly chooses between LUT and ETF at runtime to achieve an execution time and EDP that is 14%, 15% better than LUT, and 15%, 42% better

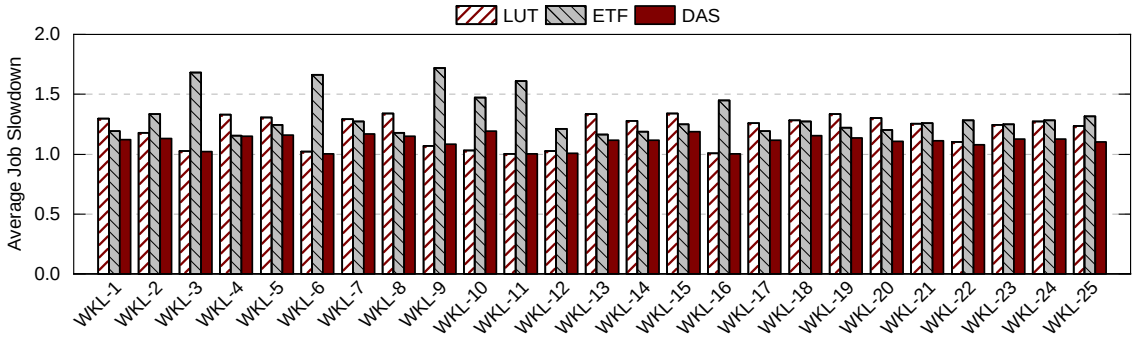

Figure 6.6: A comparison of average job slowdown of between DAS, LUT, and ETF for twenty-five workloads.

than ETF when taken individually. For workload-3, the execution time and EDP of LUT are significantly lower than ETF. DAS chooses LUT for >99% of the decisions and closely follows its execution time and EDP. These results successfully demonstrate that DAS adapts to the workloads at runtime and aptly chooses between LUT and ETF schedulers to achieve low execution time and EDP.

The same study is extended to all 40 different workloads. DAS *consistently outperforms* both LUT and ETF schedulers when they are used individually. At low data rates, i.e., when LUT is better than ETF, the DAS framework achieves more than 1.29× speed up and 45% lower EDP compared to ETF while outperforming the LUT scheduler. Moreover, the DAS framework achieves by as much as 1.28× speed up and 37% lower EDP than LUT when the workload complexity increases. In summary, DAS is always significantly better than either one of the underlying schedulers. Figure 6.6 summarizes the impact of change in workload composition on DAS performance using 25 workloads. The first three workloads are App-1 intensive, workloads 4 to 8 are WiFi intensive, and the last 11 are different mixes of applications. Average job slowdown is normalized to ETF-ideal scheduler results.

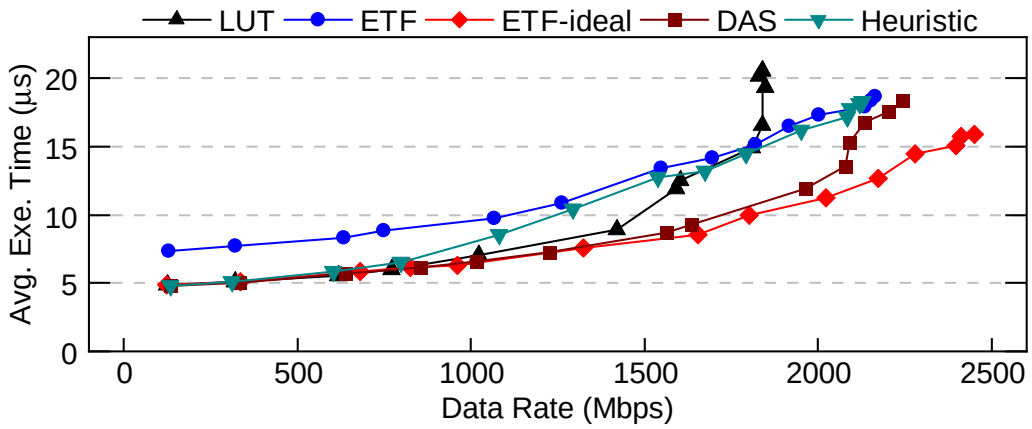

Figure 6.7: A comparison of average execution time between DAS, LUT, ETF, ETF-ideal, and the heuristic approach.

The plot shows that DAS performs better than LUT and ETF schedulers for these different scenarios, bringing the slowdown to 1. It shows that DAS moves the execution time closer to the ideal scenario where the overhead of the scheduler is zero.

We also compare DAS against a heuristic approach. This approach selects the fast scheduler when the data rate is lower than a predetermined threshold and the slow scheduler when the data rate is higher than the threshold. The predetermined threshold is selected based on the analysis of the training data used for DAS. The simulation result for this comparison for Workload-2 of Figure 6.5 is given in Figure 6.7. Results show that the heuristic approach closely mimics the behavior of LUT and ETF schedulers below and above the threshold, respectively, without exhibiting intelligent adaptability. In contrast, DAS outperforms both schedulers, achieving a 13% reduction in execution time compared to the heuristic scheduler.

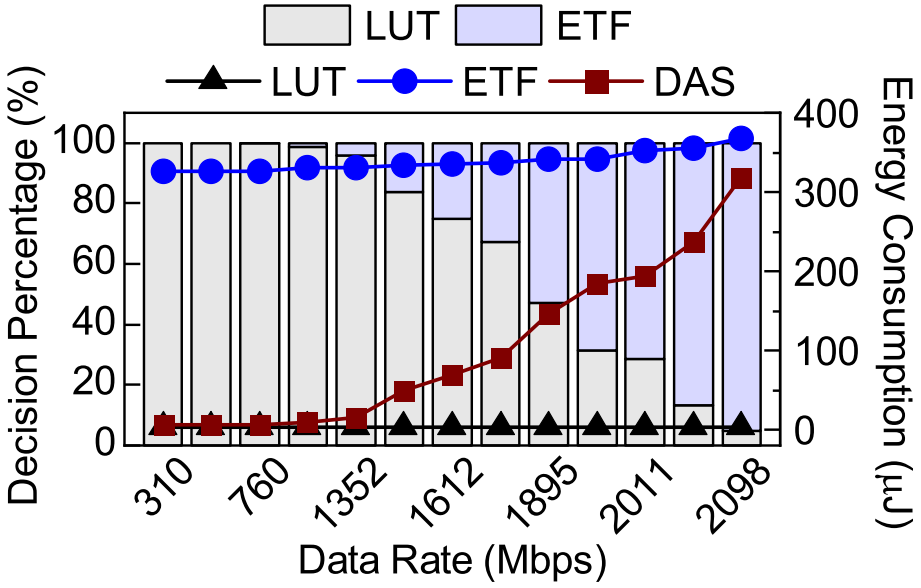

Figure 6.8: Decisions taken by the DAS framework as bar plots and total scheduling energy overheads of LUT, ETF, and DAS as line plots.

6.3.4 Scheduling Overhead and Energy Consumption Analysis

This section analyzes the runtime selections made by the DAS framework between LUT and ETF schedulers. Figure 6.8 plots the decision distribution of DAS for a workload that comprises all five applications on the primary axis. As a reminder, the low-overhead fast scheduler (LUT) performs well under low system load. In contrast, the sophisticated scheduler (ETF) is superior under heavy system load to achieve better performance and EDP. We note that DAS exploits the benefits of LUT at low system loads and the ETF scheduler under heavy loads. As the data rate increases, the DAS framework judiciously utilizes the ETF scheduler more frequently, as observed in Figure 6.8. Specifically, the DAS framework uses the LUT scheduler for *all* decisions at a low data rate of 135 Mbps and the ETF scheduler

for 95% of the decisions at the highest data rate of 2098 Mbps. At a moderate workload of 1352 Mbps, the DAS framework still uses the LUT scheduler for 96% of the decisions. As a result, the average scheduling latency overhead of the DAS framework for all workloads is 65 ns.

The secondary axis of Figure 6.8 shows the total energy overhead of using different schedulers. As DAS uses LUT and ETF approaches based on the system load, its energy consumption varies from LUT to ETF. DAS energy consumption is slightly higher than LUT for low data rates because DAS preselection model itself consumes a small amount of energy. On average of all workloads, the energy overhead of DAS per scheduling decision is 27.2 nJ.

6.4 Evaluation of DAS using FPGA Emulation

Section 6.3 presented detailed evaluations of the proposed DAS scheduler in a system-level simulation framework. This section focuses on its implementation on a real hardware platform that includes: (1) heterogeneous PEs comprising general-purpose cores and hardware accelerators and (2) a runtime framework that implements domain applications on heterogeneous SoCs and allows integrating customized schedulers. Section 6.4.1 first describes the SoC configuration, followed by an overview of the runtime environment and the data generation setup to train DAS models and their deployment in the runtime framework. Finally, Section 6.4.2 presents performance evaluations of the proposed DAS scheduler on a hardware platform.

6.4.1 Experimental Setup

DSSoC Configuration: The domain applications presented in Section 6.3.1 frequently perform the FFT and matrix multiplication operations. To this end, we construct a hardware platform comprising hardware accelerators for FFT and matrix multiplication. Additionally, we include three general-purpose cores to execute the other tasks. The full-system hardware that integrates the cores and accelerators is implemented on a Xilinx Zynq UltraScale+ ZCU102 FPGA [177].

Runtime Environment: This study utilizes the CEDR runtime framework [168] to implement DAS and evaluate its benefits for a DSSoC. CEDR allows users to compile and execute applications on heterogeneous SoC architectures. Furthermore, CEDR launches the execution of workloads comprising a mix of applications, each streaming with user-specified injection intervals. It offers a suite of schedulers and allows users to plug-and-play custom scheduling algorithms, making it a highly suitable environment for evaluating DAS. Therefore, we integrate DAS into CEDR and execute workloads on the customized domain-specific hardware. Furthermore, we implemented the LUT in software using inline assembly code and filled the task-PE assignments using the profiling information of domain applications on the target hardware.

To support the evaluation of DAS for the EDP objective, measuring power on the hardware platform at runtime is crucial. The ZCU102 FPGA integrates several on-board current and voltage sensors for the different power rails on the board [179]. These per-rail sensors allow us to accurately measure power for the processing system (which contains the CPU cores) and the programmable fabric

(which includes the hardware accelerators). Sysfs interfaces in the Linux operating system enable users to read data from these sensors [180]. To this end, we integrated functions that read the sysfs power values into CEDR. CEDR invokes these functions at runtime to accurately measure power consumption and thereby, EDP.

Training Setup: We utilized four real-world applications from the telecommunication and radar domains: WiFi-TX, Temporal Mitigation, Range Detection, and Pulse Doppler to generate the training data for the DAS preselection classifier. Fifteen different workloads are generated from these four applications by varying the constitution of the number of jobs and their injection intervals to represent a variety of data rates. For example, a workload has 80 Temporal Mitigation and 20 WiFi-TX instances, while another one has 100 Range Detection instances. Each workload is run in streaming mode and repeated for hundred trials of twelve data points on the FPGA to mitigate runtime variations due to the operating system and memory. Consequently, each data point in Figure 6.10 represents approximately 150,000 scheduled tasks with 100 jobs per trial and an average of 15 tasks per job using a specific scheduler. We utilize the same subset of performance counters described in Section 6.3.2 on the hardware platform to train the DAS preselection classifier model. The model employs a decision tree classifier with a maximum depth of 2. The choice of decision trees as the machine learning technique for the DAS preselection classifier and the tree depth are discussed in Section 6.3.2. It achieves a classification accuracy of 82.02% in choosing between the slow and the fast scheduler at runtime. The accuracy on the hardware platform is lower than observed on the system-level simulator (85.48%) due to runtime variations of the

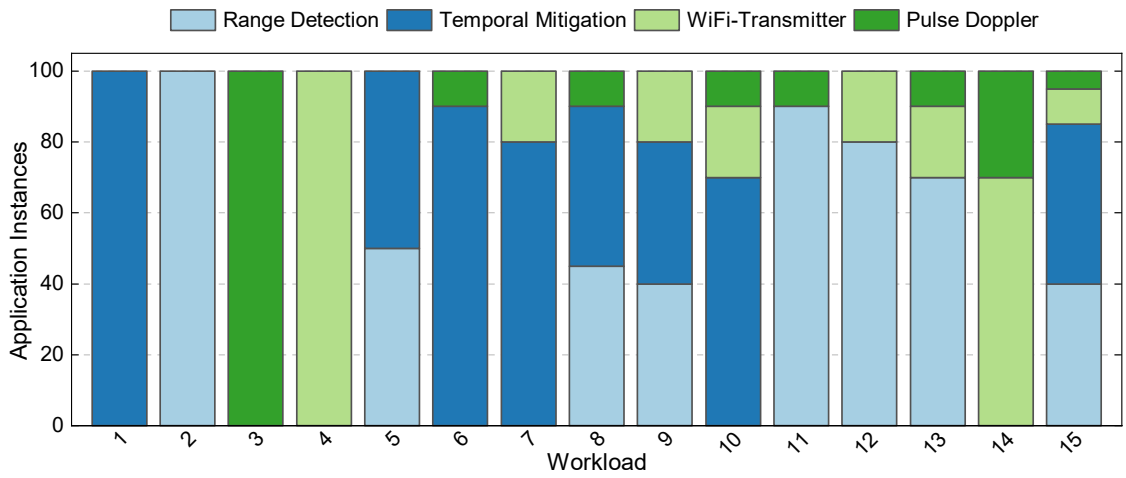

Figure 6.9: The distribution of number and type of application instances in the 15 workloads used for evaluation of the DAS framework on the runtime framework.

operating system and memory.

Runtime Framework Workload Mixes: Figure 6.9 presents the workload mixes used in the DSSoC simulator. Each workload consists of 100 jobs, and the stacked bar for each workload denotes the number of instances of each type of application. For example, Workload-7 (WKL-7) uses 80 instances of Temporal Mitigation and 20 instances of WiFi-Transmitter, and Workload-14 (WKL-14) uses 30 instances of Pulse Doppler and 70 instances of WiFi-Transmitter. The distribution is selected so that it represents a variety of low, medium, and high data rates. Also, we try to balance the distribution of applications. For example, the Pulse Doppler application has highly parallel FFT tasks which dominate the system. Therefore, we try to minimize the number of Pulse Doppler instances. Otherwise, the system is overwhelmed by the FFT tasks from a few instances of Pulse Doppler.

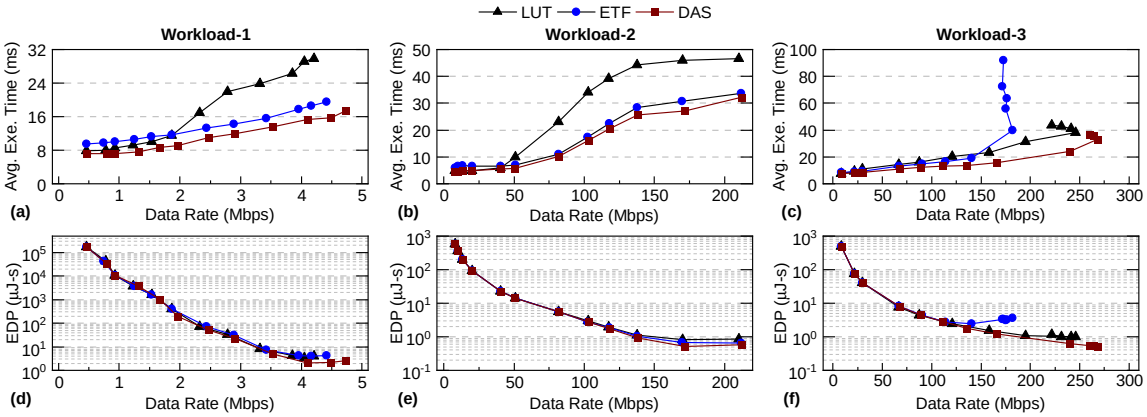

Figure 6.10: A comparison of average execution time (**a**–**c**) and energy-delay product (EDP) (**d**–**f**) between DAS, LUT, ETF on a hardware platform for three different workloads.

6.4.2 Performance Results

This section compares the average execution time and EDP of the proposed DAS framework with ETF and LUT schedulers running on the DSSoC configuration implemented on the ZCU102 FPGA. We note that the ETF-Ideal scheduler is not implemented in the FPGA since achieving zero overheads on a real system is infeasible. Figure 6.10 compares DAS with the LUT and ETF schedulers for three representative workloads. Figure 6.10a,d, Figure 6.10b,e, and Figure 6.10c,f present the average execution time and EDP results for a low data rate (Workload-1), moderate data rate (Workload-2), and high data rate workload (Workload-3), respectively. At the lower data rates of each workload, the system experiences low congestion levels and presents simpler scenarios for task scheduling. Therefore, the DAS classifier predominantly chooses the LUT (fast) scheduler resulting in similar execution times, as shown in Figure 6.10a–c. As data rates increase for

Workload-1 and Workload-2, the system experiences more congestion, and hence, the scheduling decision of LUT are less optimal. DAS thereby switches between LUT and ETF, trading off the scheduling overheads of LUT with the optimality of ETF. Consequently, DAS achieves an execution time that is notably lower than that of LUT and ETF schedulers for both Workload-1 and Workload-2. Specifically, DAS achieves an execution time improvement of up to 35% and 41% than LUT and ETF, respectively, for Workload-1, and 21% and 13% for Workload-2. The DAS framework also reduces the energy-delay product (EDP) by up to 15% for Workload-1 and up to 21% for Workload-2. As the data rate increases for Workload-3, DAS evaluates and selects the better-performing scheduler between LUT and ETF at runtime. In this scenario, the DAS framework favors the LUT scheduler since the ETF's overhead results in a longer execution time due to the substantial number of ready tasks. Specifically, DAS reduces the execution time by up to 35% and 48% and lowers EDP by up to 52% compared to LUT and ETF schedulers for Workload-3. These results show that the DAS framework on a real system dynamically adapts to the runtime conditions, leading to better performance than the schedulers it utilizes.

7 RUNTIME MONITORING FOR TASK SCHEDULING TOWARDS ROBUST DOMAIN-SPECIFIC SOCS

7.1 Background, Motivation, and Contributions

Heterogeneous architectures integrate diverse computing elements, each tailored to optimize specific objectives, resulting in enhanced performance across various fronts of optimization. Among these architectures, domain-specific systems-on-chip (SoCs) are meticulously designed to excel in particular domains such as augmented/virtual reality, autonomous driving, and telecommunication [164, 165]. They maximize energy efficiency by integrating domain-specific hardware accelerators while supporting general-purpose computing by including general-purpose cores [167, 168], effectively blending adaptability and efficiency. In the context of scheduling, the NP-complete nature of the task scheduling problem poses significant challenges to traditional algorithms as the number of processing elements (PEs) and tasks increase due to the concurrent execution of multiple applications [181, 182]. This challenge have recently led researchers to develop machine learning-based task scheduling and other dynamic resource management techniques [108, 109, 111, 91, 112, 110].

Machine learning-based policies can deliver fast and high-quality decisions tailored to a particular domain by leveraging system, application, and task information as features. They are trained using diverse workloads representing a target domain to achieve this objective. Like any machine learning model, ML-based

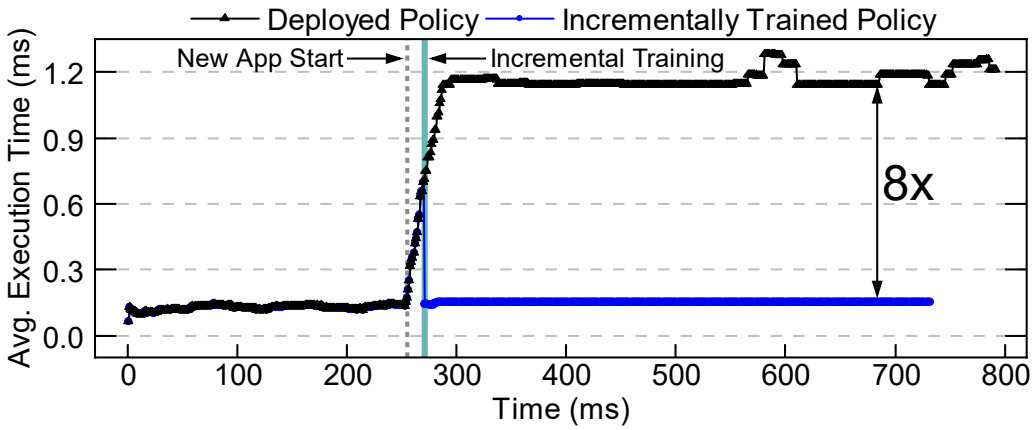

Figure 7.1: Illustration of incremental training. The gray dotted line represents the arrival of the unknown application, whereas the green line represents when the policy is updated with the incrementally trained version. The deployed and incrementally trained policies are IL-based scheduling algorithms. The execution time is 8× lower after incremental training.

schedulers operate reliably within the confines of the datasets and applications used during training. Consequently, they may fail, or their performance may deteriorate when faced with new workload scenarios, especially those involving new applications [183, 184, 185]. Therefore, there is a strong need to monitor the scheduling decisions to detect any non-robust decisions.

Figure 7.1 illustrates the variation in the execution time as an ML policy schedules streaming tasks to the PEs in an SoC. Initially, the SoC runs a mixture of applications that were used while training the ML scheduler. An unknown application replaces the previous mix at the instance marked by the gray dotted line. The average execution time begins to increase substantially after the arrival of the unknown application and converges to the execution time of the new application. A close inspection of the decisions reveals that the scheduler makes incorrect deci-

sions. As a concrete example, it fails to recognize that one of the tasks in the new application could utilize a hardware accelerator PE. Due to the incorrect decisions, the execution time is 8× longer than a scheduler trained with this new application could achieve. Indeed, if one could detect the arrival of a new application class and incrementally train the scheduler, it could achieve significantly higher performance, as depicted by the green vertical line in Figure 7.1. This example shows two crucial needs when an ML-based resource manager, such as a scheduler, is used in domain-specific SoCs. First, it must recognize the input changes (e.g., the arrival of a new application) to which the scheduler does not generalize. Second, an on-the-fly incremental training technique must adapt the scheduler to changes in data distribution over time while retaining knowledge from past data.

We propose a novel framework that achieves the following goals: (1) It monitors the actions of an ML-based scheduler, (2) it detects the input changes that deviate from the training data, and (3) it incrementally trains the ML policy to adapt to the new application. *To present a concrete implementation of the proposed framework*, we employ two runtime task schedulers, one trained using imitation learning (IL) and the other with reinforcement learning (RL). The proposed runtime monitoring is performed as a background task while an ML scheduler assigns incoming tasks to the PEs in the SoC. It first reads the features used by the ML scheduler, such as expected task execution times and PE states. Then, it computes the gradient of the trained ML policy and a coherence value using the gradient. When the gradient of the trained policy and information added by the new data samples are aligned, the coherence value is low, indicating that the current model generalizes well to the

latest data samples. In contrast, when the latest data samples are not aligned with training, the coherence increases, indicating the need for retraining. When this happens, the proposed framework incrementally updates the ML policy, adapting it to new applications while retaining past information.

The efficacy of the proposed framework is assessed using six real-world communication and radio frequency (RF) applications running on a domain-specific SoC with sixteen processing elements, including general-purpose big core clusters alongside fixed-point accelerators. Two instances of the proposed framework tailored to IL and RL schedulers monitor the SoC while a subset of the domain applications are launched. The proposed framework determines whether the IL scheduler generalizes to incoming data with over 98% accuracy. It misses only 0.59% of data points the scheduler fails to generalize. Furthermore, incrementally training the scheduler enables, on average, 4.21× faster execution time. The detection accuracy drops to 88.75% while monitoring an RL scheduler since RL policies rely on a reward function, weaker feedback than the reference label available in imitation learning. The proposed framework can still effectively flag when incremental training is needed and enable, on average, 1.32× faster execution. Finally, we implemented the proposed monitoring framework on the Nvidia Jetson Xavier NX board [3] to assess its runtime overhead. Since the proposed framework is not on the critical path, the execution time affects only (1) how fast poor scheduling decisions can be detected and (2) how frequently the monitoring process can be repeated. Even when all the steps of the proposed framework run sequentially, the worst-case execution times of the IL and RL instances are 83.74 ms and 117.53 ms,

respectively. Hence, they can be effectively used as real-time background processes that run periodically, as detailed in Section 7.4.

The main contributions of this chapter are as follows:

- A framework that continuously monitors the system, identifying unforeseen tasks and incrementally training the model as needed,
- Integration of a coherence-based detection mechanism within reinforcement and imitation learning approaches,
- Comprehensive experiments showcasing the effectiveness of the proposed framework in restoring performance,
- Runtime overhead analysis with hardware measurements.

The remainder of this chapter is organized as follows. Section 7.2 reviews the background on the coherence metric and ML schedulers. Section 7.3 describes the proposed framework and its application to IL/RL schedulers. Section 7.4 presents comprehensive experimental evaluations and hardware measurements.

7.2 Background on Coherence and ML Schedulers

This section first introduces coherence and its implications for the generalization of ML policies. Then, it overviews the use of ML schedulers in the domain-specific SoC context and describes the schedulers used in this work.

7.2.1 Background on the Coherence

Deep learning models trained with gradient descent have shown promising results in various fields, often demonstrating impressive generalization capabilities on unseen data. However, recent work notes that these networks theoretically have the capacity to memorize the training data. So, they could fail on any new input data [186, 187, 188]. Indeed, studies have shown that training with even entirely random data can lead to good training accuracy, but the models fail to generalize and exhibit poor accuracy on new data, indicating memorization. Hence, it is crucial to understand how gradient descent and the training process find solutions that generalize well among all possible solutions that fit the training data [187, 188].

One of the recent ongoing attempts to explain generalization in deep learning is "Coherent Gradients" [186, 189]. The core idea is that gradients calculated from similar training samples should be coherent, meaning they point in similar directions for generalization (rather than memorization) to happen. In other words, the theory suggests that the interaction and reinforcement between gradients from different training examples lead the model to learn features that generalize well to unseen data.

Suppose z is a sample from a batch $(\mathcal{M})$ with $M = |\mathcal{M}|$ data samples. Further, let $l_z(w)$ denote the loss function for this sample, where w represents the trainable parameters of this model. One can compute the gradient for this sample as $g_z = [\nabla l_z](w)$. Chatterjee et al. [186] quantify the coherence over these M samples using per-sample gradients. Specifically, they refer to the similarity between per-sample

gradients as *coherence* and define it as:

$$\alpha_{\mathcal{M}} = M \cdot \frac{\mathop{\mathbb{E}}_{z\sim\mathcal{M}}[g_z] \cdot \mathop{\mathbb{E}}_{z\sim\mathcal{M}}[g_z]}{\mathop{\mathbb{E}}_{z\sim\mathcal{M}}[g_z \cdot g_z]} \tag{7.1}$$

When the gradients (g_z) are perfectly aligned, the numerator and denominator will be equal, leading to maximum coherence (M). When all samples are fit, coherence will be zero, meaning the individual gradients will become zero.

During the initial training epochs, training data often shares many common features. This results in aligned gradients and, consequently, a higher coherence value. As training progresses and trainable parameters converge, new features become more specific, and the model tries to learn them individually. Consequently, the coherence value tends to decrease, as illustrated in Figure 7.2.

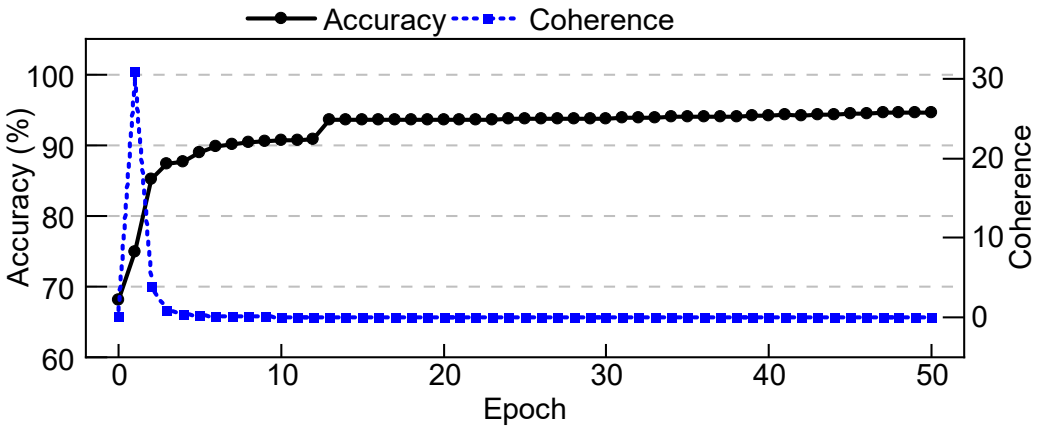

Figure 7.2: The evolution of the coherence and accuracy throughout training. The examples exhibit stronger mutual support in the early epochs, resulting in higher coherence (the right $y-$axis). As training progresses, the expected gradient of samples approaches zero, indicating that the samples no longer provide significant assistance to one another. Consequently, coherence tends to diminish towards zero by the end of the training period.

Using coherence for runtime monitoring: Gradients reinforce each other when learning takes place during the early training phases, leading to high coherence, as shown in the first few epochs of Figure 7.2. After the model has learned what is common to all the samples and the samples have been fit (in a well-generalizing manner), the coherence drops and stabilizes to a low value. When the workload falls within the generalized set at runtime (not necessarily identical to the training data), its behavior resembles the end of the training phase illustrated in Figure 7.2. Consequently, it is characterized by a low coherence value (like the latest data sample during training. However, if the new data samples deviate from the training data, their gradients would align, leading to a rise in the coherence value. Therefore, increasing coherence indicates that the model processes features from an application that it has not generalized yet. The coherence will remain high unless the ML policy, e.g., the scheduling algorithm, is incrementally trained. We leverage this observation in the proposed runtime monitoring framework. A low coherence for generalized workload indicates good performance, while a sustained high coherence suggests encountering new data that requires retraining. Using a smaller sample size of M would result in a smaller overall coherence range (zero to M in Equation 7.1), making the framework more susceptible to random noise during inference and negatively impacting accuracy. In contrast, a larger M would result in increased overhead, as discussed in Section 7.4.4 (see Table 7.2, 7.3).

While the authors of [186, 189] focus only on neural network models, we observe that the generalization theory using coherence can be used with any ML policy trained with gradient descent. Hence, we applied this framework to two

scheduling algorithms: IL and RL. The IL model is trained with neural networks, while RL model is trained with differential decision trees (DDTs), as elaborated in the following sections.

7.2.2 ML-Based Scheduling for Domain-Specific SoCs

Domain-specific SoCs are designed to deliver high performance when running applications from a target domain. A defining characteristic of these applications is processing streaming inputs for prolonged periods. For example, consider a domain-specific SoC designed for telecommunication. When the user starts the WiFi application, it processes received frames or transmits new ones for minutes, if not hours. Throughout this duration, the SoC continuously schedules the tasks that make up the WiFi transmitter and receiver chains. We envision that the proposed framework can run when a new application launches or periodically. The monitoring can repeat in the order of seconds or slower since there is no need to check the scheduler operations faster than that due to application lifetimes. Notable approaches of ML schedulers (imitation and reinforcement learning-based training) are discussed next.

IL Schedulers: Imitation learning is an ML method where an agent learns a policy (π) that mimics the behavior of an expert (π^*) using the expert's actions. The objective of imitation learning is to minimize the error between the actions taken by the agent (a_t) and the expert (a_t^*). The expert actions (a_t^*) are collected offline and paired with corresponding states (s_t, a_t^*) for the agent to learn a policy (π_θ) [190]. However, this approach has limitations, as the behavior of the expert confines the

agent's policy. To address this issue, the data aggregation (DAgger) algorithm [191] enhances the performance of imitation learning by iteratively reinforcing incorrect decision-state pairs into the training set, thereby correcting deviations and improving overall performance.

In the context of task scheduling, imitation learning-based models leverage offline training capabilities. For example, the training data is collected through executing various workloads under different system states to cover low to high congestion. During this process, an expert scheduler makes decisions for these workloads, with the data representing the system state (s_t) collected alongside the expert's policy decisions ($\pi^*(s_t)$). These system states and their corresponding action pairs are then utilized as features and target labels for supervised learning methods within the imitation learning model. Subsequently, the learned imitation learning policy (π_θ) is deployed for runtime decision-making, replacing the expert policy (π^*). The expert may be a sophisticated heuristic or a constrained programming scheduler, which can make high-quality decisions but with a significant overhead.

RL Schedulers: Unlike IL schedulers, RL schedulers do not require an expert scheduler to guide the policy toward optimal behavior [114, 113, 109]. During training, the agent interacts with the environment by taking action (a_t) as a function of the current state (s_t), such as expected task execution and earliest PE availability times. For each action, the environment gives the agent a reward (r_t) that reflects how well the action aligns with the performance objectives, such as minimizing the execution time.

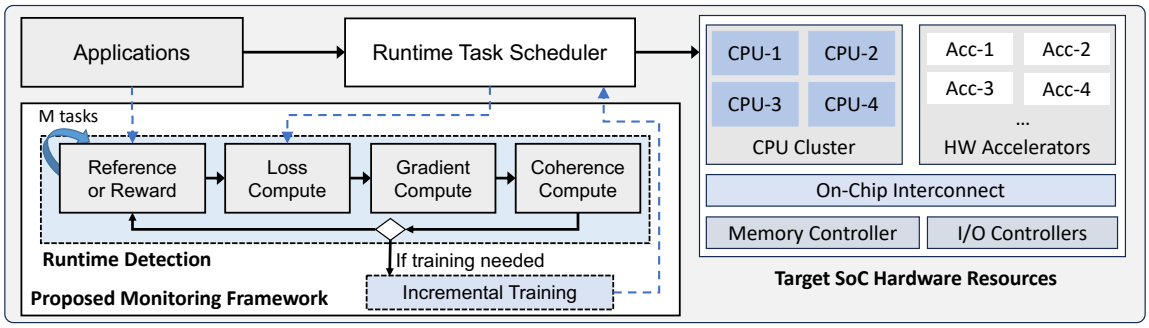

Figure 7.3: The overview of the proposed framework that monitors the scheduler decisions and application features used for decision making. It is activated to compute the coherence of a batch with M samples. The primary steps are: (1) generating the reference scheduler action for IL or reward calculation for RL, (2) loss, gradient, and coherence calculations, and (3) an optional incremental training step triggered by the coherence value.

RL training algorithms commonly use actor-critic architectures, where the actor selects the actions (a_t), and the critic evaluates their expected outcomes. Both the actor and critic are continuously updated based on the feedback from the environment in terms of reward, allowing the agent to refine its policy (π_θ) over time[192]. The agent aims to optimize policy (π_θ) that takes actions to maximize the total reward over time. The state value function can be used to find expected rewards starting from an initial state following the policy. This value function $(V_\phi(s_t))$ can be approximated with a critic network with parameter (ϕ) that returns an expected value according to the state of the environment.

7.3 Robust Monitoring of ML-Based Scheduling Algorithms

This section describes the proposed robust monitoring framework overviewed in Figure 7.3. Section 7.3.1 introduces the runtime monitoring component of the framework for detecting workload changes. Then, sections 7.3.2 and 7.3.3 present the application of the proposed framework to imitation learning and reinforcement learning schedulers, respectively. Finally, Section 7.3.4 discusses its applicability to other ML-based dynamic runtime management frameworks, and Section 7.3.5 presents the incremental training approach used in our robust monitoring framework.

7.3.1 Robust Detection of Workload Changes

The first step of the proposed robust monitoring framework is continuous monitoring to detect the variations in the workload that can lead to incorrect decisions, as shown in Figure 7.3. It is implemented as a background process to avoid any performance impact. Suppose the scheduler takes an action at time T_0. The system runs as usual by committing this action without interrupting the operation. At the same time, the background monitoring process is invoked to evaluate the quality of this action after the task is completed, as illustrated in Figure 7.4. This evaluation is performed by calling a reference scheduler with identical inputs and finding the reference action when monitoring an IL-based scheduler (detailed in Section 7.3.2). In the case of an RL-based scheduler, the reward received for this action is used to

assess its quality (detailed in Section 7.3.3). Then, the outcome of this assessment is used to compute the gradient of the ML policy. Finally, the gradient is used to compute the coherence, as described in Section 7.2.1. An insignificant change in the coherence value shows that the current ML policy handles the monitored application well. That is, the policy generalizes well to the monitored application. In contrast, a rise in coherence indicates new directions in the gradient, signifying the need to adopt the policy to address the changes in the workload. The specific details of the coherence calculation for IL- and RL-based schedulers are described in the following subsections.

Background Process Overhead: The proposed monitoring and detection framework is implemented as a background process, as mentioned above and illustrated in Figure 7.4. The system moves on with the current scheduling decision to avoid interruption since an incorrect decision only leads to transient performance degradation but not catastrophic failure. Hence, the proposed framework is not on the critical path. However, its overhead is still important since it determines how frequently the proposed framework can be called and the detection speed. Our hardware measurements indicate that the proposed monitoring and detection can be performed in the order of milliseconds, allowing frequent checks for the robustness of the ML policies. Given the types and composition of applications running on SoCs do not change in the order of seconds, the proposed framework enables runtime monitoring with negligible overhead, as detailed in Section 7.4.4.

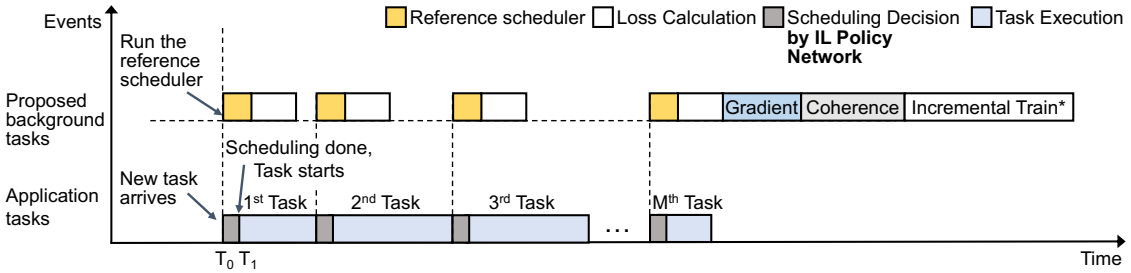

Figure 7.4: Event diagram illustrating the proposed monitoring framework *for IL schedulers*. This figure shows the tasks in series for clarity, but multiple parallel tasks can be scheduled and monitored concurrently. While monitoring IL schedulers, a trustworthy (but slower) scheduler runs in the background to determine the correct action (a_t^*). This reference and actual policy actions (a_t) for a batch with M tasks are used for the loss, gradient, and coherence calculations (detailed in Algorithm 5). The incremental training step is executed if the framework decides the IL model policy (π_θ) should be updated.

7.3.2 Application to IL-Based Schedulers

This section outlines the runtime detection framework employed for imitation learning-based task scheduling frameworks. Figure 7.4 illustrates the calculation steps in runtime monitoring for IL-based schedulers, while Algorithm 5 provides a detailed breakdown of these steps involved in the runtime detection process.

Once activated, the proposed monitoring framework processes the actions (a_t) taken by the policy (π_θ) for a sample size of M (Algorithm 5, lines 4-5). IL schedulers operate as supervised learning models, wherein an agent learns a policy (π_θ) from an expert's decision-making patterns to guide runtime scheduling decisions by generating actions (a_t) (line 6). Therefore, the runtime detection framework requires the reference targets (a_t^*), obtained by invoking the expert scheduler and collecting necessary performance metrics in the background to avoid execution

Algorithm 5 Detection Phase for the IL Scheduler

1: **Input:** Policy action set $\mathcal{M}$ with M actions, Call period of the framework P
2: **Output:** Coherence value
3: **Initialization:** ML policy π_θ with parameters θ, $\mathop{\mathbb{E}}_{z\sim\mathcal{M}}[g_z] := 0$, $\mathop{\mathbb{E}}_{z\sim\mathcal{M}}[g_z \cdot g_z] := 0$
4: Call robust monitoring framework every P timeframe
5: **while** Total number of actions $< M$ **do**
6: Get policy action a_t
7: Get ground truth a_t^* in the background using reference scheduler
8: Calculate the loss function, $\mathcal{L}_\theta$ using a_t^* and a_t as CrossEntropyLoss(a_t^*, a_t) [193]
9: **end while**
10: // gradients and coherence calculation
11: **for** sample from 1 **to** M **do**
12: Calculate gradients (g_z) using loss function $\mathcal{L}_\theta$ from current sample
13: Update estimate $\mathop{\mathbb{E}}_{z\sim\mathcal{M}}[g_z] := \mathop{\mathbb{E}}_{z\sim\mathcal{M}}[g_z] + g_z$
14: Update estimate $\mathop{\mathbb{E}}_{z\sim\mathcal{M}}[g_z \cdot g_z] := \mathop{\mathbb{E}}_{z\sim\mathcal{M}}[g_z \cdot g_z] + g_z \cdot g_z$
15: **end for**
16: Coherence $\alpha_\mathcal{M} = M \cdot \dfrac{\mathop{\mathbb{E}}_{z\sim\mathcal{M}}[g_z] \cdot \mathop{\mathbb{E}}_{z\sim\mathcal{M}}[g_z]}{\mathop{\mathbb{E}}_{z\sim\mathcal{M}}[g_z \cdot g_z]}$

time overhead (line 7). In this work, we employ a resource-intensive heuristic, the earliest task first (ETF) scheduler that loops through all ready tasks and PEs to choose the task assignment that minimizes the expected execution time [105]. The authors of the IL scheduler select ETF as the reference scheduler because its overhead grows quadratically, ranging from 0.3 ms to 8 ms. In contrast, the IL scheduler overhead grows linearly and enables nanosecond-level decisions [114, 91]. The reference actions are used to compute the loss function, denoted as $\mathcal{L}_\theta$ (line 8), in conjunction with the IL policy actions. We utilize the cross-entropy loss for $\mathcal{L}_\theta$. Then, the loss function is used to calculate the gradients g_z. Subsequently, we

calculate the expected value of the gradient vector, $\mathop{\mathbb{E}}_{z\sim\mathcal{M}}[g_z]$, by adding the gradient vectors and $\mathop{\mathbb{E}}_{z\sim\mathcal{M}}[g_z \cdot g_z]$ by adding the dot product of the gradient vectors of weights, respectively. This process can be executed efficiently, with expected values computed using running sums without storing the gradients, either incrementally or collectively, at each monitoring session's conclusion. Finally, the coherence is computed using Equation 7.1. It determines the coherence of gradients among all examples in the sample set $\mathcal{M}$, thereby detecting the unforeseen task scheduling scenarios that differ significantly from those encountered during training. We use the loss function for coherence instead of relying on accuracy because the accuracy metric misses differences when the policy generates the same actions with low confidence. Besides, accuracy remains relatively stable when a few new application instances are added to the workload mix. The loss value, in contrast, is sensitive to such variations.

7.3.3 Application to RL-Based Schedulers

This section presents the steps to apply our runtime monitoring framework to RL schedulers. During monitoring, the actor policy (π_θ) makes scheduling decisions (action a_t) for new tasks based on the SoC state (s_t). The selected PEs process the tasks as in normal operation. As described in Section 7.2.2, RL is an unsupervised learning method where both the actor and critic networks are trained during the offline training phase to maximize the reward defined as the negative execution time. Therefore, estimated state values ($V_\phi(s_t)$) from the trained critic network and rewards (r_t) expressed as the negative of the task execution times are used to

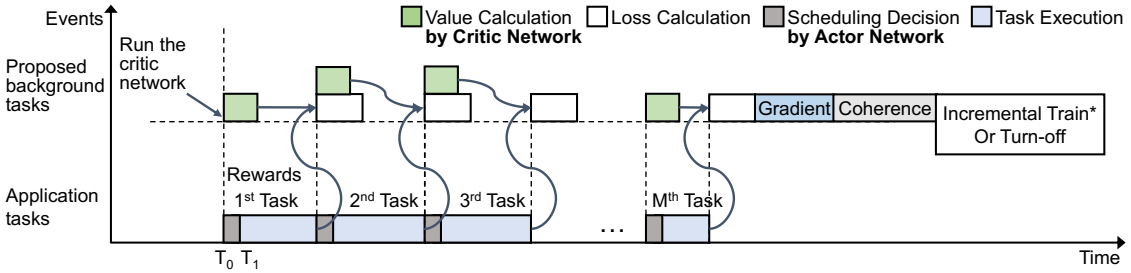

Figure 7.5: Event diagram illustrating the proposed monitoring framework *for RL schedulers*. As in Figure 7.4, the tasks are shown in series for clarity, but multiple parallel tasks can be scheduled and monitored concurrently. The proposed framework performs the loss calculation using the estimated value function $(V_\phi(s_t))$ from the critic network and rewards (r_t) from completed tasks. Then, the gradient is calculated for mini-batches (lines 13-17 in Algorithm 6), while coherence is calculated using batch coherence as given in line 19 in Algorithm 6. If the RL policy does not generalize to the current data points, it can be incrementally trained or turned off until the coherence reduces.

calculate the loss function $\mathcal{L}_\theta$ required for the gradient calculation. As new tasks arrive, the trained critic network updates the state values in the background, as illustrated in Figure 7.5. Upon task completion, rewards in terms of execution time are acquired from the PEs. These rewards and the state values are used to calculate the advantage function $(A(s_t, a_t))$ for the state-action pair, following the same equation as given in the PPO algorithm [192]:

$$A(s_t, a_t) = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t) \tag{7.2}$$

where γ represents the discount factor and $V_\phi(s_{t+1})$ is the state value after completion of the task. The loss calculation during training also uses the ratio between the updated policy and the previous policy $\rho(\theta)$. Since the policy remains fixed during inference at runtime, the probability ratio $\rho(\theta)$ remains equal to one. Thus, policy

Algorithm 6 Detection Phase for the RL Scheduler

1: **Input:** Batch $\mathcal{M}$, mini-batch $\mathcal{K}$, Framework activation period P
2: **Output:** Coherence value
3: **Initialization:** Learned policy π_θ with parameters θ, Trained critic V_ϕ with parameters ϕ, $\mathop{\mathbb{E}}_{z\sim\mathcal{K}}[g_z] := 0$, $\mathop{\mathbb{E}}_{z\sim\mathcal{K}}[g_z \cdot g_z] := 0$
4: Call robust monitoring framework every P seconds
5: // Loss calculation
6: **while** Total number of actions $< M$ **do**
7: Get policy actions a_t
8: Get value estimates $V_\phi(s_t)$ using trained critic V_ϕ
9: Get rewards ($r_t = -$task execution time) from the system for the completed tasks
10: Calculate the advantage function as loss, $\mathcal{L}_\theta$ using $V_\phi(s_t)$ and r_t for all the actions as given in the equation 7.2
11: **end while**
12: // Mini-batch gradients and coherence calculation
13: **for** mini-batch from 1 **to** K **do**
14: Calculate gradients (g_z) using average $\mathcal{L}_\theta$ from current mini-batch
15: Update estimate $\mathop{\mathbb{E}}_{z\sim\mathcal{K}}[g_z] := \mathop{\mathbb{E}}_{z\sim\mathcal{K}}[g_z] + g_z$
16: Update estimate $\mathop{\mathbb{E}}_{z\sim\mathcal{K}}[g_z \cdot g_z] := \mathop{\mathbb{E}}_{z\sim\mathcal{K}}[g_z \cdot g_z] + g_z \cdot g_z$
17: **end for**
18: Mini-batch Coherence $\alpha_K = \dfrac{\mathop{\mathbb{E}}_{z\sim\mathcal{K}}[g_z] \cdot \mathop{\mathbb{E}}_{z\sim\mathcal{K}}[g_z]}{\mathop{\mathbb{E}}_{z\sim\mathcal{K}}[g_z \cdot g_z]}$
19: Coherence $\alpha_M = M \cdot \frac{\alpha_K}{K-(K-1)\cdot\alpha_K}$

loss $\mathcal{L}_\theta$ is given by the advantage function in Equation 7.2 and used in Algorithm 6 (line 10):

$$\mathcal{L}_\theta = \rho(\theta) \cdot A(s_t, a_t)\,; \;\; \rho(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} = 1 \tag{7.3}$$

Since this loss is not directly derived from the ground truth, the resulting gradient and coherence become noisy. To address this, we split the batch $\mathcal{M}$ (with $M = |\mathcal{M}|$ samples) into a set of $\mathcal{K}$ mini-batches (with $K = |\mathcal{K}|$, each of size M/K).

Then, we use the average advantage within each mini-batch for gradient calculation. The coherence for each mini-batch and the overall batch coherence are calculated using Theorem 3 in [186] (lines 18-19 in Algorithm 6). This theorem ensures statistical equivalence of the per-sample coherence described in Equation 7.1.

7.3.4 Application to Other ML-Based DRM Algorithms

The proposed framework uses a loss function and gradients to compute the coherence for detecting workload changes. Hence, it can be applied to monitor the decisions of other ML-based schedulers and dynamic resource management (DRM) algorithms that allow runtime gradient calculation. For example, dynamic thermal and power management techniques determine the optimal voltage-frequency pairs for computing cores to meet thermal constraints while preserving performance. These algorithms encompass a variety of approaches, including IL [110, 194, 195] and RL [196, 197] methods. Our framework can work with all these methods to prevent unexpected behavior due to a mismatch between training and runtime inputs. For example, researchers in [110] employ a hierarchical imitation learning framework featuring distinct policies for frequency, core selections, and execution time predictions. Our framework can effectively monitor these policies, utilizing the described policy and expert actions outlined in the study to compute loss and subsequent steps. It can ensure robust performance across various scenarios. In summary, our framework offers monitoring support for any runtime machine learning-based framework that utilizes gradient-based optimizations, ensuring robustness and reliability across various dynamic runtime management applications.

7.3.5 Response to Significant Workload Changes

The final stage of the proposed runtime monitoring framework is the response to significant changes in the workload. The objective of this stage is to detect the significant changes in the workload to which the trained model does not generalize. We compare the coherence (α_M) for this detection against a threshold (τ) learned during training. For this purpose, we employ a simple classifier, such as a support vector machine (SVM), to learn the threshold that maximizes the detection accuracy. Coherence values lower than the threshold ($\alpha_M < \tau$) indicate that scheduler decisions are trustworthy and no intervention is required. In contrast, larger coherence values ($\alpha_M > \tau$) require action since they indicate that the model is not generalizing well to coming samples.

There are two possible responses when a significant workload change deems the scheduler unreliable. The simplest remedy is to fall back to a traditional algorithm (e.g., the reference scheduler) for actions. The ML scheduler decisions can be monitored during this time until the coherence value moves below the threshold. In this way, the SoC will be protected from unreliable ML decisions. The second option is incrementally training the scheduler to adapt to workload changes, which will conserve the advantages of using ML schedulers. The rest of this subsection describes how IL and RL schedulers respond when a significant change is triggered.

IL Scheduler: The monitoring process for the IL scheduler involves a reference scheduler whose decisions are used to compute the loss function, as shown in Figure 7.4. This implies that the reference actions (a_t^*) for the samples received during monitoring (s_t) are readily available, making incremental training a practical

option. To this end, we utilize these state-action pairs (s_t, a_t^*) to *incrementally* train the IL policy. We also measure the overhead of this training process. It takes approximately 2 ms per epoch for incremental training of the IL scheduler on the Nvidia Jetson Xavier NX board [3], a timeframe negligible compared to the domain-specific application lifecycle. The execution of the tasks continues with the previous policy to ensure continuity during this process. Subsequently, the IL scheduler starts using the new policy $(\hat{\pi})$, leading to significant benefits detailed in Section 7.4.

RL Scheduler: Unlike the IL scheduler case, RL training is unsupervised, learning from rewards (task execution time) provided by the environment (PEs in SoC) rather than a reference. Hence, the corresponding monitoring process does not involve a reference scheduler that gives correct actions. RL scheduler can be trained at runtime by using the rewards received at the end of task executions. However, the RL scheduler can make poor decisions during this time, potentially impacting the runtime of tasks it executes. If this degradation in performance is acceptable, the policy can be incrementally updated during the operation. Otherwise, turning it off may be preferable while the coherence value is above the threshold. One can also train an RL policy offline incrementally and update the scheduler if the workload changes are permanent.

7.4 Experimental Evaluation

This section presents the experimental evaluations of our framework. We begin by detailing the experimental setup in Section 7.4.1. Section 7.4.2 discusses the results obtained for the IL scheduler, while Section 7.4.3 presents the findings for the RL scheduler. Lastly, Section 7.4.4 discusses the runtime overhead of our proposed framework for both schedulers.

7.4.1 Experimental Setup

Domain-Specific SoC Configuration: The selection of the SoC configuration is tailored to the requirements of domain-specific applications. Our simulation configuration consists of sixteen PEs, comprising eight general-purpose cores utilizing the Arm big.LITTLE architecture. These cores include four Arm A57 performance and four Arm A53 low-power cores. Additionally, the SoC incorporates eight fixed-function accelerators designed for handling intensive tasks: four accelerators dedicated to Fast Fourier Transform, two for Viterbi decoding, and two for matrix multiplication. This configuration is designed based on the specific demands of the target domain applications and the computational intensities of the tasks in these applications.

Domain Applications: The evaluation of the runtime monitoring framework encompasses six real-world applications spanning the telecommunication and radio frequency domains. These applications include WiFi transmitter, WiFi receiver, temporal mitigation, lag detection, single-carrier transmitter, and single-carrier

receiver. The number of tasks for these applications varies from 7 to 34. They are mixed into the workloads spanning from lower to higher intensity levels, ensuring comprehensive coverage, as detailed in [178, 198].

Simulation Framework: We evaluated our runtime monitoring framework using an open-source discrete event-based simulator, DS3 [178]. This simulator has been validated against two commercial SoCs, namely the Odroid-XU3 [199] and the Zynq Ultrascale+ ZCU102 [177]. It enables target application simulations using different schedulers, providing a flexible environment for efficiently implementing new scheduling policies and our framework. Each simulation duration is around 2 seconds, resulting in a dynamic variation in the number of applications running, ranging from 4,000 to 40,000 instances, and an average task count ranging from 50,000 to 500,000.

7.4.2 Results Obtained with IL Scheduler

This section delves into the experimental evaluations with IL schedulers. The policy adopted for the IL scheduler comprises a neural network architecture consisting of three dense layers, each with 32 neurons. The neural network is trained using Python and TensorFlow libraries, achieving accuracies ranging between 96.1%-98.3% against the reference scheduler, ETF [105]. The policy leverages a combination of system, application, and task-level data as features to determine the cluster assignment. Then, the task is assigned to the PE within the chosen cluster, which is either available or set to become available first. *We first illustrate the proposed framework as a function of time using single- and multi-application use cases. Then, we*

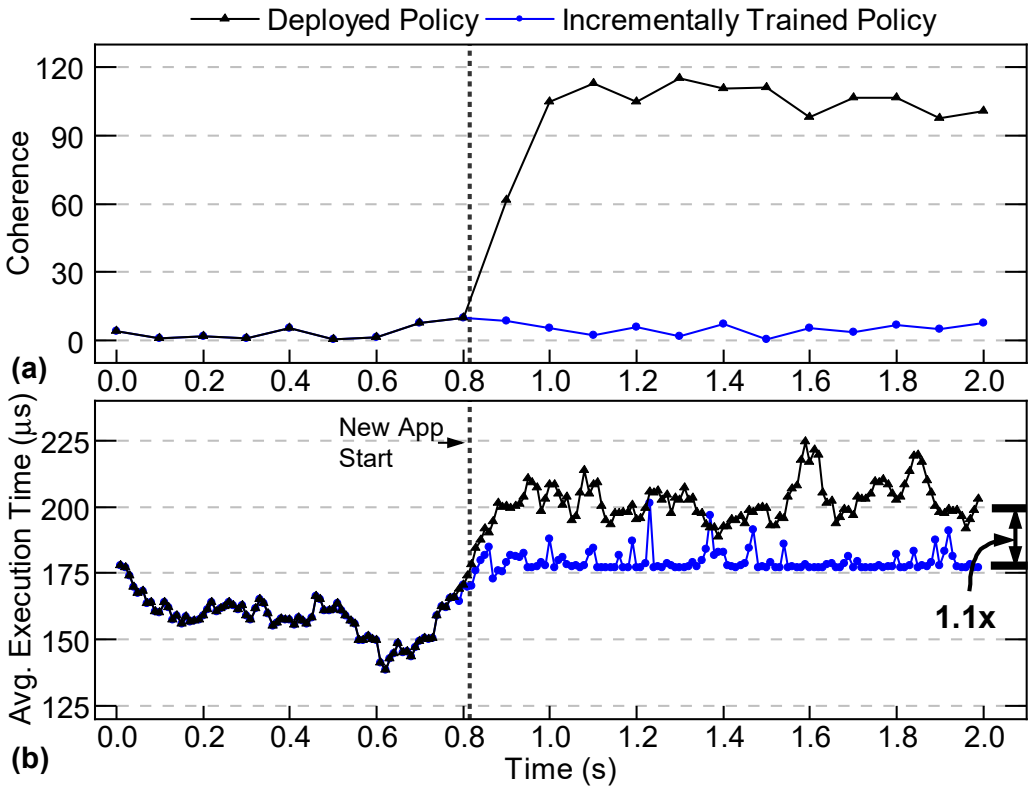

Figure 7.6: Illustration of the runtime monitoring framework with IL scheduler using two applications. The first application (WiFi transmitter) runs until the black dotted line. After that, it is replaced by a new application (lag detection) not represented in the training data.

summarize our exhaustive accuracy evaluations.

Single Application Use Case Illustration: This illustrative example starts running a domain application represented in the training dataset. As the test samples from this application arrive at runtime, the coherence value remains low, as shown in Figure 7.6(a). We emphasize that the training and test samples are different except that they come from the same application. Since the execution time varies significantly over time, it cannot be used alone to identify significant workload changes.

After running 0.8 seconds, this application is replaced with a new one that was not represented in the training dataset. The proposed monitoring framework successfully captures this change, as shown in Figure 7.6(a). The coherence increases quickly, indicating the unalignment between the trained policy and the impact of new data samples. If we do not take action (e.g., incrementally train or turn off the scheduler), the coherence remains high, and execution time varies around 200 us. In contrast, incremental training (explained in Section 7.3.5) successfully adapts the policy to the new application, as revealed by the coherence plot in Figure 7.6(a). Furthermore, the execution time reduces on average by 10%. Finally, we note that the incrementally trained policy still runs the first application optimally, i.e., the coherence remains low if it resumes running. This part is not plotted for brevity.

Multiple Application Use Case Illustration: The second example starts running a mix of five applications represented in the training dataset. The coherence computed at runtime remains low, as expected, as shown in Figure 7.7(a). After running them for about 0.25 seconds (marked by a dotted line), these applications halt, and a previously unseen application starts running. The proposed framework successfully tracks the increased coherence after this change. As in the previous example, an elevated coherence indicates that new data samples require updating the policy parameters. If the policy is not updated, coherence remains high, and the execution time rises to about 2.5 ms, as shown in Figure 7.7(b). In contrast, the proposed incremental training rapidly reduces coherence to its original value. Moreover, it achieves a remarkable performance boost (12$\times$ lower execution time) compared to no training.

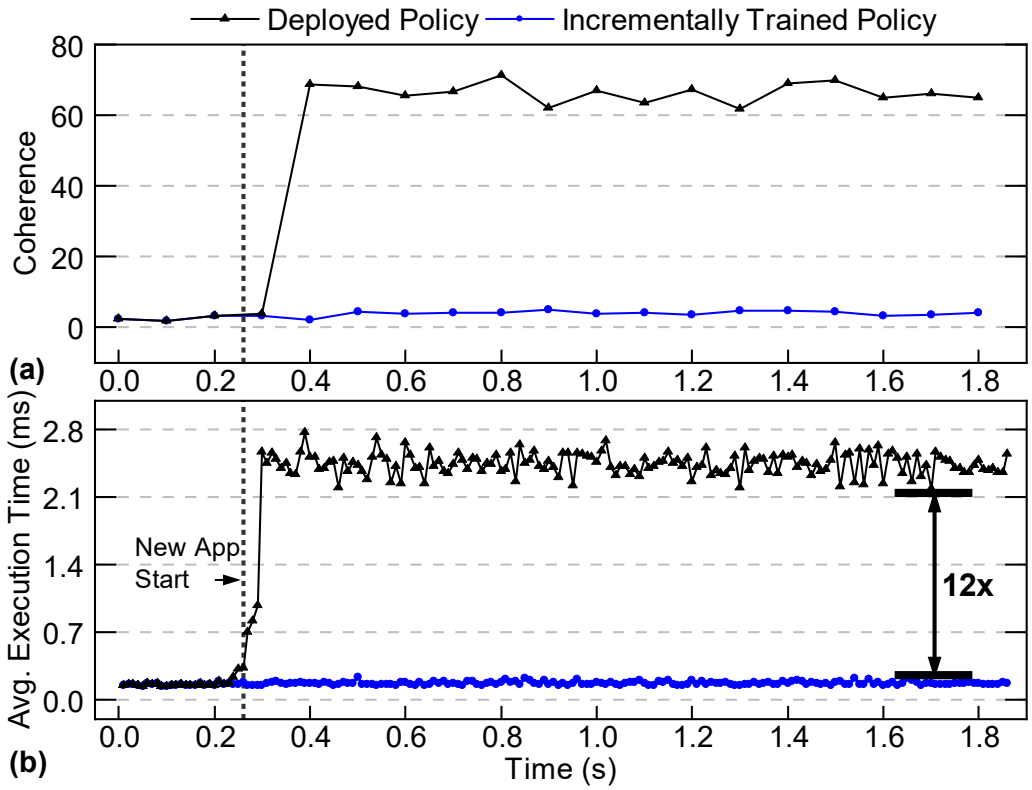

Figure 7.7: Illustration of the runtime monitoring framework with IL scheduler using a workload that is composed of six applications. Five out of six domain applications run concurrently until the black dotted line. After that, the sixth application (single-carrier receiver) is introduced.

Accuracy & Performance Summary: We prepared an extensive set of use case scenarios similar to those illustrated above. They start running a randomly selected subset of application mixes and then randomly change the applications. Single application examples start running one of the six domain applications randomly and switch to another one after a random duration. We repeated these simulations at different intensities and obtained 1221 batches. 663 out of these 1221 points indicate inputs to which the ML scheduler does not generalize. The multi-application

Table 7.1: Accuracy and execution time improvements for runtime monitoring framework on IL and RL schedulers (M=1024).

Scheduler	Monitoring Accuracy	False Negative	False Positive	Avg. Exec. Time Improvement
IL	98.39%	0.59%	1.02%	4.21×
RL	88.75%	6.20%	5.05%	1.32×

experiments start running five out of six applications concurrently (leaving one out). Then, the missing application replaces the original one. These experiments are also repeated to obtain 13767 batches. 8585 of these 13767 batches correspond to input the ML scheduler does not generalize. Overall, the combined data set comprises 14988 batches, of which 9248 batches indicate a significant input change.

The proposed runtime monitoring framework identifies whether the IL scheduler generalizes to new data points correctly 98.39% of the time, as summarized in Table 7.1 (the first row). False positives in Table 7.1 occur when our monitoring framework detects activity despite there being no new application. False negatives, on the other hand, happen when a non-generalized application appears, but our monitoring framework fails to detect it. The false positive rate for the IL scheduler is only 1.02%, i.e., it incorrectly flags a change, although the scheduler generalizes well to the input. More importantly, it almost never misses a significant input change (0.59%). Finally, the proposed framework enables 4.21× lower execution time on average when incremental training is performed. These results demonstrate the proposed framework can effectively detect when the IL scheduler makes unreliable decisions and adapt the scheduler to achieve substantial benefits.

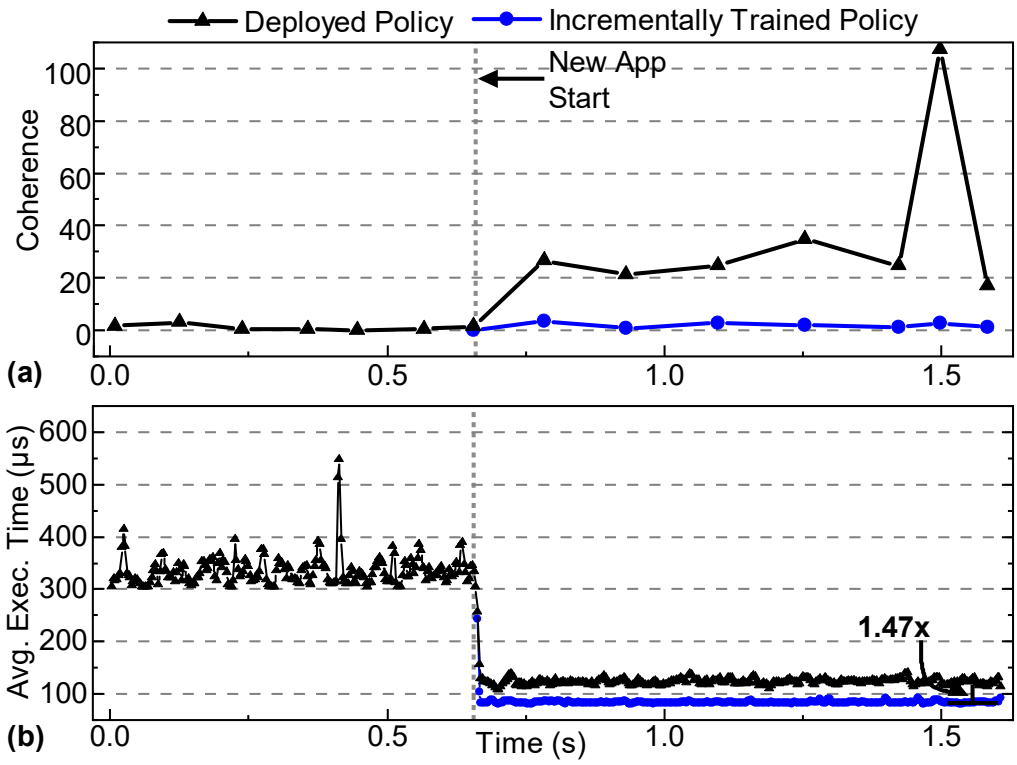

Figure 7.8: Result for the runtime monitoring framework with RL scheduler using a workload that is comprised of instances from two applications. Until the black dotted line, the first (WiFi receiver) application instances are present in the system. After that, the new application (temporal mitigation) is introduced.

7.4.3 Results Obtained with RL Scheduler

This section discusses the performance of the proposed runtime monitoring framework when applied to the RL scheduler. The RL scheduler comprises an actor policy for decision-making and a critic network for evaluation. The actor policy is responsible for scheduling decisions and is situated on the critical path of the main process. Therefore, it is implemented using a DDT, enabling scheduling in approximately 0.18 µs on the Nvidia Jetson Xavier NX board [3]. Once a scheduling

decision is made, the main process executes tasks on PEs while our framework concurrently monitors these decisions in the background. Actor-Critic policies utilize features encompassing task, application, and SoC-level information (similar to the IL scheduler described in Section 7.4.2). These policies are trained using PyTorch with an OpenAI Gym environment [200].

We conducted leave-one-out experiments with all six domain applications for a comprehensive performance evaluation. The RL scheduler generalizes well to five of these applications, even when they are excluded from training. However, it performs poorly when running the last application (temporal mitigation), indicating the RL scheduler does not generalize to this application and makes robust decisions. Our monitoring framework confirms this observation, as coherence values remain consistently low even with the arrival of new applications, except for "temporal mitigation," where coherence increases when the RL policy schedules it. Each batch (M) used in monitoring comprises 1024 samples, each divided into eight mini-batches (K) with 128 samples. The rest of this section summarizes the results.

Single Application Use Case Illustration: Like the IL experiments in Section 7.4.2, this scenario begins by executing a single application represented in the training dataset. The coherence computed by the proposed framework is low during this time, as illustrated in Figure 7.8(a). Subsequently, it is replaced with a new application, to which the RL scheduler does not generalize. The coherence value sharply increases from approximately zero to over 20 following this change, as shown in Figure 7.8(a). Correspondingly, there is a sudden decrease in execution time, as

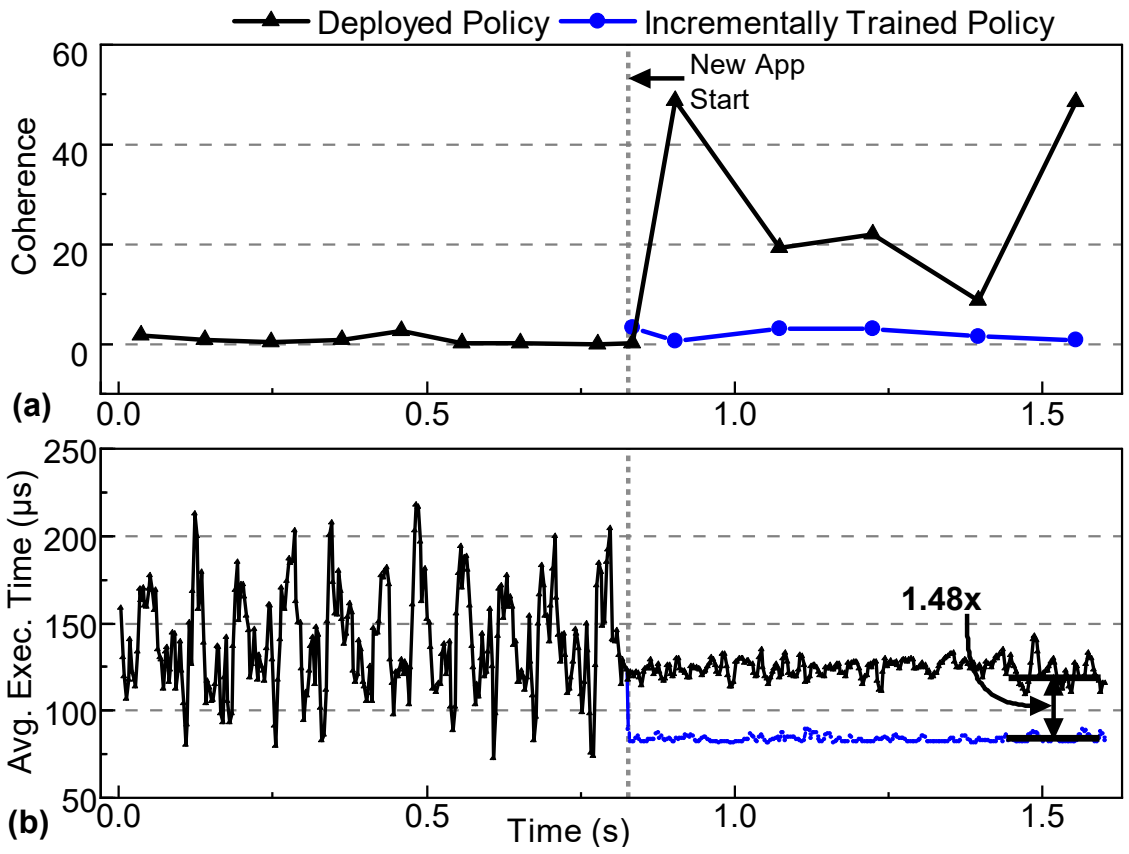

Figure 7.9: Result for the runtime monitoring framework with RL scheduler using a workload that is comprised of instances from six applications. Until the black dotted line, five out of six application instances are present in the system. After that, the sixth application (temporal mitigation) is introduced.

shown in Figure 7.8(b). This decrease occurs because the new application has inherently shorter execution times. However, it is important to note that a shorter execution time does not necessarily indicate that the RL scheduler has successfully generalized to the new application. As discussed in Section 7.3.5, the policy undergoes incremental offline training to adjust to a new application. The policy retains its performance for the initial application while being optimized for the new

one. With the incrementally trained policy, the average execution time decreases by $1.47\times$

Multiple Application Use Case Illustration: The multiple application begins running five out of six domain applications, like the IL example. Coherence during this time is low since these applications are represented during training, as shown in Figure 7.9(a). Then, a new application that is not represented in the training replaces the original mix. The proposed framework successfully captures this change, as indicated by the abrupt increase in coherence after the dotted line. The execution time varies widely during the initial period, while it has a similar average value with lower variation after launching the new application. This behavior shows that execution time is not a reliable indicator of the scheduler's generalizability. Finally, Figure 7.9(b) shows training the scheduler incrementally adapts it to the new application, enabling $1.48\times$ lower execution time.

Accuracy & Performance Summary: We conclude this section by summarizing the accuracy and performance benefits of the proposed framework with RL schedulers. Like the IL experiments, we conducted comprehensive simulations with varying application loads. For single-application examples, we assessed the monitoring framework across a total of 3685 batches (each comprising $M = 1024$ tasks). The RL scheduler fails to generalize to 666 of these batches coming from the new application. In the case of multiple applications, we evaluated our monitoring framework for over 1168 batches, with 161 batches indicating a lack of generalization. Overall, we evaluated our monitoring framework for 4853 batches. As discussed previously, the RL scheduler demonstrates inherent generalization to five applications, resulting

in fewer instances of non-generalized cases than the IL scheduler.

Table 7.1 summarizes the proposed monitoring framework's accuracy and performance benefits. It can determine whether the scheduler generalizes to the new inputs or not with 88.75% accuracy. Closing inspection reveals a 6.2% false negative rate, i.e., the frequency of failing to detect a new application. Similarly, it incorrectly flags the lack of generalization to a new application (false positive) for 5.05% of the batches. The values are lower than those obtained with the IL scheduler since there is no ground truth label during the training and monitoring of the RL scheduler. Hence, it only relies on the reward signal, a weaker indication of correctness than the ground truth. The accuracy can be improved by checking more than one batch before flagging a lack of generalization at the expense of a larger overhead. This optimization is one of the potential future research directions. Finally, when the proposed framework identifies a new application, incremental training provides, on average, a 1.32$\times$ lower execution time.

Application to Other Schedulers: The proposed framework can also be used with other scheduling algorithms besides the IL and RL schedulers considered so far. For example, Decima [108] is a graph neural network-based job scheduling algorithm targeting data clusters for streaming applications. The authors show that when the model is trained with low throughput workloads, the model poorly generalizes to high throughput workloads, leading to a 1.6$\times$ higher average execution time. This example shows a great use case for our monitoring framework. As it successfully detects the changes in the applications, specifically an unseen level of throughput in this case, it can trigger the system to take proper action, such as incremental

Table 7.2: Monitoring framework overhead for IL scheduler on Nvidia Jetson Xavier NX board [3]

Batch size (M)	**Reference scheduler (ms)**	**Loss (ms)**	**Gradient (ms)**	**Coherence (ms)**	**Total overhead (ms)**
128	3.20	1.25	7.84	0.09	12.38
256	6.40	1.92	13.70	0.21	22.23
512	12.80	3.69	26.48	0.29	43.26
1024	**25.60**	**7.50**	**50.24**	**0.40**	**83.74**

training. Indeed, when the incrementally trained version has a similar performance to a system trained with both high and low throughput workloads, the proposed framework can achieve 1.37× faster execution time. Therefore, our framework is effective for a wide range of hardware platforms and models that utilize gradient descent optimization.

7.4.4 Overhead Analysis

The proposed monitoring framework is not on the critical path since it operates in the background. An overhead analysis is still helpful since it helps determine how frequently the monitoring can be triggered. As described in Section 7.2.2, this work considers domain-specific SoC, where applications continuously process streaming inputs for extended durations after launch. The proposed monitoring framework does not need to run continuously. It can be triggered (1) when a new application launches or (2) periodically while sleeping most of the time. The overhead analysis in this section summarizes the execution overhead as a function of the batch size (M). These values determine the shortest possible monitoring period.

Table 7.2 summarizes the overhead for monitoring the IL scheduler when run-

Table 7.3: Monitoring framework overhead for RL scheduler on Nvidia Jetson Xavier NX board [3]

Batch size (M)	**Value estimates (ms)**	**Loss (ms)**	**Gradient (ms)**	**Coherence (ms)**	**Total overhead (ms)**
128	0.02	12.22	20.09	0.10	32.42
256	0.04	24.11	20.20	0.10	44.45
512	0.08	49.01	23.83	0.10	73.03
1024	**0.17**	**92.30**	**24.96**	**0.10**	**117.53**

ning on the Nvidia Jetson Xavier NX board [3]. The most time-consuming step is the gradient calculation, varying from 7.84 ms to 50.24 ms as the batch size grows from 128 to 1024. The second largest contributor is running the reference scheduler, which takes 3.2 ms to 25.6 ms. We emphasize that different components of the monitoring framework can be pipelined. For example, the reference scheduler can start running for the next task after the loss calculation begins. Hence, the total execution time in Table 7.2 is a loose upper bound. Regardless, our measurements show that the entire monitoring process takes 83.74 ms, even for the largest batch size used in our experiments, highlighted in the table. A smaller batch size can be employed to speed up the monitoring process at the expense of accuracy. The last row (bold) highlights the setting in our experiments, while evaluations shown for all batch sizes are listed in Table 7.2. This means the monitoring can be repeated in the background with this period if needed. However, in practice, we expect a longer period, on the order of seconds, since the application composition in the target SoCs rarely changes.

Table 7.3 summarizes the monitoring and detection overhead for RL schedulers as a function of the batch size (M). The loss and gradient calculations dominate

the total execution time for RL schedulers. The loss takes longer than those for the IL scheduler since loss for IL is the mean squared error, but RL requires solving Equation 7.2. As in the IL scheduler, the value estimate, loss, and gradient calculations can be pipelined. In the worst case, when all steps are performed sequentially, the total execution time varies from 32.42 ms to 117.53 ms as the batch size grows from 128 to 1024. The last row (bold) highlights the setting in our experiments. Like in the IL case, all batch sizes in the table lead to effective monitoring. Hence, the proposed framework can run as a real-time background task to monitor RL schedulers.

8 CONCLUSIONS AND FUTURE DIRECTIONS

Domain-specific AI hardware is designed to meet the unique computational demands of artificial intelligence applications, providing superior performance and efficiency compared to general-purpose processors. As AI workloads grow increasingly complex, there is a critical need for both design-time and runtime optimizations to maximize the hardware's capabilities. Design-time optimizations ensure that the hardware architecture is tailored for peak efficiency, while runtime optimizations allow the system to adapt dynamically to varying workload demands, ensuring sustained high performance. This dissertation addressed significant challenges in domain-specific hardware and optimization frameworks for AI and other computationally intensive applications, using strategies for design-time and runtime optimizations. On the design-time front, the development of a heterogeneous big-little chiplet-based In-Memory Computing (IMC) architecture, which leverages a combination of big and little IMC-based chiplets coupled with an optimal Network-on-Package (NoP) configuration, showcased the potential for improved performance and efficiency in chip design. This architecture facilitated better resource management and energy efficiency, which is essential for handling the increasing demands of AI workloads. It achieved up to 2.8$\times$ higher IMC utilization and up to 329$\times$ improvement in the energy-delay-area product (EDAP) compared to homogeneous chiplet IMC architectures. Additionally, an energy-efficient on-chip training framework, PHR, utilizing a resistive RAM-based IMC accelerator, demonstrated the capability to enhance the accuracy of mmWave-

based human pose estimation models, contributing to more precise and practical AI applications. It improved the MPJPE by 23.89% using customization with on-chip training. Then, the communication-aware sparse neural network optimization framework, CANNON, addressed the need for efficient neural network models that consider data communication bottleneck with its communication-aware sparse training technique, optimizing the overall system performance. It resulted in up to 6.8× improvement in the EDP with respect to the neural networks with pruning and state-of-the-art mapping techniques without any significant accuracy degradation.

On the runtime optimization front, the dynamic adaptive scheduling framework provided a balanced approach to task scheduling by combining fast, low-overhead algorithms with more complex, high-overhead scheduling methods, ensuring optimal task execution across varying workloads. This allowed the system to adapt dynamically to changing task requirements and resource availability in runtime. Experimental results with five streaming applications showed that DAS achieved up to 1.29× speedup and 45% lower EDP than the underlying schedulers. Finally, the runtime monitoring framework for machine learning-based task scheduling algorithms introduced a mechanism to enable robust domain-specific SoCs with runtime performance assessment and adjustment. It detected whether the trained scheduler generalizes to the current workload with 88.75% to 98.39% accuracy and enabled 1.1× to 14× faster execution time when the scheduler is incrementally trained.

In summary, this dissertation addressed the critical challenges in the optimization of design time and runtime of domain-specific AI hardware by making the

following contributions:

- A heterogeneous big-little chiplet-based IMC architecture that utilizes a big and little IMC-based chiplets [28],
- An energy-efficient on-chip training framework with a resistive RAM-based in-memory computing accelerator for the customization of mmWave-based human pose estimation models [29],
- A communication-aware sparse neural network optimization framework [24],
- A dynamic adaptive scheduling framework that combines fast, low overhead and complex, high overhead tasks scheduling algorithms [30, 31],
- A runtime monitoring framework for ML-based task scheduling algorithms to enable robust domain-specific SoCs.

8.1 Future Directions

Neural Network Optimization: Extending the communication-aware sparse neural network optimization framework to support a broader range of neural network architectures and applications would enhance its versatility and applicability. Furthermore, adapting the framework to optimize other architectures for AI acceleration can provide a generalized optimization solution.

Dynamic Adaptive Scheduling: Another promising direction is integrating advanced machine learning techniques into the dynamic scheduling framework to further enhance its adaptability and efficiency. Developing more sophisticated

algorithms that can predict workload patterns and resource demands with greater accuracy could significantly improve the performance of domain-specific hardware.

Runtime Monitoring: Expanding the runtime monitoring framework to include more comprehensive metrics and feedback mechanisms would provide deeper insights into system performance and enable real-time optimizations. This can involve developing new monitoring tools that capture performance indicators or integrating advanced analytics to provide more granular, actionable feedback. Another direction for monitoring may include the consideration of dynamic thermal power management frameworks that affect the performance of the underlying hardware at runtime.

BIBLIOGRAPHY

[1] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–27, 2019.

[2] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[3] Nvidia. Jetson Xavier NX Developer Kit. `https://www.nvidia.com/en-us/autonomous-machines/embedded-Syst./jetson-xavier-nx/`, 2014. [Online; accessed 29 Sep. 2022].

[4] Gokul Krishnan, Sumit K Mandal, Manvitha Pannala, Chaitali Chakrabarti, Jae-Sun Seo, Umit Y Ogras, and Yu Cao. Siam: Chiplet-based scalable in-memory acceleration with mesh for deep neural networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–24, 2021.

[5] Anirban Shafiee, Ali Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. ISAAC: A Convolutional Neural Network Accelerator with in-situ Analog Arithmetic in Crossbars. *ACM/IEEE ISCA*, 2016.

[6] Nan Wu, Lei Deng, Guoqi Li, and Yuan Xie. Core placement optimization for multi-chip many-core neural network systems with reinforcement learning. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 26(2):1–27, 2020.

[7] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[8] Li Deng, Jinyu Li, Jui-Ting Huang, Kaisheng Yao, Dong Yu, Frank Seide, Michael Seltzer, Geoff Zweig, Xiaodong He, Jason Williams, et al. Recent advances in deep learning for speech research at microsoft. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 8604–8608. IEEE, 2013.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[10] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE international conference on computer vision*, pages 2722–2730, 2015.

[11] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *nature*, 542(7639):115–118, 2017.

[12] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

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

[14] Deepak Ghimire, Dayoung Kil, and Seong-heum Kim. A survey on efficient convolutional neural networks and hardware acceleration. *Electronics*, 11(6):945, 2022.

[15] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks. *arXiv preprint arXiv:1802.04680*, 2018.

[16] Mucong Ding, Kezhi Kong, Jingling Li, Chen Zhu, John Dickerson, Furong Huang, and Tom Goldstein. Vq-gnn: A universal framework to scale up graph neural networks using vector quantization. *Advances in Neural Information Processing Systems*, 34:6733–6746, 2021.

[17] Zhenhua Liu, Yunhe Wang, Kai Han, Wei Zhang, Siwei Ma, and Wen Gao. Post-training quantization for vision transformer. *Advances in Neural Information Processing Systems*, 34:28092–28103, 2021.

[18] Jie-Fang Zhang, Ching-En Lee, Chester Liu, Yakun Sophia Shao, Stephen W Keckler, and Zhengya Zhang. Snap: An efficient sparse neural acceleration processor for unstructured sparse deep neural network inference. *IEEE Journal of Solid-State Circuits*, 56(2):636–647, 2020.

[19] Mostafa Mahmoud, Isak Edo, Ali Hadi Zadeh, Omar Mohamed Awad, Gennady Pekhimenko, Jorge Albericio, and Andreas Moshovos. Tensordash: Exploiting sparsity to accelerate deep neural network training. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 781–795. IEEE, 2020.

[20] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. Learning n: m fine-grained structured sparse neural networks from scratch. *arXiv preprint arXiv:2102.04010*, 2021.

[21] Zhi-Gang Liu, Paul N Whatmough, and Matthew Mattina. Systolic tensor array: An efficient structured-sparse gemm accelerator for mobile cnn inference. *IEEE Computer Architecture Letters*, 19(1):34–37, 2020.

[22] Chaoyang Zhu, Kejie Huang, Shuyuan Yang, Ziqi Zhu, Hejia Zhang, and Haibin Shen. An efficient hardware accelerator for structured sparse convolutional neural networks on fpgas. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(9):1953–1965, 2020.

[23] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing dnn pruning to the underlying

hardware parallelism. *ACM SIGARCH Computer Architecture News*, 45(2):548–560, 2017.

[24] A Alper Goksoy, Guihong Li, Sumit K Mandal, Umit Y Ogras, and Radu Marculescu. Cannon: Communication-aware sparse neural network optimization. *IEEE Transactions on Emerging Topics in Computing*, 2023.

[25] Jian Meng, Li Yang, Xiaochen Peng, Shimeng Yu, Deliang Fan, and Jae-Sun Seo. Structured pruning of rram crossbars for efficient in-memory computing acceleration of deep neural networks. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 68(5):1576–1580, 2021.

[26] Chen Yang, Yishuo Meng, Kaibo Huo, Jiawei Xi, and Kuizhi Mei. A sparse cnn accelerator for eliminating redundant computations in intra-and inter-convolutional/pooling layers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(12):1902–1915, 2022.

[27] Nvidia Ampere Architecture In-Depth. `https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth`. accessed date: Jun. 28, 2024.

[28] Gokul Krishnan, A Alper Goksoy, Sumit K Mandal, Zhenyu Wang, Chaitali Chakrabarti, Jae-sun Seo, Umit Y Ogras, and Yu Cao. Big-little chiplets for in-memory acceleration of dnns: A scalable heterogeneous architecture. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2022.

[29] A. Alper Goksoy, Sizhe An, and Umit Y Ogras. Energy-efficient on-chip training for customized home-based rehabilitation systems. In *Proceedings of the 60th IEEE/ACM Design Automation Conference*, 2023.

[30] A Alper Goksoy, Sahil Hassan, Anish Krishnakumar, Radu Marculescu, Ali Akoglu, and Umit Y Ogras. Theoretical validation and hardware implementation of dynamic adaptive scheduling for heterogeneous systems on chip. *Journal of Low Power Electronics and Applications*, 13(4):56, 2023.

[31] A Alper Goksoy, Anish Krishnakumar, Md Sahil Hassan, Allen J Farcas, Ali Akoglu, Radu Marculescu, and Umit Y Ogras. Das: Dynamic adaptive scheduling for energy-efficient heterogeneous socs. *IEEE Embedded Systems Letters*, 14(1):51–54, 2021.

[32] Gauthaman Kim, Jinwoo Murali, Heechun Park, Eric Qin, Hyoukjun Kwon, Venkata Chaitanya Krishna Chekuri, Nael Mizanur Rahman, Nihar Dasari, Arvind Singh, Minah Lee, et al. Architecture, Chip, and Package Codesign Flow for Interposer-Based 2.5D Chiplet Integration Enabling Heterogeneous IP Reuse. *IEEE TVLSI*, 2020.

[33] Eric Vivet, Pascal Guthmuller, Yvain Thonnart, Gael Pillonnet, César Fuguet, Ivan Miro-Panades, Guillaume Moritz, Jean Durupt, Christian Bernard, Didier Varreau, et al. IntAct: A 96-core Processor with Six Chiplets 3D-stacked on an Active Interposer with Distributed Interconnects and Integrated Power Management. *IEEE JSSC*, 2020.

[34] Jingyang Pal, Saptadeep Liu, Irina Alam, Nicholas Cebry, Haris Suhail, Shi Bu, Subramanian S Iyer, Sudhakar Pamarti, Rakesh Kumar, and Puneet Gupta. Designing a 2048-Chiplet, 14336-Core Waferscale Processor. In *ACM/IEEE DAC*, 2021.

[35] Leila Ma, Yenai Delshadtehrani, Cansu Demirkiran, José L Abellán, and Aiav Joshi. TAP-2.5 D: A Thermally-Aware Chiplet Placement Methodology for 2.5 D Systems. In *IEEE DATE*, 2021.

[36] Hao Zheng, Ke Wang, and Ahmed Louri. A Versatile and Flexible Chiplet-based System Design for Heterogeneous Manycore Architectures. In *ACM/IEEE DAC*, 2020.

[37] Mengdi Wang, Ying Wang, Cheng Liu, and Lei Zhang. Network-on-Interposer Design for Agile Neural-Network Processor Chip Customization. In *ACM/IEEE DAC*, 2021.

[38] Uneeb Rathore, Sumeet Singh Nagi, Subramanian Iyer, and Dejan Marković. A 16nm 785gmacs/j 784-core digital signal processor array with a multi-layer switch box interconnect, assembled as a 2× 2 dielet with 10µm-pitch inter-dielet i/o for runtime multi-program reconfiguration. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 52–54. IEEE, 2022.

[39] Jieming Bharadwaj, Srikant Yin, Bradford Beckmann, and Tushar Krishna. Kite: A Family of Heterogeneous Interposer Topologies Enabled via Accurate Interconnect Modeling. In *ACM/IEEE DAC*, 2020.

[40] Zhanhong Tan, Hongyu Cai, Runpei Dong, and Kaisheng Ma. Nn-baton: Dnn workload orchestration and chiplet granularity exploration for multichip accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1013–1026. IEEE, 2021.

[41] Walker J Turner, John W Poulton, John M Wilson, Xi Chen, Stephen G Tell, Matthew Fojtik, Thomas H Greer, Brian Zimmer, Sanquan Song, Nikola Nedovic, et al. Ground-referenced signaling for intra-chip and short-reach chip-to-chip interconnects. In *2018 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–8. IEEE, 2018.

[42] Min SH Aung et al. The automatic detection of chronic pain-related expression: requirements, challenges and the multimodal emopain dataset. *IEEE Trans. on Affective Computing*, 7(4):435–451, 2015.

[43] Aleksandar Vakanski et al. A data set of human body movements for physical rehabilitation exercises. *Data*, 3(1):2, 2018.

[44] Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Realtime multi-person 2d pose estimation using part affinity fields. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 7291–7299, 2017.

[45] Ke Sun, Bin Xiao, Dong Liu, and Jingdong Wang. Deep high-resolution representation learning for human pose estimation. In *Proc. IEEE/CVF Conf. on Computer Vision and Pattern Recognition*, pages 5693–5703, 2019.

[46] Sizhe An and Umit Y Ogras. Mars: mmwave-based assistive rehabilitation system for smart healthcare. *ACM Trans. on Embedded Computing Syst.*, 20:1–22, 2021.

[47] Arindam Sengupta, Feng Jin, Renyuan Zhang, and Siyang Cao. mm-pose: Real-time human skeletal posture estimation using mmwave radars and cnns. *IEEE Sensors Journal*, 20(17):10032–10044, 2020.

[48] Hongfei Xue et al. mmmesh: Towards 3d real-time dynamic human mesh construction using millimeter-wave. In *Proc. 19th Int. Conf. on Mobile Syst., Applications, and Services*, pages 269–282, 2021.

[49] Shinhyun Choi et al. Sige epitaxial memory for neuromorphic computing with reproducible high performance based on engineered dislocations. *Nature Materials*, 17(4):335–340, 2018.

[50] Xuehai Song, Linghao Qian, Hai Li, and Yiran Chen. Pipelayer: A Pipelined Reram-based Accelerator for Deep Learning. In *IEEE HPCA*, pages 541–552, 2017.

[51] Alessandro Grossi et al. Resistive ram endurance: Array-level characterization and correction techniques targeting deep learning applications. *IEEE Trans. on Electron Devices*, 66(3):1281–1288, 2019.

[52] Biresh Kumar Joardar et al. Learning to train cnns on faulty reram-based manycore accelerators. *ACM Trans. on Embedded Computing Syst.*, 20(5s):1–23, 2021.

[53] Xiaochen Peng et al. Dnn+ neurosim v2. 0: An end-to-end benchmarking framework for compute-in-memory accelerators for on-chip training. *IEEE TCAD*, 40(11):2306–2319, 2020.

[54] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.

[55] Babak Hassibi, David G Stork, and Gregory J Wolff. Optimal brain surgeon and general network pruning. In *IEEE international conference on neural networks*, pages 293–299. IEEE, 1993.

[56] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.

[57] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In *International Conference on Machine Learning*, pages 2498–2507. PMLR, 2017.

[58] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable Training of Artificial Neural Networks with Adaptive Sparse Connectivity Inspired by Network Science. *Nature communications*, 9(1):1–12, 2018.

[59] Foroozan Karimzadeh, Ningyuan Cao, Brian Crafton, Justin Romberg, and Arijit Raychowdhury. Hardware-aware pruning of dnns using lfsr-generated pseudo-random indices. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2020.

[60] Chaoqun Chu, Yanzhi Wang, Yilong Zhao, Xiaolong Ma, Shaokai Ye, Yunyan Hong, Xiaoyao Liang, Yinhe Han, and Li Jiang. Pim-prune: Fine-grain dcnn pruning for crossbar-based process-in-memory architecture. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[61] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8612–8620, 2019.

[62] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(3):1–18, 2017.

[63] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 806–814, 2015.

[64] Wojciech Romaszkan, Tianmu Li, and Puneet Gupta. 3pxnet: Pruned-permuted-packed xnor networks for edge machine learning. *ACM Transactions on Embedded Computing Systems (TECS)*, 19(1):1–23, 2020.

[65] Valentin Radu, Kuba Kaszyk, Yuan Wen, Jack Turner, José Cano, Elliot J Crowley, Björn Franke, Amos Storkey, and Michael O'Boyle. Performance Aware Convolutional Neural Network Channel Pruning for Embedded GPUs. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 24–34, 2019.

[66] Gokul Krishnan, Xiaocong Du, and Yu Cao. Structural pruning in deep neural networks: A small-world approach. *arXiv preprint arXiv:1911.04453*, 2019.

[67] Geng Yuan, Payman Behnam, Yuxuan Cai, Ali Shafiee, Jingyan Fu, Zhiheng Liao, Zhengang Li, Xiaolong Ma, Jieren Deng, Jinhui Wang, Mahdi Bojnordi, Yanzhi Wang, and Caiwen Ding. Tinyadc: Peripheral Circuit-aware Weight Pruning Framework for Mixed-signal DNN Accelerators. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 926–931. IEEE, 2021.

[68] Siling Yang, Weijian Chen, Xuechen Zhang, Shuibing He, Yanlong Yin, and Xian-He Sun. Auto-prune: Automated DNN Pruning and Mapping for ReRAM-based Accelerator. In *Proceedings of the ACM International Conference on Supercomputing*, pages 304–315, 2021.

[69] Jilan Lin, Zhenhua Zhu, Yu Wang, and Yuan Xie. Learning the sparsity for reram: Mapping and pruning sparse neural network for reram based accelerator. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pages 639–644, 2019.

[70] Ling Liang, Lei Deng, Yueling Zeng, Xing Hu, Yu Ji, Xin Ma, Guoqi Li, and Yuan Xie. Crossbar-aware neural network pruning. *IEEE Access*, 6:58324–58337, 2018.

[71] Tzu-Hsien Yang, Hsiang-Yun Cheng, Chia-Lin Yang, I-Ching Tseng, Han-Wen Hu, Hung-Sheng Chang, and Hsiang-Pang Li. Sparse reram engine: Joint exploration of activation and weight sparsity in compressed neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 236–249, 2019.

[72] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244, 2009.

[73] Youcef Saad. Sparskit: a basic tool kit for sparse matrix computations-version 2. 1994.

[74] Shaden Smith and George Karypis. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, pages 1–7, 2015.

[75] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Trans. on Parallel and Distrib. Syst.*, 13(3):260–274, 2002.

[76] Luiz F Bittencourt, Rizos Sakellariou, and Edmundo RM Madeira. DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish

Time Algorithm. In *IEEE Euromicro Conf. on Parallel, Distrib. and Network-based Process.*, pages 27–34, 2010.

[77] Chandandeep Singh Pabla. Completely Fair Scheduler. *Linux Journal*, (184), 2009.

[78] Mihaela-Andreea Vasile, Florin Pop, Radu-Ioan Tutueanu, Valentin Cristea, and Joanna Kołodziej. Resource-Aware Hybrid Scheduling Algorithm in Heterogeneous Distributed Computing. *Future Generation Computer Systems*, 51:61–71, 2015.

[79] Hoeseok Yang and Soonhoi Ha. Ilp based data parallel multi-task mapping/scheduling technique for mpsoc. In *2008 International SoC Design Conference*, volume 1, pages I–134. IEEE, 2008.

[80] Luca Benini, Davide Bertozzi, and Michela Milano. Resource Management Policy Handling Multiple Use-Cases in MpSoC Platforms using Constraint Programming. In *Logic Program.: 24th Int. Conf.*, pages 470–484, 2008.

[81] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple Linux Utility for Resource Management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[82] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[83] Kallia Chronaki et al. Task Scheduling Techniques for Asymmetric Multi-core Systems. *IEEE Trans. on Parallel and Distrib. Systems*, 28(7):2074–2087, 2016.

[84] Junyan Zhou. Real-time Task Scheduling and Network Device Security for Complex Embedded Systems based on Deep Learning Networks. *Microprocessors and Microsystems*, 79:103282, 2020.

[85] Alireza Namazi, Saeed Safari, and Siamak Mohammadi. CMV: Clustered Majority Voting Reliability-aware Task Scheduling for Multicore Real-time Systems. *IEEE Trans. on Reliability*, 68(1):187–200, 2018.

[86] Guoqi Xie, Gang Zeng, Liangjiao Liu, Renfa Li, and Keqin Li. Mixed Real-Time Scheduling of Multiple DAGs-based Applications on Heterogeneous Multi-core Processors. *Microprocessors and Microsystems*, 47:93–103, 2016.

[87] Tang Xiaoyong, Kenli Li, Zeng Zeng, and Bharadwaj Veeravalli. A Novel Security-Driven Scheduling Algorithm for Precedence-Constrained Tasks in Heterogeneous Distributed Systems. *IEEE Transactions on Computers*, 60(7):1017–1029, 2010.

[88] Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 7(5):506–521, 1996.

[89] Rizos Sakellariou and Henan Zhao. A Hybrid Heuristic for DAG Scheduling on Heterogeneous Systems. In *Int. Parallel and Distributed Processing Symposium*, page 111. IEEE, 2004.

[90] Andreas Prodromou, Ashish Venkat, and Dean M Tullsen. Agon: A scalable competitive scheduler for large heterogeneous systems. *arXiv preprint arXiv:2109.00665*, 2021.

[91] Anish Krishnakumar et al. Runtime Task Scheduling using Imitation Learning for Heterogeneous Many-core Systems. *IEEE Trans. on CAD of Integr. Circuits and Syst.*, 39(11):4064–4077, 2020.

[92] Ravindra Jejurikar and Rajesh Gupta. Energy-aware Task Scheduling with Task Synchronization for Embedded Real-time Systems. *IEEE Trans. on CAD of Integr. Circuits and Syst.*, 25(6):1024–1037, 2006.

[93] Poopak Azad and Nima Jafari Navimipour. An Energy-aware Task Scheduling in the Cloud Computing using a Hybrid Cultural and Ant Colony Opti-

mization Algorithm. *Int. Journal of Cloud Applications and Computing*, 7(4):20–40, 2017.

[94] Sanjeev Baskiyar and Rabab Abdel-Kader. Energy Aware DAG Scheduling on Heterogeneous Systems. *Cluster Computing*, 13(4):373–383, 2010.

[95] Vishnu Swaminathan and Krishnendu Chakrabarty. Real-Time Task Scheduling for Energy-Aware Embedded Systems. *Journal of the Franklin Institute*, 338(6):729–750, 2001.

[96] Othon Tomoutzoglou, Dimitris Mbakoyiannis, George Kornaros, and Marcello Coppola. Efficient job offloading in heterogeneous systems through hardware-assisted packet-based dispatching and user-level runtime infrastructure. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(5):1017–1030, 2019.

[97] Dimitrios Mbakoyiannis, Othon Tomoutzoglou, and George Kornaros. Energy-performance considerations for data offloading to fpga-based accelerators over pcie. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(1):1–24, 2018.

[98] Achim Streit. A Self-tuning Job Scheduler Family with Dynamic Policy Switching. In *Workshop on Job Scheduling Strategies for Parallel Process.*, pages 1–23. Springer, 2002.

[99] Mohammad I Daoud and Nawwaf Kharma. A Hybrid Heuristic–genetic Algorithm for Task Scheduling in Heterogeneous Processor Networks. *Journal of Parallel and Distrib. Computing*, 71(11):1518–1531, 2011.

[100] Cristina Boeres, Alexandre Lima, and Vinod EF Rebello. Hybrid Task Scheduling: Integrating Static and Dynamic Heuristics. In *Proc. of 15th Symp. on Computer Arch. and High Perform. Computing*, pages 199–206, 2003.

[101] Renato Cordeiro et al. Ecg-based authentication using timing-aware domain-specific architecture. *IEEE trans. on computer-aided design of integr. circuits and syst.*, 39(11):3373–3384, 2020.

[102] Jörg Fickenscher, Frank Hannig, and Jürgen Teich. Dsl-based acceleration of automotive environment perception and mapping algorithms for embedded cpus, gpus, and fpgas. In *Archit. of Comput. Syst.*, pages 71–86, 2019.

[103] Daniele Parravicini, Davide Conficconi, Emanuele Del Sozzo, Christian Pilato, and Marco D Santambrogio. Cicero: A domain-specific architecture for efficient regular expression matching. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–24, 2021.

[104] Teng Wang et al. Via: A novel vision-transformer accelerator based on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):4088–4099, 2022.

[105] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D Anger, and Chung-Yee Lee. Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. *SIAM Journal on Computing*, 18(2):244–257, 1989.

[106] Hamid Arabnejad and Jorge G Barbosa. List Scheduling Algorithm for Heterogeneous Systems by an Optimistic Cost Table. *IEEE Trans. on Parallel and Distributed Systems*, 25(3):682–694, 2013.

[107] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

[108] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. In *ACM Special Interest Group on Data Communication*, pages 270–288. 2019.

[109] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource Management with Deep Reinforcement Learning. In *ACM Workshop on Hot Topics in Networks*, pages 50–56, 2016.

[110] Anderson L Sartor, Anish Krishnakumar, Samet E Arda, Umit Y Ogras, and Radu Marculescu. HiLITE: Hierarchical and Lightweight Imitation Learning for Power Management of Embedded SoCs. *IEEE Computer Architecture Letters*, 19(1):63–67, 2020.

[111] Zhao Tong, Xiaomei Deng, Hongjian Chen, Jing Mei, and Hong Liu. QL-HEFT: A Novel Machine Learning Scheduling Scheme Base on Cloud Computing Environment. *Neural Comput. and Appl.*, 32:5553–5570, 2020.

[112] Xiaojie Wang, Zhaolong Ning, Song Guo, and Lei Wang. Imitation Learning Enabled Task Scheduling for Online Vehicular Edge Computing. *IEEE Transactions on Mobile Computing*, 21(2):598–611, 2020.

[113] Zhiming Hu, James Tu, and Baochun Li. Spear: Optimized dependency-aware task scheduling with deep reinforcement learning. In *2019 IEEE 39th Int. Conf. on Distrib. Comput. Syst. (ICDCS)*, pages 2037–2046, 2019.

[114] Toygun Basaklar, A. Alper Goksoy, Anish Krishnakumar, Suat Gumussoy, and Umit Y. Ogras. Dtrl: Decision tree-based multi-objective reinforcement learning for runtime task scheduling in domain-specific system-on-chips. *ACM Trans. Embed. Comput. Syst.*, 22(5s), 2023.

[115] Azalia Mirhoseini et al. Device Placement Optimization with Reinforcement Learning. In *International Conference on Machine Learning-Volume 70*, pages 2430–2439. JMLR. org, 2017.

[116] Yuan Wen, Zheng Wang, and Michael FP O'Boyle. Smart Multi-Task Scheduling for OpenCL Programs on CPU/GPU Heterogeneous Platforms. In *International Conf. on High Performance Computing*, pages 1–10, 2014.

[117] Ankur Mallick, Kevin Hsieh, Behnaz Arzani, and Gauri Joshi. Matchmaker: Data drift mitigation in machine learning for large-scale systems. *Proceedings of Machine Learning and Systems*, 4:77–94, 2022.

[118] Samuel Ackerman, Eitan Farchi, Orna Raz, Marcel Zalmanovici, and Parijat Dube. Detection of data drift and outliers affecting machine learning model performance over time. *arXiv preprint arXiv:2012.09258*, 2020.

[119] Limin Yang et al. CADE: Detecting and explaining concept drift samples for security applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2327–2344, 2021.

[120] Denis Moreira Dos Reis, Peter Flach, Stan Matwin, and Gustavo Batista. Fast unsupervised online drift detection using incremental kolmogorov-smirnov test. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1545–1554, 2016.

[121] Ahsanul Haque, Latifur Khan, and Michael Baron. Sand: Semi-supervised adaptive novel class detection and classification over data stream. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[122] Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *International Conference on Learning Representations*, 2019.

[123] Zhiao Shi, Emmanuel Jeannot, and Jack J Dongarra. Robust task scheduling in non-deterministic heterogeneous computing systems. In *2006 IEEE int. conf. on cluster comput.*, pages 1–10. IEEE, 2006.

[124] Saining Xie, Alexander Kirillov, Ross Girshick, and Kaiming He. Exploring Randomly Wired Neural Networks for Image Recognition. In *IEEE/CVF ICCV*, 2019.

[125] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan,

et al. Searching for Mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1314–1324, 2019.

[126] Gokul Krishnan, Sumit K Mandal, Chaitali Chakrabarti, Jae sun Seo, Umit Y Ogras, and Yu Cao. Interconnect-aware Area and Energy Optimization for In-Memory Acceleration of DNNs. *IEEE Design & Test*, 37(6):79–87, 2020.

[127] Sumit K Mandal, Gokul Krishnan, Chaitali Chakrabarti, Jae-Sun Seo, Yu Cao, and Umit Y Ogras. A Latency-Optimized Reconfigurable NoC for In-Memory Acceleration of DNNs. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 10(3):362–375, 2020.

[128] Sumit K Mandal, Gokul Krishnan, A. Alper Goksoy, Gopikrishnan Ravindran Nair, Yu Cao, and Umit Y Ogras. COIN: Communication-Aware In-Memory Acceleration for Graph Convolutional Networks. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2022.

[129] Liangzhen Kwon, Hyoukjun Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. Heterogeneous Dataflow Accelerators for Multi-DNN Workloads. In *IEEE HPCA*, 2021.

[130] Ranggi Hwang, Taehun Kim, Youngeun Kwon, and Minsoo Rhu. Centaur: A Chiplet-based, Hybrid Sparse-Dense Accelerator for Personalized Recommendations. In *2020 ACM/IEEE 47th Annual ISCA*, pages 968–981. IEEE, 2020.

[131] Yuan Li, Ahmed Louri, and Avinash Karanth. Scaling deep-learning inference with chiplet-based architecture and photonic interconnects. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 931–936. IEEE, 2021.

[132] Yuan Li, Ahmed Louri, and Avinash Karanth. Spacx: Silicon photonics-based scalable chiplet accelerator for dnn inference. In *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, pages 1–13, 2022.

[133] Yuan Li, Ke Wang, Hao Zheng, Ahmed Louri, and Avinash Karanth. Ascend: A scalable and energy-efficient deep neural network accelerator with photonic interconnects. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2022.

[134] Ravi Mahajan, Robert Sankman, Neha Patel, Dae-Woo Kim, Kemal Aygun, Zhiguo Qian, Yidnekachew Mekonnen, Islam Salama, Sujit Sharan, Deepti Iyengar, et al. Embedded multi-die interconnect bridge (emib)–a high density, high bandwidth packaging interconnect. In *2016 IEEE 66th Electronic Components and Technology Conference (ECTC)*, pages 557–565. IEEE, 2016.

[135] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 45–54, New York, NY, USA, 2017. ACM.

[136] Trevor Mudge. Power: A First-class Architectural Design Constraint. *Computer*, 34(4):52–58, 2001.

[137] MICRON. Datasheet for DDR4 Model. `https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/4gb_ddr4_dram_2e0d.pdf` Accessed 29 Mar. 2021, 2014.

[138] Saransh Imani, Mohsen Gupta, Yeseong Kim, and Tajana Rosing. FloatPIM: In-memory Acceleration of Deep Neural Network Training with High Precision. In *ACM/IEEE ISCA*, 2019.

[139] David Greenhill et al. 3.3 A 14nm 1GHz FPGA with 2.5 D Transceiver Integration. In *2017 IEEE ISSCC*. IEEE, 2017.

[140] William J Poulton, John W Dally, Xi Chen, John G Eyles, Thomas H Greer, Stephen G Tell, and C Thomas Gray. A 0.54 pJ/b 20Gb/s Ground-Referenced Single-Ended Short-Haul Serial Link in 28nm CMOS for Advanced Packaging Applications. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, 2013.

[141] Michael Su, Bryan Black, Yu-Hsiang Hsiao, Chien-Lin Changchien, Chang-Chi Lee, and Hung-Jen Chang. 2.5 d ic micro-bump materials characterization and imcs evolution under reliability stress conditions. In *2016 IEEE 66th Electronic Components and Technology Conference (ECTC)*, pages 322–328. IEEE, 2016.

[142] Chester Liu, Jacob Botimer, and Zhengya Zhang. A 256gb/s/mm-shoreline aib-compatible 16nm finfet cmos chiplet for 2.5 d integration with stratix 10 fpga on emib and tiling on silicon interposer. In *2021 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–2. IEEE, 2021.

[143] Saurabh Sinha, Greg Yeric, Vikas Chandra, Brian Cline, and Yu Cao. Exploring Sub-20nm FinFET Design with Predictive Technology Models. In *DAC 2012*, pages 283–288. IEEE, 2012.

[144] CHIPS Alliance (INTEL). EMIB PHY RTL. `https://github.com/chipsalliance/aib-phy-hardware`, 2021. [Online; Accessed 20-April-2022].

[145] Sizhe An, Yin Li, and Umit Ogras. mRI: Multi-modal 3d human pose estimation dataset using mmwave, RGB-d, and inertial sensors. In *Proc. of NeurIPS Datasets and Benchmarks Track*, 2022.

[146] Puck ME Schuivens et al. Impact of the covid-19 lockdown strategy on vascular surgery practice: more major amputations than usual. *Annals of Vascular Surgery*, 69:74–79, 2020.

[147] Texas Instruments. Datasheet. `https://www.ti.com/lit/ds/symlink/iwr1443.pdf`, 2014. [Online; accessed 8 Apr. 2022].

[148] Microsoft. Kinect sensor. `https://developer.microsoft.com/en-us/windows/kinect/`, 2014. [Online; accessed 29 Aug. 2022].

[149] Catalin Ionescu, Dragos Papava, Vlad Olaru, and Cristian Sminchisescu. Human3. 6m: Large scale datasets and predictive methods for 3d human

sensing in natural environments. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 36(7):1325–1339, 2013.

[150] Chinthaka Gamanayake, Lahiru Jayasinghe, Benny Kai Kiat Ng, and Chau Yuen. Cluster Pruning: An Efficient Filter Pruning Method for Edge AI Vision Applications. *IEEE Journal of Selected Topics in Signal Processing*, 14(4):802–816, 2020.

[151] Pai-Yu Chen et al. NeuroSim: A Circuit-level Macro Model for Benchmarking Neuro-inspired Architectures in Online Learning. *IEEE TCAD*, 37(12):3067–3080, 2018.

[152] Nan Jiang et al. A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator. In *IEEE ISPASS*, pages 86–96, 2013.

[153] Kaiming He et al. Deep Residual Learning for Image Recognition. In *IEEE CVPR*, pages 770–778, 2016.

[154] Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. Densenet: Implementing efficient convnet descriptor pyramids. *arXiv preprint arXiv:1404.1869*, 2014.

[155] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[156] Ya Le and Xuan Yang. Tiny imagenet visual recognition challenge. *CS 231N*, 7(7):3, 2015.

[157] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[158] Yann LeCun. The mnist database of handwritten digits. *http://yann.lecun.com/exdb/mnist/*, 1998.

[159] Erdős Paul and Rényi Alfréd. On random graphs i. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.

[160] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[161] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.

[162] Ximing Qiao et al. Atomlayer: A Universal Reram-based CNN Accelerator with Atomic Layer Computation. In *IEEE/ACM DAC*, 2018.

[163] Jonathan Frankle. Openlth: A framework for lottery tickets and beyond. `https://github.com/facebookresearch/open_lth`, 2020.

[164] John L Hennessy and David A Patterson. A New Golden Age for Computer Architecture. *Commun. of the ACM*, 62(2):48–60, 2019.

[165] D Green et al. Heterogeneous Integration at DARPA: Pathfinding and Progress in Assembly Approaches. *ECTC, May*, 2018.

[166] RF Convergence: From the Signals to the Computer by Dr. Tom Rondeau (Microsystems Technology Office, DARPA). `https://futurenetworks.ieee.org/images/files/pdf/FirstResponder/Tom-Rondeau-DARPA.pdf`. [Online; last accessed 15-May-2022.].

[167] Kasra Moazzemi, Biswadip Maity, Saehanseul Yi, Amir M Rahmani, and Nikil Dutt. HESSLE-FREE: Heterogeneous Systems Leveraging Fuzzy Control for Runtime Resource Management. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–19, 2019.

[168] Joshua Mack, Sahil Hassan, Nirmal Kumbhare, Miguel Castro Gonzalez, and Ali Akoglu. Cedr-a compiler-integrated, extensible dssoc runtime. *ACM Transactions on Embedded Computing Systems (TECS)*, 2022.

[169] Philippe Magarshack and Pierre G Paulin. System-on-chip Beyond the Nanometer Wall. In *Proc. of Design Automation Conference*, pages 419–424, 2003.

[170] Young-Kyu Choi et al. In-depth Analysis on Microarchitectures of Modern Heterogeneous CPU-FPGA Platforms. *ACM Transactions on Reconfigurable Technology and Systems*, 12(1):1–20, 2019.

[171] Anish Krishnakumar, Umit Ogras, Radu Marculescu, Mike Kishinevsky, and Trevor Mudge. Domain-specific architectures: Research problems and promising approaches. *ACM Transactions on Embedded Computing Systems*, 22(2):1–26, 2023.

[172] Tobias Beisel, Tobias Wiersema, Christian Plessl, and André Brinkmann. Cooperative Multitasking for Heterogeneous Accelerators in the Linux Completely Fair Scheduler. In *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 223–226, 2011.

[173] Mary L McHugh. The Chi-square Test of Independence. *Biochemia Medica*, 23(2):143–149, 2013.

[174] ZeBu server 4. `https://www.synopsys.com/verification/emulation/zebu-server.html`. accessed date: Jan. 2, 2020.

[175] Veloce2 emulator. `https://www.mentor.com/products/fv/emulation-systems/veloce`. accessed date: Jan. 2, 2020.

[176] Joshua Mack, Nirmal Kumbhare, Anish NK, Umit Y Ogras, and Ali Akoglu. User-Space Emulation Framework for Domain-Specific SoC Design. In *2020 IEEE Int. Parallel and Distrib. Process. Symp. Workshops*), pages 44–53, 2020.

[177] Zynq ZCU102 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html, Accessed 10 March 2023.

[178] Samet Egemen Arda et al. DS3: A System-Level Domain-Specific System-on-Chip Simulation Framework. *IEEE Trans. on Computers*, 69(8):1248–1262, 2020.

[179] Xilinx - Accurate Power Measurement. `https://www.xilinx.com/developer/articles/accurate-design-power-measurement.html`. accessed date: Feb. 11, 2023.

[180] Sysfs Interface in Linux. `https://www.kernel.org/doc/Documentation/hwmon/sysfs-interface`. accessed date: Feb. 11, 2023.

[181] J.D. Ullman. NP-Complete Scheduling Problems. *Journal of Computer and System Sciences*, 10(3):384 – 393, 1975.

[182] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1979.

[183] Yue Wu et al. Large scale incremental learning. In *Proc. of the IEEE/CVF Conf. on Comput. Vis. and Pattern Recognit.*, 2019.

[184] Francisco M Castro, Manuel J Marín-Jiménez, Nicolás Guil, Cordelia Schmid, and Karteek Alahari. End-to-end incremental learning. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 233–248, 2018.

[185] Weitang Liu, Xiaoyun Wang, John Owens, and Yixuan Li. Energy-based out-of-distribution detection. *Adv. in Neural Inf. Process. Syst.*, 33, 2020.

[186] Satrajit Chatterjee and Piotr Zielinski. On the generalization mystery in deep learning. *arXiv preprint arXiv:2203.10036*, 2022.

[187] Dimitris Kalimeris, Gal Kaplun, Preetum Nakkiran, Benjamin Edelman, Tristan Yang, Boaz Barak, and Haofeng Zhang. Sgd on neural networks learns functions of increasing complexity. *Advances in neural inf. process. syst.*, 32, 2019.

[188] Stanislav Fort, Paweł Krzysztof Nowak, Stanislaw Jastrzebski, and Srini Narayanan. Stiffness: A new perspective on generalization in neural networks. *arXiv preprint arXiv:1901.09491*, 2019.

[189] Satrajit Chatterjee. Coherent gradients: An approach to understanding generalization in gradient descent-based optimization. In *International Conference on Learning Representations*, 2020.

[190] Faraz Torabi, Garrett Warnell, and Peter Stone. Behavioral cloning from observation. *arXiv preprint arXiv:1805.01954*, 2018.

[191] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A Reduction of Imitation Learning and Structured Prediction To No-Regret Online Learning. In *Proc. of the Int. Conf. on Art. Intel. and Stat.*, pages 627–635, 2011.

[192] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[193] Cross Entropy Loss. `https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html`.

[194] Khurram Bhatti, Cecile Belleudy, and Michel Auguin. Power management in real time embedded systems through online and adaptive interplay of dpm and dvfs policies. In *2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, pages 184–191. IEEE, 2010.

[195] Ryan Gary Kim et al. Imitation Learning for Dynamic VFI Control in Large-Scale Manycore Systems. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 25(9):2458–2471, 2017.

[196] Fakhruddin Muhammad Mahbub ul Islam and Man Lin. Hybrid dvfs scheduling for real-time systems based on reinforcement learning. *IEEE Systems Journal*, 11(2):931–940, 2015.

[197] Fakhruddin Muhammad Mahbub ul Islam, Man Lin, Laurence T Yang, and Kim-Kwang Raymond Choo. Task aware hybrid dvfs for multi-core real-time systems using machine learning. *Information Sciences*, 433:315–332, 2018.

[198] DS3 Simulator. `https://github.com/segemena/DS3.git`.

[199] Hardkernel. ODROID-XU3. `https://wiki.odroid.com/old_product/odroid-xu3/odroid-xu3` Accessed 20 Mar. 2020, 2014.

[200] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.