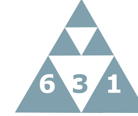


UNIVERSITÉ DE LILLE

École Doctorale 631 — Mathématiques-Sciences du numérique et de leurs interactions
Inria

Predictable CNN Inference and Real-Time Scheduling on Multicore Embedded Platforms
–
Inférence prédictible des CNN et ordonnancement temps réel sur des plateformes embarquées multicœurs

Présentée et soutenue publiquement le 18 novembre 2025

CHIARA DAINI

Thèse de doctorat d'INFORMATIQUE

dirigée par GIUSEPPE LIPARI

présentée et soutenue publiquement le date devant un jury composé de :

Gilles GRIMAUD	Professeur - Université de Lille	(président)
Claire PAGETTI	Directrice de Recherche - ONERA/DTIM	(rapporteuse)
Frank SINGHOFF	Professeur - Université de Brest/CNRS	(rapporteur)
Mitra NASRI	Maître de Conférence - Eindhoven University of Technology	(examinatrice)
Giuseppe LIPARI	Professeur - Université de Lille	(directeur)
Houssam Eddine ZAHAF	Maître de Conférence - Nantes Université	(co-encadrant)

Declaration on the Use of Artificial Intelligence Tools

To write this thesis, I have used Copilot to assist with language refinement, grammar correction and formatting suggestions. The content presented is my own work, and I have ensured that all sources are properly cited.

Abstract

Modern embedded systems incorporate sophisticated functionalities such as Machine Learning (ML). Many such applications have strict timing demands and their real-time execution is crucial for the system to be functional. Due to their computationally intensive demands and large energy consumption, ML algorithms are usually executed on high-end hardware like Graphic Processing Units (GPUs) and Tensor Processing Units (TPUs). However, it can be useful to execute ML algorithms on low-power embedded systems, for reasons of cost, power consumption, privacy, latency, etc. Executing ML algorithms on embedded platforms effectively is challenging, because it is necessary to optimize size and performance of the models, without compromising model accuracy.

One of the most popular class of algorithm in machine learning is the Convolutional Neural Network (CNN), which is widely used for image recognition, object detection, and audio classification tasks. CNNs are known to be computationally intensive and require a large amount of memory to store the model parameters and intermediate results. This makes them particularly challenging to execute on resource-constrained embedded systems.

The platform on which CNNs are executed is also a important factor for achieving real-time performance. Many embedded systems are based on multicore architectures, which allow executing multiple tasks in parallel. However, these architectures also introduce challenges related to resource contention, such as memory access and cache coherence. To address these challenges, researchers have proposed the use of scratchpad memories, which provide deterministic memory access times and eliminate the interference typical of cache-based systems. This makes them particularly suitable for real-time applications, as they allow guaranteeing the timing constraints of the system.

In this thesis, we address the real-time execution of CNN inference on multicore scratchpad-based systems. We first propose a formal model that captures the characteristics of CNN inference tasks, considering their connection patterns and data dependencies. The model is based on a task graph representation, where both computational tasks and memory transfers are represented as nodes and their dependencies as edges.

As a first contribution, we use this model to develop an Integer Linear Programming (ILP) formulation that optimally allocates computational tasks and memory transfers across cores while respecting timing and memory constraints. This yields optimal task allocation under strict real-time requirements.

In the second contribution, we introduce a complete scheduling framework for real time CNN inference on multicore scratchpad-based systems, called CNN-PAS (CNN Partitioning, Allocation and Scheduling). CNN-PAS provides heuristic algorithms for partitioning the CNN into a graph of threads, allocating threads and data transfers, and assigning scheduling parameters to threads and data transfers. The framework also includes a schedulability analysis of threads and data transfers, which allows verifying whether the system can meet the timing constraints of the CNN inference task.

Résumé

Les systèmes embarqués modernes intègrent des fonctionnalités sophistiquées telles que et le Machine Learning (ML). Nombreuses applications de ce type présentent des exigences temporelles strictes, et leur exécution en temps réel est cruciale pour assurer le bon fonctionnement du système. En raison de leurs exigences de calcul intensives et de leur consommation d'énergie, les algorithmes de ML sont généralement exécutés sur des plateformes matérielles puissantes, telles que les Graphic Processing Units (GPUs) ou les Tensor Processing Units (TPUs). Exécuter des algorithmes de ML sur des plateformes embarquées représente un défi, mais aussi une opportunité. Cela permet d'exécuter des algorithmes lourds sur des dispositifs à faible consommation énergétique, avec des gains en efficacité énergétique, en latence, en confidentialité et en coût.

Parmi les algorithmes de ML les plus populaires, on trouve les réseaux de neurones convolutifs (Convolutional Neural Networks, CNN), largement utilisés pour la reconnaissance d'images, la détection d'objets et la classification audio. Les CNN sont connus pour nécessiter d'une grande quantité de mémoire pour stocker les paramètres du modèle et les résultats intermédiaires. Cela les rend particulièrement difficiles à exécuter sur des systèmes embarqués aux ressources limitées.

La plateforme sur laquelle les CNN sont exécutés joue également un rôle important dans la réalisation des performances temps réel. De nombreux systèmes embarqués reposent sur des architectures multicœurs, qui permettent l'exécution parallèle de plusieurs tâches. Toutefois, ces architectures introduisent aussi des défis liés au partage des ressources, tels que l'accès à la mémoire et la cohérence du cache. Pour répondre à ces défis, des chercheurs ont proposé l'utilisation de mémoires scratchpad, qui offrent des temps d'accès mémoire déterministes et éliminent les interférences typiques des systèmes basés sur le cache. Cela les rend particulièrement adaptées aux applications temps réel, car elles permettent de garantir les contraintes temporelles du système.

Dans cette thèse, nous nous intéressons à l'exécution en temps réel de l'inférence des CNN sur des systèmes multicœurs avec mémoire scratchpad. Nous proposons d'abord un modèle formel qui capture les caractéristiques des tâches d'inférence CNN, en tenant compte de leurs connexions et dépendances de données. Le modèle repose sur une représentation par graphe de tâches, où les tâches de calcul et les transferts de mémoire sont représentés comme des nœuds, et leurs dépendances comme des arêtes.

En première contribution, nous utilisons ce modèle pour développer une Integer Linear Programming (ILP) qui alloue de manière optimale les tâches de calcul et les transferts de mémoire entre les cœurs, tout en respectant les contraintes temporelles et mémoire. Cela permet une allocation optimale des tâches sous des exigences strictes de temps réel.

En seconde contribution, nous introduisons un framework complet de ordonnancement pour l'inférence CNN temps réel sur des systèmes multicœurs avec mémoire scratchpad, appelé CNN-PAS (CNN Partitioning, Allocation and Scheduling). CNN-PAS propose des algorithmes heuristiques pour partitionner le CNN en un graphe de threads, allouer les threads et les transferts de données, et attribuer les paramètres d'ordonnancement aux threads et aux transferts. Le framework inclut également une analyse de l'ordonnançabilité des threads et des transferts, permettant de vérifier si le système peut respecter les contraintes temporelles de la tâche d'inférence CNN.

Acknowledgements

Without a doubt a PhD is far from being easy to achieve. The last three years and a half have been a true challenge in many aspects. I moved to another city, met new people and had to adapt to an unfamiliar way of working: being a researcher.

Unfortunately, things did not go as I had hoped. I struggled to find my path and questioned my decision about pursuing the academic career more than once. Until I met my two current supervisors: Giuseppe Lipari and Houssam Zahaf. They guided me throughout this long journey, helping grow both as a researcher and as a person. They taught me invaluable skills, like presenting in front of an expert audience, technical details about my work and much more. Without them I certainly would not have finished my thesis. I am deeply grateful for everything they have done. They are great supervisors and I believe any student would be lucky to pursue a PhD under their guidance.

I would thank also the members of my jury. Firstly, Prof. Claire Pagetti and Prof. Frank Singhoff for accepting to read and evaluate my work. I thank also Prof. Gilles Grimaud and Prof. Mitra Nasri for be members of my jury and accepting to examine the research presented in this dissertation.

I want to thank from the bottom of my heart my husband André, who has being my biggest supporter since the day we met. He has always stood by my side, without hesitation and no matter the circumstances. He was there in my darkest moments and in the brightest ones as well. He truly is my source of strength and love.

A big thank you goes to the members of the Cambium team at Inria Paris, who hosted me for more than two years and made me feel welcome, even though we did not work in the same research domain. In particular, I would like to thank François Pottier, who, aware of the difficulties I faced with my thesis, never hesitated to help me and kindly welcomed me into his team. I am also very grateful to Alexandre Moine and Clément Blaudeau, two PhD colleagues with whom I shared many moments of fun and laughter in the middle of long working days.

Many people have patiently listened to me talk about my papers, my code, CNNs, and so on, for years. For that, I have to say thank you to the best friends I could ever ask for: Beatrice, my best friend since high school, who knows me so well and has never let me down; Carlo, who supported me throughout my daily Parisian life; and Filip, who started as a colleague and has now become one of my closest friends.

Last but not least, I would like to thank my mum and my sister for being such a wonderful family and for always being proud of me.

Contents

I Background 17

1 Introduction 19
1.1 General Context: The Convergence of AI and Real-Time Embedded Systems 19
1.2 Problem Statement: Predictability on Multicore Embedded Architectures 20
1.3 Research Gap 20
1.4 Proposed Approach and Contributions 21
1.5 Thesis Outline 22

2 Convolutional Neural Networks for Embedded Systems 23
2.1 Background on Convolutional Neural Networks 23
2.1.1 Input Data Representation 24
2.1.2 Tensors 24
2.2 Convolution Neural Networks Architecture 25
2.2.1 Pre-processing Layers 26
2.2.2 Activation Functions 27
2.2.3 Convolutional Layers 28
2.2.4 Pooling Layers 29
2.2.5 Fully Connected Layers 31
2.3 Training and Inference Phases 31
2.3.1 Training Phase 32
2.3.2 Inference Phase 33
2.4 History of CNNs 33
2.4.1 Origin 33
2.4.2 Evolution 33
2.5 CNNs in Embedded Systems 35
2.5.1 TinyML 36
2.6 Benchmarks 37
2.6.1 ResNet-8 38
2.6.2 MCUNet-Tiny 38
2.6.3 DS-CNN 40
2.7 State of the Art in Real-Time CNNs for Embedded Systems 41
2.7.1 TinyML Frameworks 41
2.7.2 Model Optimization and Reduction Techniques 42
2.7.3 Predictable and Certifiable Implementation 42
2.7.4 Performance Optimization at the Implementation Level 43
2.8 Conclusion 43

3 Real Time Systems 45
3.1 Liu and Layland Model 45
3.2 Real-Time Scheduling and Analysis 47
3.2.1 Scheduling Characteristics 47
3.2.2 Preemptive vs Non-Preemptive Scheduling 48
3.2.3 Time-Triggered vs. Event-Triggered 49
3.3 Event-Triggered Scheduling 49
3.3.1 Scheduling Algorithms 49
3.4 Uniprocessor vs. Multiprocessor Scheduling 53
3.4.1 Multiprocessor Task Models 55

3.5 Real-Time Multicore Embedded Platforms 57
3.5.1 Uniprocessor vs. Multicore Architectures 57
3.5.2 Classification of Multicore Architectures 57
3.6 The Memory Subsystem 59
3.6.1 The Problem with Caches in Real-Time Systems 59
3.6.2 Scratchpad Memories as a Predictable Alternative 60
3.7 Shared Bus and Main Memory Contention 61
3.7.1 PREM and AER Model 61
3.8 Conclusion 62

II Contributions 63

4 Parallel Real-Time Execution of CNN Applications 65
4.1 AECR-DAG Task Model 65
4.1.1 Computational Nodes 66
4.1.2 Communication Nodes 67
4.2 Hardware Platform Model 68
4.2.1 Memory Architecture 68
4.2.2 Data Transfer 68
4.3 Conclusion 69

5 Optimizing CNN Inference on Multicore Scratchpad Architectures 71
5.1 Execution Model 71
5.1.1 Localized Operations 71
5.1.2 Memory Transfer Cost 72
5.1.3 Merge of Localized Operations 75
5.2 ILP formulation 76
5.2.1 ILP Problem Definition 77
5.3 Strategies to solve the problem 83
5.3.1 Global ILP 83
5.3.2 Distributed ILP 83
5.4 Experimentation and Analysis 84
5.4.1 Experimental Setup 84
5.4.2 Results 85
5.4.3 Analysis of Optimal Mapping Structures 88
5.4.4 Scalability 89
5.5 Conclusions 90

6 CNN-PAS: A Partitioning, Allocation and Scheduling Framework 91
6.1 Methodology Overview 91
6.2 Task Graph partitioning 92
6.3 Allocation 96
6.4 Memory transfers 98
6.4.1 Precedence encoding 99
6.4.2 Competing communication tasks 100
6.5 Scheduling Analysis 102
6.5.1 Scheduling algorithm for threads 102
6.5.2 Scheduling algorithm for communication nodes 105
6.6 Experimental evaluation 106
6.6.1 Benchmarks 106
6.6.2 Experimental Setup 107
6.6.3 Results 108

6.6.4 Comparison vs. MicroNets . . . 110
6.6.5 Applicability . . . 111
6.7 Model Instantiation from a Network Definition File . . . 111
6.8 Conclusion . . . 112

III Conclusion and Future Work **115**

7 Conclusion and Future Work **117**
7.1 Contributions . . . 117
7.2 Future Work . . . 118
7.3 Research Output . . . 118
7.3.1 Publications . . . 119
7.3.2 Software . . . 119

List of Figures

2.1 Representation of a neural network with multiple layers of neurons. 23
2.2 A neuron applies a function F_N to its inputs X to produce an output Y. . . . 24
2.3 Illustration of data representation as tensors. Left: A single-channel tensor of size $H \times W$, analogous to a grayscale image. Right: A three-channel tensor of size $H \times W \times C$, analogous to an RGB image. 25
2.4 An example of zero-padding. 26
2.5 Flattening a 3D tensor into a 1D vector. Each color corresponds to a channel. 27
2.6 Illustration of a 3D convolution operation. 29
2.7 Illustration of depthwise and pointwise convolutions. 29
2.8 Illustration of average pooling and max pooling operation. 30
2.9 Illustration of global average pooling operation. 31
2.10 Timeline of CNNs evolution. 34
2.11 Comparison of active power consumption between TinyML systems and those supported by MLPerf. The scale of power consumption is logarithmic. 36
2.12 A diagram of a ResNet residual block. The "identity" shortcut connection allows the model to learn a residual function, $F(x)$, which is added to the original input x. 38
2.13 Architecture of the ResNet-8 model. 39
2.14 Architecture of the MCUNet-Tiny model. 39
2.15 Block of MCUNet-Tiny. 40
2.16 Architecture of the DS-CNN model. The model consists of two parallel streams, each with multiple blocks, followed by fully connected layers and an output layer. 40
2.17 A representative convolutional block from the DS-CNN model. This is a basic VGG-style block consisting of sequential convolutional layers (typically two or three) followed by a max-pooling layer. 41

3.1 Example of a periodic task. 47
3.2 Preemptive RM scheduling example with three periodic tasks. 50
3.3 Preemptive EDF scheduling example with three periodic tasks. 53
3.4 Visual comparison of partitioned and global scheduling, showing individual vs. shared ready-queues. 54
3.5 A compact vertical DAG representing a task graph. Nodes v_i denote tasks, and labeled edges e_{i-j} denote dependencies between tasks. 56
3.6 Fork-Join structure model. 56
3.7 Shared Memory Architecture model. 58
3.8 Distributed Memory Architecture model. 58
3.9 Example of a structural cache hierarchy for a quad-core system. 59
3.10 Cache Hierarchy in Multicore Systems. 60
3.11 The AER task model. 61

4.1 A visual representation of a computational node v_i^l. 67
4.2 Transformation of a logical inter-core dependency into an explicit memory transfer node within the DAG. The dashed edge in (a) is replaced by the path through the non-preemptive memory node μ_k in (b). 67

4.3 The proposed hardware platform model. Each of the M cores has a private Code Memory (CM) and two scratchpad (SPM). The two SPM memories are connected to a shared bus, which also provides access to the main DRAM and a DMA Engine. . . . 68

5.1 Execution sequence of a localized operation v. . . . 72
5.2 Linearization of a 3D tensor of size [2][3][2] into a 1D memory array, illustrating the row-major order. . . . 73
5.3 Mapping of a $3 \times 3 \times 2$ sub-tensor from a $4 \times 4 \times 2$ tensor to linearized memory. Accessing the sub-tensor requires fetching three separate, non-contiguous batches of data. . . . 74
5.4 Example of merged localized operations 76
5.5 Visualization of the binary indicator b_0^l for operation v_0^l assigned to core p. . . . 78
5.6 Visualization of the binary variable $\mathbf{u}_p^l$. Operations v_0^l and v_5^l, shown in red, are assigned to core p. . . . 80
5.7 Visualization of the binary variable $\mathbf{u}_p^l$ and $\mathbf{c}_p^l$. Operations v_0^l and v_5^l, shown in red, are assigned to core p. . . . 81
5.8 Structure of the Global ILP strategy. The entire CNN dataflow graph, spanning all layers, is considered as a single optimization problem constrained by one overall deadline. . . . 83
5.9 Example of a CNN with 4 layers and the precedence constraints between localized operations. The red rectangle indicates the deadline constraint. . . . 84
5.10 Illustration of the impact of granularity on the number of variables in the optimization problem. . . . 85
5.11 Illustration of the `im2col` operation. Each colored column in the output matrix corresponds to a colored sliding window of the same color from the input. . . . 86
5.12 Comparison of the number of pixels copied into the SPM for different image sizes ($m \times m$) using a 3x3 kernel and three approaches. . . . 87
5.13 Execution time versus image dimensions for the two optimization techniques, considering different granularities. . . . 87
5.14 Comparison of global and distributed optimization outcomes as a function of granularity and density. Successful configurations are indicated by circles. The shaded regions reflect the areas of successful optimization. . . . 88
5.15 Visualization of an optimal mapping for an 8x8 layer onto 4 cores. . . . 89

6.1 Overall flow of the CNN-PAS framework, showing the six main phases and their iterative nature. . . . 92
6.2 Illustration of the horizontal partitioning phase for a value of $h = 1$. A single layer's feature map is divided by one horizontal cut into two independent, parallel threads. . . . 93
6.3 Example of a CNN structure. . . . 94
6.4 Graph $\mathcal{G}$ partitioned with $h = 1$. . . . 94
6.5 Resulting graph $\mathcal{G}'$ after horizontal partitioning with $h = 1$. . . . 95
6.6 Illustration of the graph transformation to model memory transfers. . . . 99
6.7 Request bound function for communication node μ_k 103
6.8 Minimum schedulable deadline for various CNNs in isolation and in concurrency as a function of their computational complexity (MACs). . . . 108
6.9 Number of horizontal partitions generated for different CNN models as a function of the assigned execution deadline. . . . 109
6.10 Operations vs. Deadline for our framework on an AURIX TriCore, compared with specific state-of-the-art MicroNet models executed on an Arm Cortex-M platform. . . . 110

List of Tables

2.1 Comparison of TinyML Performance Against Existing Technologies. 37
2.2 Key characteristics of selected benchmark CNN architectures. 38

6.1 Summary of CNN models used in the evaluation. 107
6.2 Measured WCET for primitive operations and data transfers on the AURIX TC3XXXs. 107
6.3 Average scratchpad memory occupancy, core utilization and framework execution time for different CNN models under various deadlines. 109

Part I

Background

Chapter 1

Introduction

1.1 General Context: The Convergence of AI and Real-Time Embedded Systems

Machine Learning (ML) is rapidly expanding beyond its traditional limits in high-performance computing, becoming integral to a new generation of complex, intelligent systems. This trend is particularly evident in the domain of embedded platforms, where sophisticated functionalities are no longer exclusively cloud-based but are now executed directly on-device. This migration toward the edge is prominent in domains such as autonomous driving, industrial automation and intelligent sensing. It is is driven by critical application requirements that centralized processing cannot meet, including the need for lower latency, enhanced data privacy, and reduced energy consumption [1].

In many of these applications, ML algorithms operate in contexts where their execution is subject to strict timing constraints. For example, in industrial automation, a visual fault detection system on a high-speed production line must classify a component as “accepted” or “rejected” within milliseconds, before the next item takes its place. Similarly, a collaborative robot must process visual input to halt its motion and avoid collision with a human operator. In these scenarios, the system’s correctness depends not only on the logical result of a computation but also on the physical instant at which that result is produced.

TinyML [2] is a generic term used to describe a class of ML applications that run on resource-constrained devices, such as microcontrollers (MCUs). It refers to algorithms and models that are small enough to fit within the limited memory and processing capabilities of these devices, while still delivering useful functionality. TinyML applications often involve lightweight models that can perform tasks like image classification, speech recognition, or anomaly detection in real-time, without relying on cloud-based resources. The most interesting phase of a ML is the inference phase, where the model is applied to new data to make predictions or decisions. This phase is particularly critical in real-time applications, as it must be executed within strict timing constraints to ensure the system’s responsiveness and reliability.

In this thesis, we focus on a specific ML model: Convolutional Neural Networks (CNNs). They have become foundational in many applications, particularly where image and video data are prevalent, such as in autonomous vehicles, robotics and smart cameras. They are designed to learn spatial hierarchies of features from input data, making them particularly effective for tasks like image recognition and object detection. CNNs have a unique architectural structure that is well-suited for processing grid-like data. However, CNNs are known to be computationally intensive and require substantial memory to store model parameters and intermediate data, that’s why they are typically implemented on high-performance hardware such as Graphics Processing Units (GPUs) or Tensor Processing Units (TPUs). This poses a significant challenge for their deployment on resource-constrained embedded microcontrollers.

This convergence of ML’s computational demands with the strict temporal requirements of embedded systems creates a need for methodologies that can guarantee the predictable, real-time execution of CNNs on hardware with limited resources.

1.2 Problem Statement: Predictability on Multicore Embedded Architectures

Modern embedded systems are increasingly built on multicore architectures to meet their high computational demands. While these architectures provide the necessary processing power, they introduce a fundamental challenge to temporal predictability. The simultaneous execution of tasks across multiple cores leads to contention for shared resources, particularly the memory subsystem—including shared caches, communication buses, and main memory. This interference can lead to highly pessimistic Worst-Case Execution Time (WCET) estimations.

Conventional cache-based architectures intensify this issue. The unpredictable nature of cache hits and misses, combined with complex eviction patterns, makes it exceedingly difficult to derive tight bounds on task execution times and communication delays. To overcome this unpredictability, high-integrity real-time systems are increasingly designed with scratchpad memories (SPMs). A scratchpad is a software-managed local memory that provides deterministic, low-latency access. This architecture offers superior temporal predictability but shifts the responsibility of data management entirely to the software; every data transfer between main memory and a core's local scratchpad must be explicitly achieved.

This leads to the central problem addressed in this thesis: **how can a CNN be systematically partitioned and scheduled on a multicore platform with scratchpad memories to guarantee real-time performance?**

Solving this problem requires a methodology that co-designs the scheduling of both computations and memory transfers. It is not sufficient to simply allocate operations to cores; one must also orchestrate the corresponding data movements across the shared bus to prevent contention and ensure that all data dependencies are met within strict temporal bounds.

Classic convolution optimization techniques, such as the use of General Matrix-Matrix Multiplication (GEMM) [3] with the `im2col` transformation, are highly effective for maximizing average-case performance and throughput. However, such methods conflict with the requirements of hard real-time systems. The `im2col` process creates a large intermediate matrix, significantly increasing memory overhead and leading to contention on the shared bus. This memory-intensive approach introduces timing unpredictability, making it difficult to derive tight WCET bounds. Therefore, a new approach is needed that prioritizes predictable, contention-free memory management over average-case speed.

The core challenge, therefore, is to develop a framework that can partition the complex graph of a CNN, allocate its operations to cores, and generate a contention-free schedule for all computations and data transfers that meets end-to-end deadlines.

1.3 Research Gap

The existing literature on TinyML and the deployment of CNNs on embedded systems primarily focuses on optimizing for model size and average inference speed, often neglecting the need for temporal guarantees. While this work has been important in making ML models feasible on resource-constrained hardware, it leaves a significant gap concerning their application in real-time contexts. We identify three principal areas where current research leaves room for improvement:

- Lack of real-time guarantees in TinyML frameworks. Frameworks like TensorFlow Lite for Microcontrollers (TFLM) [4], uTensor [5] and CMSIS-NN [6] have successfully facilitated the deployment of ML models on MCUs by reducing their memory footprint and optimizing computational kernels. However, their design objective is to achieve low average latency in isolation. The literature based on these frameworks often interprets "real-time" as being fast enough for a specific application's functionality, but this does

not satisfy the stringent requirements of classical real-time systems, which demand formal guarantees on respect of timing constraints. A significant gap exists in the analysis of ML workloads from a scheduling theory perspective, particularly when they must be co-scheduled with other critical tasks under a Real-Time Operating System (RTOS).

- Unpredictability of multicore COTS hardware. To meet the computational demand of CNNs, a logical solution is to employ multicore architectures. However, the use of shared resources, especially in Commercial-Off-The-Shelf (COTS) processors, introduces significant unpredictability. Concurrent access to shared caches, communication buses, and main memory leads to contention and interference, which can drastically inflate task execution times and make tight WCET analysis intractable. While some research explores mapping DNNs onto hardware, it often targets cache-based systems or specialized accelerators, and does not address the explicit, contention-free scheduling of memory transfers needed for predictable real-time performance.
- Absence of a systematic scheduling methodology. Architectures with scratchpad memories, combined with execution models offer a path toward predictability. Moreover, the use of a comprehensive task model, such as the AECR-DAG (Acquisition-Execution-Communication-Restitution Directed Acyclic Graph) [7], can help structure the representation of CNNs in a way that facilitates scheduling. However, there is a lack of systematic methodologies that can take a high-level CNN specification and automatically partition it into parallel threads, allocate those threads to cores, and generate a provably correct schedule for both computations and explicit memory transfers, all while respecting end-to-end deadlines.

1.4 Proposed Approach and Contributions

To address the identified research gaps, this thesis develops a comprehensive methodology for the predictable, real-time execution of CNN inference on multicore embedded platforms with scratchpad memories. Our approach is centered on the co-scheduling of computational tasks and memory transfers to eliminate contention and provide formal timing guarantees. The main scientific contributions are as follows:

1. **An extended real-time task model for CNNs.** We establish a formal model for representing CNN inference on multicore platforms with scratchpads. We adopt the AECR-DAG model, first introduced in [7], which partitions a task's sub-components into four distinct stages: *Acquisition* of input data, *Execution* of computational operations, *Communication* of intermediate results between local memories, and *Restitution* of the final output. We extend this model to explicitly characterize the memory usage of computational nodes, providing a structured representation that captures both the data dependencies and memory access patterns inherent in CNNs.
2. **An optimal ILP-based mapping.** Building on this model, we define the concept of a *localized operation* as the smallest schedulable unit of computation. We then formulate an Integer Linear Programming (ILP) problem that finds the optimal mapping of these operations to cores. The objective is to minimize end-to-end latency by strategically merging operations to reduce data movement, all while respecting the platform's memory and real-time constraints. This ILP formulation provides a formal definition of the problem and serves as a baseline for the following contributions.
3. **CNN-PAS framework.** We develop CNN-PAS, a complete framework designed to find schedulable configurations for complex CNNs. CNN-PAS performs its tasks in three distinct phases:

- Partitioning: the high-level CNN graph is partitioned into a set of concurrent, schedulable threads;
- Allocation: these threads are allocated to cores and, when a dependency exists between two threads on two different cores, data transfers are introduced;
- Scheduling: a schedule for both the computational threads and the memory transfers is generated by assigning intermediate offsets and deadlines to enforce all precedence constraints.

4. **A schedulability analysis and experimental validation.** We provide a complete schedulability analysis to formally verify that the schedule generated by CNN-PAS meets all real-time constraints. The effectiveness of the entire methodology is validated through extensive experiments on representative CNN benchmarks.

1.5 Thesis Outline

The remainder of this thesis is organized as follows:

Part I provides the background necessary for this work. Chapter 2 introduces the fundamentals of CNNs and their application in embedded systems. Chapter 3 covers the principles of real-time systems, including task models and scheduling theory. It also details the target multicore hardware architecture with a focus on its scratchpad-based memory subsystem.

Part II presents the main contributions of this thesis. Chapter 4 establishes the formal AECR-DAG task model used to represent CNN inference. Chapter 5 details the ILP-based formulation for optimal mapping and scheduling. Chapter 6 presents the CNN-PAS framework, including its partitioning, allocation and scheduling heuristics, along with the schedulability analysis.

Chapter 7 summarizes the research findings, reiterates the contributions and discusses potential directions for future work.

Chapter 2

Convolutional Neural Networks for Embedded Systems

Convolutional Neural Networks (CNNs) have become a central component in modern machine learning (ML), widely used in diverse applications ranging from visual perception and object detection to audio analysis, such as keyword spotting, and time-series sensor data processing. In recent years, there has been a growing interest in bringing intelligence closer to the data source, enabling real-time, private, energy-efficient and scalable Artificial Intelligence (AI) across a wide range of application domains. This has led to increased interest in deploying CNN models on microcontrollers and other low-power embedded platforms, a trend commonly referred to as *TinyML*. While CNNs are traditionally resource-intensive, advances in model design and deployment strategies have made it possible to run them effectively in environments with strict timing, memory and energy constraints.

This chapter provides an overview of CNN fundamentals with an emphasis on their applicability in embedded systems. We discuss the computational structure of CNNs and the challenges they pose when targeting architectures with limited on-chip resources.

2.1 Background on Convolutional Neural Networks

Neural Networks (NN) are inspired by the structure and functioning of the human brain, consisting of interconnected nodes (neurons) organized in layers as shown in Figure 2.1. These networks implement a sequence of mathematical functions F_N that map input data X to output predictions Y, see Figure 2.2. Each function F_N corresponds to a specific layer in the network, and the overall computation is represented as the composition $F_N(X) = Y$. A typical neural network includes an input layer, one or more hidden layers and an output layer. The number of hidden layers defines the network's depth; when this depth exceeds one, the network is referred to as a Deep Neural Network (DNN).

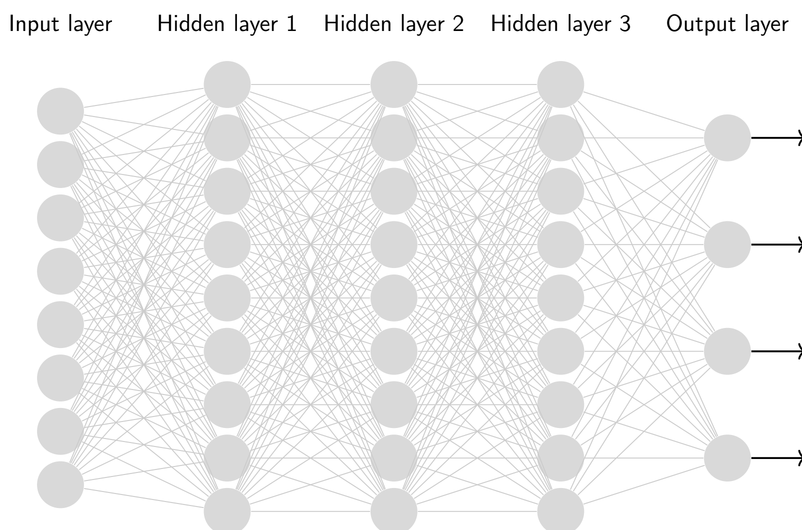


Figure 2.1: Representation of a neural network with multiple layers of neurons.

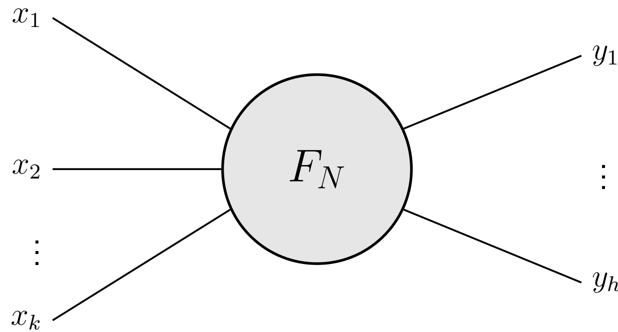


Figure 2.2: A neuron applies a function F_N to its inputs X to produce an output Y.

Convolutional Neural Networks (CNNs) are a specialized class of DNNs [8] particularly well-suited for processing data with a grid-like structure. They have become the standard architecture for image recognition, object detection, and audio classification, among others. The input data is a tensor whose elements represent features at specific locations within the grid—for example, pixel values in an image or frequency-bin energies in an audio spectrogram. The core strength of CNNs is their ability to automatically learn a hierarchy of features directly from this data. By processing the input through a series of specialized layers, they can identify simple local patterns (like edges or textures) in early layers and compose them into more complex, abstract features (like objects or phonetic structures) in deeper layers.

2.1.1 Input Data Representation

The computational models examined in this thesis process various types of input data that can be represented as structured, multi-dimensional arrays. This grid-like format is fundamental to the application of CNNs, which are architected to efficiently extract features from such spatially or temporally organized data.

An example is the digital image. An *image* is an array of pixels, where each pixel has a specific color or intensity value. The array's dimensions, its height and width, define the image's resolution. Images may also possess a third dimension for channels, often referred to as *depth*. For example, a grayscale image has a single channel, whereas a standard RGB image has three channels (red, green, and blue).

Beyond computer vision, other input data can be transformed into a similar format. In audio processing, for example, a 1D audio signal is typically converted into a 2D representation, such as a spectrogram or a matrix of Mel-Frequency Cepstral Coefficients (MFCCs) [9]. In this representation, one axis corresponds to time frames and the other to frequency bands. This 2D structure is analogous to a single-channel image.

Regardless of the source modality, elements within these structures are indexed by their coordinates. A position in a 2D grid is denoted by a tuple (i, j). For multi-channel data, an additional index (c) specifies the channel, resulting in the tuple (i, j, c).

Figure 2.3 illustrates this grid-like structure using the image as a clear visual analogy.

This generalized representation of data as a multi-dimensional array is formally defined in machine learning as a *tensor*.

2.1.2 Tensors

In the context of machine learning, a *tensor* is a mathematical object that describes a multilinear relationship between sets of algebraic objects related to a vector space and can be defined as:

$$\mathcal{T} \in \mathbb{R}^{d_1 \times d_2 \times \cdots \times d_n}$$

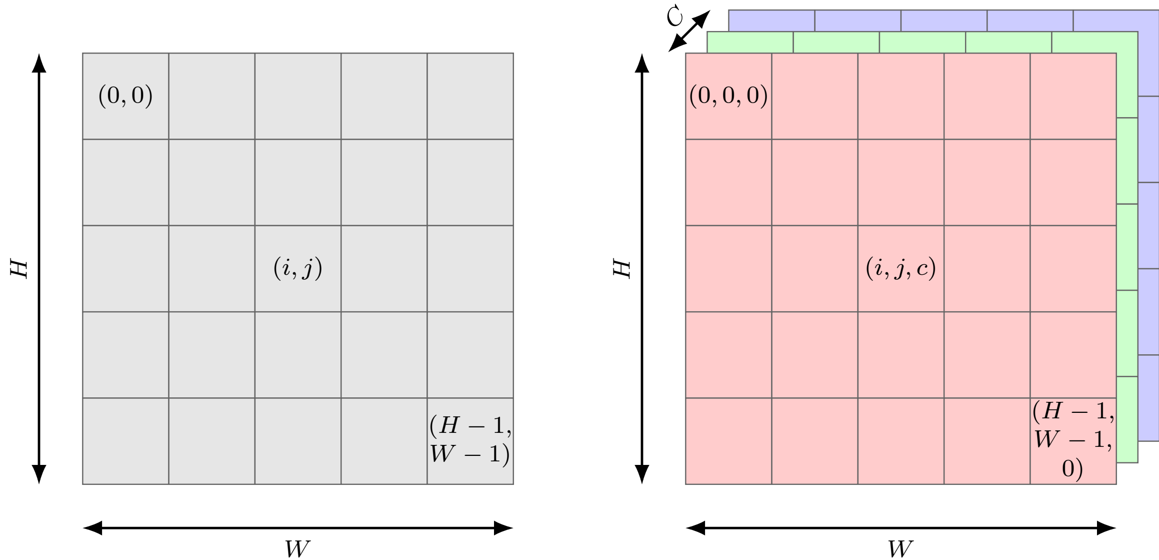


Figure 2.3: Illustration of data representation as tensors. Left: A single-channel tensor of size $H \times W$, analogous to a grayscale image. Right: A three-channel tensor of size $H \times W \times C$, analogous to an RGB image.

where n is the number of dimensions (also called the *rank* of the tensor), d_k is the size of the tensor along the k-th dimension, $\mathbb{R}$ denotes that each element is a real number.

Tensors are a generalization of vectors and matrices to higher dimensions. A tensor can be thought of as a multi-dimensional array, where each dimension corresponds to a specific aspect of the data. For example:

- **1D Tensor (Vector):** $\mathcal{T} \in \mathbb{R}^{d_1}$ — a feature vector of length d_1.
- **2D Tensor (Matrix):** $\mathcal{T} \in \mathbb{R}^{d_1 \times d_2}$ — a matrix with height d_1 and width d_2.
- **3D Tensor:** $\mathcal{T} \in \mathbb{R}^{d_1 \times d_2 \times d_3}$ — a matrix with height d_1, width d_2 and 3 channels d_3.
- **4D Tensor:** $\mathcal{T} \in \mathbb{R}^{d_1 \times d_2 \times d_3 \times d_4}$ — a batch of matrices, where d_1 is the batch size, followed by spatial dimensions d_1 and d_2 and number of channels d_3.

Tensors are a fundamental data structure, enabling efficient representation and manipulation of complex data. They are used to store input data, model parameters and intermediate results during computation.

2.2 Convolution Neural Networks Architecture

CNNs are composed of several types of layers, each serving a specific purpose in the network's operation. The most common layers in a CNN include:

- **Pre-processing Layers**: typically used to adjust the input dimensions. Common techniques are *padding*, which adds rows and columns (usually filled with zeros) around the input data, and *flattening*, which converts multi-dimensional feature maps into a one-dimensional vector;
- **Activation Function**: apply an activation function (e.g., ReLU, Sigmoid, Tanh, Softmax) to introduce non-linearity into the model;

- **Convolutional Layers**: apply convolution operations to the input data, extracting local features by sliding a filter (or *kernel*) across the input. The output of this operation is a feature map that highlights the presence of specific patterns in the input;

- **Pooling Layers**: reduce the spatial dimensions of the feature maps, retaining only the most important information. Common pooling methods include max pooling, min pooling, average pooling, and global average pooling;

- **Fully Connected Layers**: combine features learned by previous layers and produce the final output (e.g., class probabilities).

The architecture of a CNN can vary significantly depending on the specific task and the complexity of the input data.

2.2.1 Pre-processing Layers

Pre-processing layers are used to prepare the input data for subsequent operations within the CNN. Two common pre-processing operations are **padding** and **flattening**.

Padding. Padding is typically applied before convolution operations to control the spatial dimensions of the output and ensure that features at the borders are not discarded. It involves adding rows and columns, usually filled with zeros, around the input tensor.

Given a 2D input tensor $\mathcal{T}$ of shape $d_h \times d_w$, a padding P of shape $p_h \times p_w$ applied to all sides results in an output tensor $\mathcal{T}'$ of shape:

$$d'_h = d_h + 2p_h, \quad d'_w = d_w + 2p_w \tag{2.1}$$

This operation preserves the spatial resolution of the feature map. An example of padding is shown in Figure 2.4, where a 4×4 input matrix is padded with $p_h = 1$ and $p_w = 1$ to a 6×6 matrix by adding a border of zeros.

0	0	0	0	0	0
0	$(0,0)$	$(0,1)$	$(0,2)$	$(0,3)$	0
0	$(1,0)$	$(1,1)$	$(1,2)$	$(1,3)$	0
0	$(2,0)$	$(2,1)$	$(2,2)$	$(2,3)$	0
0	$(3,0)$	$(3,1)$	$(3,2)$	$(3,3)$	0
0	0	0	0	0	0

Figure 2.4: An example of zero-padding.

Flattening. Flattening is used to convert multi-dimensional feature maps into a one-dimensional vector.

For a tensor $\mathcal{T} \in \mathbb{R}^{d_h \times d_w \times d_d}$, the flattening operation produces a vector $\mathbf{v} \in \mathbb{R}^d$ with $d = d_h \times d_w \times d_d$, defined by

$$\mathbf{v}[k] = \mathcal{T}[i, j, c] \quad \text{where} \quad \begin{cases} k = c \cdot (d_w \cdot d_h) + j \cdot d_w + i, \\ 0 \leq i < d_h, \\ 0 \leq j < d_w, \\ 0 \leq c < d_c. \end{cases}$$

This transformation maintains the order of elements in the original matrix. An example of flattening a 3D tensor (with shape $5 \times 5 \times 3$) into a 1D vector is shown in Figure 2.5.

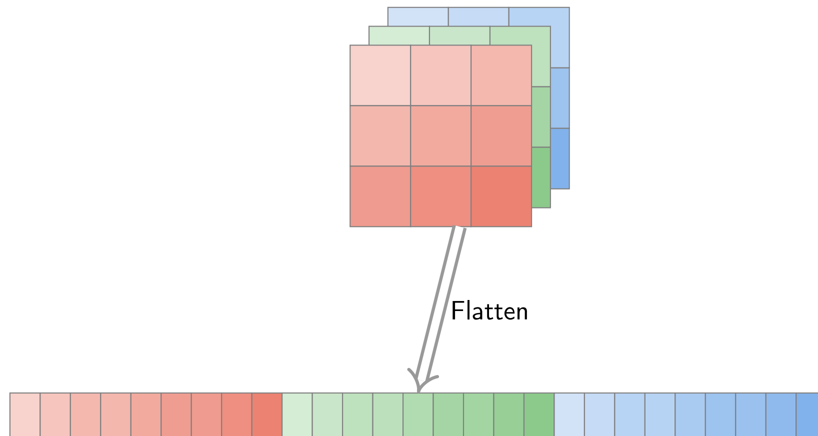


Figure 2.5: Flattening a 3D tensor into a 1D vector. Each color corresponds to a channel.

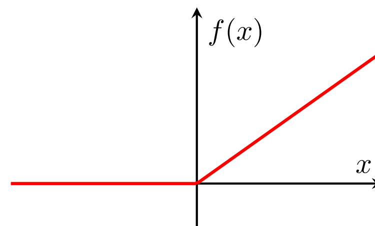
2.2.2 Activation Functions

Activation functions are non-linear transformations applied to the output of a neuron or layer in a neural network. They are normally used to decide whether a neuron should be activated or not, keeping the outputs that are relevant for the following layers and discarding the irrelevant ones. Without activation functions, the network would behave like a linear model, limiting its ability to learn complex patterns in the data.

Common activation functions include:

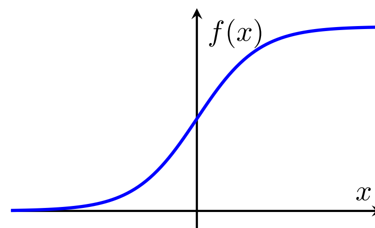
ReLU (Rectified Linear Unit)

$$f(x) = \max(0, x)$$



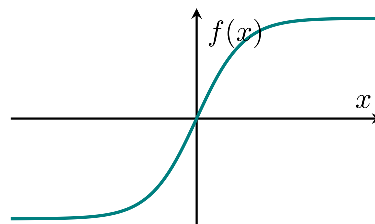
Sigmoid

$$f(x) = \frac{1}{1 + e^{-x}}$$



Tanh (Hyperbolic Tangent)

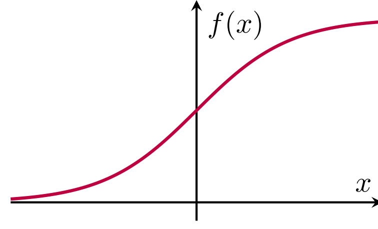
$$f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



Softmax

$$f(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}, \quad i = 1, \ldots, K$$

where K is the number of classes.



2.2.3 Convolutional Layers

The convolution operation is the core of a CNN, enabling it to learn features present in the input image, such as edges, textures, and shapes. This operation is performed by applying a filter (or *kernel*) to the input data. The filter is a smaller matrix that slides over the input data, performing element-wise multiplication and summing the results to produce an output value for each position.

Let $X \in \mathbb{R}^{H \times W \times D}$ be the input tensor, where H, W, and D denote the height, width, and depth (number of channels) respectively.

Let $K \in \mathbb{R}^{k_h \times k_w \times D \times k_n}$ be the filter bank. Here, k_h and k_w are the spatial dimensions of a filter, D is the depth of each filter (which must match the input depth), and k_n is the number of filters.

Let $s = (s_h, s_w)$ be the *stride* and $p = (p_h, p_w)$ be the *symmetric padding* applied to the height and width of the input tensor.

The convolution operation produces an output tensor $Y \in \mathbb{R}^{o_h \times o_w \times k_n}$. The spatial dimensions of Y are computed as:

$$o_h = \left\lfloor \frac{H - k_h + 2p_h}{s_h} \right\rfloor + 1 \tag{2.2}$$

$$o_w = \left\lfloor \frac{W - k_w + 2p_w}{s_w} \right\rfloor + 1 \tag{2.3}$$

Each of the k_n filters slides across the (padded) input tensor to produce a 2D feature map. The value at position (x, y) in the n-th output feature map is calculated by performing a dot product between the filter weights and the corresponding input region, to which a bias term is added. This is formulated as:

$$Y_{x,y,n} = \sum_{d=1}^{D} \sum_{i=1}^{k_h} \sum_{j=1}^{k_w} K_{i,j,d,n} \cdot X_{s_x+i, s_y+j, d} \tag{2.4}$$

for all $1 \leq x \leq o_h$, $1 \leq y \leq o_w$, and $1 \leq n \leq k_n$. The top-left corner of the input region is determined by the output coordinates and stride:

$$s_x = s_h \cdot (x - 1), \quad s_y = s_w \cdot (y - 1)$$

The indices on X refer to the coordinates in the input tensor after padding has been applied.

The depth of the output tensor Y is determined by the number of filters, k_n, as each filter is specialized to detect a different visual feature (e.g., an edge, a specific color gradient, or a texture). When a single filter convolves with the input tensor, it produces one two-dimensional feature map. This map highlights the locations within the input where the filter's target feature is present. The final output tensor is simply the stack of these k_n unique feature maps.

Figure 2.6 illustrates this process. In the example, three distinct $2 \times 2 \times 3$ filters slide over a $5 \times 5 \times 1$ input tensor with a stride of $(1, 1)$. Assuming zero padding, the convolution with each filter produces a separate $4 \times 4 \times 1$ feature map. Stacking these three maps results in a final output tensor Y with dimensions $4 \times 4 \times 3$.

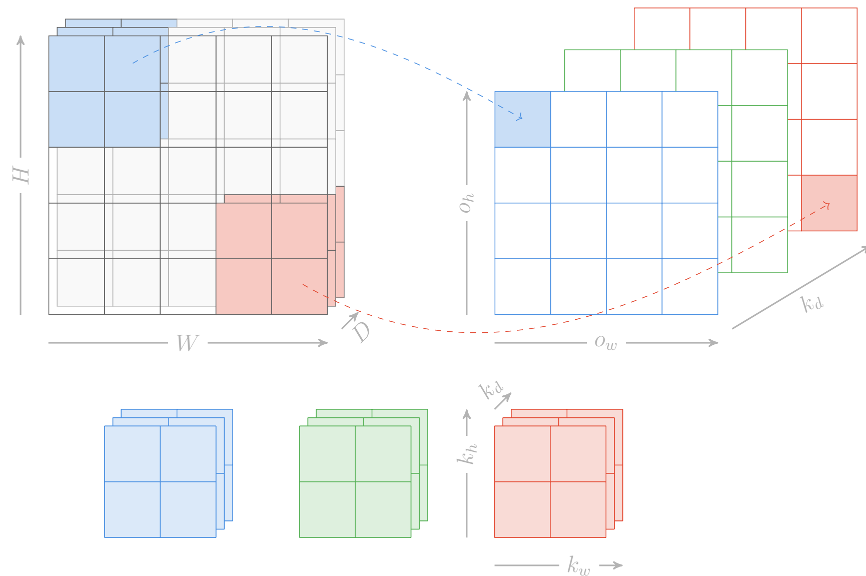


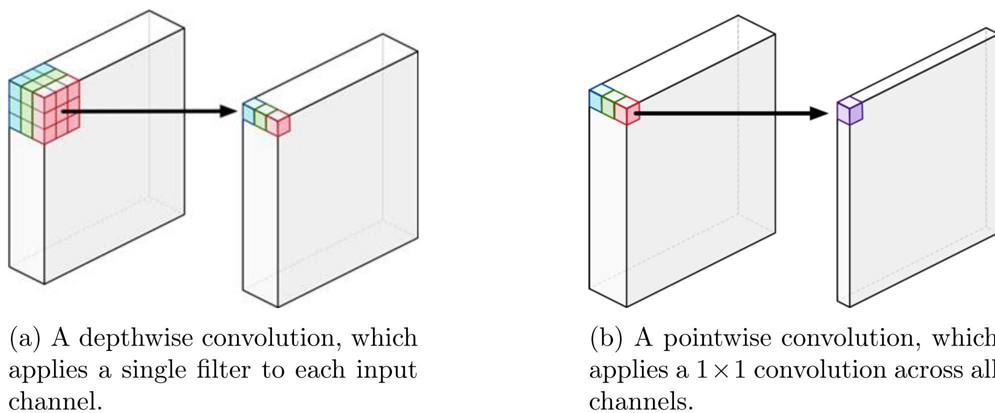
Figure 2.6: Illustration of a 3D convolution operation.

Convolutional operations can have a high computational cost, especially for large input tensors and multiple filters. Recent advancements in CNN architectures have introduced various optimizations to reduce this cost, such as *pointwise* and *depthwise convolutions*.

Depthwise convolution [10] applies a single spatial filter (of size $(k_h \times k_w)$) to each input channel independently. This filter analyses spatial information within each of the C channels but does not combine them, producing an intermediate tensor of shape $(o_h \times o_w \times C)$.

Pointwise convolution [11]—a standard convolution with a $(1 \times 1 \times C)$ kernel—is applied to the output of the depthwise stage. This step computes linear combinations of the channels to fuse them and produce the final output tensor of shape $(o_h \times o_w \times 1)$.

In Figure 2.7a and Figure 2.7b, we illustrate the depthwise and pointwise convolutions, respectively.



(a) A depthwise convolution, which applies a single filter to each input channel.

(b) A pointwise convolution, which applies a 1×1 convolution across all channels.

Figure 2.7: Illustration of depthwise and pointwise convolutions.

2.2.4 Pooling Layers

Pooling layers reduce the spatial dimensions of the input tensor, decreasing the computational load and introducing a degree of translation invariance. A pooling operation moves a window across the input and aggregates values using a function such as maximum, minimum, or average.

Let $X \in \mathbb{R}^{H\times W\times D}$ be the input tensor, and let the pooling kernel K have shape $k_h \times k_w$. The stride is defined as $s = (s_h, s_w)$ and the padding as $p = (p_h, p_w)$, with $s_h, s_w, p_h, p_w \in \mathbb{N}$. Let $T \in \{\max, \min, \text{avg}\}$ denote the pooling type.

The output tensor $Y \in \mathbb{R}^{o_h\times o_w\times D}$ has dimensions:

$$o_h = \left\lfloor \frac{H - k_h + 2p_h}{s_h} \right\rfloor + 1 \tag{2.5}$$

$$o_w = \left\lfloor \frac{W - k_w + 2p_w}{s_w} \right\rfloor + 1 \tag{2.6}$$

For each channel $d \in [1, D]$, the output at position (x, y) is computed as:

$$Y_{x,y,d} = \begin{cases} \max\limits_{i,j} X_{s_x+i,\, s_y+j,\, d}, & \text{if } T = \max \\ \min\limits_{i,j} X_{s_x+i,\, s_y+j,\, d}, & \text{if } T = \min \\ \dfrac{1}{k_h k_w} \sum\limits_{i=1}^{k_h} \sum\limits_{j=1}^{k_w} X_{s_x+i,\, s_y+j,\, d}, & \text{if } T = \text{avg} \end{cases} \tag{2.7}$$

where $s_x = s_h \cdot (x - 1)$, $s_y = s_w \cdot (y - 1)$.

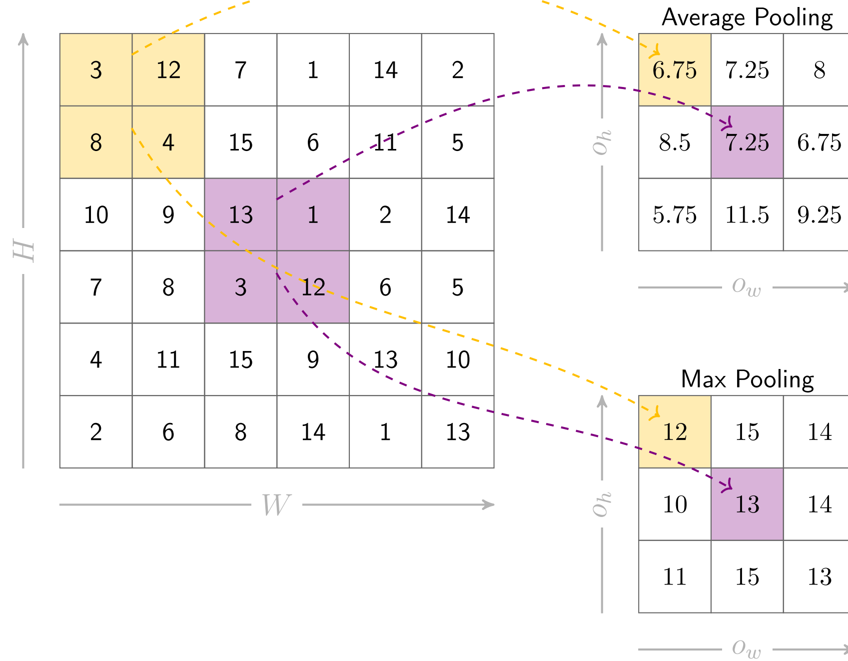


Figure 2.8: Illustration of average pooling and max pooling operation.

In Figure 2.8, we illustrate the average pooling and max pooling operation. The input tensor is a 6×6 and the pooling kernel is 2×2 with a stride of $(2, 2)$. The output tensor is a 3×3 tensor, where each element is computed by applying the pooling operation to the corresponding region in the input tensor.

Another common pooling operation is *global average pooling*. Global average pooling computes the average of all values in each channel, resulting in a single value per channel. This operation is often used before fully connected layers to reduce the spatial dimensions to $1x1$ while retaining the depth of the input tensor.

Let $X \in \mathbb{R}^{H \times W \times D}$ be the input tensor. The global average pooling operation produces an output tensor $Y \in \mathbb{R}^{1 \times 1 \times D}$, where each channel d is computed as:

$$Y_{1,1,d} = \frac{1}{H \cdot W} \sum_{i=1}^{H} \sum_{j=1}^{W} X_{i,j,d} \tag{2.8}$$

Global average pooling is particularly useful for reducing the number of model parameters in the final layers of a CNN, a critical advantage for deployment in resource-constrained environments like embedded systems. A representation of the global average pooling operation is shown in Figure 2.9.

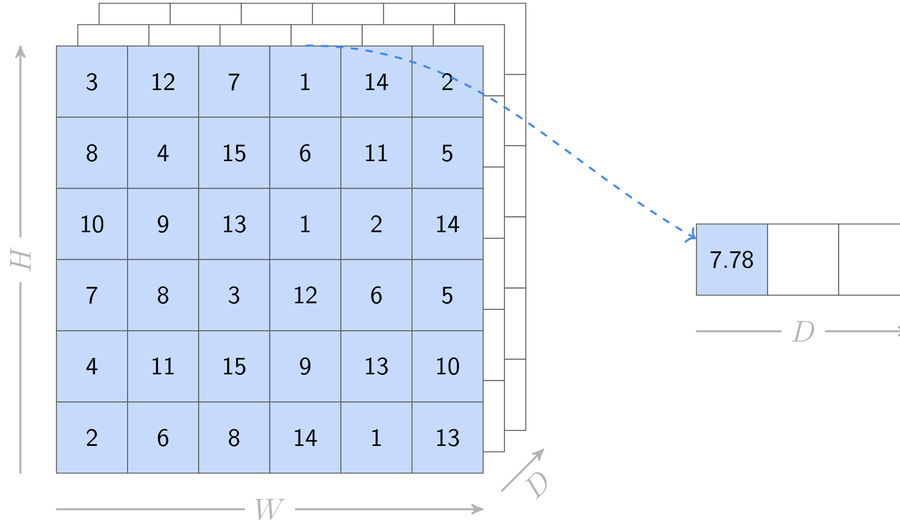


Figure 2.9: Illustration of global average pooling operation.

2.2.5 Fully Connected Layers

Fully connected layers are responsible for integrating and classifying features extracted by the preceding layers. In these layers, each output neuron is connected to every input neuron, enabling global feature combination.

Let the input be a vector $X \in \mathbb{R}^{n_{\text{in}}}$, where n_{in} is the number of input features. Let the weight matrix be $W \in \mathbb{R}^{n_{\text{in}} \times n_{\text{out}}}$, and the bias vector $b \in \mathbb{R}^{n_{\text{out}}}$.

The fully connected operation produces an output vector $Y \in \mathbb{R}^{n_{\text{out}}}$, defined as:

$$Y = XW + b \tag{2.9}$$

Element-wise, the output at position j is computed as:

$$Y_j = \sum_{i=1}^{n_{\text{in}}} X_i \cdot W_{i,j} + b_j \tag{2.10}$$

for all $j \in [1, n_{\text{out}}]$.

2.3 Training and Inference Phases

The lifecycle of a CNN consists of two main phases: *training* and *inference*. In this thesis we will focus on the inference phase, but for completeness we will describe both phases. Each phase has distinct goals, computational characteristics and resource requirements.

2.3.1 Training Phase

During training, the CNN learns to recognize patterns in the input data by adjusting its weights based on the provided ground truth labels. This phase is computationally intensive and typically requires large datasets and significant processing power. The goal is to minimize the difference between the predicted outputs and the actual labels, thereby improving the model's accuracy. The training process involves several steps:

- Forward Propagation: the input data is passed through the network, layer by layer, to compute the predicted outputs. Each layer applies its respective operations (convolution, activation, pooling, etc.) to transform the input data into a final output;

- Loss Computation: The predicted outputs are compared to the ground truth labels using a loss function (e.g., cross-entropy for classification tasks). The loss function quantifies the difference between the predicted and actual outputs, providing a measure of how well the model is performing;

- Backward Propagation: The gradients of the loss with respect to the model's weights are computed using the chain rule. This process involves propagating the error backward through the network, layer by layer, to determine how much each weight contributed to the overall loss;

- Weight Updates: The weights are adjusted based on the computed gradients using an optimization algorithm such as stochastic gradient descent (SGD) or Adam. The weights are updated in the direction that minimizes the loss, effectively allowing the model to learn from its mistakes.

This process is repeated for multiple iterations (epochs) over the training dataset until the model converges to a satisfactory level of performance. The training phase is resource-intensive, requiring substantial memory and computational power to handle large datasets and complex models. It often involves techniques such as data augmentation, regularization, and early stopping to improve generalization and prevent overfitting, which is a common modeling error that occurs when a model learns the training data too well. An overfit model captures not only the underlying patterns within the data but also the noise and random fluctuations specific to the training set. Consequently, while the model may exhibit extremely high accuracy on the data it was trained on, it fails to generalize to unseen data, leading to poor performance in real-world applications.

There exist several frameworks and libraries that facilitate the training of CNNs, such as TensorFlow [12], PyTorch [13] and Keras [14].

TensorFlow is a popular open-source machine learning framework developed by Google. It provides a flexible and efficient platform for building and training neural networks, including CNNs. TensorFlow supports both high-level APIs for ease of use and low-level APIs for fine-grained control over model architecture and training. It offers a wide range of tools for data preprocessing, model training, and deployment, making it suitable for both research and production environments. For embedded deployment, TensorFlow includes *TensorFlow Lite* and *TensorFlow Lite Micro*, which target mobile and microcontroller platforms, respectively.

PyTorch is another widely used open-source machine learning framework, developed by Facebook's AI Research lab. It is known for its dynamic computation graph, which allows for more intuitive model building and debugging. PyTorch provides a rich set of libraries and tools for training CNNs, including automatic differentiation, graphic processing unit (GPU) acceleration, and support for various neural network architectures.

Keras is a high-level neural networks API written in Python, designed to enable fast experimentation with deep learning models. It can run on top of TensorFlow. Keras provides

a user-friendly interface for building and training CNNs, making it accessible to both beginners and experienced practitioners. It simplifies the process of defining model architectures, compiling models, and training them on datasets, while still allowing for customization and flexibility when needed.

2.3.2 Inference Phase

After the training phase is complete, the CNN enters the inference phase. Inference refers to the process of using the trained model to make predictions on new, unseen data. Unlike training, inference is performed with fixed model parameters and does not involve updating the model. Normally, inference is performed online, meaning that the model processes input data in real-time or near-real-time to generate predictions.

During inference, the input data is passed through the network in a forward pass, and the output is computed using the learned weights. This phase is generally less computationally demanding than training, but it is critical for real-time and embedded applications where latency, memory usage, and energy consumption are important constraints.

2.4 History of CNNs

CNNs have had significant evolution, leading to increasingly deeper, more accurate and more efficient architectures. In this section we present an overview of the history of CNNs and the most used models. For a comprehensive review, see the survey by Li et al. [15].

2.4.1 Origin

The foundations of CNNs date back to the 1980s, with early inspiration from the Neocognitron, a hierarchical visual recognition model introduced by Fukushima [16, 17]. Unlike modern CNNs, the Neocognitron relied on unsupervised competitive learning mechanism [18].

A significant milestone was reached in 1989 when LeCun et al. proposed the first supervised multilayer CNN, known as ConvNet, trained using the backpropagation algorithm [19, 20] on the MNIST dataset, a set of 28×28 grayscale images of handwritten digits. This model achieved promising results on tasks such as handwritten digit and zip code recognition [21]. The architecture was further improved in 1998 with the introduction of LeNet-5, which became a benchmark model for document recognition and character classification. LeNet-5's effectiveness in optical character recognition and fingerprint identification led to its adoption in commercial applications, such as ATMs and banking systems.

Despite its simplicity, LeNet-5 contains the fundamental building blocks of modern CNNs: 2D convolutional layers, 2D pooling layers, and fully connected layers. Each convolutional and fully connected layer is followed by a non-linear activation function, specifically the hyperbolic tangent. The final output layer uses a softmax function for classification.

2.4.2 Evolution

The evolution of CNNs has been led by advancements in both algorithmic strategies and hardware capabilities. In the early 2000s, CNNs faced significant limitations due to low computational resources and the lack of efficient parallel processing techniques. Deep CNNs, in particular, were challenging to train, often requiring extensive time and suffering from issues such as exploding gradients when traditional activation functions like the sigmoid were used.

A turning point occurred in 2006 when Hinton et al. demonstrated that "layer-wise pre-training" could effectively overcome the gradient instability issues that hit deep networks, renewing interest in deep learning research [22]. This approach provided a more effective

initialization method compared to random weight initialization, particularly when using either supervised or unsupervised learning. Around the same time, researchers began moving away from sigmoid activations toward alternatives like ReLU and tanh, which offered more stable gradient propagation [23].

Another significant development was the adoption of max-pooling instead of traditional subsampling techniques, as demonstrated by Ranzato et al. [24], which improved the learning of translation-invariant features. Concurrently, the increasing use of GPUs revolutionized the training of deep CNNs. GPU acceleration, enabled by platforms like NVIDIA's CUDA (introduced in 2007), allowed for faster and more scalable training [25].

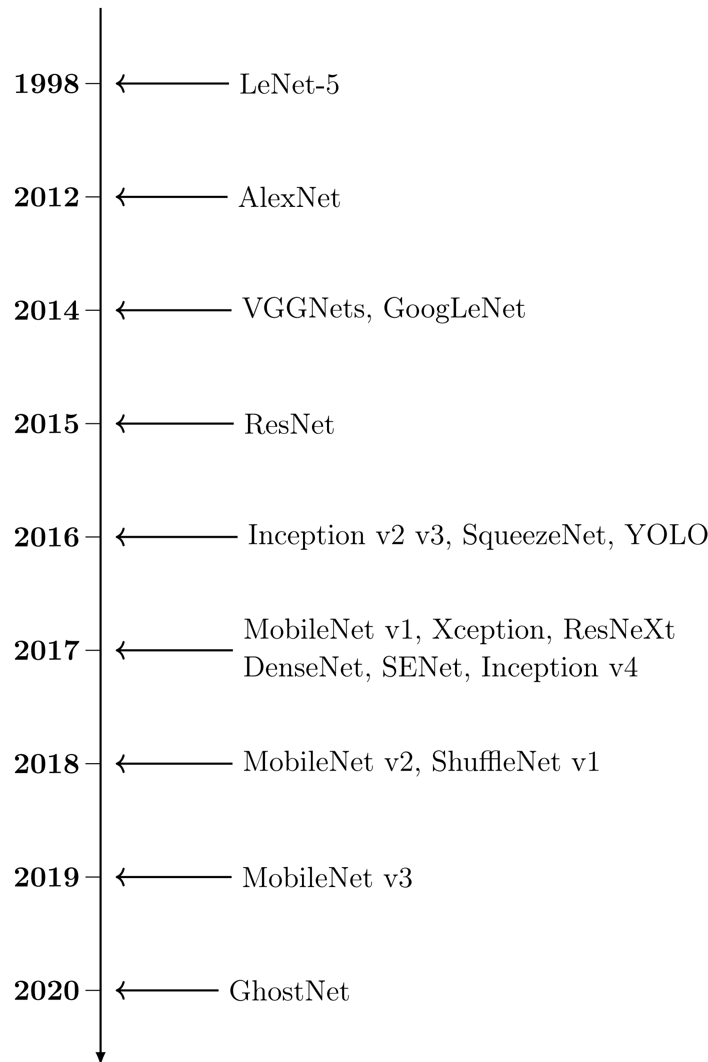


Figure 2.10: Timeline of CNNs evolution.

Large-scale annotated datasets such as ImageNet [26] and PASCAL VOC [27] also played an inportant role in this evolution. These datasets, along with benchmark challenges like the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), provided the foundation for objective performance evaluation of deep models and fostered intense research activity in computer vision.

Prior to the rise of large-scale deep CNNs, compact task-specific architectures like CifarNet were developed to demonstrate the effectiveness of CNNs on small datasets such as CIFAR-10 [28]. These models highlighted the capability of relatively shallow networks to outperform traditional handcrafted approaches in constrained environments.

The pivotal moment for modern CNNs came in 2012 with the introduction of AlexNet [29]. Competing in the ILSVRC, AlexNet achieved a top-5 error rate of 15.3%, a better result compared to all non-deep learning approaches. The success of AlexNet comes from several key innovations, including the use of the ReLU activation function, dropout for regularization, and leveraging GPUs for training acceleration.

Following AlexNet, a series of increasingly sophisticated architectures were developed, each introducing novel concepts that pushed the boundaries of performance and efficiency. A timeline of CNNs evolution is shown in Figure 2.10. The most significant of these are:

- VGGNets [30]. Introduced by the Visual Geometry Group at Oxford, these models are a series of CNNs, including VGG-11, VGG-16 and VGG-19 that demonstrated that increasing network depth improves the performance of the network. Their key design principle was simplicity, exclusively using 3×3 convolutional filters, instead of 5×5 or larger filters, to achieve a large receptive field. Parameters dropped by around 45%.
- GoogLeNet (Inception) [31]. The winner of ILSVRC 2014, the Inception models created a wider network by performing convolutions with multiple kernel sizes (1×1, 3×3, and 5×5) in parallel within the same layer, allowing the model to capture features at various scales simultaneously. It also made extensive use of 1×1 convolutions for dimensionality reduction to maintain computational efficiency.
- YOLO (You Only Look Once) [32]. This model introduced a fundamentally new architecture for object detection. Instead of relying on complex pipelines that process an image in several steps, YOLO examines the entire image in a single pass. From this single look, the network directly predicts both the location (bounding boxes) and the class of all objects simultaneously. Its success has led to an influential family of successor models, including YOLOv2 and YOLOv3.
- ResNet (Residual Network) [33]. This architecture, which won the ILSVRC 2015, solved a fundamental barrier to training extremely deep networks. It introduced the concept of residual blocks with shortcut or skip connections (see Section 2.6.1 for more details). These connections allow the gradient to bypass layers and flow directly through the network, enabling the successful training of networks hundreds or even thousands of layers deep.
- MobileNets. This family of models was designed by Google specifically for high performance on resource-constrained platforms, such as mobile and embedded devices. There are three versions of MobileNets to date, namely MobileNet v1 [34], MobileNet v2 [35], and MobileNet v3 [36]. Their core innovation is the use of depth-wise separable convolutions. This factorization reduces the computational cost and model size with only a minor trade-off in accuracy.

Another major advancement has been the use of Neural Architecture Search (NAS) to automate the design of efficient CNNs. By leveraging techniques such as reinforcement learning [37], gradient-based search [38], and evolutionary algorithms, NAS frameworks can generate architectures optimized for specific tasks and hardware platforms.

These innovations represent a significant evolution in CNN design, emphasizing the need for scalable, generalizable, and hardware-aware architectures that meet the growing demand for on-device inference and real-time processing in modern AI applications.

2.5 CNNs in Embedded Systems

In Section 2.4, we discussed various CNN models from a high-level architectural perspective, without delving into the specific hardware platforms on which they can be deployed. In this

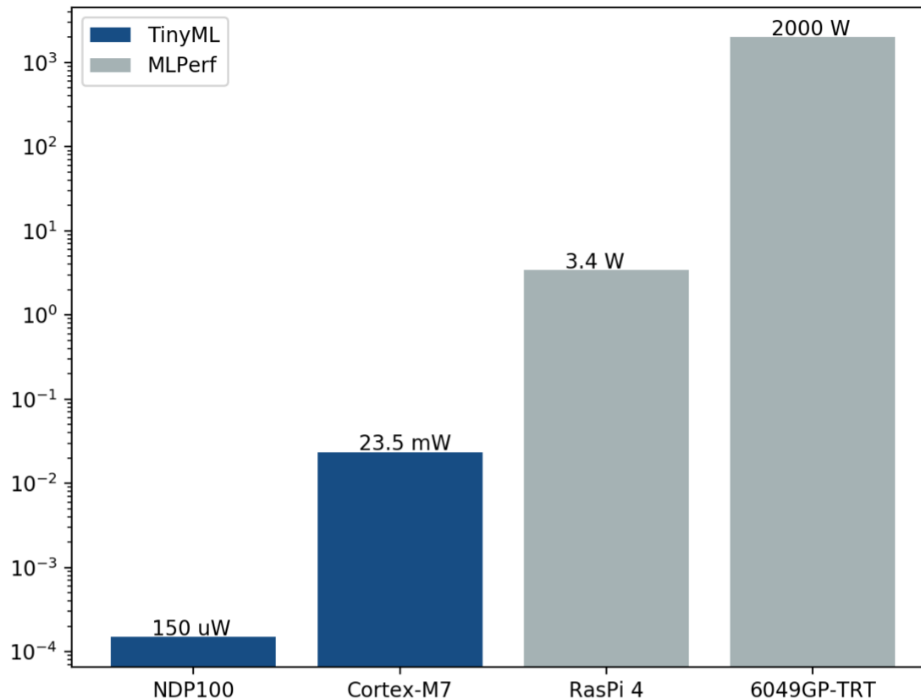


Figure 2.11: Comparison of active power consumption between TinyML systems and those supported by MLPerf. The scale of power consumption is logarithmic.

section, we shift our focus toward embedded systems and introduce the concept of *TinyML*—a subfield of machine learning that focus on deploying models on resource-constrained devices.

2.5.1 TinyML

ML models can be designed with different performance targets depending on the deployment environment. *TinyML* [2] refers to the implementation of ML models, especially NN on resource-constrained devices. It was inspired by Mobile ML's features [39] (low latency, resource limits, moderate cost) and its development grew as a result of the technical breakthrough in the field of Internet of Things (IoT) and microcontrollers (MCUs). These systems operate under strict limitations in terms of memory, compute power, and energy consumption.

TinyML is successfully applied in various application areas, such as predictive maintenance in industry, patient diagnosis in healthcare, crop monitoring in agriculture, and wildlife conservation in environmental applications. TinyML technology is driven by the necessity to integrate intelligence in a wide range of applications that were previously not viable owing to the high power and computing needs of standard ML models.

To better understand the constraints of TinyML, we can compare it with the more traditional class of machine learning applications that run on high-performance computing platforms, which we refer to as *MLPerf*. MLPerf [40] refers to the class of machine learning applications that run on high-performance computing platforms such as GPUs, TPUs, or cloud-based server clusters. These models are typically large, deep architectures optimized for maximum accuracy and scalability, with far fewer constraints on memory or power.

The differences between TinyML and MLPerf are illustrated in Figure 2.11, which compares the active power consumption of TinyML systems with those supported by MLPerf. The figure shows that TinyML systems consume orders of magnitude less power than MLPerf systems, highlighting the significant differences in resource constraints and operational environments. Figure 2.11 can be found in [41].

The key advantages of TinyML include [2]:

- Reduced Latency: by processing data directly on the device, TinyML models achieve a much faster response time, which is critical for real-time applications like image and speech recognition;

- Offline Capability: models can operate without an internet connection, making them suitable for deployment in remote areas with poor or no connectivity;

- Improved Privacy and Security: since sensitive information is kept on the device and not sent to the cloud, user privacy is significantly enhanced, which helps comply with data protection regulations;

- Low Energy Consumption: TinyML algorithms and the low-power components they run on are designed to be computationally efficient, consuming power in the order of milliwatts or microwatts. This allows devices to run unplugged for weeks or even years;

- Reduced Cost: by minimizing data transmission to the cloud, costs associated with bandwidth and storage are lowered. The low energy footprint also contributes to cost savings.

Table 2.1: Comparison of TinyML Performance Against Existing Technologies.

Technologies	**Latency**	**Privacy Consideration**	**Accuracy**	**Power Consumption**	**Memory Consumption**
Cloud Computing	10-500 ms	Very Low	86-94%	50-1000 W	GBs to TBs
Fog Computing	5-400 ms	Low	85-93%	10-100 W	MBs to GBs
Edge Computing	0.70-350 ms	Low	80-90%	1-10 W	Few KB
TinyML	0.18-300 ms	Very High	80-90%	25-300 mW	Few KB

Table 2 [2] compares TinyML with cloud, fog, and edge computing in terms of latency, data privacy, accuracy, power consumption, reliability, and memory consumption. Cloud computing refers to the use of remote servers to process and store data, while fog computing refers to a distributed computing model that brings computation closer to the data source. Edge computing refers to processing data on local devices, such as smartphones or IoT devices, rather than sending it to the cloud. The table shows that TinyML has advantages over other technologies in terms of latency, privacy, power consumption, memory consumption and reliability.

2.6 Benchmarks

To evaluate the performance of TinyML models, several benchmarks have been established to provide standardized metrics to compare the accuracy, latency, memory, and energy consumption of NN architectures targeting low-power devices. A list of benchmarks can be found in [41].

In our evaluation (see Chapter 6), we selected three CNNs that are commonly used in TinyML benchmarks: ResNet-8, MCUNet-Tiny, and DS-CNN. We report in Table 2.2 the key characteristics of these models, including the number of parameters, the common task they are used for, and the typical input shape.

Table 2.2: Key characteristics of selected benchmark CNN architectures.

Model	Parameters	Common Task	Typical Input Shape
ResNet-8	∼11k	Image Classification	32×32×3 (RGB Image)
MCUNet-Tiny	<300k	Image Classification	84×84×3 (RGB Image)
DS-CNN	∼35k	Keyword Spotting	49×10×1 (Spectrogram)

2.6.1 ResNet-8

ResNet-8 [33] is a simplified version of the original ResNet architecture, designed for low-resolution image tasks such as CIFAR-10 classification.

ResNet was introduce in 2015 by He et al. and is known for its use of residual connections to enable training of very deep networks. A residual block contains two convolutional layers with batch normalization and ReLU activation. Also at the end of each block, a skip connection is added to the output. This allows the network to learn faster and achieve better performance on complex tasks. Residual connections allow gradients to flow more easily through the network, mitigating the vanishing gradient problem and enabling the training of networks with hundreds or even thousands of layers. The architecture of a residual block is shown in Figure 2.12.

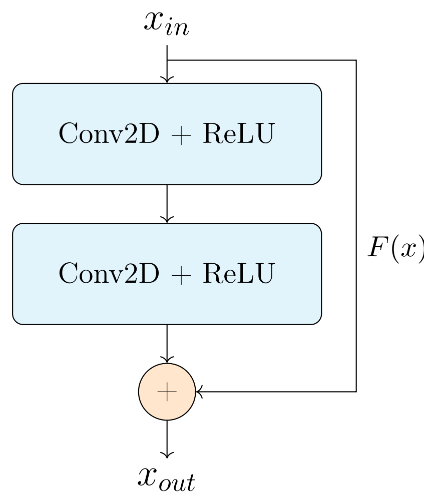


Figure 2.12: A diagram of a ResNet residual block. The "identity" shortcut connection allows the model to learn a residual function, $F(x)$, which is added to the original input x.

The model we selected is a compact version of ResNet (see Figure 2.13), making it suitable for low-power devices. It has approximately 11,000 parameters and is typically trained on the CIFAR-10 dataset, with an input resolution of 32×32×3 pixels. It contains one convolutional layer at the input, followed by 3 residual blocks, each with two convolutional layers. The model uses a global average pooling layer before the final fully connected layer for classification. Each convolutional layer in the residual block uses a 3×3 kernel, stride of 1 and pooling of 1, and the model employs ReLU activation functions after each convolution. The final output layer uses softmax activation for multi-class classification.

2.6.2 MCUNet-Tiny

MCUNet-Tiny [42] comes from a family of image classification CNNs specifically designed for deployment on microcontrollers. It was first introduced in 2020 by Lin et al. and is

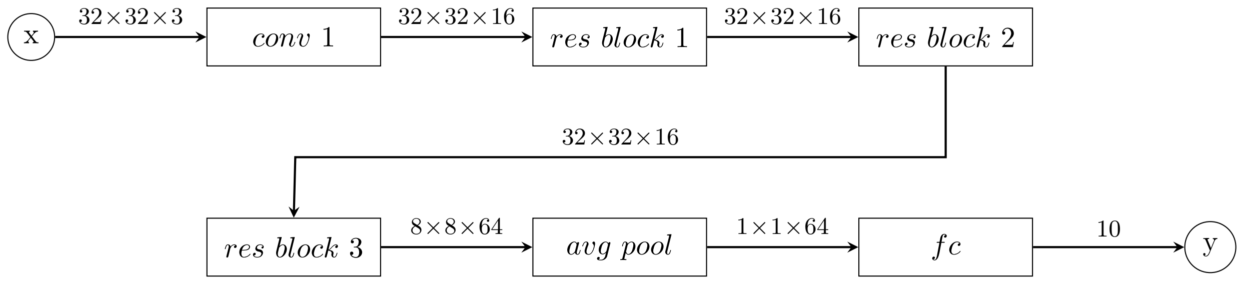


Figure 2.13: Architecture of the ResNet-8 model.

optimized for low-power devices, achieving high accuracy with a small number of parameters. MCUNet-Tiny has fewer than 300,000 parameters and can be trained on datasets like CIFAR-10 or ImageNet. It supports input resolutions ranging from 84×84 to 160×160 pixels, depending on the target accuracy and hardware constraints.

MCUNet-Tiny's architecture consists of multiple blocks, each containing a depthwise convolution followed by a pointwise convolution, with ReLU activation functions applied after each convolution. The model also includes a global average pooling layer before the final fully connected layer for classification. The architecture is designed to be modular, allowing for easy adaptation to different input resolutions and hardware constraints.

In our evaluation (Chapter 6), we will focus on the MCUNet-Tiny model with an input resolution of 84×84 pixels, which is suitable for CIFAR-10 classification tasks. A Figure 2.14 illustrates the structure of the whole MCUNet-Tiny model and Figure 2.15 shows a single block of the MCUNet-Tiny architecture, which consists of a pointwise convolution, a depthwise convolution, a pointwise convolution, and a skip connection that adds the input to the output of the projection convolution.

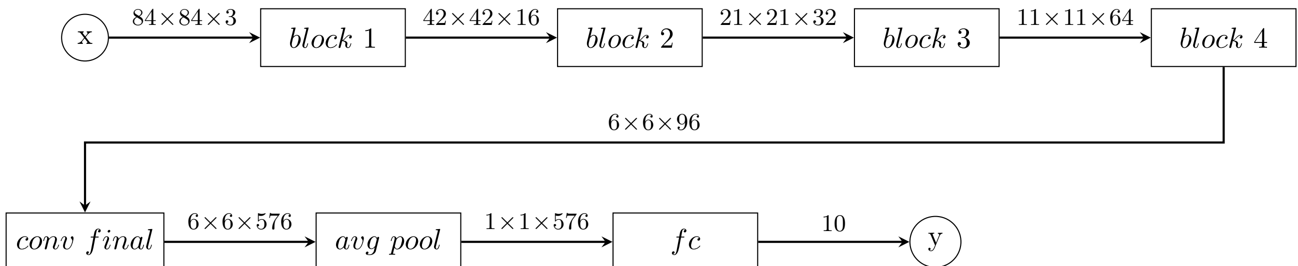


Figure 2.14: Architecture of the MCUNet-Tiny model.

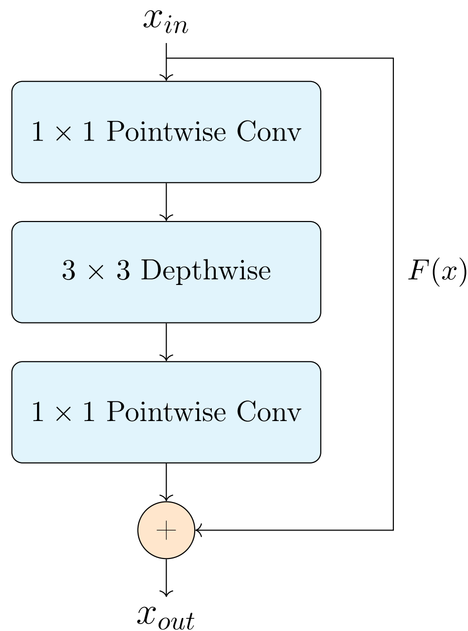


Figure 2.15: Block of MCUNet-Tiny.

2.6.3 DS-CNN

DS-CNN [43] proposes a dual-stream architecture for heart sound classification, leveraging both time-domain and frequency-domain features. DS-CNN is a compact CNN architecture that uses depthwise separable convolutions to reduce the number of parameters and computational cost. It has approximately 35,000 parameters and is typically trained on datasets like PhysioNet's heart sound dataset [44], with an input resolution of $49 \times 10 \times 1$ pixels. The architecture consists of multiple convolutional layers, each followed by batch normalization and ReLU activation functions. The model also includes a global average pooling layer before the final fully connected layer for classification.

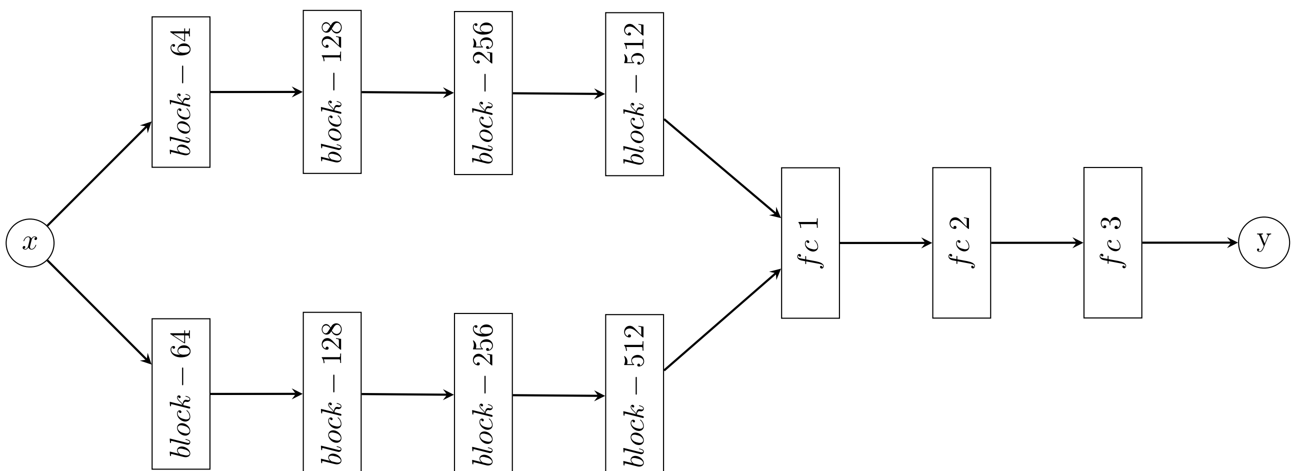


Figure 2.16: Architecture of the DS-CNN model. The model consists of two parallel streams, each with multiple blocks, followed by fully connected layers and an output layer.

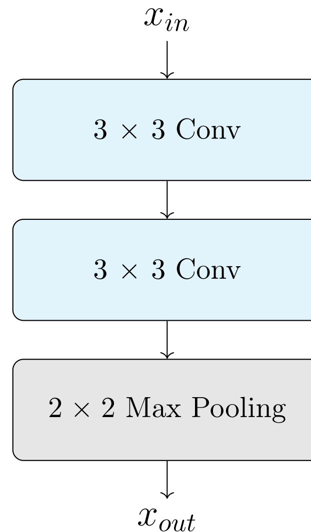


Figure 2.17: A representative convolutional block from the DS-CNN model. This is a basic VGG-style block consisting of sequential convolutional layers (typically two or three) followed by a max-pooling layer.

In Figure 2.16, we show the architecture of the DS-CNN model, which consists of two parallel streams, each with multiple blocks, followed by fully connected layers and an output layer. Each block contains two convolutional layers and a max pooling layer, as we can see in the detailed view of a single block in Figure 2.17. Every block has a number of filters that are applied to the input data, and the output of each block is passed to the next block in the stream. The two streams are merged at the fully connected layer, which produces the final output of the model.

2.7 State of the Art in Real-Time CNNs for Embedded Systems

The deployment of CNNs on embedded systems is a rapidly evolving field. The state of the art can be analyzed through several key areas: frameworks designed for resource-constrained devices, algorithmic techniques for model reduction, and specialized strategies for predictable and high-performance implementation.

2.7.1 TinyML Frameworks

The deployment of ML models onto resource-constrained microcontrollers has been significantly advanced by specialized frameworks. In [6], the authors present CMSIS-NN, an optimized software kernel for deploying NNs on ARM Cortex-M processors. Another notable work is uTensor [5], which provides a framework for deploying ML models on micro-controllers, focusing on efficient memory usage and computational performance. Lastly, a very popular framework is TensorFlow Lite for Micro-controllers (TFLM) [4], a lightweight version of TensorFlow Lite that supports a wide range of operations and provides tools for model quantization and optimization. To standardize the evaluation of different TinyML models and frameworks, Banbury et al. [41] proposed the MLPerf Tiny benchmark suite, which provides a standard for evaluating performance on common embedded tasks.

While these frameworks provide the foundational tools for running ML models on MCUs, a primary challenge is the transition from achieving low latency to guaranteeing predictable real-time performance. A growing body of work implements ML on embedded platforms for applications requiring timely responses, such as emotion recognition [45], violence detection [46] and agricultural object detection [47]. In these contexts, “real-time” typically means

that the average inference speed is sufficient for the application's functionality. However, this interpretation does not satisfy the stringent requirements of classical real-time systems, which demand formal guarantees on schedulability and bounded execution times.

A significant research gap exists in the analysis of these models from a scheduling theory perspective. Even foundational research that produces highly efficient model architectures, such as MicroNets [48], focuses on optimizing for low latency and memory usage in isolation. The literature concerning the integration of ML workloads into systems with multiple concurrent tasks, managed by a Real-Time Operating System (RTOS), is notably limited. Specifically, there is a lack of research on deriving the Worst-Case Execution Time (WCET) for complex CNNs and analyzing their schedulability alongside other critical tasks. This issue reflects a challenge in real-time computing, such as ensuring predictable performance on multi-core processors [49], yet its application to the characteristics of ML inference remains an underdeveloped area. Moreover in [50], the author lists several methods to design a CNN for embedded systems. More in detail, it is discussed how to chose some parameters in order to optimize the CNN for embedded systems.

2.7.2 Model Optimization and Reduction Techniques

To address the significant resource constraints of MCUs, researchers have developed numerous techniques to reduce the computational and memory footprint of CNNs. The ZIP-CNN methodology, proposed by Garbay [51], focuses on creating a framework to estimate the impact of these reduction techniques on a target MCU's latency, energy consumption, and memory before implementation. The primary reduction techniques include:

- Quantization: this technique reduces the numerical precision of model parameters and activations, typically from 32-bit floating-point (FP32) to 8-bit integer (INT8) or smaller formats. Quantization significantly decreases the memory footprint and can accelerate inference, especially on MCUs that lack a Floating Point Unit (FPU);

- Pruning: this method involves removing redundant parameters (weights, filters, or channels) from the network. Pruning can be unstructured, creating sparse matrices that are difficult to accelerate without specialized hardware, or structured, where entire filters or channels are removed, resulting in smaller, dense models that are easier to implement efficiently;

- Knowledge Distillation: this is a training strategy where a smaller "student" network is trained to mimic the output of a larger, more complex "teacher" network. This allows the smaller model to achieve higher accuracy than if it were trained from scratch alone, making it possible to deploy compact yet powerful models.

2.7.3 Predictable and Certifiable Implementation

For safety-critical applications, such as those in avionics, average-case performance is insufficient. These systems require formal guarantees of predictability and certifiability.

In [52], the authors introduce ACETONE, a programming framework designed to meet safety objectives such as traceability and WCET computation for ML applications. ACETONE focuses on generating predictable C code from trained feed-forward deep neural networks, ensuring semantic preservation. A core contribution of this work is the formalization of the semantics of each network layer as a mathematical function, which serves as a reference to guarantee that the generated C code is semantically equivalent to the training environment. The framework addresses key certification objectives, including traceability from the model description to the final executable and the ability to perform static timing analysis. The C code is statically generated and suited for a single-core bare-metal platform [53].

2.7.4 Performance Optimization at the Implementation Level

Beyond model-level reduction, significant performance gains can be achieved by optimizing the low-level implementation of core CNN operations. The convolution layer is typically the most computationally intensive part of a CNN.

A common and highly effective optimization is to reformulate the convolution operation as a General Matrix-Matrix Multiplication (GEMM). While this allows leveraging highly optimized routines, standard BLAS (Basic Linear Algebra Subprograms) libraries are often not designed for certifiable systems, as they may use dynamic memory allocation or complex optimizations that hinder WCET analysis. The work on ACETONE was extended to include efficient, predictable, and traceable implementations of GEMM-based convolution routines to address this specific issue [54]. In a similar context, Juan et al. [55] propose a portable convolution algorithm based on the BLIS realization of the GEMM kernel, which mitigates the memory overhead associated with the traditional `im2col` transform by avoiding intermediate memory usage.

Memory and execution time optimization techniques for CNN inference have also been explored using Integer Linear Programming (ILP). The authors of TASO [56] introduce a domain-specific optimization for CNN models that selects primitive operations to implement convolutional layers, optimizing for execution time and memory consumption. In a related work, LEMON [57] proposes a flexible ILP approach for mapping DNN layers onto accelerators, considering memory hierarchy constraints to optimize energy efficiency and latency. These methods, however, often target architectures based on cache memories where it is not possible to directly control memory contention on the bus. We address this problem by using scratchpad memories and scheduling memory transfers to avoid contention.

2.8 Conclusion

In this chapter, we have provided an overview of CNNs, their architecture, and their deployment in embedded systems. We started by introducing the fundamental components of CNNs, including convolutional layers, activation functions, pooling layers, and fully connected layers. We then discussed the training and inference phases of CNNs, highlighting the differences between these two stages. We also explored the evolution of CNN architectures, from the early models to recent advancements. We emphasized the importance of adapting CNNs for resource-constrained environments, leading to the emergence of TinyML as a subfield focused on deploying machine learning models on low-power devices. To ground our analysis, we introduced the three benchmark models that will be central to our work: ResNet-8, MCUNet-Tiny, and DS-CNN. We conclude this chapter by reviewing the state of the art in deploying CNNs on embedded systems, discussing frameworks, model optimization techniques, and strategies for achieving predictable and certifiable implementations.

Chapter 3

Real Time Systems

A real-time system is a computer system that must react within a strict time limit to external events. The correctness of its behavior depends not only on the logical results of the computation, but also on the physical instant at which these results are produced [58].

A real-time system interacts with its physical environment through sensors, which gather data, and actuators, which perform actions. Given the dynamic nature of the environment, the system's response to a detected event must occur within a bounded time delay for it to be correct. The correctness of the system, therefore, depends on the temporal validity of its internal state in relation to the state of the environment.

The severity of the consequences resulting from a failure to meet temporal constraints provides a crucial basis for classifying real-time systems. While various taxonomies exist, a primary classification distinguishes between three levels of criticality:

- Hard Real-Time Systems: in these systems, failing to meet even a single deadline constitutes a some kind of damage to the surrounding environment, potentially leading to catastrophic consequences, especially when humans are involved. Key examples include avionics flight control systems, medical devices such as pacemakers, air traffic control, and industrial robots on an assembly line.

- Soft Real-Time Systems: for these systems, missing one or more deadlines is not considered a failure but rather a degradation in the system's quality of service. The system continues to operate with reduced utility. An example of a soft real-time system is an audio application, where missing a deadline might result in a momentary glitch in the quality, but the overall system remains functional.

The work presented in this thesis will focus exclusively on challenges and solutions related to hard real-time systems.

To guarantee that these timing constraints are respected, it is essential to perform a formal analysis of the system's temporal behavior before it is deployed. The analysis must consider all factors that can influence timing, which requires creating accurate models of the system's fundamental components.

The rest of this chapter is therefore dedicated to establishing the foundations of real-time systems.

3.1 Liu and Layland Model

In real-time systems, tasks are the fundamental units of computation that must be executed within specific timing constraints. A task tipically models a *thread* of execution, which is a sequence of instructions that can be scheduled by the operating system. A task can be activated multiple times during the execution of a system and each activation is called a *job* or *instance*. Let's define a real-time task set by $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task $\tau_i \in \mathcal{T}$ is modeled as an infinite sequence of jobs $\{j_i^1, j_i^2, \ldots, j_i^k\}$.

Each job j_i^ℓ is characterized by three timing parameters:

- a_i^ℓ: the *arrival time*, i.e., the time at which the job becomes ready for execution;

- d_i^ℓ: the *absolute deadline*, by which the job must complete its execution to be considered timely;

- c_i^ℓ: the *execution time*, i.e., the worst-case computation time required by the job.

Based on the pattern of job arrivals, tasks can be classified into the following categories:

- **Periodic tasks**: jobs are released at strictly regular intervals, defined by a fixed period.
- **Sporadic tasks**: jobs arrive irregularly, but with a guaranteed minimum inter-arrival time between consecutive jobs.
- **Aperiodic tasks**: job releases do not follow any predictable pattern.

The seminal work by Liu and Layland [59] introduced what is the most widely used task model in real-time systems. In this model, tasks can be described by a tuple $\tau_i = (C_i, D_i, T_i)$, where:

- **Execution time** (C_i): the worst-case execution time (WCET) required for a task. It represents the maximum time a task can take to complete its execution and it's equal to:
$$C_i = \max_{\forall \ell} c_i^\ell \tag{3.1}$$
Several techniques exist to estimate the WCET of a task and exhaustive surveys can be found in [60, 61].
- **Deadline** (D_i): the time by which a job must complete its execution. For periodic tasks, this is often equal to the period ($D_i = T_i$) and in this case it is called *implicit deadline*. If the deadline is less than the period, it is called *constrained deadline*. If the deadline is greater than the period, it is called *arbitrary deadline*.
- **Period** (T_i): the time interval between consecutive activations of a periodic task. For sporadic tasks, it is the minimum inter-arrival time.

Periodic task sets are also distinguished in two classes: *synchronous* and *asynchronous*. Synchrony refers to the activation time of the first job of a task, called *offset* ϕ_{τ_i}. A synchronous task set is one in which all tasks have the same offset, i.e., $\phi_{\tau_i} = \phi_{\tau_j}$ for all i, j. In an asynchronous task set, the offsets can be different, i.e., $\phi_{\tau_i} \neq \phi_{\tau_j}$ for some i, j [62].

Besides the execution time C_i, another parameter that is often used to characterize a task is the worst-case response time (R_i). The worst case response time is not a characteristic of the task itself, but it depends on the scheduling and hence on the presence of other tasks.

Let f_i^k the finishing time of the k-th job of task τ_i. Job j_i^k has executed c_i^k units of execution between $\left[a_i^k, f_i^k\right]$. Therefore we can define:

$$R_i = \max_{\forall k}\{f_i^k - a_i^k\} \tag{3.2}$$

For the task to be feasible (i.e., to meet its deadlines), it must hold that $R_i \leq D_i$.

Another important characteristic in the Liu and Layland model is the task's *utilization*. It measures of how much computing resources a task requires relative to its period and is defined as the ratio of the execution time to the period:

$$U_i = \frac{C_i}{T_i} \tag{3.3}$$

In Figure 3.1, we illustrate a periodic task τ_i with period $T_i = 8$, deadline $D_i = 4$, and execution time $C_i = 2$. The task's utilization is $U_i = \frac{C_i}{T_i} = \frac{2}{8} = 0.25$, indicating that it occupies 25% of the CPU time during its period.

Several models have been proposed to extend the Liu and Layland model to cover more complex task sets. We will present in Section 4 the models that we will use to represent the system under analysis.

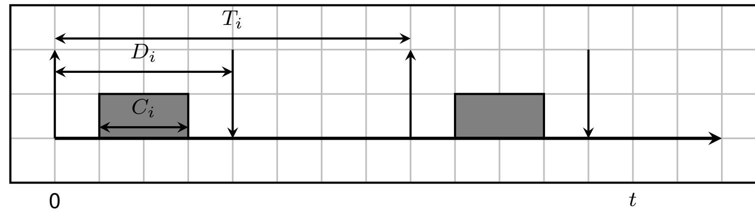


Figure 3.1: Example of a periodic task.

3.2 Real-Time Scheduling and Analysis

Once the workload of a real-time system has been characterized using appropriate task models, the challenge is to schedule these tasks such that all timing constraints are met. The *scheduler* is the component of the operating system responsible for deciding which task should execute on the processor at any given point in time. The logic used by the scheduler is dictated by a *scheduling algorithm*.

3.2.1 Scheduling Characteristics

Before going into the details of scheduling algorithms, it is important to give more definitions and terminology.

Preemption

Preemption refers to the ability of the scheduler to interrupt a currently running task to allow a higher-priority task to execute. The preempted task's state is saved, allowing it to resume later from the exact point of interruption.

Work conserving

A scheduler is considered work-conserving if it never leaves the processing unit idle while there is at least one task in the ready queue waiting to be executed. In other words, a work-conserving scheduler ensures that the CPU is always busy as long as there is pending work to be done.

Schedulability

A task τ_i is said to be schedulable by a given scheduling algorithm if its R_i is no greater than its relative deadline D_i. Consequently, a task set $\mathcal{T}$ is schedulable if every task $\tau_i \in \mathcal{T}$ is schedulable.

Feasibility

A task set $\mathcal{T}$ is considered feasible on a specific hardware platform if there exists at least one scheduling algorithm capable of scheduling all possible job sequences generated by $\mathcal{T}$ without any deadline misses.

Optimality

A real-time scheduling algorithm is optimal within a certain class of algorithms if it can schedule every task set that all other algorithm in that same class can schedule.

Schedulability Test

A schedulability test is an algorithm that determines if a task set can be scheduled by a specific algorithm on a given platform. These tests are categorized as follows:

- A sufficient test guarantees that if a task set passes the test, it is indeed schedulable. However, failing the test does not necessarily imply it is not schedulable.
- A necessary test ensures that if a task set is schedulable, it will pass the test. Consequently, failing a necessary test proves the task set is not schedulable.
- An exact test is one that is both sufficient and necessary. It provides a definitive answer regarding the schedulability of a task set.

Sustainability

A scheduling algorithm is considered sustainable if the schedulability of the system is guaranteed to be maintained even if some task parameters improve. Specifically, a reduction in the execution time of any job, or an increase in its period (inter-arrival time), will not result in any other job missing its deadline.

Blocking

Blocking is the phenomenon where a higher-priority job is prevented from executing because a lower-priority job holds a required resource by a higher-priority job.

Priority Inversion

This directly follows from blocking. Priority inversion happens when the highest priority job is not allowed to execute and lower priority job is executed instead. It can happen in a non-preemptive system or in systems with mutually exclusive resources protected by locks [63]. In these systems it is important to bound the blocking time of the higher priority jobs using an appropriate locking protocol.

3.2.2 Preemptive vs Non-Preemptive Scheduling

Scheduling algorithms can be categorized based on several criteria. A fundamental distinction is between *preemptive* and *non-preemptive* scheduling.

In a preemptive system, the scheduler has the authority to interrupt a currently executing job to run a different, higher-priority job that has become ready. The state of the interrupted (preempted) job is saved, allowing it to resume its execution later from the exact point of interruption. The primary advantage of preemption is responsiveness. High-priority jobs, which may be associated with critical events, do not have to wait for lower-priority jobs to finish their execution.

However, preemption is not without its costs. The process of saving and restoring task states, known as a *context switch*, introduces runtime overhead. More critically, preemption complicates the analysis of shared resources. If a low-priority job is preempted while holding a resource (e.g., a mutex), a higher-priority job that requires the same resource will be forced to wait. Mitigating this requires sophisticated resource access protocols, such as the Priority Inheritance Protocol (PIP) or the Priority Ceiling Protocol (PCP) [63].

In contrast, a non-preemptive scheduler allows a task to run to completion once it has been allocated the processor. A job only releases the CPU when it completes, or if it voluntarily blocks itself (e.g., to wait for an I/O operation). The main benefit of this approach is its simplicity and predictability at the task level. It avoids the complexities of priority inversion

caused by preemption, simplifying the management of shared resources. The overhead of context switching is also reduced, as it only occurs when a task naturally completes.

The principal drawback of non-preemptive scheduling is the potential for significant blocking. A high-priority job that arrives just after a long, low-priority job has started execution must wait for the entire duration of that low-priority job. This blocking time can be unacceptably long and makes it much harder to guarantee the deadlines of urgent tasks. Schedulability analysis in non-preemptive systems must therefore explicitly account for the longest possible blocking time imposed by any lower-priority task.

3.2.3 Time-Triggered vs. Event-Triggered

Beyond the model of preemption, scheduling architectures can be classified by what triggers scheduling decisions. This leads to two distinct paradigms: *time-triggered* and *event-triggered*.

In a time-triggered (TT) system, all scheduling decisions are made at predefined points in time, determined by a clock. The activation of tasks and the dispatching logic are based on a static schedule that is computed offline. This schedule is like a timetable, dictating exactly which task runs in which time slot. The primary advantage of the TT approach is its high degree of predictability; the system's behavior over time is deterministic and easy to verify. However, it is inherently inflexible. The system cannot easily react to unexpected events, and if a task finishes its work early, the processor may sit idle until the next predefined time slot begins, a property known as being *non-work-conserving*.

In contrast, event-triggered (ET) system makes scheduling decisions at runtime in response to system events, such as a task becoming ready, a task completing, or a resource being released. The most common form of ET scheduling is priority-driven, where the scheduler always selects the highest-priority ready task to execute. This approach is highly flexible and efficient, as it can react immediately to urgent events. The trade-off is that its behavior is less deterministic upfront, requiring more complex schedulability analysis (such as the methods discussed later in this chapter) to provide timing guarantees.

3.3 Event-Triggered Scheduling

In this thesis, we focus on event-triggered scheduling due to its flexibility and responsiveness to dynamic system conditions. We will explore the most common event-triggered scheduling algorithms, which are based on the concept of task priorities.

3.3.1 Scheduling Algorithms

The choice of scheduling algorithm is based on the choice of priorities assignment. There exists two main approaches to assigning priorities to tasks: (i) fixed priority scheduling and (ii) dynamic priority scheduling. Each approach has its own characteristics and is suitable for different types of real-time systems.

In fixed priority scheduling, priorities of tasks are assigned statically and do not change during the system's execution. This means that once a task is assigned a priority, it retains that priority for all its jobs throughout the system's lifetime.

In dynamic priority scheduling, the priority of a job can change at any moment during its execution. This allows for more flexible scheduling decisions based on the current state of the system and the deadlines of jobs.

We will present now the most common scheduling algorithms used in real-time systems, categorized by their priority assignment approach.

Static-Priority Scheduling

This class of algorithms employs a static-priority scheme, where each task's priority is assigned offline and remains constant throughout execution. While the scheduling algorithm remains the same, executing the highest-priority ready task, the approaches differ in the policy they use to assign these priorities. Three seminal approaches define this category:

- **Rate-Monotonic (RM) Scheduling:** this algorithm assigns priorities based on the task's activation rate. Tasks with shorter periods (higher rates) are assigned higher priorities. The intuition is that tasks that arrive more frequently require more urgent attention. For a task τ_i, its priority P_i is inversely proportional to its period T_i ($P_i \propto \frac{1}{T_i}$). RM is the optimal static-priority algorithm for uniprocessor systems with implicit-deadline ($D_i = T_i$) periodic tasks [59]. An example of RM scheduling is shown in Figure 3.2, where three periodic tasks are scheduled based on their rates: τ_1 has $C_1 = 2$, $T_1 = 5$; τ_2 has $C_2 = 2$, $T_2 = 6$; and τ_3 has $C_3 = 2$, $T_3 = 10$. The task with the shortest period (τ_1) is assigned the highest priority, followed by τ_2 and then τ_3.

- **Deadline-Monotonic (DM) Scheduling:** a generalization of RM, this algorithm assigns priorities based on tasks' relative deadlines. Tasks with shorter relative deadlines receive higher priorities ($P_i \propto \frac{1}{D_i}$). DM is the optimal static-priority algorithm for uniprocessor systems with constrained-deadline ($D_i \leq T_i$) tasks. When deadlines are implicit, DM is equivalent to RM [64].

- **Optimal Priority Assignment (OPA):** while RM and DM provide effective heuristics, they are not optimal for all task models, such as those with arbitrary deadlines ($D_i > T_i$). The Optimal Priority Assignment algorithm, developed by Audsley, provides a method to find a feasible priority ordering for any task set, if one exists [65]. The algorithm iteratively assigns priorities from the lowest to the highest. At each step, it searches for a task that can meet its deadline if assigned the current (lowest available) priority level, assuming all other unassigned tasks have higher priorities. If such a task is found, it is assigned the priority, removed from the set of unassigned tasks, and the process repeats for the next priority level.

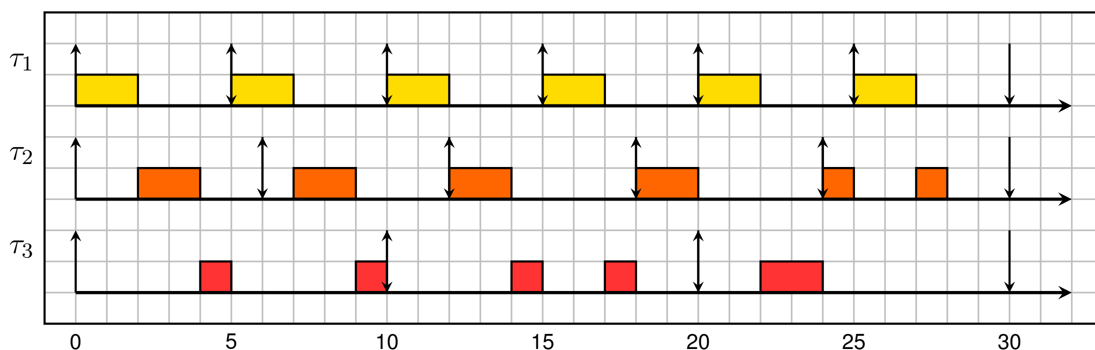


Figure 3.2: Preemptive RM scheduling example with three periodic tasks.

To verify if a task set is schedulable under a given static-priority assignment, two primary methods of analysis are used.

Utilization-Based Tests. These tests provide a quick, sufficient condition for schedulability by comparing the total processor utilization $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ against a calculated bound.

The most well-known is the Liu and Layland bound for RM scheduling [59]:

$$U \leq n(2^{1/n} - 1)$$

If the total utilization of a set of n implicit-deadline tasks is below this bound, the task set is guaranteed to be schedulable by RM. As $n \to \infty$, the bound converges to $\ln(2) \approx 0.693$. While simple, this test is not exact; a task set may still be schedulable even if its utilization exceeds this bound. Utilization-based tests are used for preemptive scheduling.

Response-Time Analysis (RTA). Response-Time Analysis provides an exact (necessary and sufficient) schedulability test for preemptive scheduling. The processor idea is to calculate R_i for each task and verify that it is less than or equal to its deadline ($R_i \leq D_i$).

The worst-case response time of a task τ_i is the sum of its C_i and the maximum possible interference from higher-priority tasks. This can be calculated using the following iterative recurrence relation:

$$R_i^{(k+1)} = C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j \tag{3.4}$$

where $hp(i)$ is the set of tasks with higher priority than τ_i. The term $\lceil \frac{R_i^{(k)}}{T_j} \rceil$ represents the maximum number of times a higher-priority task τ_j can preempt τ_i during its response time.

The calculation starts with an initial guess, typically $R_i^{(0)} = C_i$, and iterates until the value converges (i.e., $R_i^{(k+1)} = R_i^{(k)}$). The task τ_i is schedulable if the iteration converges to a value $R_i \leq D_i$. If at any point $R_i^{(k)} > D_i$, the task is deemed unschedulable, and the iteration can be stopped. The entire task set is schedulable if and only if all tasks are schedulable. This method was formalized in foundational works by Joseph and Pandya [66] and Audsley et al. [65].

Dynamic-Priority Scheduling

- **Earliest Deadline First (EDF) Scheduling:** in EDF, priority is not assigned to tasks, but to individual jobs. At any point in time, the job with the earliest absolute deadline d_i^ℓ has the highest priority. As new jobs with earlier deadlines arrive, they will preempt any currently executing job with a later deadline.

  An example of EDF scheduling is shown in Figure 3.3, where three sporadic tasks are scheduled based on their deadlines: τ_1 has $C_1 = 2$, $T_1 = 5$; τ_2 has $C_2 = 2$, $T_2 = 6$; and τ_3 has $C_3 = 2$, $T_3 = 10$. The task with the earliest deadline is always executed first, leading to a dynamic schedule that adapts to the arrival of new jobs.

  For independent, preemptive, sporadic tasks with implicit deadlines ($D_i = T_i$), EDF is optimal for all the classes of uniprocessors. A simple and exact schedulability test is to check if the total processor utilization does not exceed the capacity of one processor:

$$\sum_{\tau_i \in \mathcal{T}} U_i \leq 1. \tag{3.5}$$

  For more general cases, such as tasks with constrained deadlines ($D_i \leq T_i$), this utilization-based test is necessary but not sufficient. The exact schedulability of a system scheduled with EDF can be determined using *Processor Demand Analysis*, proposed by Baruah et al. [67]. This method checks if the cumulative computational demand of all tasks within any interval of time never exceeds the length of that interval. This analysis is a feasibility test. However, since EDF is an optimal algorithm, proving a task set is feasible is the same as proving it is schedulable.

This is formalized using the concept of the *demand bound function*, $\text{dbf}(\mathcal{T}, t)$, which calculates the maximum possible execution demand of a task set $\mathcal{T}$ in any time interval of length t. It is the sum of the demand bound functions of each individual task $\tau_i \in \mathcal{T}$:

$$\text{dbf}(\mathcal{T}, t) = \sum_{\tau_i \in \mathcal{T}} \text{dbf}_i(t) \tag{3.6}$$

where the demand bound function for a single sporadic task τ_i is given by:

$$\text{dbf}_i(t) = \max\left(0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) \cdot C_i \tag{3.7}$$

This formula calculates the worst-case execution demand for task τ_i in any interval of length t, which occurs when one job arrives at the start of the interval and subsequent jobs arrive as rapidly as possible.

An exact schedulability test can then be formulated based on this function.

> **Theorem 1: Baruah et al. [67]**
>
> A sporadic task set $\mathcal{T}$ is schedulable by EDF on a single processor if and only if:
>
> 1. The total utilization $U_{\mathcal{T}} = \sum_{\tau_i \in \mathcal{T}} \frac{C_i}{T_i} \leq 1$, and
> 2. For all $t > 0$, the total processor demand does not exceed the available time: $\text{dbf}(\mathcal{T}, t) \leq t$.

While the condition $\text{dbf}(\mathcal{T}, t) \leq t$ must hold for all $t > 0$, it sufficient to check a bounded number of values of t. The check only needs to be performed at absolute deadlines of tasks up to a certain upper bound L^*, typically the hyperperiod of the task set or a tighter calculated bound where the processor demand is guaranteed to be less than the available time [67]. This makes the exact test computationally feasible.

- **Least Laxity First (LLF) Scheduling:** Like EDF, LLF [68, 69] is an optimal dynamic-priority scheduling algorithm for uniprocessor systems. However, instead of deadlines, it uses the concept of laxity (or slack time) to assign priorities. At any time t, the laxity $L_i(t)$ of a job is the maximum amount of time its execution can be delayed without missing its deadline. It is calculated as:

$$L_i(t) = d_i - t - C_i'(t) \tag{3.8}$$

  where d_i is the job's absolute deadline, t is the current time, and $C_i'(t)$ is the remaining execution time of the job.

  The LLF scheduler assigns the highest priority to the job with the smallest non-negative laxity. While optimal, LLF is rarely used in practice due to its high overhead. When two or more jobs have similar laxity, the scheduler can switch between them excessively, leading to a high number of context switches that degrade system performance. For this reason, EDF is generally preferred.

- **P-Fair (Proportional Fair) Scheduling:** P-Fair scheduling [70, 71] is an optimal algorithm designed primarily for multiprocessor systems. Its goal is to ensure that every task τ_i with utilization U_i makes progress at a perfectly steady rate, as if it were running on a dedicated processor of speed U_i.

  To achieve this, P-Fair divides time into discrete slots (quanta) and aims to keep the difference (or *lag*) between a task's actual processor allocation and its ideal allocation

close to zero at all times. This ensures a proportional and fair distribution of CPU time across all tasks.

P-Fair is important because it is one of the few known optimal scheduling algorithms for multiprocessor systems (it can schedule any task set where the total utilization does not exceed the number of processors, $\sum U_i \leq M$). However, its practical use is limited by its high overhead, as it can cause a large number of preemptions and task migrations between processors to maintain its strict fairness property.

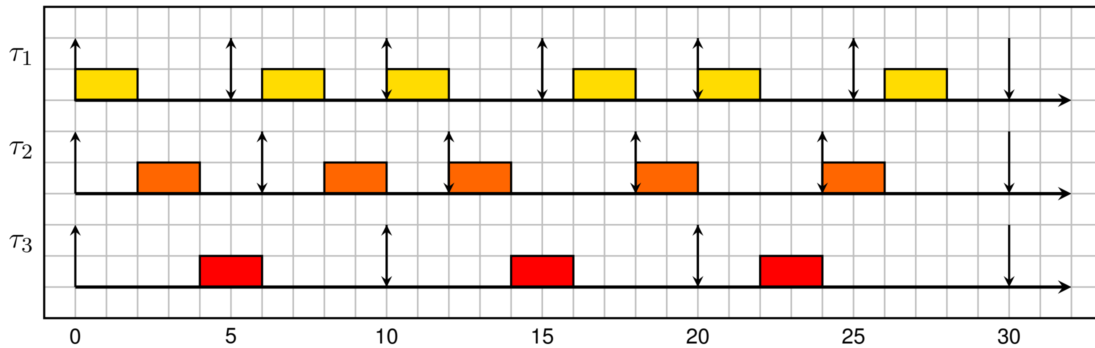


Figure 3.3: Preemptive EDF scheduling example with three periodic tasks.

3.4 Uniprocessor vs. Multiprocessor Scheduling

The principles and complexity of real-time scheduling change dramatically when moving from a single processor to a multiprocessor platform. While uniprocessor scheduling is a mature and well-understood field, multiprocessor scheduling introduces challenges that are still an active area of research.

Uniprocessor Scheduling. On a platform with a single processor, the scheduling problem is purely temporal: the scheduler's only decision is *when* to execute a given task. As discussed in Section 3.3.1, this problem is largely studied. Optimal online algorithms exist: RM is optimal for static-priority tasks with implicit deadlines and EDF is optimal for dynamic-priority tasks. For any set of tasks that is theoretically schedulable, EDF can successfully schedule it, provided the total utilization does not exceed 100%. Furthermore, exact schedulability tests, such as RTA for static-priority systems and Processor Demand Analysis for EDF, provide verifiable guarantees of system correctness.

Multiprocessor Scheduling. The introduction of multiple processors adds a spatial dimension to the scheduling problem: the scheduler must decide not only *when* a task should run, but also *where* (on which processor). This additional complexity invalidates many of the optimality results from uniprocessor theory and introduces new challenges. A survey by Davis and Burns provides a comprehensive overview of the field [72]. The main approaches to multiprocessor scheduling can be categorized as follows:

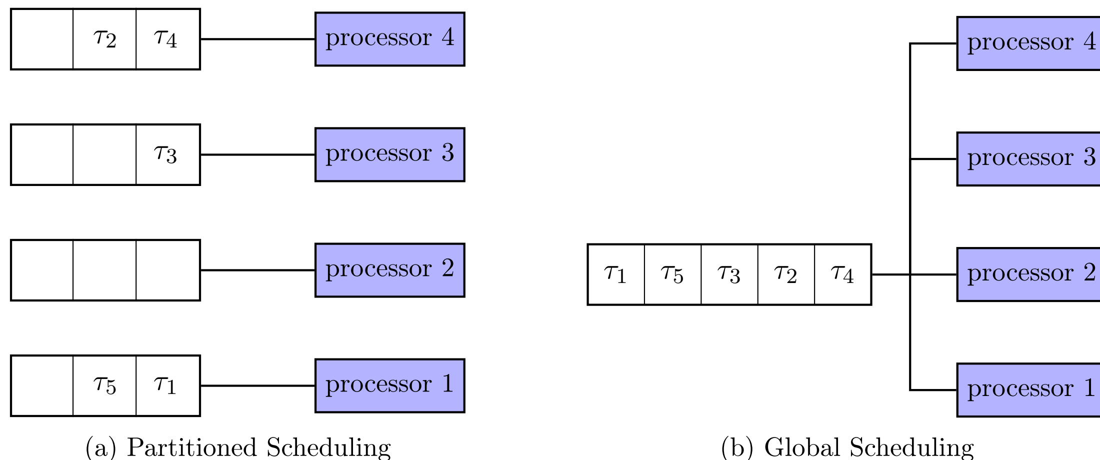
- **Partitioned Scheduling:** in this approach, the problem is divided into two stages. First, each task is statically assigned to a specific processor. Second, each processor independently schedules its assigned tasks using a standard uniprocessor scheduling algorithm (e.g., local RM or EDF).
  - **Advantages:** its primary advantage is simplicity. Once tasks are assigned, we can reuse the benefit of well-established uniprocessor analysis techniques [73]. The runtime overhead is low, as tasks do not migrate between processors, which also leads to better cache performance.

  - **Disadvantages:** The main drawback is the inflexibility of the initial task allocation. This assignment is an NP-hard problem, making it difficult to find an optimal distribution. This can lead to situations where a task set is deemed unschedulable even with significant resources available, as the system cannot exploit unused capacity on one processor to alleviate load on another. Furthermore, the static nature makes it inconvenient to reschedule tasks in response to changing workloads.

- **Global Scheduling:** This approach maintains a single, system-wide queue for all ready tasks. Whenever a processor becomes available, the scheduler selects the highest-priority task from this global queue to execute. Consequently, a task is not bound to a specific processor and may be preempted on one processor and later resume on another, a feature known as *task migration*.

  - **Advantages:** Global scheduling offers superior flexibility and resource efficiency. It provides automatic load balancing, as the processing load is naturally distributed across all processors. This ability to reclaim and utilize otherwise idle processor time often leads to a lower average-case response time for tasks.
  - **Disadvantages:** The approach suffers from higher overheads and weaker predictability. The single ready queue becomes a point of contention, requiring costly inter-processor synchronization mechanisms [74]. The primary drawback is the cost of task migration, which leads to a loss of cache affinity and can significantly increase worst-case execution times.

- **Hybrid Scheduling:** Various hybrid approaches exist to find a compromise between the two extremes. A common example is *clustered scheduling*, where processors are grouped into clusters. Scheduling is partitioned between clusters, but global within each cluster. This limits task migration to a smaller set of processors, balancing schedulability and overhead.



(a) Partitioned Scheduling

(b) Global Scheduling

Figure 3.4: Visual comparison of partitioned and global scheduling, showing individual vs. shared ready-queues.

In this thesis, the scheduling analysis will be performed using the partitioned scheduling approach, exploiting the simplicity of the uniprocessor scheduling theory. As this is proved to be a NP-hard problem, existing real-time literature suggests the use of heuristics such as Best-fit and Worst-fit [75, 76] to allocate tasks on processor processors.

Best-Fit heuristic attempts to pack tasks as tightly as possible. When allocating a new task, Best-Fit assigns it to the processor that will have the least remaining capacity (i.e.,

highest utilization) after the task is added. The goal is to leave other processors with large chunks of free capacity, potentially making it easier to fit large tasks later.

Worst-Fit heuristic takes the opposite approach. It allocates a new task to the processor that will have the most remaining capacity (i.e., lowest utilization) after the task is added. The goal is to evenly distribute the load across all processors, which can sometimes lead to better overall schedulability by avoiding the creation of highly utilized processors.

3.4.1 Multiprocessor Task Models

Several models have been developed to represent tasks in multiprocessor systems, starting from the basic Liu and Layland one.

In general, a parallel task is said to be: (i) *rigid* if the number of processors simultaneously used by the task is fixed a priori, (ii) *moldable* if the number of processors simultaneously used by the task is not fixed but determined before the execution, and (iii) *malleable* if the number of processors simultaneously used by the task can change at runtime.

The most important are:

Directed Acyclic Graph (DAG) Model

At its most fundamental, a graph is an abstract representation of relationships, consisting of a set of vertices (or nodes) that represent objects, and a set of edges that represent connections between those objects.

Graphs can be categorized as either undirected or directed. In an undirected graph, edges simply signify a symmetric relationship between two vertices. In a directed graph, however, each edge has a specific orientation, pointing from a source vertex to a target vertex. This directionality allows to model non symmetric relationships like precedence, where one operation must complete before another can begin.

A key concept in directed graphs is a cycle, which is a path along a sequence of directed edges that starts and ends at the same vertex. A cycle represents a circular dependency—a logical impossibility in a sequence of operations (e.g., operation A cannot start until B is done, but B cannot start until A is done). A graph that is both directed and contains no cycles is known as a Directed Acyclic Graph (DAG). The absence of cycles guarantees a well-defined and executable order of operations, ensuring a clear forward progression from input data to final output.

Therefore, we can define $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ to be a directed acyclic graph where $\mathcal{V}$ is the set of vertices (nodes) and $\mathcal{E}$ is the set of edges (dependencies). Each vertex $v \in \mathcal{V}$ represents a computation or operation, and each edge $(e, v) \in \mathcal{E}$ indicates that operation e must be completed before operation v can start. Edges generate a relationship of precedence between vertices.

In Figure 3.5, we can see a compact vertical representation of a DAG.

Fork-Join Model

This model represents tasks that can be split into multiple parallel branches (fork) and later synchronized back together (join). In this model, all parallel segments in a task shares the same number of processors, and that tasks of this model have implicit deadline.

A fork-join task τ_i is defined by the tuple $\tau_i = ((C_{i,1}, P_{i,2}, C_{i,3}, \ldots, C_{i,s_i}), m_i, T_i)$, where:

- s_i is the number of segments.
- m_i is the degree of parallelism, representing the number of threads executed in each parallel segment ($m_i \leq m$, where m is the total number of system processors).
- T_i is the task's period and relative deadline.

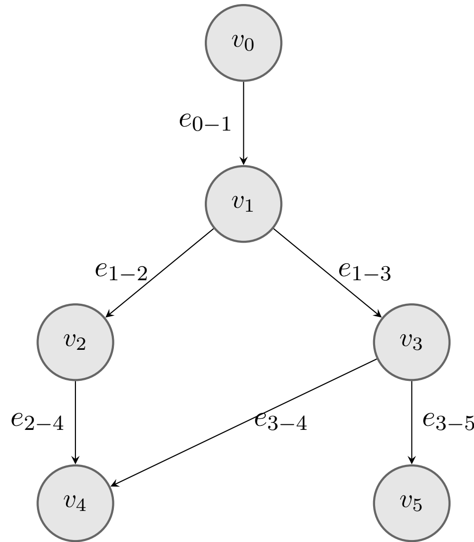


Figure 3.5: A compact vertical DAG representing a task graph. Nodes v_i denote tasks, and labeled edges e_{i-j} denote dependencies between tasks.

- $C_{i,j}$ denotes theWCET of a *sequential* segment, for $j \in \{1, 3, \ldots, s_i\}$.
- $P_{i,j}$ denotes the WCET of each of the m_i threads within a *parallel* segment, for $j \in \{2, 4, \ldots, s_i - 1\}$. All m_i threads execute concurrently.

This model is illustrated in Figure 3.6, where sequential segments $C_{i,j}$ are shown as single tasks, and parallel segments $P_{i,j}$ are represented as multiple concurrent tasks.

The fork-join model can also be represented as a DAG, where $C_{i,j}$ and $P_{i,j}$ are nodes, and edges represent the precedence constraints.

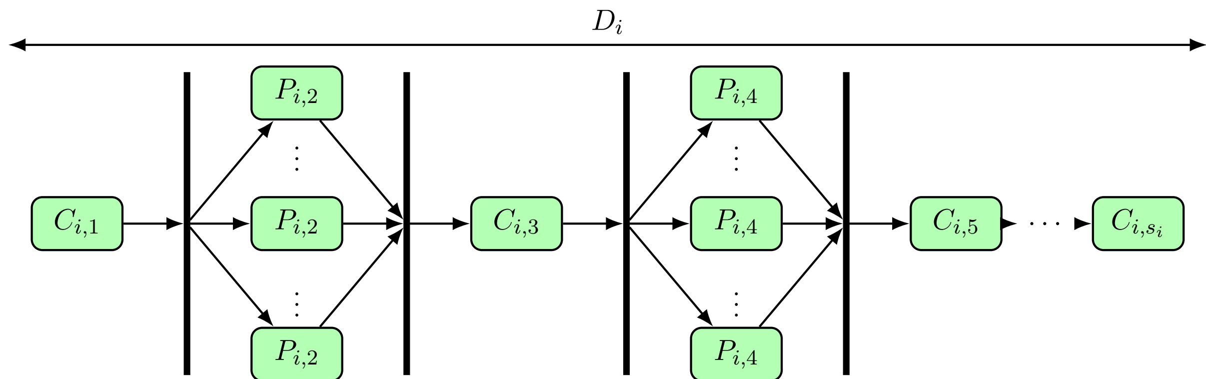


Figure 3.6: Fork-Join structure model.

GANG Model

The GANG model describes a parallel task where all its constituent threads must be scheduled to run simultaneously. The term "gang" refers to this group of threads that are treated as a single, indivisible unit by the scheduler: either all m_i processors are allocated to the task's threads, or none are. This synchronized and simultaneous execution is often referred to as *lockstep* execution.

A GANG task τ_i is characterized by the tuple $\tau_i = (C_i, T_i, D_i, m_i)$, where:

- C_i is the worst-case execution time (WCET) of the task, which is the duration for which all m_i threads execute.

- T_i is the period of the task.
- D_i is the relative deadline, often implicit ($D_i = T_i$).
- m_i is the fixed number of processors required by the task for its entire execution.

Unlike the DAG or Fork-Join models, the GANG model does not expose internal parallelism or precedence constraints to the scheduler. From the scheduler's perspective, the task is a single block that requires m_i processors for C_i time.

After having introduced uniprocessor and multiprocessor scheduling with the most used parallel task models, in the next section we will talk about real time platforms.

3.5 Real-Time Multicore Embedded Platforms

For decades, performance improvements in computer systems were driven by increasing the clock frequency and architectural complexity of single-core processors, a trend famously described by Moore's Law [77, 78]. However, by the mid-2000s, this approach hit fundamental physical limits, primarily the "power wall," where increasing clock speeds led to unmanageable heat dissipation and power consumption. Faced with this barrier, the industry switched from making single cores faster to integrating multiple, often simpler, processing cores onto a single integrated circuit. This shift to multicore architectures led to a new era of parallel computing, unlocking significant performance gains for a wide range of applications.

3.5.1 Uniprocessor vs. Multicore Architectures

A uniprocessor system consists of a single Central Processing Unit (CPU). A single CPU can execute one task at a time, at any given moment. As a result of their hardware constraints, uniprocessor systems cannot perform efficient parallel processing, yet, multitasking can be supported in such systems. Multitasking on a uniprocessor means that more than one task can be executed on a time-sharing basis, where the CPU rapidly switches between tasks, giving the illusion of simultaneous execution.

In contrast, a multicore architecture integrates two or more cores, onto a single integrated circuit. This design enables true hardware parallelism, where multiple tasks can execute genuinely simultaneously, each on a different core. This parallel execution capability can lead to significant improvements in computational throughput and overall performance, especially for applications that can be decomposed into parallel sub-tasks.

The adoption of multicore platforms in embedded real-time systems had been slower, primarily due to these reasons:

- the increasing complexity of programming parallel applications. Parallel tasks need to be coordinated by using complex synchronization mechanisms. Also parallel task access and communicate through shared memory which requires mutual exclusion and complex locking protocols;
- the speed-up of parallel applications is limited by contention on shared resources (memory and bus). The gains in terms of WCET are not evident (see Section 3.7).

3.5.2 Classification of Multicore Architectures

Multicore architectures can be classified based on several design characteristics. From the perspective of real-time systems, the two most important classifications concern the composition of the cores and the organization of the memory subsystem.

The first classification distinguishes between homogeneous and heterogeneous architectures:

- **Homogenous architectures** consist of multiple identical cores. The symmetric design semplifies task scheduling and load balancing, as all cores are capable of executing the same set of instructions and have the same performance characteristics;
- **Heterogeneous architectures**, in contrast, integrate different types of cores onto the same chip. This can include a mix of high-performance cores and energy-efficient cores, as the ARM big.LITTLE architecture [79], or a combination of general-purpose CPUs and specialized accelerators like Digital Signal Processors (DSPs) or Graphics Processing Units (GPUs). This second type of architecture is the most common in modern embedded systems. Performance between different core types are not related.

While core composition is important, the memory architecture is arguably the most critical factor influencing timing predictability. Here, the primary distinction is between shared and distributed memory models:

- **Shared Memory Architectures** use a global memory space accessible by all cores, allowing them to access the same main memory through a shared bus (Figure 3.7). This simplifies programming, as cores can communicate and share data implicitly by reading from and writing to the same memory locations. The vast majority of embedded multicore platforms are based on this design;

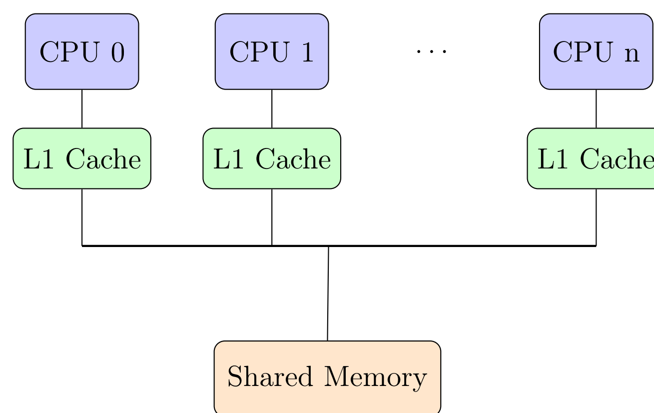


Figure 3.7: Shared Memory Architecture model.

- **Distributed Memory Architectures** consist of multiple local memory spaces, each associated with a specific core. Communication between cores is handled explicitly through an interconnection network, typically via message passing (Figure 3.8).

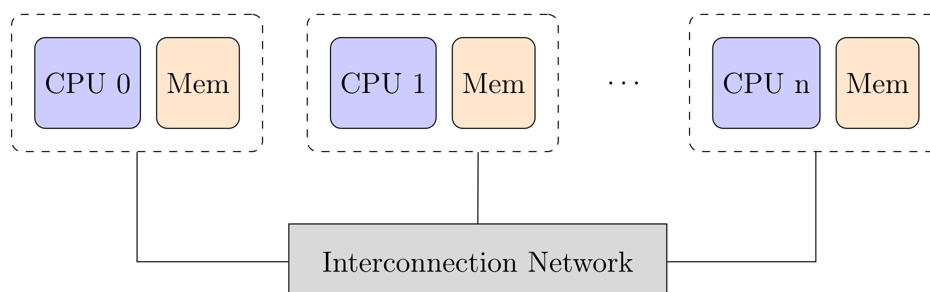


Figure 3.8: Distributed Memory Architecture model.

While a multicore architecture can provide significant performance improvements, it also introduces new challenges for real-time systems, typically resource sharing and timing unpredictability.

3.6 The Memory Subsystem

The challenge of resource sharing and timing unpredictability is rooted almost entirely within the memory subsystem. While the processing cores themselves can operate in parallel, they must ultimately contend for access to a common set of memory resources to fetch instructions and manipulate data. The performance and, more critically, the timing behavior of any single core are therefore no longer independent but are instead coupled with the memory access patterns of all other cores in the system.

3.6.1 The Problem with Caches in Real-Time Systems

The heart of the memory subsystem in most multicore embedded systems is a cache hierarchy, which is designed to improve performance by storing frequently accessed data closer to the processing cores. Caches are small, fast memory units that sit between the CPU and the main memory (RAM). Usually each core has its own private small Level 1 (L1) cache, and there may be a shared Level 2 (L2) cache or even larger caches at Level 3 (L3) or beyond. Figure 3.9 illustrates a typical structural arrangement.

This hierarchical structure is designed to bridge the performance gap between the fast CPU cores and the slower main memory. The relationship between cache level, access speed, and capacity is conceptually shown as a pyramid in Figure 3.10.

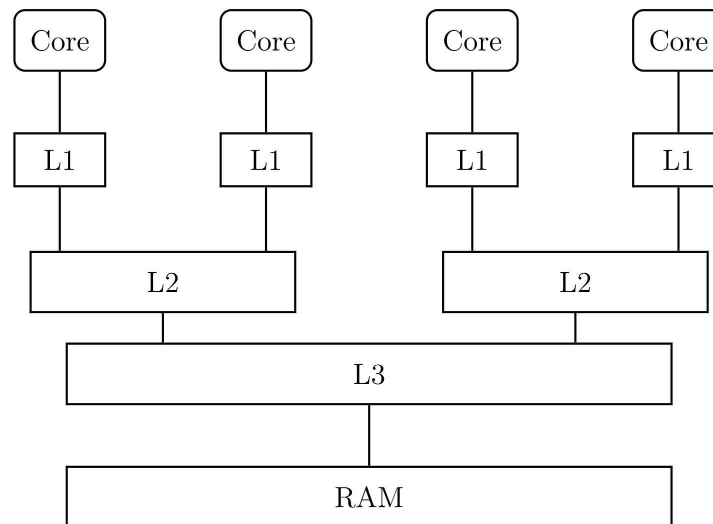


Figure 3.9: Example of a structural cache hierarchy for a quad-core system.

This is made to bridge the performance gap between the CPU and the slower main memory. The cache stores copies of frequently accessed data and instructions, so that subsequent requests for the same data can be served quickly from the cache (a cache hit) rather than incurring the high latency of a main memory access (a cache miss). While this mechanism is highly effective at improving average-case performance, its inherent unpredictability makes it a significant liability for real-time systems [80].

The unpredictability of caches stems from several sources of timing interference, which can be broadly categorized as follows:

- **Intra-task Interference:** a task can evict its own cache lines. This occurs when the task's working set is larger than the cache or when different memory locations used by the task map to the same cache set, causing conflict misses;
- **Intra-core Interference:** on a preemptive system, a task's cached data can be evicted by a different task that preempts it on the same core. When the original task resumes, it

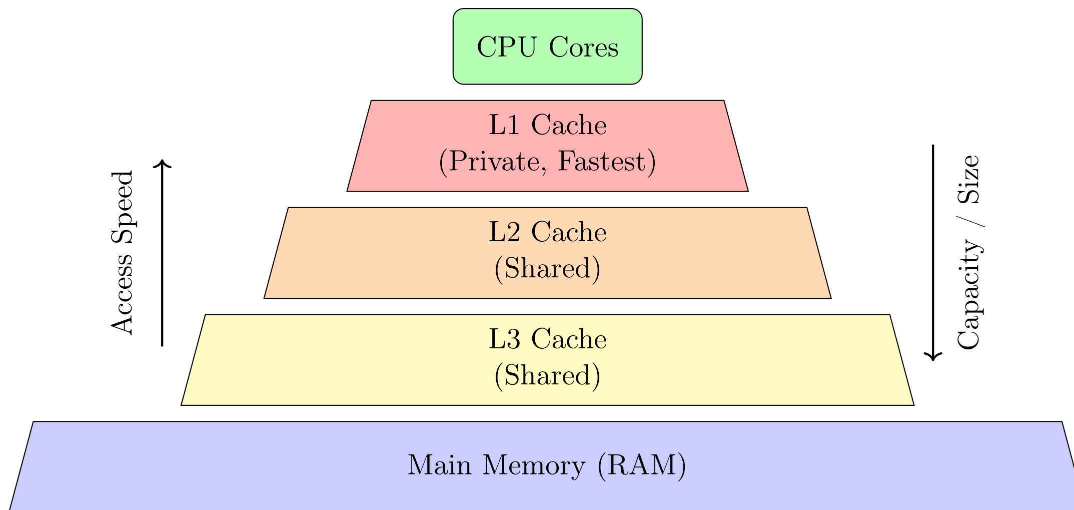


Figure 3.10: Cache Hierarchy in Multicore Systems.

suffers a burst of cache misses as it reloads its data, a delay known as the Cache-Related Preemption Delay (CRPD);

- **Inter-core Interference:** in a multicore system, multiple cores can access the same shared cache (e.g., L2 or L3). If one core evicts data that another core needs, it can lead to cache misses and increased access times for the affected core. This is particularly problematic in systems where tasks are distributed across multiple cores, as the timing behavior of one task can be influenced by the memory access patterns of others.

Due to these factors, calculating a tight Worst-Case Execution Time (WCET) for a task becomes challenging. Many progresses in the field of real-time systems have focused on developing techniques to analyze and bound the timing behavior of tasks in the presence of caches [81, 82, 83, 84]. However, these techniques often rely on pessimistic assumptions to ensure safety, such as treating every memory access as a potential cache miss.

3.6.2 Scratchpad Memories as a Predictable Alternative

As an alternative to caches, some multicore embedded systems use Scratchpad Memories (SPMs). A SPM is software-managed, on-chip static RAM, similar to a Level 1 (L1) cache. However, unlike caches, scratchpad memories do not automatically store data. Instead, they require explicit management by the programmer or compiler, which must decide what data to load into the scratchpad and when to evict it.

While this adds a layer of software complexity, it provides several profound advantages for real-time systems: (i) deterministic access times, since there is no concept of a cache hit or miss; (ii) elimination of hardware-managed conflicts, as the software has absolute control over data placement; and (iii) strong temporal isolation, because a core's access to its private SPM is not impacted by the memory activity of other cores.

However, the need for explicit management introduces its own set of challenges. Legacy programs cannot be easily executed on SPM-based systems without modification, and the software is responsible for managing the limited SPM space efficiently. This has led to significant research into developing efficient static and dynamic allocation techniques [85, 86, 87]. Furthermore, SPMs and caches present different fragmentation trade-offs. Caches suffer from internal fragmentation (wasted space if an application does not use an entire cache line), whereas SPMs suffer from external fragmentation (difficulty in managing variably-sized data blocks in a contiguous memory space).

Many modern multicore platforms offer solutions with scratchpad memories, such as the NXP S32 series, Renesas R-Car SoCs, and Infineon AURIX TriCore.

3.7 Shared Bus and Main Memory Contention

The second main factor of temporal unpredictability in a multicore processor that we consider in this work is contention for shared resources. When multiple cores or DMA engines attempt to access main memory simultaneously, a hardware arbiter grants access to one master while forcing the others to wait. A critical issue for real-time systems is that these arbiters are typically priority-agnostic; they treat requests from all sources equally, often using simple policies like round-robin or first-come-first-served [88]. This means a low-priority task can delay a high-priority one, creating an uncontrolled source of priority inversion at the hardware level and making it extremely difficult to bound the worst-case memory access latency.

To manage this complexity, two general approaches have emerged in the literature: (i) attempting to formally analyze the worst-case interference to derive a safe, often pessimistic, bound on the delay; or (ii) enforcing temporal isolation at the system level to prevent interference from occurring in the first place, for example by using time-triggered bus schedules. In the first approach, the hardware platform and the task execution must be modeled accurately: the memory access profiles of all tasks are extracted and combined with each other to estimate the worst-case profile. In general, it is difficult to precisely compute the worst-case interference profile, which likely leads to include scenarios that might never occur, therefore over-estimating the worst-case interference cost. The second approach tends to prevent interference by enforcing time isolation (e.g. time partitioning schemes like MemGuard [89]).

A third intermediate approach is to use a task model that separates memory access phases from computation phases. Two examples of such models are the PREM and AER models, which we will describe in the next section.

3.7.1 PREM and AER Model

In the PREM (Periodic Real-time Execution Model) model [90], the task code is divided into a set of scheduling intervals executed sequentially at run-time: compatible intervals and predictable intervals. Predictable intervals are divided into two phases: a *non-preemptive* memory phase and a *preemptive* execution phase. In the memory phase, the core accesses to the main memory to perform data fetches [91, 90]. At the end of the memory phase, all required data is available in the core local memory. Therefore, during the computation phase, the task can perform computations without any need to access to the main memory.

The AER model [49] introduces a more structured approach to task execution by separating the memory and computation phases. Under the AER model, tasks are divided into three distinct phases: acquisition (A), execution (E) and restitution (R), as it is possible to see in Figure 3.11. The acquisition phase is responsible for fetching the necessary data from the main memory, the execution phase performs the computations and the restitution phase writes back the results to the main memory. This separation allows for better management of memory accesses and computation, leading to improved predictability and performance and adding the possibility of parallel execution of tasks.

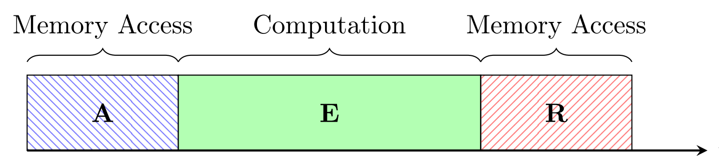


Figure 3.11: The AER task model.

The structure of the AER model aligns well with the practical execution flow of many data-intensive computational workloads, such as those found in digital signal processing, image processing, or linear algebra. These tasks naturally follow the A-E-R pattern:

- during the acquisition (A) phase, the task performs memory-intensive operations to fetch a block of input data and any required parameters;
- once all data is available in local memory, the execution (E) phase begins, performing the core computation without accessing the main memory bus;
- finally, the restitution (R) phase transfers the resulting output data back to main memory.

3.8 Conclusion

This chapter has provided a comprehensive overview of the fundamental principles of real-time systems, establishing the theoretical groundwork necessary for the analysis presented in this thesis.

We started with the basics for single-core processors, including the classic Liu and Layland task model and key scheduling methods like Rate-Monotonic (RM) and Earliest Deadline First (EDF). We then moved to modern multi-core processors, comparing partitioned vs. global scheduling and introducing models for parallel tasks like DAGs and Fork-Join.

Most importantly, this chapter explained the biggest challenge for getting predictable timing on multi-core chips: the memory system. We showed how unpredictable delays come from two main sources: interference in the cache and competition for the shared memory bus.

To help solve these issues, we looked at two key ideas: using Scratchpad Memories (SPMs) instead of caches and using task models like PREM and AER that separate memory access from computation.

With this background, the next chapter will now use these concepts to build a solution. We will develop a new system model and scheduling framework to make complex applications run predictably on multi-core hardware.

PART II

Contributions

Parallel Real-Time Execution of CNN Applications

The preceding chapters have presented a brief overview of the foundation upon which this work is built.

We have now all the elements to proceed to the contributions. Before doing that, it is important to establish the objective set in this thesis: to execute a CNN inference application on a multicore embedded platform featuring distributed scratchpad memories, together with other real-time tasks, such that all deadlines are met. To achieve this, we need to (i) define an appropriate task model for CNN inference, (ii) model the hardware platform, and (iii) develop a methodology to transform the CNN into the chosen task model and test its schedulability on the target platform.

To do so, in this chapter we introduce the task model we will use to represent CNN inference tasks: the AECR-DAG model. This model combines the AER model, which separates memory-bound operations from computation, with the DAG model, which captures the complex dependencies between CNN layers. Moreover, we describe the hardware platform model, a multicore architecture with scratchpad memories and DMA engines.

4.1 AECR-DAG Task Model

After introducing the AER and the DAG models in Chapter 3, we can define the task model used to represent CNN inference tasks in this thesis: the AECR-DAG model.

AECR-DAG model has been introduced in [7]. This model combines the AER model's structured approach to task execution with the DAG's ability to represent complex dependencies between operations and enhances temporal predictability by decoupling a task's computation phase from its memory access phases. It partitions a task into four distinct stages: an *Acquisition* phase, where necessary data is fetched from one external memory to the local memory of the core where the task is assigned; an *Execution* phase, where the computation is performed using only local data, thus avoiding contention on shared resources; a *Communication* phase, where data is transferred between local memories, from one execution stage to the next one; and a *Restitution* phase, where results are transferred from the local memory to an external memory. We extend the model with the characterization of the memory usage of the computational nodes. This allows for a structured representation that captures both the computational dependencies and the memory access patterns inherent in CNNs.

We consider that a task $\tau_i = (\mathcal{G}_i, D_i, T_i)$ is characterized by three parameters:

- the task structure expressed as a DAG $\mathcal{G}_i$;
- the period (or minimal interarrival time) T_i;
- the relative deadline D_i.

We assume that a task structure is described by a DAG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_1, v_2, \cdots, v_n\}$ is the set of nodes and $\mathcal{E}$ is the set of edges.

An edge $e(v_i, v_j) \in \mathcal{E}$ models a precedence constraint and related communication between nodes v_i and v_j. This means that node v_j must wait for the result of node v_i before it can be performed. In other words, the data produced by v_i is used as input for v_j.

The set of *immediate predecessors* of node v_j, denoted by $\mathsf{pred}(v_j)$, is the set of all nodes v_i such that there exists an edge $e(v_i, v_j)$. The set of *predecessors* of v_j is the set of all nodes for which there exists a path toward v_j.

Similarly, we define the set of *immediate successors* of v_j, denoted by $\mathsf{succ}(v_j)$, as the set of all nodes v_k such that there exists an edge $e(v_j, v_k)$. The set of *successors* of v_j is the set of nodes for which there is a path from v_j.

The vertices $\mathcal{V}$ of the task graph are partitioned into two disjoint subsets: a set of computational nodes, denoted $\mathcal{V}^C$, and a set of communication nodes, $\mathcal{V}^M$.

4.1.1 Computational Nodes

Computational nodes $v \in \mathcal{V}^C$ represent the *execution* (E) phase of the AER model and correspond to the operations performed by the CNN's layers (e.g., convolution, pooling, activation functions). These nodes are organized into a sequence of layers, $L_0, L_1, \ldots, L_k$. A specific computational node is denoted by v_i^l, signifying it is the i-th node in layer L_l. The total set of vertices is the union of all layers: $\mathcal{V}^C = \bigcup_{l=0}^{k} L_l$.

An edge $e(v_i^l, v_j^p) \in \mathcal{E}$ models a precedence constraint and the associated data communication between node v_i^l from layer L_l and node v_j^p from layer L_p. This means that node v_j^p must wait for the result of node v_i^l before it can be performed. The directed, acyclic nature of the graph ensures that $l < p$.

The set of predecessors $\mathsf{pred}(v_j^p)$, is the set of all nodes $\{v_i^l \mid e(v_i^l, v_j^p) \in \mathcal{E}\}$. Nodes in the input layer L_0 have no predecessors.

Similarly, the set of successors $\mathsf{succ}(v_i^l)$, is the set of all nodes $\{v_j^p \mid e(v_i^l, v_j^p) \in \mathcal{E}\}$. Nodes in the final layer L_k have no successors.

While in simple feed-forward networks an edge typically connects a node v_i^{l-1} to a node v_j^l, our DAG model is more general. It can represent advanced architectures with skip-connections (e.g., ResNet) by allowing an edge to connect non-adjacent layers, where $l < p - 1$. This flexibility is a key strength of the DAG representation for modern CNNs.

Each computational node $v_i^l = (f_i, \alpha_i, \beta_i, \gamma_i)$ is characterized by:

- f_i is the function that contains the code to be executed (e.g. the convolution function). This function contains the implementation of the CNN layer's operation;

- α_i is the set of input data buffers required by f_j. The total size of these buffers, α_i, corresponds to the amount of data that must be loaded from an external memory to the local memory of the core the task will be executed on, during the preceding acquisition phase. The data itself is produced by the set of immediate predecessor nodes, $\mathsf{pred}(v_j)$;

- β_i is the set of output data buffers produced by f_j. The size of these buffers, β_i, corresponds to the amount of data that must be written from the local memory to the external memory of the core its successor is assigned, during the subsequent restitution phase. This data is consumed by the set of immediate successor nodes, $\mathsf{succ}(v_j)$;

- γ_i represents the private memory footprint of the node, which includes the stack space and any temporary internal buffers required by f_j during its execution.

Computation nodes are assigned to cores during the allocation phase, that will be explored in Sections 5 and 6.

The execution of a computational node v_j is **preemptive**, allowing the scheduler to interrupt it to execute a higher-priority task.

In Figure 4.1, we can see a visual representation of a computational node v_i^l.

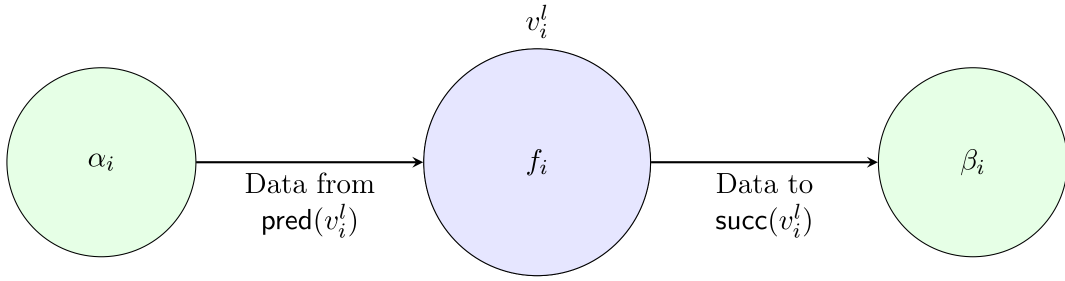


Figure 4.1: A visual representation of a computational node v_i^l.

4.1.2 Communication Nodes

Communication nodes form the subset $\mathcal{V}^M = \{\mu_1, \mu_2, \ldots, \mu_s\}$. These nodes represent the memory-bound *acquisition* (A) and *restitution* (R) phases of the AER model. Unlike computational nodes, they do not belong to a specific layer but instead model the data flow required to traverse an edge between layers.

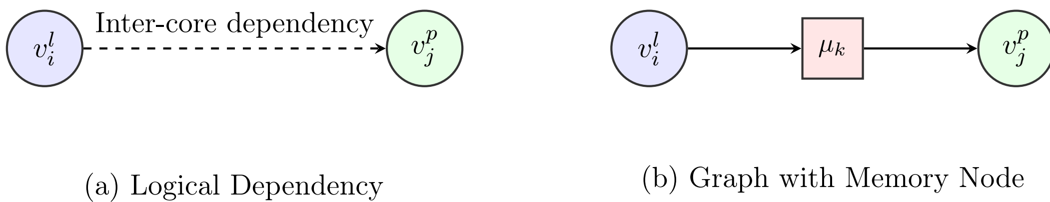
A communication node $\mu_k \in \mathcal{V}^M$ is instantiated in the graph for each data dependency that crosses a core boundary. For every edge $e(v_i^l, v_j^p)$ between two computational nodes that are allocated to different cores, the edge is replaced by a path through μ_k, creating two new edges: $e(v_i^l, \mu_k)$ and $e(\mu_k, v_j^p)$. This transformation is illustrated in Figure 4.2.

The cost of this node is determined by the volume of data that must be transferred over the bus. This explicit modeling makes the mapping from logical data dependencies to physical bus transactions clear and analyzable.

A *communication node* of the task graph, $\mu_k = (\delta_k, \mathsf{src}_k, \mathsf{dst}_k)$ is characterized by a source memory src_k, a destination memory dst_k and a transfer time δ_k.

Communication nodes are used to represent transfer of data between computation nodes that are allocated on different cores. Therefore, most communication nodes have one single predecessor and a single successors that are computation nodes. The exception are the communication nodes that represent transfer of data to/from the DRAM.

A defining characteristic of all communication nodes is that they are executed **non-preemptively**. Once a data transfer is initiated on a shared bus, it runs to completion without interruption, ensuring predictable access times.



(a) Logical Dependency

(b) Graph with Memory Node

Figure 4.2: Transformation of a logical inter-core dependency into an explicit memory transfer node within the DAG. The dashed edge in (a) is replaced by the path through the non-preemptive memory node μ_k in (b).

The resulting task graph $\mathcal{G}_i = (\mathcal{V}^C \cup \mathcal{V}^M, \mathcal{E})$ captures the complete structure of the CNN inference task, including both computational and memory transfer operations. This model allows us to analyze the task's execution flow, memory requirements, and scheduling behavior on a multicore platform. A more complete example of a CNN structure will be shown in Section 6.

4.2 Hardware Platform Model

We can now define the hardware platform model on which the AECR-DAG task model will be executed. The reference architecture described in this thesis is a simplified version of the Tricore Infineon AURIX TC3XXX. However, the model is general enough to be applied to any multicore platform with scratchpad memory.

We consider a multicore platform composed of M cores, denoted as $\Pi = \{\pi_1, \ldots, \pi_M\}$. Each core has access to a hierarchy of local memories and a shared main DRAM. To ensure temporal predictability, we establish a precise model for memory organization and access.

4.2.1 Memory Architecture

Each core π_p is associated with the following memory units:

- Local Code Memory: This memory contains the executable code for the operating system and any software threads assigned to that core. We assume a static partitioning of threads, meaning the code for a thread is fixed to a specific core's memory;
- Two separate scratchpad data memories that contain global data and stack memory for the threads executing on the core. We denote the first scratchpad memory of core p with σ_p^1 and the second with σ_p^2, their capacity is $\texttt{cap}(\sigma_p^1)$ and $\texttt{cap}(\sigma_p^2)$.

In addition to the cores and their memories, the platform model includes dedicated Direct Memory Access (DMA) engines. A DMA is a hardware component designed to manage data transfers between memory locations—such as from the main DRAM to a core's scratchpad—independently of the main processor cores. The primary advantage of using a DMA is that it offloads memory copy operations from the CPUs, allowing computation and communication to occur in parallel. While the DMA is busy moving data across the bus, the cores can continue to execute their computational tasks using their local memories. Our reference architecture includes two such DMA engines to facilitate efficient data movement.

The cores, their local memories, and the main DRAM are interconnected via a crossbar switch. This architecture is depicted in Figure 4.3.

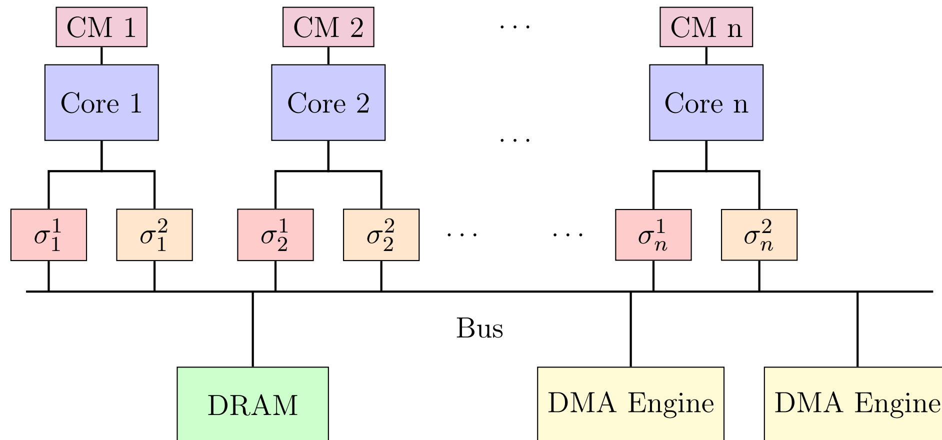


Figure 4.3: The proposed hardware platform model. Each of the M cores has a private Code Memory (CM) and two scratchpad (SPM). The two SPM memories are connected to a shared bus, which also provides access to the main DRAM and a DMA Engine.

4.2.2 Data Transfer

Data transfers between any two memories (e.g., local-to-local or main-to-local) are performed by DMA engines. To prevent unpredictable access times and inter-core interference, we enforce

a strict protocol: code executing on a core π_p exclusively accesses its own local memories, σ_p^1 and σ_p^2. All data movement between memories is explicitly managed via the DMA.

The crossbar switch allows for parallel, non-interfering accesses. For instance, a core can access its scratchpad σ_p while a DMA engine transfers data to its external memory λ_p without contention. However, interference occurs when a core and a DMA engine attempt to access the *same* local memory simultaneously.

On our reference platform, the DMA has priority over the core. Consequently, if a core initiates a memory access while a DMA transfer to that same memory is in progress, the core's access is stalled until the DMA transfer completes. This behavior is modeled as a preemption mechanism at the memory access level, where core execution is blocked by higher-priority DMA activity. This blocking delay is a critical parameter for the schedulability analysis detailed in Section 6.

4.3 Conclusion

In this chapter, we have introduced the AECR-DAG task model, which combines the structured approach of the AER model with the dependency representation capabilities of DAGs. It provides a clear framework for analyzing and optimizing CNN inference tasks on multicore platforms with scratchpad memories.

We have also detailed the hardware platform model, which includes a multicore architecture with local and scratchpad memory, and a DMA engine for data transfers. This platform is designed to ensure temporal predictability and efficient memory access, addressing the unique challenges posed by CNN inference tasks.

Chapter 5

Optimizing CNN Inference on Multicore Scratchpad Architectures

The preceding chapter established the AECR-DAG task model as a formal framework for representing CNNs and detailed a predictable multicore hardware platform equipped with scratchpad memories. Building upon this foundation, this chapter addresses the central problem of optimizing the inference process.

The core of our approach is to decompose the high-level CNN graph into a set of fine-grained, manageable tasks. To this end, we first introduce a formal execution model centered around the concept of a *localized operation*, which encapsulates both computational and memory transfer costs. This model allows for a precise analysis of execution time. Subsequently, we explore the strategy of merging these localized operations to reduce memory bus contention.

To solve the complex problem of mapping and scheduling these operations, we formulate the optimization task as an Integer Linear Programming (ILP) problem. We present two distinct strategies: a global approach that optimizes the entire network simultaneously and a distributed approach that handles each layer independently.

5.1 Execution Model

Having established the hardware platform model, the next step is to define how a CNN is executed on it. This section, therefore, introduces the execution model.

5.1.1 Localized Operations

A *localized operation* represents the smallest unit of work within the CNN model and it correspond to a computational node into the AECR-DAG graph, as introduced in Section 4.1.1. It denotes a set of arithmetic operations that can only be executed after loading the necessary input data into one of the scratch-pad memories (SPMs) of the assigned processor. A localized operation corresponds, for example, to a single convolution operation or a pooling operation within a CNN layer. It is the minimal unit of computation that can be scheduled and executed on a core.

Definition 2. *(Localized operation) A localized operation v is defined by the computation it must perform and the data it requires.*

- $\mathsf{C}(v)$*: the computational cost, representing the time required to execute all arithmetic operations within v once its data is available locally;*
- $\mathsf{M}(v)$*: the memory-set, representing all data accessed by the operation.*

The total time to complete a localized operation depends on both its computational cost and the time taken to fetch its memory-set.

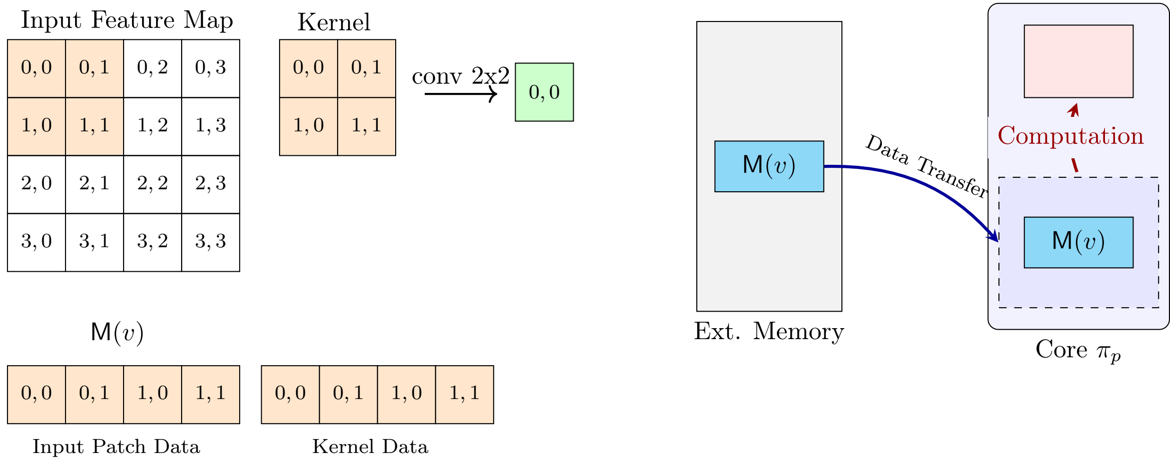


Figure 5.1: Execution sequence of a localized operation v.

Definition 3. *(Execution time of a localized operation) Let v be a localized operation. Its total execution time, $\mathsf{E}(v)$, is the sum of its computation time and its memory transfer time:*

$$\mathsf{E}(v) = \mathsf{C}(v) + \mathsf{cost_memory}(\mathsf{M}(v)) \tag{5.1}$$

where the function $\mathsf{cost_memory}(\cdot)$ computes the total time required to transfer the memory-set $\mathsf{M}(v)$. This cost will be defined in the following Section 5.1.2.

Figure 5.1 illustrates the data requirements and execution flow of a localized operation, v. The left side shows that an operation (e.g., at output $(0, 0)$) requires a corresponding data patch from the input tensor as well as the kernel weights. This complete memory-set is $\mathsf{M}(v)$. The right side shows the execution process on a hardware core, where this memory-set is moved from external memory to a local buffer before computation.

5.1.2 Memory Transfer Cost

To calculate the execution time $\mathsf{E}(v)$ of a localized operation v, we need to determine the time required to transfer its memory-set $\mathsf{M}(v)$. The function $\mathsf{cost_memory}(\cdot)$ computes the time to transfer a memory-set to a local memory. This cost is influenced by different factors:

- Data Linearization: multi-dimensional tensors are stored as one-dimensional arrays in physical memory, typically following a row-major ordering convention, as illustrated in Figure 5.2. This linear storage is typically handled automatically by the compiler or the memory management unit;

- Non-contiguous Access: operations like convolution require accessing a sub-tensor or patch. Due to linearization, this patch often maps to multiple, non-contiguous segments in the 1D memory space;

- Batched Transfers: to manage non-contiguous access, the memory controller groups data fetches into batches. Each batch corresponds to a single contiguous block of data. Initiating each batch transfer incurs a fixed latency overhead;

- Coalesced Transfers: within a batch, data is moved in fixed-size chunks, or coalesced units, to maximize bus utilization.

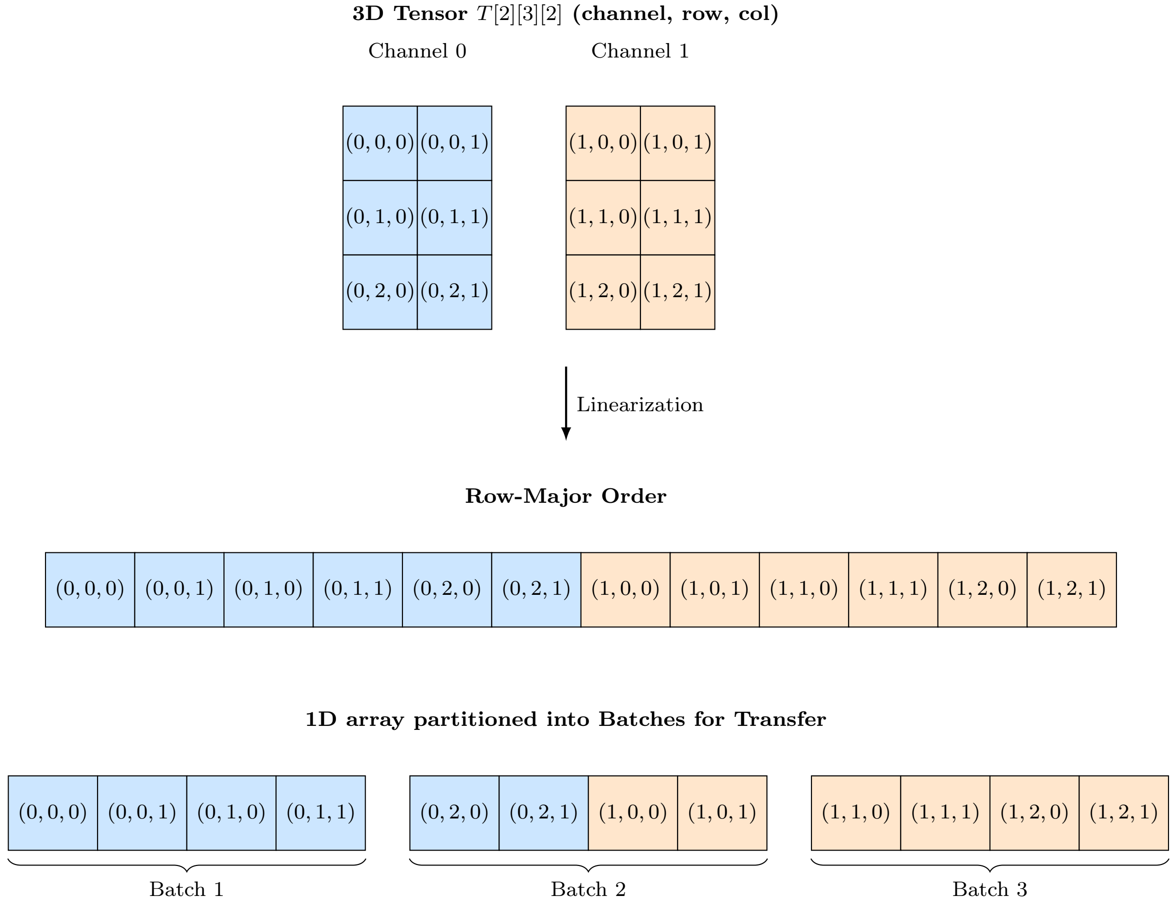


Figure 5.2: Linearization of a 3D tensor of size [2][3][2] into a 1D memory array, illustrating the row-major order.

The function $\mathsf{cost_memory}(\cdot)$ computes the total time to transfer a memory-set $\mathsf{M}(v)$ by summing the latency overhead for all batches and the time required for the actual data movement. The memory-set is partitioned into N_B batches, where each batch represents a contiguous block of data. The total cost is formulated as:

$$\mathsf{cost_memory}(\mathsf{M}(v)) = N_B \cdot \text{cost}_{\text{batch}} + \left\lceil \frac{|\mathsf{M}(v)|}{\Delta} \right\rceil \cdot \text{cost}_{\text{xfer}} \tag{5.2}$$

where:

- N_B is the number of non-contiguous memory batches required to fetch $\mathsf{M}(v)$. This term represents the total latency overhead;
- $|\mathsf{M}(v)|$ is the total number of data elements in the memory-set;
- Δ is the size (in elements) of a single coalesced memory transfer;
- $\text{cost}_{\text{batch}}$ is the fixed latency cost, in clock cycles, to initiate one batch transfer;
- $\text{cost}_{\text{xfer}}$ is the cost, in clock cycles, to transfer one coalesced unit of size Δ.

Example 5.1.1. *Consider an operation that requires a* $3 \times 3 \times 2$ *sub-tensor from a larger* $4 \times 4 \times 2$ *(*$H \times W \times C$*) input tensor. The input tensor is stored in memory in row-major order. The memory address for an element at* $[h][w][c]$ *is calculated as* $h * W * C + w * C + c$*.*

A) 3D Input Tensor ($4 \times 4 \times 2$)
Required Patch ($3 \times 3 \times 2$)

B) 1D Linearized Memory
Total Elements: $4 \times 4 \times 2 = 32$

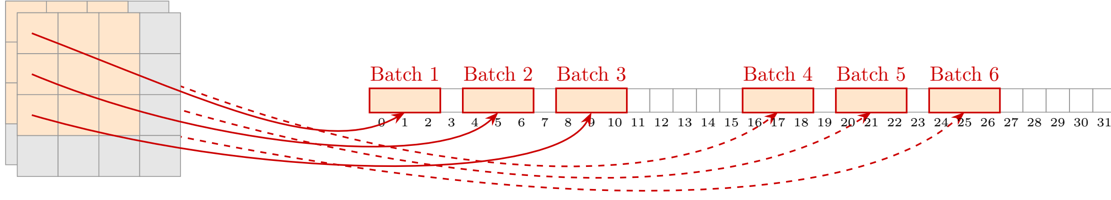


Figure 5.3: Mapping of a $3 \times 3 \times 2$ sub-tensor from a $4 \times 4 \times 2$ tensor to linearized memory. Accessing the sub-tensor requires fetching three separate, non-contiguous batches of data.

The access pattern for the required $3 \times 3 \times 2$ patch from a $4 \times 4 \times 2$ tensor stored in CHW format is as follows:

- ***Channel 0:***
  - *Row 0 ($h = 0$): elements $[0][0][0]$ to $[0][0][2]$. This is a contiguous block at indices $0 - 2$, forming Batch 1.*
  - *Row 1 ($h = 1$): Elements $[0][1][0]$ to $[0][1][2]$. This is a contiguous block at indices $4 - 6$, forming Batch 2.*
  - *Row 2 ($h = 2$): Elements $[0][2][0]$ to $[0][2][2]$. This is a contiguous block at indices $8 - 10$, forming Batch 3.*
- ***Channel 1:***
  - *Row 0 ($h = 0$): Elements $[1][0][0]$ to $[1][0][2]$. This is a contiguous block at indices $16 - 18$, forming Batch 4.*
  - *Row 1 ($h = 1$): Elements $[1][1][0]$ to $[1][1][2]$. This is a contiguous block at indices $20 - 22$, forming Batch 5.*
  - *Row 2 ($h = 2$): Elements $[1][2][0]$ to $[1][2][2]$. This is a contiguous block at indices $24 - 26$, forming Batch 6.*

The data is spread across six non-contiguous segments in memory. Therefore, the number of batches $N_B = 6$, as illustrated in Figure 5.3.

Let's assume the following hardware-dependent parameters:

- *Latency per batch, $cost_{batch} = 100$ cycles.*
- *Coalesced transfer size, $\Delta = 4$ elements.*
- *Cost per coalesced transfer, $cost_{xfer} = 8$ cycles.*

The total size of the memory-set is $|\mathsf{M}(v)| = 3 \times 3 \times 2 = 18$ elements. Using the cost function:

$$\begin{aligned}
\mathsf{cost_memory}(\mathsf{M}(v)) &= N_B \cdot cost_{batch} + \left\lceil \frac{|\mathsf{M}(v)|}{\Delta} \right\rceil \cdot cost_{xfer} \\
&= 6 \cdot 100 + \left\lceil \frac{18}{4} \right\rceil \cdot 8 \\
&= 600 + \lceil 4.5 \rceil \cdot 8 \\
&= 600 + 5 \cdot 8 = 640 \; cycles
\end{aligned}$$

The total memory transfer cost is 640 cycles. This is composed of 600 cycles of latency overhead for initiating the six separate batch transfers and 40 cycles for the actual data movement.

5.1.3 Merge of Localized Operations

To optimize the execution of the CNN, we can group multiple localized operations that are assigned to the same processor core. This process, which we define as a *merge operation*, combines several fine-grained operations into a single, coarser-grained one.

First, we formally define the assignment of operations to cores.

Definition 4. *(Mapping function) The mapping function,* $\mathsf{map}(v)$*, returns the processor core* p *to which a localized operation* v *is allocated:*

$$\mathsf{map}(v) = p$$

Using this, we can define the merge operation. A merge is only possible for operations that are mapped to the same core and belong to the same layer of the CNN.

Definition 5. *(Merge of localized operations) A merge of a set of localized operations,* Θ*, results in a new, single localized operation,* v'*, that encapsulates the work of all operations in* Θ*. The following conditions must hold:*

- ***Co-location:*** *all operations in* Θ *must be mapped to the same core, which will also be the location of the new merged operation* v'*.*

$$\forall v \in \Theta : \mathsf{map}(v) = \mathsf{map}(v')$$

- ***Layer Constraint:*** *all operations must belong to the same CNN layer,* l*.*

The properties of the new localized operation v' *are defined as follows:*

- ***Computational Cost:*** *the total computational cost of* v' *is the sum of the costs of the individual operations.*

$$\mathsf{C}(v') = \sum_{v \in \Theta} \mathsf{C}(v)$$

- ***Memory-Set:*** *the memory-set of* v' *is the union of the memory-sets of its constituent operations.*

$$\mathsf{M}(v') = \bigcup_{v \in \Theta, \mathsf{pred}(v) \notin \Theta} \mathsf{M}(v)$$

The primary benefit of the merge operation lies in the formulation of $\mathsf{M}(v')$. By taking the union of only the necessary input data, we significantly reduce the total data that needs to be transferred over the bus, if the operations in Θ share common input data.

Example 5.1.2. *Consider the example presented in Figure 5.4. It includes a first convolution step with a 2x2 kernel, followed by a pooling step (for instance a max function on 2x2). The lower part of the figure shows the memory mapping (here row-major) of the input data in the RAM (we assume that the weights of the kernel are already pre-loaded into the SPMs). The tuples of numbers in the figure represent the coordinates of the data in the tensor.*

The chosen strategy is to first copy this data into the SPMs before performing the operations. We notice that different convolution operations share common data (for example, the calculation represented by the black square shares data with the calculation represented by the red square). It is therefore possible to reduce copies by merging certain convolution operations,

thus relieving the data bus during transfers. However, we lose the ability to parallelize the work. Conversely, if the two calculations were done on different cores, the overall computation time would be shorter, but some data would need to be copied multiple times. Figure 5.4 shows the data transfer, denoted (1), to the SPM after grouping localized operations by column. The data required for the convolution products of the first column is transferred into core 0, for the second column into core 1, and so forth. We notice that the data is redundant on the different SPMs.

The second step is the convolution calculation, denoted (2) on Fig. 5.4. This step is performed in parallel on the different cores without memory contention.

Next, note that for pooling operations, it is also necessary to share data that may have been calculated on different cores. Therefore, memory transfers based on the grouping of pooling localized actions are also necessary. For example, if pooling is performed by row on a different core, core 0 needs a transfer step as shown in the figure (step (3)).

The final operation involves performing the pooling steps in parallel (step (4) in the figure).

Through this example, we see that the choice of merging localized operations can influence overall temporal performance. It is necessary to balance between data locality to minimize transfers and leverage burst memories, while maintaining parallelism to benefit from multiple computing cores.

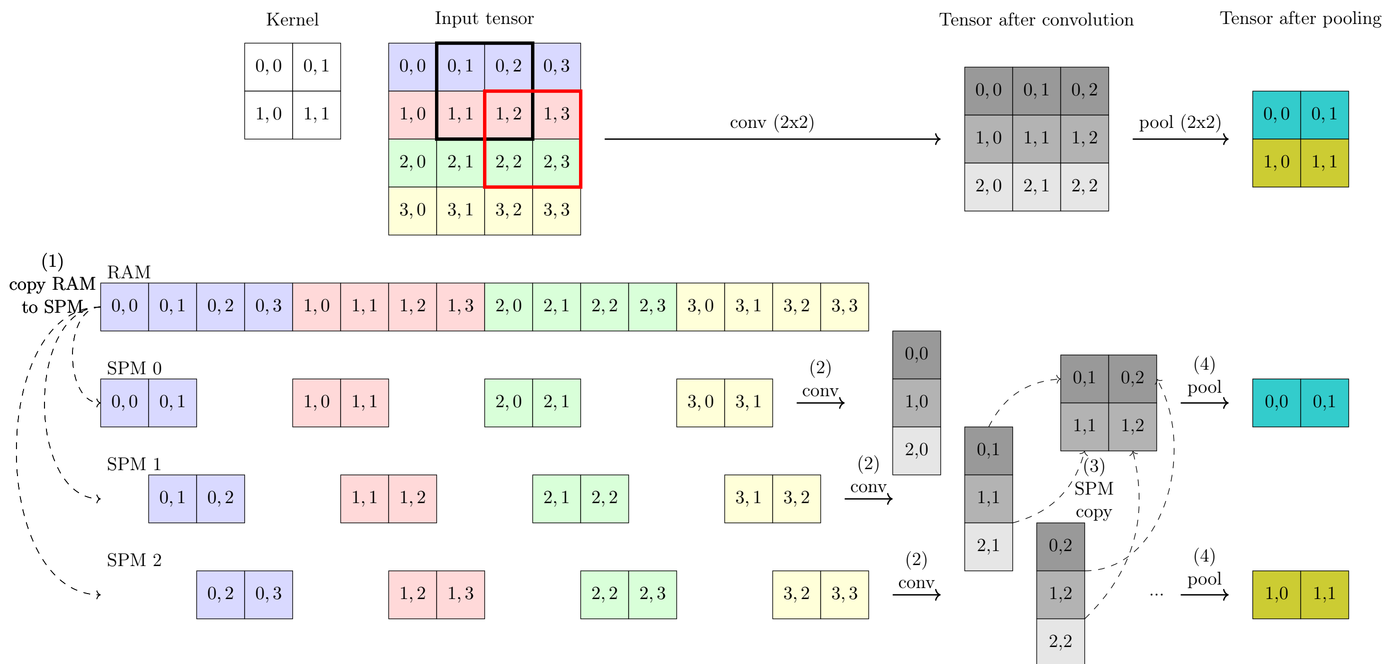


Figure 5.4: Example of merged localized operations

5.2 ILP formulation

Having defined the execution model based on localized operations and the merge strategy, the central challenge is to determine an optimal mapping of tasks to cores. This requires deciding:

1. Which localized operations to merge to create coarser-grained tasks.
2. To which processor core each of these tasks should be assigned (mapping).

To solve this problem and find a provably optimal balance, we formulate it as an Integer Linear Programming (ILP) model. ILP provides a powerful and formal framework for expressing our optimization goals and system constraints. This section details this formulation.

We first define the decision variables that capture the merging and mapping choices. We then establish the set of constraints that ensure the validity of the solution. Finally, we formulate the objective function.

5.2.1 ILP Problem Definition

A Linear Programming (LP) problem involves *optimizing* a *linear objective function* subject to linear constraints. Its standard form is:

$$\text{Maximize: } \mathbf{c}^T\mathbf{x}, \text{subject to: } A\mathbf{x} \leq \mathbf{b}, \mathbf{x} \in \mathcal{X}$$

where $\mathbf{c}^T\mathbf{x}$ is the objective function, $\mathbf{x}$ is a vector of m variables, $\mathbf{c}$ and $\mathbf{b}$ are coefficient vectors, and A is a coefficient matrix. The constraints $A\mathbf{x} \leq \mathbf{b}$ and $\mathbf{x} \in \mathcal{X}$ define the feasible region for optimization. LP solvers, using methods like the Simplex algorithm [92], efficiently find optimal solutions when $\mathcal{X} = \mathbb{R}^m$. If the variables are defined over the domain of integers, as in our case, the problem becomes an Integer Linear Programming (ILP) problem with $\mathcal{X} = \mathbb{Z}^m$, including binary variables as $\{0, 1\}$.

To implement and solve ILP problems, several commercial and open-source solvers are available. Prominent commercial solvers like Gurobi [93] and CPLEX [94] are known for their high performance and are widely used in academia and industry. In this work, we utilize CPLEX.

To model our optimization problem as an ILP, we first identify and define key parameters. While our method is general and can be applied to n-dimensional tensors, for simplicity, we focus here on 2D tensors representing images. We consider L layers of convolution or pooling. For a layer l, we introduce the following variables:

- n_h^l: number of columns in the input tensor for layer l.
- n_w^l: number of rows in the input tensor for layer l.
- N^l: number of localized operations for layer l.

The number of localized operations in a layer is equal to the product of the dimensions of the tensor of the next layer: Therefore, we have $N^l = n_h^{l+1} \times n_w^{l+1}$.

In this first contribution, we focus on optimizing the implementation of a single CNN. The objective is to allocate localized operations on cores, by merging them whenever appropriate and possible, so to ensure that the complete execution of the CNN meets its deadline, i.e. the execution of all operations must complete no later than D time units from its activation. Considering various parameters such as input size, the number of layers, and others, the design space for implementation can become extremely large to explore. To address this, we propose two distinct approaches.

The first proposed approach models the problem of implementing all layers as a single ILP. This method integrates all constraints into one optimization problem. The global approach gives better solutions, but it tends to be much more complex due to the increased size of the design space choices. When all constraints are respected, the CNN completes before it deadline, hence we look for a feasible solution and we set the objective function to zero to stop the ILP resolution at the first feasible solution.

In the second approach, we explore the design space by addressing the optimization of individual layers separately. For each layer, the objective is to minimize its completion time. The completion time of a given layer is set as the starting time for the next one. Each layer is optimized independently using a separate ILP formulation, and the execution of layers does not overlap in time. By optimizing sequentially from the first layer to the last, we aim to increase the chances of reducing the overall completion time, thereby ensuring the CNN meets its deadline.

In the remainder of this section, we first present a general ILP model, which will serve as the baseline ILP for the different approaches. Then, we describe each specific approach in detail.

Operation mapping. The binary decision variable $\mathbf{a} \in \{0, 1\}$ expresses the allocation of localized operation v_i of layer l on core p. For ease of presentation, we extend the notation of localized operation v_i to v_i^l to include its layer. Variable $\mathbf{a}_p^{i,l}$ is defined as follows:

$$\mathbf{a}_p^{i,l} = \begin{cases} 1 & \text{if } v_i^l \text{ is mapped to core } p \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

As the localized operation v_i^l must only be allocated to one processor at a time, the sum of all the a variables for a single localized operation across each processor must be equal to 1:

$$\sum_{p \in [1..P]} \mathbf{a}_p^{i,l} = 1 \tag{5.4}$$

Memory transfer. For each localized operation v_i^l, we have a vector corresponding to the size of the input tensor, representing the data that need to be loaded when the localized operation v_i^l is allocated to a given core. We use $b^{i,l}(x, y)$ to indicate whether data (x, y) is used as input by localized operation v_i^l, where (x, y) represents the coordinates of a specific data element within the input tensor. Thus, we can write:

$$b^{i,l}(x, y) = \begin{cases} 1 & \text{if the data } (x, y) \text{ is used by } v_i^l \\ 0 & \text{otherwise} \end{cases} \tag{5.5}$$

Note that $b^{i,l}(x, y)$ are input constants and not decision variables. In Figure 5.5, we visualize the binary indicator matrix $b^{i,l}$ for a localized operation v_0^l, where the red cells indicate the operation v_0^l and the orange cells indicate the data (x, y) that are used as input by the operation.

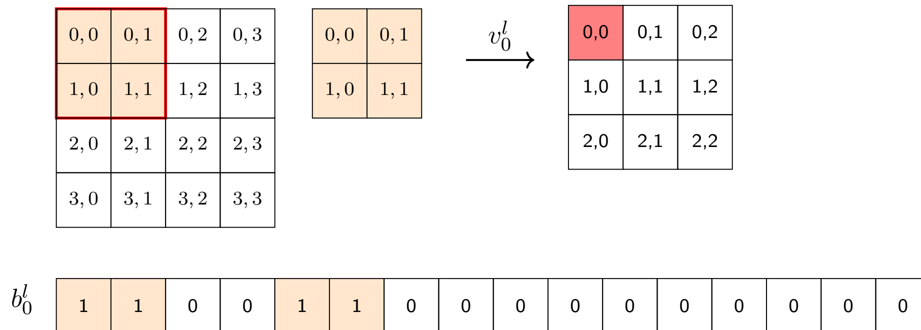


Figure 5.5: Visualization of the binary indicator b_0^l for operation v_0^l assigned to core p.

In the same vein, we define a function that associates an input tensor data with the localized operation that was used to compute it. In the presence of skip connections, input data for layer l can be produced by localized operations in *multiple* preceding layers. We denote this mapping with a generalized function $g^l(x, y) \subseteq \{(i, l')\}$, which returns the set of operations $v_i^{l'}$ (from layers $l' < l$) that compute the data (x, y) used as input by layer l. The functions $b^{i,l}(x, y)$ and $g^l(x, y)$ describe the structure of the dataflow graph and are provided as input to the model.

We now define the binary variables $\mathbf{u}_p^l(x, y)$, which indicate whether data (x, y) needs to be loaded on processor p to compute any localized operation in layer l. The variable $\mathbf{u}_p^l(x, y)$ is

equal to 1 if the input data (x, y) is accessed by any localized operation assigned to processor p, and is not already available in its memory.

To determine this, we observe that (x, y) may be generated by one or more operations $v_i^{l'}$ in previous layers. Thus:

$$\mathbf{u}_p^l(x,y) = \left(1 - \bigvee_{(i,l')\in g^l(x,y)} \mathbf{a}_p^{i,l'}\right) \cdot \left(\bigvee_{i\in[1..N^l]} \mathbf{a}_p^{i,l} \cdot b^{i,l}(x,y)\right) \tag{5.6}$$

The first term evaluates whether data (x, y) is already present in the local memory of processor p, i.e., if any of the predecessors of that data were assigned to p. The second term checks whether any operation on processor p requires this data. In Figure 5.6, we visualize the binary variable $\mathbf{u}_p^l$ for localized operations v_0^l and v_5^l assigned to core p. The red cells indicate the operations, while the orange cells indicate the data (x, y) that need to be loaded on core p. The two operations share one pixel, $(1, 1)$.

For the first layer, all input data are assumed to be stored in RAM and thus unavailable in any processor's local memory. Hence, for layer $l = 1$, we consider the first term to evaluate as 1.

While Equation (5.6) is expressed compactly, its non-linear nature, involving logical disjunctions (OR) and products, makes it unsuitable for a standard ILP solver. However, it can be transformed into an equivalent set of linear constraints. The logical OR, for instance, can be linearized using a common "Big-M" formulation.

To illustrate, let us linearize the term that determines if any operation on core p requires data (x, y). We can define an auxiliary binary variable $R_p^l(x, y)$ representing this condition:

$$R_p^l(x,y) = \bigvee_{j\in\mathcal{N}_l(x,y)} a_p^{j,l}$$

where $\mathcal{N}_l(x, y)$ is the set of operations in layer l that require data (x, y). This non-linear relationship is modeled with the following two linear inequalities.

First, an upper-bound constraint ensures that $R_p^l(x, y)$ can only be 1 if at least one relevant operation is assigned to the core:

$$R_p^l(x,y) \le \sum_{j\in\mathcal{N}_l(x,y)} a_p^{j,l} \tag{5.7}$$

Second, a constraint using a large constant, M, enforces the relationship in the reverse direction:

$$\sum_{j\in\mathcal{N}_l(x,y)} a_p^{j,l} \le R_p^l(x,y) \cdot M \tag{5.8}$$

Here, M is a sufficiently large constant, such as the total number of operations in the set $|\mathcal{N}_l(x, y)|$. If any $a_p^{j,l}$ on the left-hand side is 1, the sum becomes at least 1, which forces $R_p^l(x, y)$ to be 1 to satisfy the inequality. Conversely, if $R_p^l(x, y)$ is 0, this constraint forces all $a_p^{j,l}$ in the sum to be 0.

Together, these constraints ensure that $R_p^l(x, y)$ is 1 if and only if at least one of its input variables is 1. The full non-linear expression in Equation 5.6 can be linearized by applying this technique along with standard product linearization for binary variables.

Coalesced data. Now, let us suppose that the first input data is aligned with a coalesced block for memory transfer[1]. We denote N_Δ^l as the number of coalesced blocks needed to

[1] Some compilers have specific flags that can be used to align data in memory. For example, GCC has the '-malign-*' options to enforce array alignments.

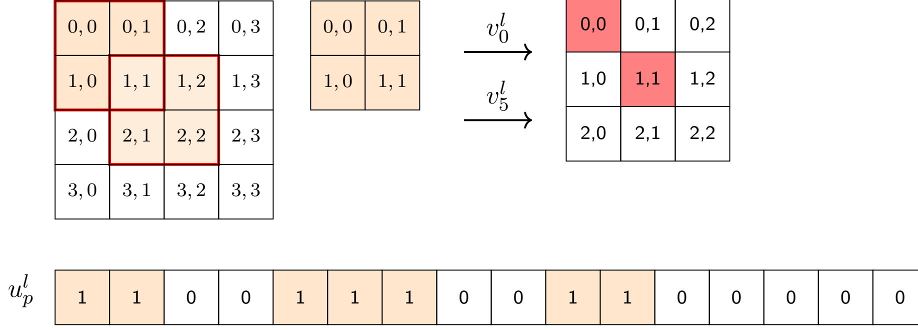


Figure 5.6: Visualization of the binary variable $\mathbf{u}_p^l$. Operations v_0^l and v_5^l, shown in red, are assigned to core p.

transfer the entire input tensor for layer l, *i.e.* $N_\Delta^l = \left\lceil \frac{n_w^l . n_h^l}{\Delta} \right\rceil$. Let $\mathbf{c}_p^l(k)$ indicate whether the k-th coalesced block needs to be transferred to processor p to compute layer l. We have for all $k \in [1..N_\Delta^l]$:

$$\mathbf{c}_p^l(k) = \bigvee_{\substack{i \in [1 \ldots \Delta] \\ \wedge k \cdot \Delta + i < N_\Delta^l}} \mathbf{u}_p^l \left((k \cdot \Delta + i) \ \mathbf{mod} \ n_w^l, \left\lfloor \frac{k \cdot \Delta + i}{n_w^l} \right\rfloor \right) \tag{5.9}$$

Eq. 5.9 indicates that the k-th coalesced block is transferred to processor p if at least one of the data elements in the block is loaded on that processor. The OR operation can be linearized as explained above.

The data produced by localized operations are stored in the SPM memory of the core on which the operation was performed. Therefore, elements of the output tensor of a layer are distributed across multiple memories. The current model does not account for this distribution. To address this issue, we assume that computed data for layer l is stored such that the memory address on each SPM for a data point (x_1, y_1) is smaller than for (x_2, y_2) if $x_1 + n_w^l \cdot y_1 \leq x_2 + n_w^l \cdot y_2$ (however, there is no ordering between addresses on different SPMs). Under these conditions, the calculation of $\mathbf{c}_p^l$ serves as an upper bound on the number of memory transfers required to copy data from another SPM. The exact addressing problem is solved at code generation stage.

In Figure 5.7, we visualize the binary variable $\mathbf{c}_p^l$ for localized operations v_0^l and v_5^l assigned to core p. Instances of vector $\mathbf{c}_p^l$ are 1 if at least one of the data (x, y) in the coalesced block k has to be loaded on core p.

Let $\mathbf{n}_p^l$ represent the number of memory loads required for core p to compute layer l, taking into account the coalesced blocks. The number of memory loads $\mathbf{n}_p^l$ is given by summing the indicator function $\mathbf{c}_p^l(k)$ over all possible blocks k:

$$\mathbf{n}_p^l = \sum_{k \in [1..N_\Delta^l]} \mathbf{c}_p^l(k) \tag{5.10}$$

Execution time. Memory transfer from the RAM to the SPM or between SPMs is performed sequentially across the cores. This means that core p memory transfer depends on the completion of memory transfers from all previous cores. Hence, the time taken to perform memory transfers on a core p for a layer l is defined as:

$$t_p^l(\text{comm}) = \sum_{k=0}^{p} n_k^l \cdot (\text{cost}_{\text{xfer}} + \text{cost}_{\text{batch}}) \tag{5.11}$$

where $N_{B,k}^l$ is the number of coalesced blocks transferred to core k.

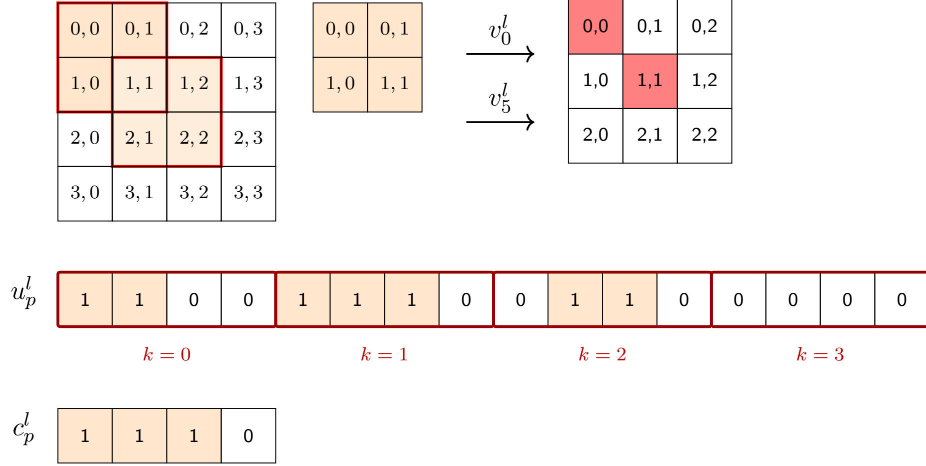


Figure 5.7: Visualization of the binary variable $\mathbf{u}_p^l$ and $\mathbf{c}_p^l$. Operations v_0^l and v_5^l, shown in red, are assigned to core p.

Once the memory transfer for core p is completed, computation on that core can start. Hence, the time taken to perform operations on a core p for a layer l is defined as:

$$\mathbf{t}_p^l(\text{op}) = \sum_{i \in [1..N^l]} \mathbf{a}_p^{i,l} \cdot \mathsf{C}(v_i^l) \tag{5.12}$$

The total completion time for core p for a layer l is the sum of its memory transfer time and its computation time:

$$\mathbf{t}_p^l = \mathbf{t}_p^l(\text{comm}) + \mathbf{t}_p^l(\text{op}) \tag{5.13}$$

Memory size constraint. The memory size constraint limits the total memory usage on processor p to its specified maximum $\mathsf{cap}(\sigma_p)$, introduced in Chapter 4, so we have for each $p \in [1..P]$:

$$\mathbf{n}_p^l \cdot \Delta + \sum_{i \in [1..N^l]} \mathbf{a}_p^{i,l} \leq \mathsf{cap}(\sigma_p) \tag{5.14}$$

where the first term represent the size used to store input data and the second term to store output.

Precedence Constraint. To ensure that the merge is performed correctly, we need to verify the precedence constraints.

Definition 6. *(Precedence Constraint for localized operations)*

Given two localized operation $(v_1, v_2) \in \mathcal{G}$, it exists a precedence constraint if there is an edge $e = (v_1, v_2) \in \mathcal{E}$ in the graph $\mathcal{G}$. This means that each precedence constraint ensures that a localized operation v cannot start execution until all its predecessor operations, denoted as $\mathsf{pred}(v)$*, have been completed.*

Formally, for each localized operation $v \in V$, the following condition must hold:

$$\max_{v' \in \mathsf{pred}(v)} \mathsf{end}(v') \leq \mathsf{start}(v), \tag{5.15}$$

where $\mathsf{end}(v')$ *denotes the completion time of operation v', and* $\mathsf{start}(v)$ *denotes the start time of v.*

This guarantees that dependencies between operations are respected and that no operation can be executed out of order.

Definition 7. *(Precedence Constraint for Merges)*

Given two merges Θ_1 *and* Θ_2*, it exists a precedence constraint between them if there is a precedence constraint between any pair of localized operations belonging to* Θ_1 *and* Θ_2*, respectively.*

In other words, if there exists a localized operation $v_1 \in \Theta_1$ *and* $v_2 \in \Theta_2$ *such that* $(v_1, v_2) \in \mathcal{E}$*, then a precedence constraint is established between the two merges.*

Implementation. We now discuss the implementation of the ILP. As previously described in Definition 3, all operations within a given layer that are assigned to the same core are merged into a single operation, which is executed by a single thread. Before starting the computation, the thread synchronizes on a barrier, i.e., semaphores, to ensure that all the data are available. Data transfers—either between RAM and SPM or across different SPMs—are managed by a dedicated thread manager, which synchronizes with other threads and ensures that data are moved from the appropriate sources to the corresponding destinations and in the correct order. The thread execution model is illustrated in Listing 5.1, which presents a pseudo-code representation of the execution loop for a thread handling the merge operation Θ_i. Synchronization barriers, used to coordinate thread execution, are available in all major operating systems.

Theorem 8

Let $\Theta_1, \Theta_2, \ldots, \Theta_m$ be sets of merged localized operations obtained as a feasible solution of the ILP. We assign each Θ_i to a thread τ_i, and threads are synchronized using synchronization barriers. Then, the execution order of the operations respects the precedence constraints defined by the ILP.

Proof. The proof follows from the synchronization properties of the barrier and the thread execution model. For a given merge Θ, the thread assigned to Θ cannot start its execution until all preceding merges $\Theta' \in \mathsf{pred}(\Theta)$ have completed their data transfers and are ready for synchronization.

Thus, the synchronization enforced by the barrier ensures that the execution order of merges respects the precedence constraints over the localized operations presented in the original Graph $\mathcal{G}$.

□

Listing 5.1: Thread execution loop for merge operation Θ_i

```
void *thread(void *arg)
{
    int i = // Index of merged operations
    while(true) {
        // synchronize on input data
        sem_wait(&input[i]);

        // Operation \( \Theta_i \)
        compute_merge_operation(i);

        // signals end of operation
        sem_post(&output[i]);
    }
}
```

5.3 Strategies to solve the problem

The model described in the previous section may involve a large number of mapping variables for localized operations which are dependent through data locality. As we will see in Section 5.4, this can lead to significant computation times or even make finding a solution infeasible. To address this issue, we propose two different approaches as we briefly mentioned in Section 5.2.

5.3.1 Global ILP

In the case of the first approach, we aim at optimizing the layers together rather than treating them separately. This integrated approach allows for a more holistic optimization meaning that the entire CNN inference is optimized in a way that maximizes efficiency and performance, taking into account the interactions and dependencies between the layers. By doing so, we can achieve a better balance between the layers, potentially leading to improved overall performance.

To represent the deadline bound, we simply introduce a relation between execution duration and deadline:

$$\sum_{l \in [1..L-1]} \mathbf{t}_p^l \leq D \tag{5.16}$$

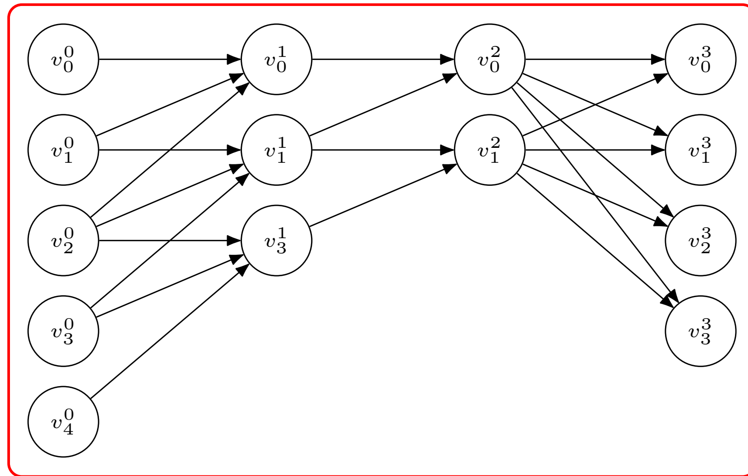


Figure 5.8: Structure of the Global ILP strategy. The entire CNN dataflow graph, spanning all layers, is considered as a single optimization problem constrained by one overall deadline.

Here we do not define an objective function and only try to find a feasible solution.

5.3.2 Distributed ILP

The second approach involves decoupling each layer and solving the problem independently for each of them. Here, the strategy will be to distribute the computational load of the layer across multiple processing cores while minimizing the completion time, only verifying on the last layer that the deadline is met.

We can write the objective function for a layer l as follows:

$${}^*t_p^l = \min \max_{p \in [1..P]} \{\mathbf{t}_p^l\} \tag{5.17}$$

and for the last layer L, we simply aim to verify that the deadline D is indeed respected, as follows:

$$\mathbf{t}_p^L + \sum_{l \in [1..L-1]} {}^*t_p^l \leq D \tag{5.18}$$

This constraint imposes that the completion time of the last layer plus the sum of those of the previous layers must be less than the overall deadline.

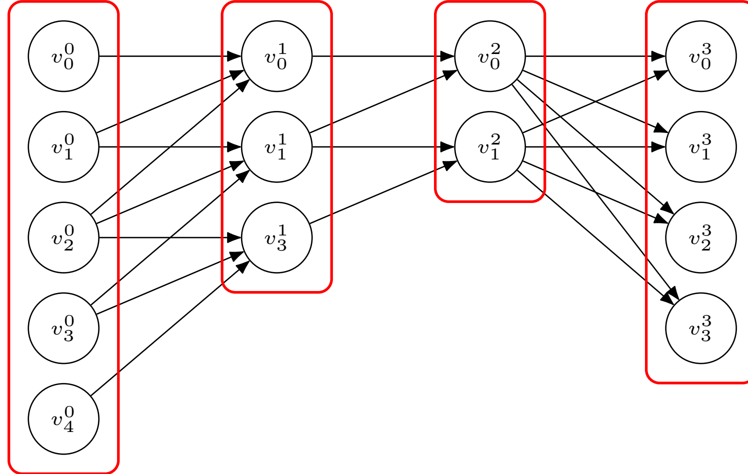


Figure 5.9: Example of a CNN with 4 layers and the precedence constraints between localized operations. The red rectangle indicates the deadline constraint.

5.4 Experimentation and Analysis

To evaluate the effectiveness of our model, we conducted a series of experiments. the goal was to test the performance of our ILP formulation and the two strategies for solving the problem in a simulated environment. We present the experimental setup, the results of our experiments, and an analysis of the outcomes.

5.4.1 Experimental Setup

The target platform used in our experiments is a multicore embedded platform with 6 cores, each equipped with a SPM of 24,000 units, using 16-bit floating-point data. The coalesced block size is set to $\Delta = 4$ pixels.

In our evaluation, we focus on three aspects: the amount of duplicated data, the execution time of the optimization process and the feasibility of the solution. Data duplication refers to the process of creating copies of data to facilitate faster computations, such as when transforming convolution operations into matrix multiplications in GEMM-based techniques. Execution time refers to how long the optimizer takes to find a solution. Feasibility concerns whether the optimizer finds a solution that allows the CNN to complete within the selected deadline.

We also consider a metric we refer to as *density* δ, defined as the ratio of the total execution time required to execute the CNN sequentially (with no parallelization) to the CNN's deadline D. This metric serves as a normalization factor, allowing the comparison of different optimization techniques across various CNN architectures and image dimensions. The larger the delta, the more parallelized and accelerated the solution will be, as the density of operations per core increases and the deadline decreases.

Example 5.4.1. *Suppose a CNN takes 100 ms to execute sequentially, and the deadline is 50 ms. Then,* $\delta = \frac{100}{50} = 2$*. In this case, the system must achieve at least a* $2\times$ *speedup through parallelization to meet the deadline. Conversely, if* $\delta < 1$*, the CNN can meet the deadline even without parallelization.*

Additionally, we introduce the concept of *granularity* γ, which refers to grouping multiple operations together into indivisible units. By setting a granularity value, we can reduce the number of variables in the optimization problem, thereby simplifying the model and speeding up the optimization process. The operations grouped by the granularity parameter follow a deliberate pattern. From our tests, we observed that the optimizer tends to group operations on sequential pixels. This behavior is likely due to the way the image is stored in row-major order, meaning that operations on adjacent pixels are naturally grouped together.

In Figure 5.10, we illustrate how the granularity affects the number of variables needed to represent the operations in a convolutional layer. The input matrix is a 5×5 image, and the kernel size is 2×2 with a stride of 1. The first row shows the operations with a granularity of $\gamma = 1$, where each operation corresponds to a distinct variable a. This results in four variables needed to complete one row. In the second row, with a granularity of $\gamma = 2$, two operations are grouped together, allowing us to complete one row with only two variables for the same image input.

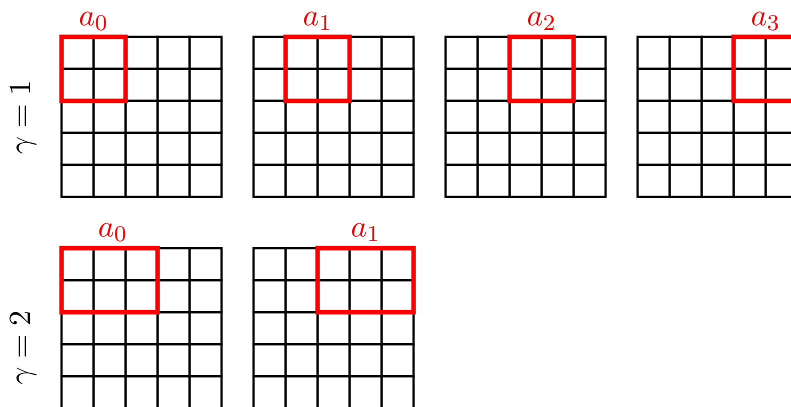


Figure 5.10: Illustration of the impact of granularity on the number of variables in the optimization problem.

It is important to distinguish granularity from the merge operation Θ. While granularity is a fixed parameter defined before optimization, the merge operation focuses on combining multiple localized operations assigned to the same core. The granularity defines how operations are structured at a finer level before optimization, while the merge operation is a higher-level abstraction that groups these operations based on their assignment to cores.

5.4.2 Results

We test our model using a diverse set of randomly generated CNNs. This approach allows for a more comprehensive analysis of the model's performance across various network configurations. We generate up to 20 CNNs, each containing 2 to 4 convolutional layers and 2 to 4 pooling layers, providing a range of complexity levels. The convolutional kernels are uniformly sized at 3×3, with the number of output channels varying from 20 to 64, ensuring that the networks remain representative of common CNN structures while also introducing variability in their architecture. We decided for these tests to not introduce any skip connections, depthwise and pointwise convolutions. This choice simplifies the model and allows us to focus on the core aspects of the optimization problem without the added complexity introduced by these advanced techniques.

The first experiment we conducted was aimed at comparing the amount of data copied during execution for the GEMM approach and our two strategies. For clarity, we briefly summarize how the GEMM technique works. In a convolution operation, a kernel slides over the input image. Using `im2col`, this sliding window is unrolled into columns of a large matrix, which can then be multiplied by the kernel matrix. For example, consider a 3×3 input image and a 2×2 kernel. The `im2col` operation reshapes the input image into a matrix where each column represents a local patch of the image that will be multiplied by the kernel. The resulting matrix will have dimensions 4×4, as each column corresponds to a 2×2 patch from the input image, and there are four such patches in total. This example can be visualized in Figure 5.11.

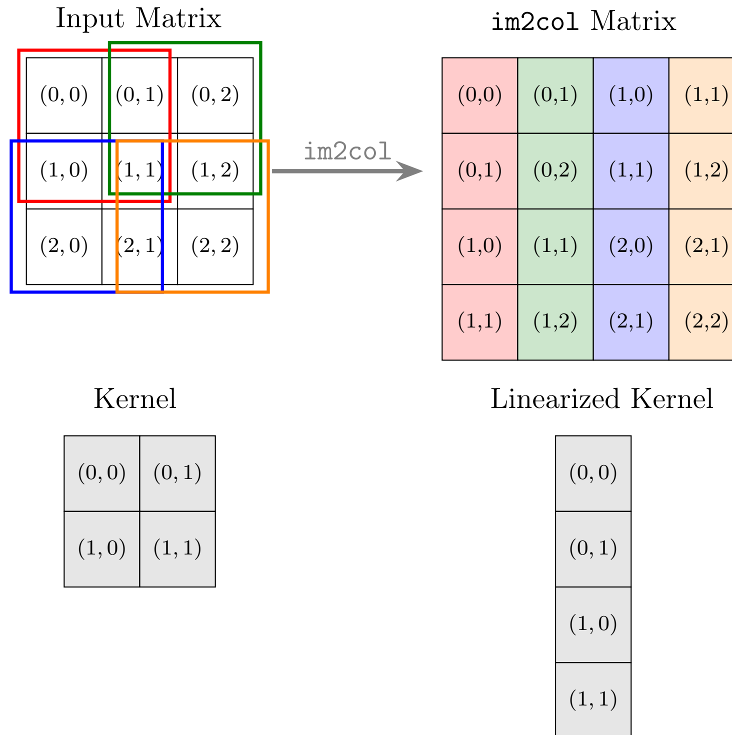


Figure 5.11: Illustration of the `im2col` operation. Each colored column in the output matrix corresponds to a colored sliding window of the same color from the input.

In this experiment, we evaluated the data transfer requirements for various image sizes, ranging from 10×10 to 50×50, processed using a 3×3 convolutional kernel. The granularity γ and the density δ were set to 1, chosen to isolate the impact of the memory management strategies and provide a clear comparison between the three approaches.

The results shown in Figure 5.12 demonstrate that GEMM method, which uses the `im2col` transformation, is particularly memory-intensive, as it unrolls image data into a matrix suitable for matrix multiplication, effectively increasing the data footprint. On the other hand, the Distributed and Global strategies exhibit a reduced memory transfer requirement, showcasing their potential for resource-constrained systems.

In the second experiment, we evaluate the performance of the optimization approaches by selecting a fixed density δ while varying the size of the input image and the granularity γ. The input images range from $10 \times 10 \times 1$ pixels to $90 \times 90 \times 1$ pixels. In this context, the dimensions represent the width, height, and depth of the image, respectively, as it is standard in CNNs. We set the density parameter $\delta = 4$ for this test. We also vary γ from 2 to 4, so to investigate how granularity impacts execution time.

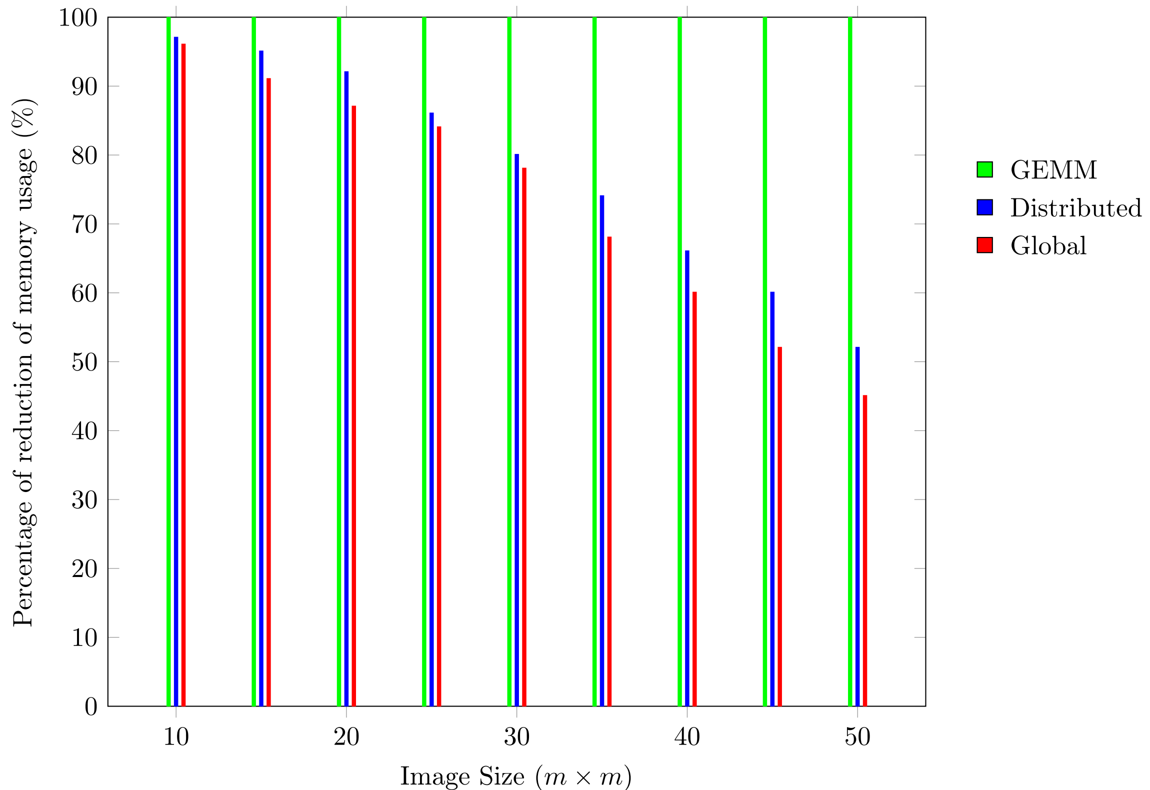


Figure 5.12: Comparison of the number of pixels copied into the SPM for different image sizes ($m \times m$) using a 3x3 kernel and three approaches.

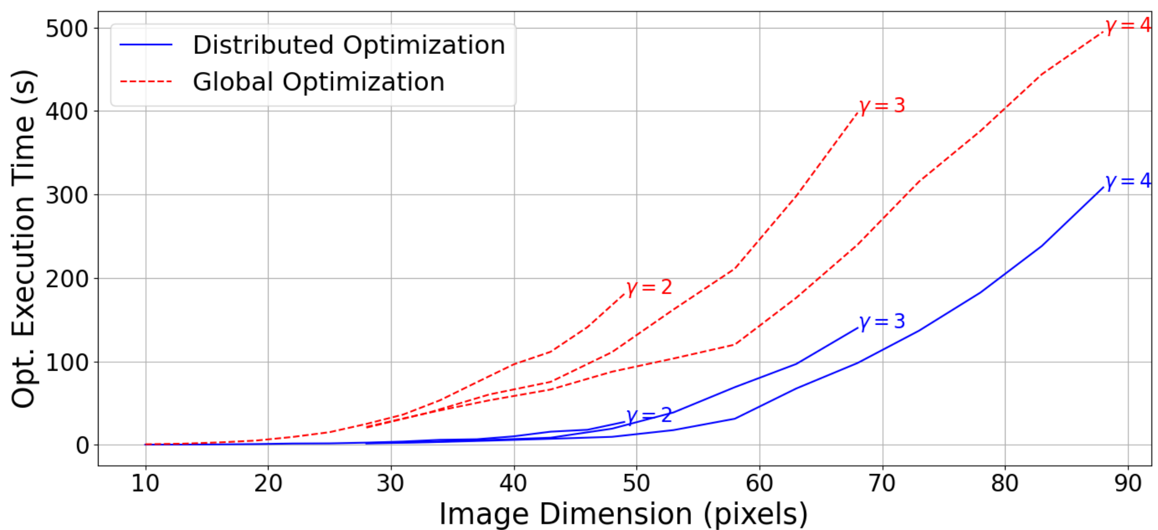


Figure 5.13: Execution time versus image dimensions for the two optimization techniques, considering different granularities.

The plot in Figure 5.13 compares optimization execution times for distributed optimization and global optimization across various image dimensions and granularities. All the results presented are for feasible solutions, where both temporal and memory constraints are met. The results indicate that execution time increases with image dimension for both methods. However, global optimization exhibits significantly longer optimization execution times compared to distributed optimization, especially for larger image sizes. This difference is attributed to

the greater number of variables involved in global optimization, as it processes the entire CNN simultaneously, whereas distributed optimization handles one layer at a time. Importantly, the curves in this figure represent the average execution times from the tests performed across all 20 CNNs. Increasing γ leads to a notable reduction in the number of variables in the optimization problem, which allows both techniques to handle larger images more efficiently. For the same image dimension, a higher granularity results in a lower optimization execution time. This is because grouping operations together at higher granularity reduces the complexity of the problem, thereby speeding up the optimization process.

The third experiment was conducted to evaluate the success rate of each technique, crossing the two key variables density δ and granularity γ.

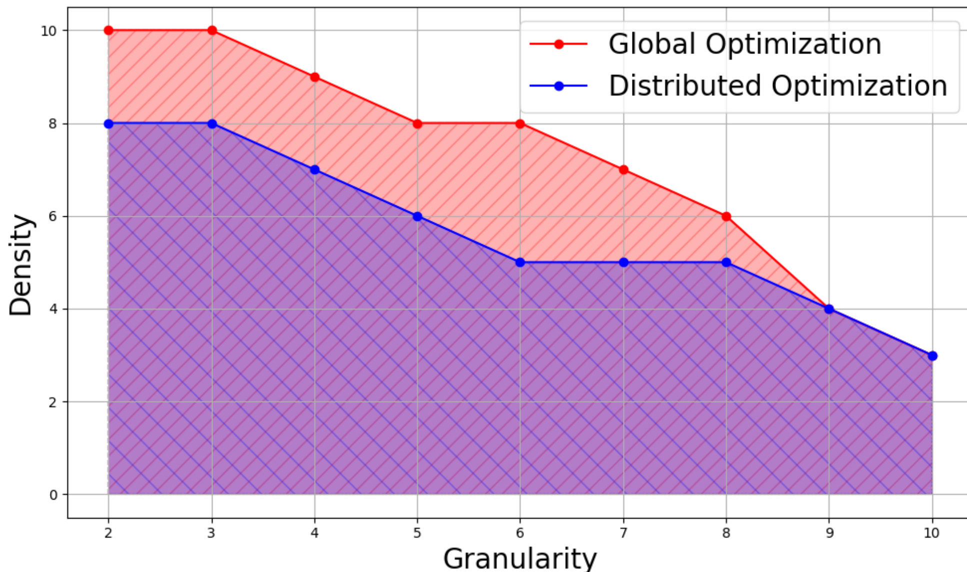


Figure 5.14: Comparison of global and distributed optimization outcomes as a function of granularity and density. Successful configurations are indicated by circles. The shaded regions reflect the areas of successful optimization.

From the first experiment, we observed that global optimization takes significantly longer to find a solution compared to distributed optimization. However, this speed comes at a cost: while distributed optimization may yield faster results, it often struggles to find solutions that global optimization can achieve. In Figure 5.14, the shaded areas beneath each line highlight the feasible solutions for their respective techniques. Notably, the global optimization demonstrates a greater capacity to identify feasible solutions, as indicated by the larger red area compared to the blue area. Importantly, the results depicted in this figure represent the average success rates from the tests performed across all 20 CNNs.

5.4.3 Analysis of Optimal Mapping Structures

Beyond quantitative performance metrics, it is instructive to analyze the qualitative structure of the solutions generated by our ILP model. As mentioned in our discussion of granularity (Section 5.4.1), we observed that the optimizer tends to group operations on sequential pixels to align with the row-major memory layout. To provide clear evidence for this behavior, we designed an experiment to visualize the optimizer's mapping strategy.

We tasked the Distributed ILP optimizer with mapping a single convolutional layer with an 8×8 output feature map onto a 4-core platform. We conducted 20 trials. The ILP optimizer was configured with $\delta = 1$ and the granularity parameter was set to $\gamma = 1$. This setup allows us to observe the inherent preferences of the model when minimizing memory costs.

The consistent mapping structure produced by the optimizer across 20 trials is visualized in Figure 5.15, where each cell in the grid represents a localized operation, colored according to the core it was assigned to.

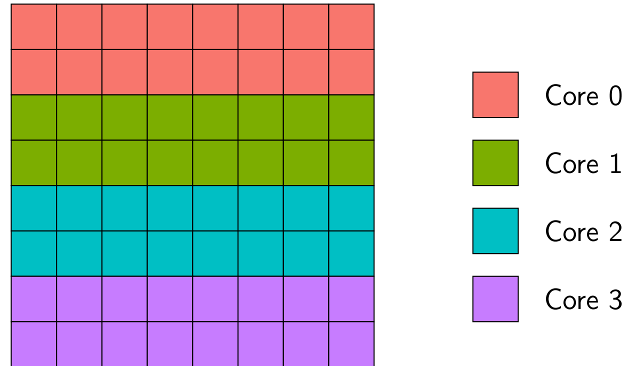


Figure 5.15: Visualization of an optimal mapping for an 8x8 layer onto 4 cores.

The result shown in Figure 5.15 reveals that the optimizer does not scatter operations randomly across cores. Instead, it systematically creates contiguous horizontal partitions.

This behavior is a direct consequence of the memory cost model defined in Section 5.1.2. Since input tensors are stored in row-major order, assigning adjacent rows to the same core maximizes data reuse and minimizes the number of non-contiguous memory batches (n_k^l). Fetching data for different rows often breaks memory contiguity, incurring a significant latency penalty ($\text{cost}_{\text{batch}}$) that the optimizer correctly identifies and avoids.

Furthermore, this partitioning strategy is reinforced by the objective to minimize the makespan (the maximum completion time across all cores(eq. 5.17)). By assigning each core a distinct horizontal band of the feature map, the total workload is evenly distributed. Each core receives a similar number of operations to compute, leading to balanced execution times. This prevents a scenario where one core finishes early and sits idle while another is overburdened, which would result in a longer overall execution time. Thus, the horizontal structure is optimal for both minimizing memory latency and balancing the computational load.

5.4.4 Scalability

The primary challenge of the proposed ILP-based method is its computational complexity, which directly impacts its scalability to larger, more intricate CNN models. Our investigation into this issue followed a clear progression of strategies aimed at managing this complexity.

Our initial approach was the Global ILP, which optimizes the entire network simultaneously. While theoretically capable of finding the highest quality solutions, this method proved to be the least scalable. By considering all variables and constraints across every layer at once, the problem size grows exponentially, leading to prohibitively long optimization times for even moderately sized networks.

To mitigate this, we developed the Distributed ILP strategy. By partitioning the problem and optimizing it layer-by-layer, this approach significantly reduces the number of variables in each optimization step. As shown in our results (Figure 5.13), this strategy drastically decreased the time required to find a solution. However, this speed introduced a new compromise: by making locally optimal decisions for each layer, the distributed strategy sometimes fails to find a globally feasible solution for tight deadlines, a limitation illustrated in Figure 5.14.

As a further technique to manage complexity, we introduced the concept of granularity (γ). This parameter allows for a manual reduction in the number of decision variables by

grouping primitive operations before optimization. We found this to be a powerful lever for reducing optimization time in both the Global and Distributed strategies.

Ultimately, our findings on scalability highlight a fundamental trade-off between optimization effort and solution feasibility. The Global model offers the best chance of finding a feasible solution for challenging deadlines but at a high computational cost. The Distributed model and the granularity parameter provide practical tools to reduce this cost, at the risk of simplifying the problem to a point where an otherwise valid solution might be missed. A designer must therefore choose the appropriate strategy based on the complexity of the CNN and the stringency of its performance constraints.

5.5 Conclusions

In this chapter, we proposed a formal model based on decomposing CNNs into fine-grained *localized operations*. To solve the mapping and scheduling of these operations, we formulated the problem as an ILP task and presented two distinct solution strategies: a holistic Global ILP and a layer-by-layer Distributed ILP.

Our experimental evaluation brought to light several key insights. First, both proposed strategies significantly reduce the memory transfer overhead compared to the conventional `im2col`/GEMM approach, which is critical for memory-constrained embedded systems. Second, we identified a clear trade-off between the two ILP strategies: the Distributed approach is significantly faster to solve, making it more practical for larger networks, whereas the Global approach, while more computationally expensive, is capable of finding feasible solutions for more demanding performance constraints (i.e., higher density values). This highlights a fundamental trade-off between optimization time and solution quality. Finally, we demonstrated that the concept of *granularity* is an effective parameter to manage the complexity of the optimization problem.

Chapter 6

CNN-PAS: A Partitioning, Allocation and Scheduling Framework

The preceding chapter introduced an ILP methodology to derive an optimal mapping and schedule for a single CNN on a multicore scratchpad architecture. That work established a theoretical performance bound, demonstrating how to minimize latency by optimizing computation and data transfers.

However, real-world embedded systems rarely run a single application in isolation. They must concurrently manage a diverse set of tasks—from sensor processing and control loops to communication protocols. The static, offline optimality achieved for one CNN does not guarantee schedulability when the system is under load from other real-time tasks.

This chapter addresses this critical gap by shifting focus from single-task optimality to multi-task schedulability.

To do this, we present the CNN Partitioning, Allocation, and Scheduling (CNN-PAS) framework. CNN-PAS is a multi-phase methodology designed to transform a set of high-level task specifications into a schedulable implementation on a multicore platform. Unlike the ILP solver, CNN-PAS employs an iterative process of graph partitioning, thread allocation, communication modeling, and schedulability analysis to systematically find a feasible configuration for the entire system.

6.1 Methodology Overview

The central objective of this work is to schedule a set of real-time tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ on the target multicore platform to ensure all deadlines are met.

To address this challenge, we introduce CNN-PAS, which stands for *CNN Partitioning, Allocation and Scheduling*.

For each task, the developer specifies a graph $\mathcal{G}$ of computation nodes that represent the functional part of the application. At the beginning, each graph only contains computations, and an edge between nodes represents the data flowing across the application as well as the precedence constraint.

In particular, one or more of these task graphs represents a CNN as described in Section 4. A single node graph represents a classical periodic Liu&Layland task. We refer to these initial representations as *graph specifications*. As CNN-PAS progress across the different phases, it will transform the specification graphs into *implementation graphs*, allocating nodes into threads, threads onto cores, adding communication nodes to represent data transfer and adding intermediate deadlines, before checking schedulability.

More specifically, our CNN-PAS consists of an iterative sequence of phases as follows.

- **Phase 1**: *graph partitioning*. In this phase we partition the nodes of graph $\mathcal{G}$ into a set of concurrent threads to be executed sequentially or in parallel on the platform. The result is a graph of threads $\mathcal{G}'$ for every task τ_i.
- **Phase 2**: *thread allocation*. In this phase, the threads are allocated on the cores. We will use a simple allocation heuristic based on worst-fit to maximize the chances of

executing the nodes in parallel. If no allocation can be found, we go back to the phase 1 and produce a different partition of the nodes into threads.

- **Phase 3**: *modeling data transfer*. In this phase, we further transform the graphs by adding *communication nodes* between threads allocated on different cores. Communication nodes model data transfer between local memories, and they are performed by the DMA engines. The results will be a new graph $\mathcal{G}''$ which encompasses threads and communication nodes.

- **Phase 4**: *encoding precedence constraints*. In this phase we assign offsets and intermediate deadlines to threads and communication nodes so to implicitly enforce the precedence between nodes in graph $\mathcal{G}''$.

- **Phase 5**: *allocation of communication nodes*. Here we partition the communication tasks to the two DMA engines, and we check their schedulability. Again, if the allocation is not feasible, we go back to phase 1.

- **Phase 6**: *scheduling analysis*. In this phase, we check that all the deadlines computed in the previous step are respected when executing the threads on the multicore platform. If the analysis returns *false*, we go back to Phase 1 to try a different partitioning of the task graph specifications.

In Figure 6.1 we illustrate the overall flow of the CNN-PAS framework. In the next sections we will explain each phase in turn.

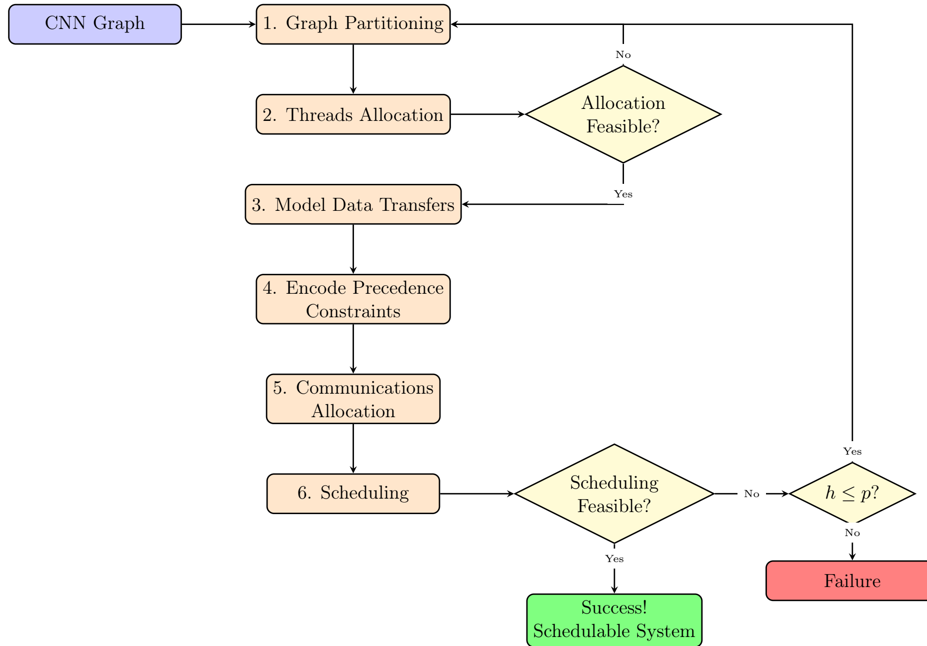


Figure 6.1: Overall flow of the CNN-PAS framework, showing the six main phases and their iterative nature.

6.2 Task Graph partitioning

In the initial graph $\mathcal{G}$, each node represents a single operation in the CNN. However, this fine-grained representation is not suitable for efficient scheduling on parallel architectures. The purpose of this phase is to merge nodes of $\mathcal{G}$ into a coarser graph $\mathcal{G}'$. The nodes of $\mathcal{G}'$, which we term threads, represent the fundamental units for parallel execution and scheduling.

The nodes are mergeable if they are part of the same layer and can be executed in parallel without violating data dependencies.

The transformation from $\mathcal{G}$ to $\mathcal{G}'$ is governed by a single integer parameter, h, which defines the number of horizontal cuts applied to the feature maps of layer. The decision to partition layers horizontally is not arbitrary; it is a direct consequence of the findings from our ILP-based analysis in the previous chapter. As demonstrated in Section 5.4.3, an optimal mapping naturally creates contiguous horizontal partitions to align with the row-major memory layout of feature maps. This structure minimizes costly non-contiguous memory transfers between rows, providing a clear, evidence-based motivation for our partitioning strategy. We decided that the layers eligible for horizontal partitioning are those that have a feature map with a height greater than one, i.e., $h \geq 1$. The fully connected layers are not partitioned, and they are always implemented by a single thread. This is because fully connected layers do not have a height dimension in their feature maps, and they are typically small enough to be efficiently handled by a single thread.

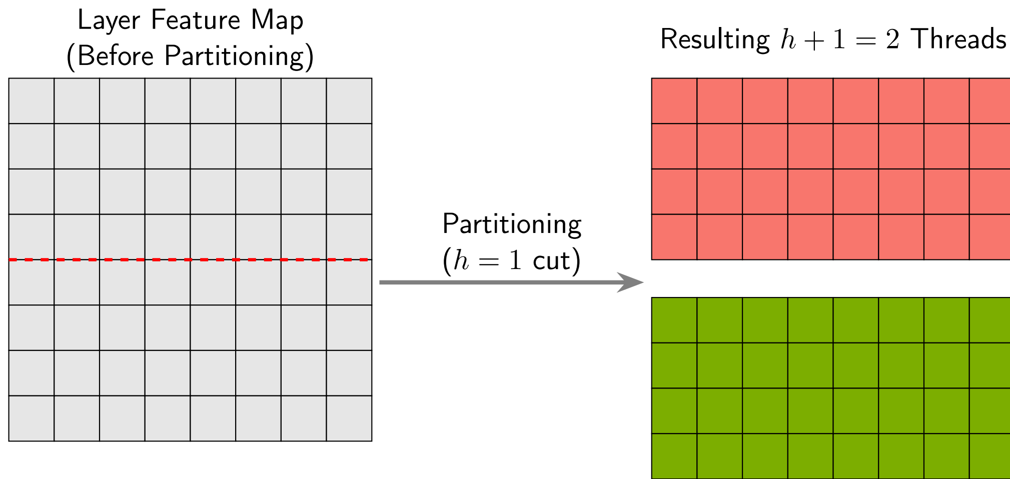


Figure 6.2: Illustration of the horizontal partitioning phase for a value of $h = 1$. A single layer's feature map is divided by one horizontal cut into two independent, parallel threads.

Consider Figure 6.3 and let us apply a horizontal partitioning with $h = 1$: we obtain the graph shown in Figure 6.4. As you can see, each layer is divided into two threads and there is no overlapping of nodes between threads. After the partitioning, nodes are merged into single threads, as shown in Figure 6.5.

When a set of nodes $\mathcal{S}_j = \{v_1, \dots, v_k\}$ from $\mathcal{G}$ is merged into a thread th in $\mathcal{G}'$, the thread's parameters are computed as follows:

- **Predecessors:** the thread inherits the union of all unique predecessors of the nodes in $\mathcal{S}_j$. That is, a node v_p is a predecessor of th if it produces an output used by any $v_j \in \mathcal{S}_j$. For example, in Figure 6.5, both threads th_0^2 and th_1^2 have th_0^1 and th_1^1 as predecessors, because both threads depend on outputs from both threads in the previous layer.

- **Buffer sizes (α, β):** input buffer size α is the union of all input buffers required by the nodes in $\mathcal{S}_j$, because there is no aliasing, while output buffer size β is the sum of all output buffers required by the nodes in $\mathcal{S}_j$, because each node produces a unique output. Formally, we have:

$$\alpha(\mathsf{th}) = \bigcup_{v_j \in \mathcal{S}_j} \alpha(v_j), \quad \beta(\mathsf{th}) = \sum_{v_j \in \mathcal{S}_j} \beta(v_j)$$

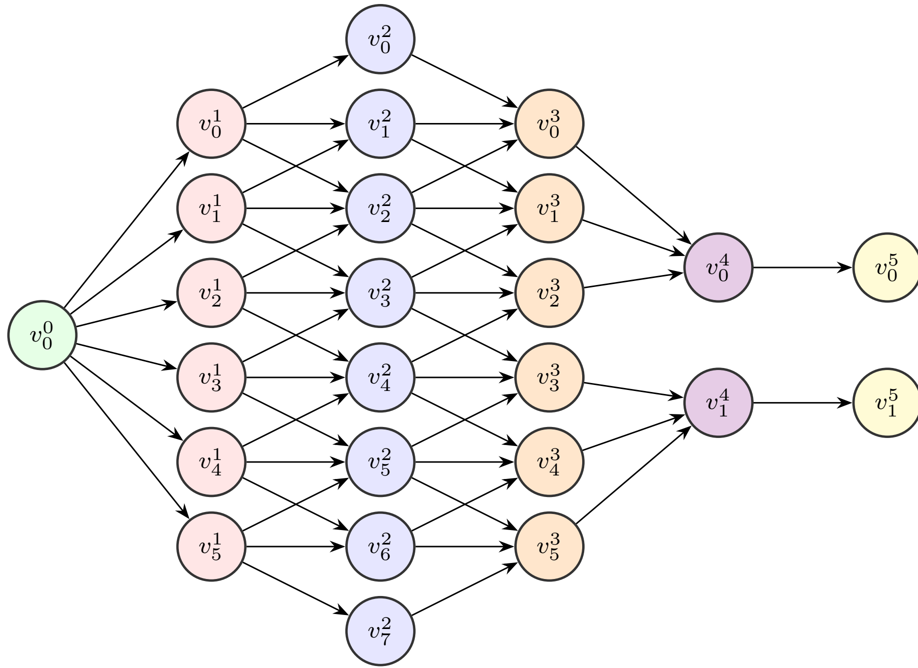


Figure 6.3: Example of a CNN structure.

- **Static memory (γ):** The thread's code and stack memory is the sum of the memory requirements of its nodes:

$$\gamma(\mathsf{th}) = \sum_{v_j \in \mathcal{S}_j} \gamma(v_j)$$

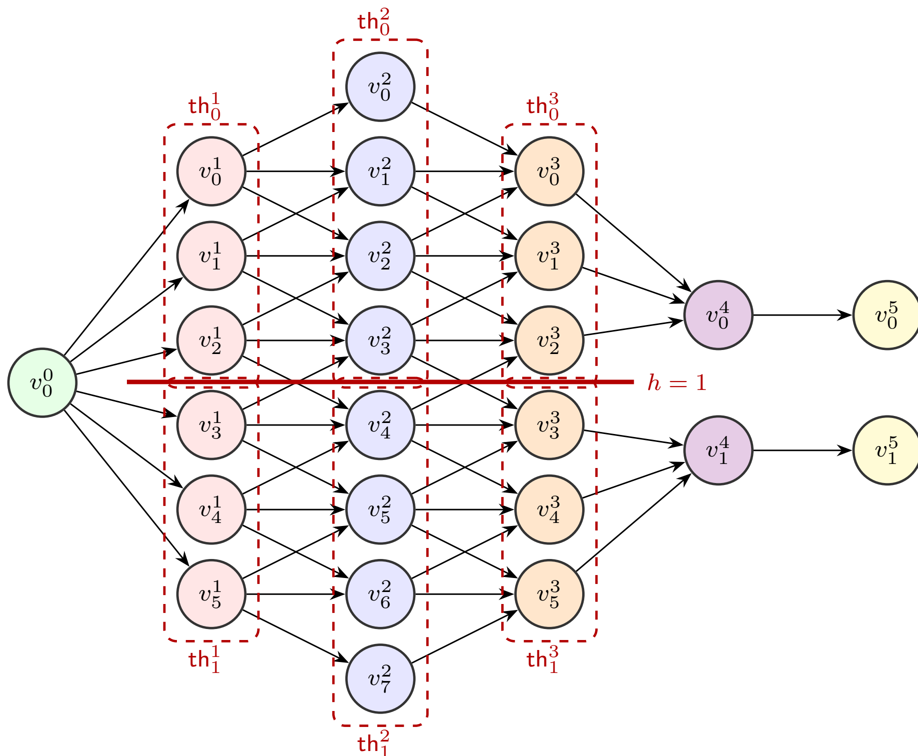


Figure 6.4: Graph $\mathcal{G}$ partitioned with $h = 1$.

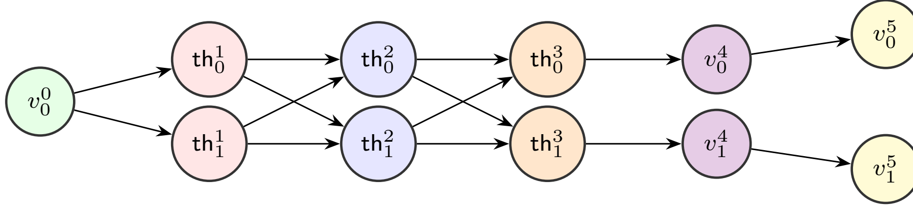


Figure 6.5: Resulting graph $\mathcal{G}'$ after horizontal partitioning with $h = 1$.

As shown in Figure 6.5, every thread in one layer may be connected to every thread in the next layer; this directly follows from our definition of thread predecessors (each merged thread inherits the union of predecessors of its constituent nodes). To avoid unnecessary data copies we assign a cost to each inter-thread edge that reflects the input buffers that must be available for the destination thread.

Formally, the set-valued cost of an edge $(\mathsf{th}_u, \mathsf{th}_z)$ is the set of input buffers required by the destination thread th_z, which by definition is the union of the output buffers of its predecessor threads:

$$\text{cost}(\mathsf{th}_u, \mathsf{th}_z) = \bigcup_{v \in \mathsf{v}(\mathsf{th}_z)} \beta(v), \tag{6.1}$$

where pred(th_z) denotes the predecessors of thread th_z and $\beta(\cdot)$ is the output-buffer set.

To illustrate this, consider Figure 6.5 where th_0^2 depends on both th_0^1 and th_1^1. Referring back to the original graph in Figure 6.4, th_0^2 requires input buffers from nodes v_0^1, v_1^1, v_2^1, and v_3^1, where v_0^1, v_1^1, v_2^1 belong to th_0^1 and v_3^1 belongs to th_1^1. Therefore, the cost of the edges are:

$$\text{cost}(\mathsf{th}_0^1, \mathsf{th}_0^2) = \bigcup \beta(v_0^1, v_1^1, v_2^1), \quad \text{cost}(\mathsf{th}_1^1, \mathsf{th}_0^2) = \beta(v_3^1).$$

Our allocation framework performs an iterative search to generate a feasible graph $\mathcal{G}'$. The process begins with the coarsest graph ($h = 0$). If mapping and scheduling fail, h is incremented to create a finer-grained graph. This procedure continues until a feasible schedule is found or the upper bound $h = M$ is reached, where M is the number of available processor cores. Partitioning a layer into more threads than available cores introduces scheduling overhead without further parallelism; if no feasible schedule is found, the model is considered infeasible for the target platform.

The process of generating $\mathcal{G}'$ for a given h is detailed in Algorithm 1. The complexity of this algorithm is $O(m \cdot k)$ where m is the number of cores and k is the number of layers for graph $\mathcal{G}$.

Algorithm 1 Task Graph Partitioning

1: **Input:** Task graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, number of horizontal cuts h.
2: **Output:** Merged graph $\mathcal{G}'_i = (\mathcal{V}', \mathcal{E}')$.
3: $\mathcal{V}' \leftarrow \emptyset$; $\mathcal{E}' \leftarrow \emptyset$
4: Let `map_v_to_s` be an empty map from original nodes to new threads.
   // Phase 1: Create threads and calculate their parameters
5: **for** each layer Y_i in $\mathcal{G}$ **do**
6:     **if** Y_i is partitionable **then**
7:         Partition nodes of Y_i into $h+1$ sets: $\mathcal{S}_1, \dots, \mathcal{S}_{h+1}$.
8:     **else**
9:         Group all nodes of Y_i into a single set $\mathcal{S}_1$.
10:     **end if**
11:     **for** each set $\mathcal{S}_j$ found above **do**
12:         Create a new thread th_j and add it to $\mathcal{V}'$.
13:         ▷ Set parameters based on the definitions in the text
14:         $\alpha(\mathsf{th}_j) \leftarrow \bigcup_{v \in \mathcal{S}_j} \alpha(v)$
15:         $\beta(\mathsf{th}_j) \leftarrow \sum_{v \in \mathcal{S}_j} \beta(v)$
16:         $\gamma(\mathsf{th}_j) \leftarrow \sum_{v \in \mathcal{S}_j} \gamma(v)$
17:         For each node $v \in \mathcal{S}_j$, set `map_v_to_s`$[v] \leftarrow \mathcal{S}_j$.
18:     **end for**
19: **end for**
   // Phase 2: Add edges between threads based on original dependencies
20: **for** each original edge $(u, v) \in \mathcal{E}$ **do**
21:     Let $\mathsf{th}_u \leftarrow$ map_v_to_s$[u]$. Let $\mathsf{th}_v \leftarrow$ map_v_to_s$[v]$.
22:     **if** $\mathsf{th}_u \neq \mathsf{th}_v$ and the edge $(\mathsf{th}_u, \mathsf{th}_v)$ is not already in $\mathcal{E}'$ **then**
23:         Add edge $(\mathsf{th}_u, \mathsf{th}_v)$ to $\mathcal{E}'$.
24:     **end if**
25: **end for**
26: **return** $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$

6.3 Allocation

In this phase, we allocate threads to cores, taking into account the computational load, the memory requirements and the communications. To address this, we explored two classic allocation heuristics: best-fit and worst-fit.

Our initial approach was a best-fit strategy. This heuristic attempts to pack threads as tightly as possible, prioritizing the allocation of a thread to a core that is already running one of its predecessors. The goal was to minimize data movement over the shared bus. However, we observed that this strategy was counterproductive for CNNs; it tended to serialize the execution by packing entire layers onto a single core whenever possible, failing to exploit the parallelism offered by the multicore architecture.

Consequently, we adopted a worst-fit strategy, which inherently promotes parallelism by distributing the workload as evenly as possible across all available cores. This approach forms the basis of our allocation algorithm. Threads are sorted in a topological order, and for each thread, we calculate a score for every possible core/SPM combination. The thread is then allocated to the location with the best score. The score is designed to find the least loaded core (the "worst fit") while still providing a bonus for allocations that reduce data transfers.

The pseudo-code for the allocation of one task graph is described in Algorithm 2. Threads are first sorted by topological order. Then, for every thread (line 4), we compute a score for every local memory m_p (either σ_p^1 or σ_p^2) on core π_p (line 6).

We first compute the required memory `MemReq` at line 7 as the sum of the input buffer, output buffer and stack frame. Function `pred_memory()` computes the sum of the sizes of the output buffers of the immediate predecessors that have already been allocated to the same memory. This is subtracted from the requirements of the current thread; in fact, when two threads that are in a predecessor/successor relationship are allocated on the same core and the same local memory, the successor thread can reuse the output buffer of the predecessor as its input buffer, saving some precious memory.

Then, we compute the score of this allocation (line 8) by summing the thread utilization to the utilization $U(p)$ of the threads already allocated on core π_p. We add an additional positive score for every immediate predecessor allocated on the same memory, so to avoid communication and reuse the buffers.

At line 9, we check that the thread indeed fits into the memory and that the utilization is less than a predefined bound (in the experiments we selected a bound of 80% for every core). If the thread fits into the core, the score is added to a list of scores.

After checking every local memory and core, we select the memory and the core with the best score. If the score list is empty, the heuristic allocation algorithm fails and we go back and select a different partitioning of the graph. Otherwise, the thread is allocated and we proceed with the next thread.

For the special case of single thread tasks, the above algorithm is equivalent to a worst-fit algorithm [75, 76] with an additional check on the memory capacity.

Algorithm 2 Thread Allocation Algorithm

1: **Input:** Task graph $\mathcal{G}$, set of local memories $\mathcal{M}$
2: **Output:** Mapping of threads to cores and local memories
3: Sort threads in topological order
4: **for** each thread $\texttt{th}_i \in \texttt{sorted}(\mathcal{G})$ **do**
5: $\quad$ $\texttt{ScoreList} = \emptyset$
6: $\quad$ **for** each memory $m_p \in \mathcal{M}$ **do**
7: $\quad\quad$ $\texttt{MemReq}(m_p) = \alpha_i + \beta_i + \gamma_i - \texttt{pred_memory}(i, m_p)$
8: $\quad\quad$ $\texttt{Score}(m_p) = 1 - (U(p) + \frac{C_i}{T_i}) + \texttt{pred_score}(\texttt{i}, \texttt{m}_\texttt{p})$
9: $\quad\quad$ **if** $\texttt{MemReq}(m_p) + \texttt{AllocMem}(m_p) \leq \texttt{cap}(m_p)$ **then**
10: $\quad\quad\quad$ **if** $U(p) + \frac{C_i}{T_i} \leq \texttt{UtilBound}(p)$ **then**
11: $\quad\quad\quad\quad$ Add $\texttt{Score}(m)$ to `ScoreList`
12: $\quad\quad\quad$ **end if**
13: $\quad\quad$ **end if**
14: $\quad$ **end for**
15: $\quad$ **if** `ScoreList` is empty **then return** False
16: $\quad$ **end if**
17: $\quad$ Select m_p with the best score from `ScoreList`
18: $\quad$ Allocate $\texttt{th}_i$ to core p and memory m_p
19: $\quad$ Update $U(p)$ and $\texttt{AllocMem}(m_p)$
20: **end for**

The complexity of Algorithm 2 is $O(m \cdot k \cdot c)$ where c is the number of memories per core and $m \cdot k$ is the maximum number of threads that can be generated from the previous step.

6.4 Memory transfers

Once threads are allocated on the cores, we add communication nodes to the graph. Communication nodes are responsible for copying data from the output buffer of the source thread to the input buffer of the destination thread. In addition to communications between local memories, we need to consider communications to/from the central DRAM. Therefore, we extend set $\mathcal{M}$ to include one more element representing the DRAM: $\mathcal{M}' = \mathcal{M} \cup \{\mathsf{DRAM}\}$.

For each pair of threads th_i and th_j that are allocated on different SRAMs, where th_i is an immediate predecessor of th_j, we add a communication node μ_k to the graph. The edge $e(\mathsf{th}_i, \mathsf{th}_j)$ is removed, and edges $e(\mathsf{th}_i, \mu_k)$ and $e(\mu_k, \mathsf{th}_j)$ are added to enforce the precedence constraints: communication μ_k can start only after thread th_i has finished executing; and it must complete before thread th_j can start executing. The amount of data to be transferred depends on β_i and α_j, that is on the input and output buffer sizes. We do not add any communication node if th_i and th_j are allocated on the same SRAM memory: in fact, th_j can re-use (part of) the output buffer of th_i as its input buffer.

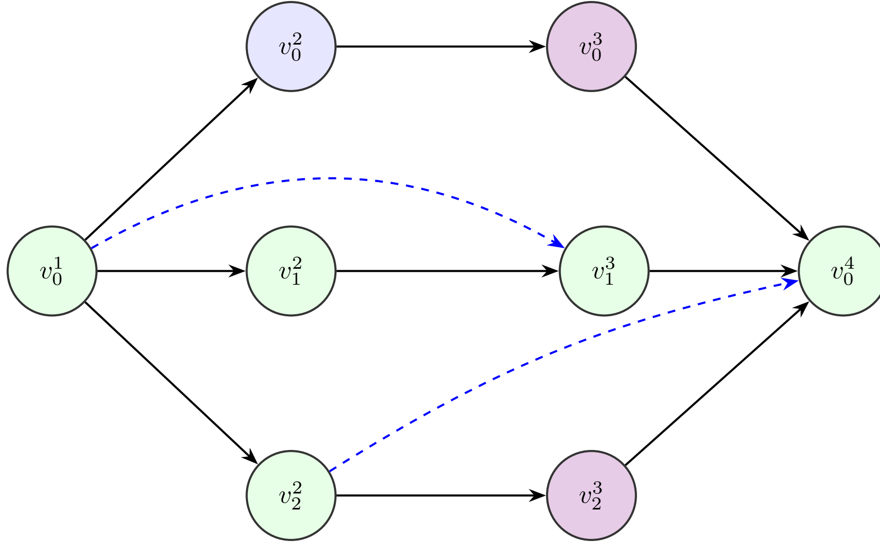
Furthermore, we add a single source th_{source} and a single sink th_{sink} to the task graph: these are "fake" threads that have zero computation time and that do not need to be executed on a core; they represent the activation and the completion of the task graph. We then add communication nodes from th_{source} to all threads with no predecessor in the graph $\mathcal{G}'$; these represent the transfer of the input data from the central DRAM to the local memory of the threads. In the same way, we add communication nodes from every thread with no successor to the th_{sink} fake thread, representing the transfer of the output data from the local memories to the main DRAM.

The volume of data transferred between threads depends on their input and output buffer sizes. As discussed in Section 6.2, each thread produces a unique portion of the output feature map. To quantify data movement, we assign a *cost* to each edge $e(\mathsf{th}_i, \mathsf{th}_j)$ in the task graph, representing the amount of data that must be transferred from thread th_i to thread th_j, as in Equation 6.1. Only the required data are transferred, and the contiguity of the buffers ensures that a single memory copy operation is sufficient per edge.

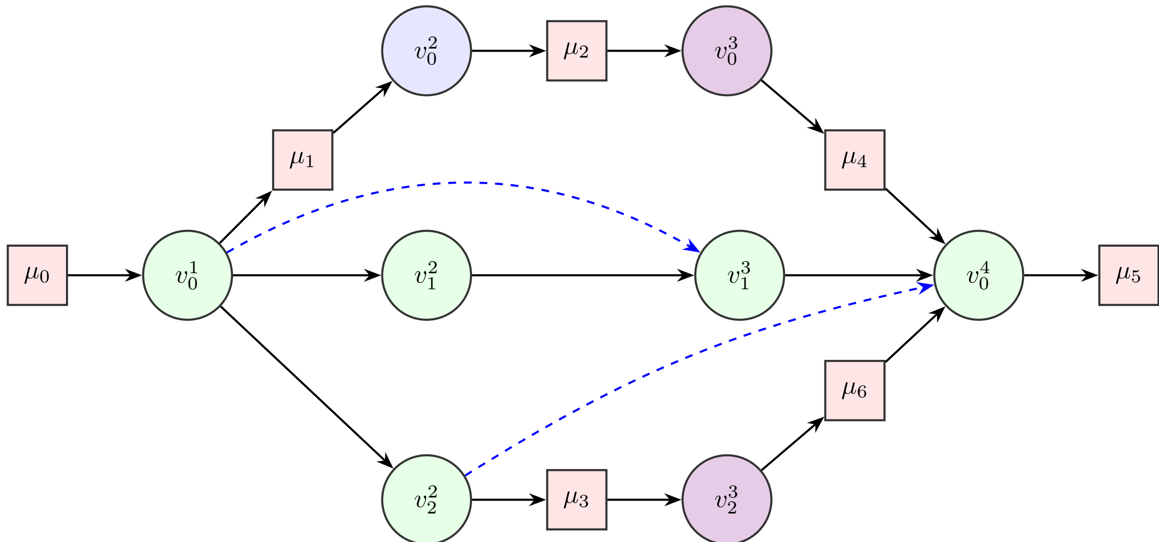
The results of this operation is a new graph $\mathcal{G}''$ containing threads and communications nodes where needed.

This transformation is illustrated in Figure 6.6. The top graph, (a), shows an example of a task graph $\mathcal{G}'$ after the partitioning and allocation phases, consisting only of computational threads. The different colors indicate that the threads are mapped to different cores.

The bottom graph, (b), shows the resulting graph $\mathcal{G}''$ after this phase is complete. The rectangular, red communication nodes (μ_0, μ_k) have been inserted on the edges between threads that reside on different cores (e.g., between v_0^1 and v_0^2) and to connect the graph to the main DRAM (μ_0 and μ_5). Communication is not needed between threads of the same color (e.g., between v_2^2 and v_2^3), as they are mapped to the same core and can share data through their local scratchpad memory.



(a) Task graph $\mathcal{G}'$ showing computational threads and their dependencies..



(b) Resulting graph $\mathcal{G}''$ after adding explicit communication nodes (μ_k) for data transfers between cores.

Figure 6.6: Illustration of the graph transformation to model memory transfers.

6.4.1 Precedence encoding

To enforce the precedence constraints defined by the graph $\mathcal{G}''$, we assign intermediate offsets and deadlines to each thread and communication node. This technique ensures that a successor node is only scheduled after its predecessors have completed, thereby implicitly respecting the task graph's dependencies without requiring complex and costly synchronization mechanisms like spin-locks or inter-processor interrupts.

While many methods exist for the intermediate deadline assignment problem([95, 96]), we employ an approach more specific for CNN applications and their DAG structure. The idea behind it is to exploit the layered structure of CNNs and assign the same intermediate deadline to all the nodes within the same layer.

Each node is assigned a *layer index* $x(v_i)$, defined as the number of edges in the longest path from any source node to v_i. The *next layer* $z(v_i)$ is defined as the smallest layer index among the successors of v_i. Using these definitions, the algorithm computes a workload per

layer and proportionally distributes the end-to-end deadline across all layers. Sink nodes (with no successors) are assigned a deadline equal to the end-to-end deadline of the graph.

Algorithm 3 Encoding Algorithm

1: **Input:** Task graph $\mathcal{G}''$, D
2: **Output:** Intermediate offsets and deadlines for all threads in $\mathcal{G}''$
3: Sort nodes topologically and compute for each thread th_i: layer index $x(\mathsf{th}_i)$ and next layer $z(\mathsf{th}_i)$
4: **for** each layer l and processor p **do**
5: $\quad C_{\mathsf{th}}(l,p) \leftarrow \sum_{x(\mathsf{th}_i)\leq l<z(\mathsf{th}_i)} \frac{C(\mathsf{th}_i)}{z(\mathsf{th}_i)-x(\mathsf{th}_i)}$
6: **end for**
7: $C_{\max}(l) \leftarrow \max_p C_{\mathsf{th}_i}(l,p), \quad C_{\text{tot}} \leftarrow \sum_l C_{\max}(l)$
8: **if** $C_{\text{tot}} > D$ **then**
9: $\quad$ **return** infeasible
10: **end if**
11: offset$(0) \leftarrow 0$
12: **for** $l = 0$ to k **do**
13: $\quad D(l) \leftarrow C_{\max}(l) \cdot D/C_{\text{tot}}$
14: $\quad$ offset$(l) \leftarrow$ offset$(l-1) + D(l-1)$
15: **end for**
16: **for** each thread th_i **do**
17: $\quad$ offset$(\mathsf{th}_i) \leftarrow$ offset$(x(\mathsf{th}_i))$
18: $\quad \text{deadline}(\mathsf{th}_i) \leftarrow \begin{cases} D, & \text{if } \mathsf{th}_i \text{ is sink} \\ \text{offset}(z(\mathsf{th}_i)), & \text{otherwise} \end{cases}$
19: **end for**

The complexity of the deadline assignment algorithm is bounded by the number of nodes and processors; that is, it is equal to $O(m \cdot k)$.

6.4.2 Competing communication tasks

Definition 9 (Competing communication nodes)**.** *Two communication nodes are* competing*, and we write* $\mu_k \bowtie \mu_h$*, if*

- *they have either source or destination in common:*

$$\mu_k \bowtie \mu_h \Leftrightarrow \mathsf{src}_k = \mathsf{src}_h \vee \mathsf{src}_k = \mathsf{dst}_h \vee \mathsf{src}_h = \mathsf{dst}_k \vee \mathsf{dst}_h = \mathsf{dst}_k$$

- *and, they can be scheduled at the same time; this may happen if the two communication tasks belong to two different sporadic task graphs, or if they belong to the same task graph and have overlapping scheduling windows:*

$$\mu_k \bowtie \mu_h \Leftrightarrow \mu_k \in \tau_i$$

For a communication node μ_k*, the set of* competing communications *is defined as:*

$$\mathsf{comp}(\mu_k) = \{\mu_h \mid \mu_k \bowtie \mu_h\}$$

Notice that the relation $\bowtie$ is symmetric and reflexive, but not transitive. Competing communications cannot be executed in parallel; in fact, if we try to transfer two data buffers from/to the same memory, they will be sequentialized by the memory controller. Conversely, two non competing communication nodes can be executed in parallel thanks to the crossbar switch architecture.

Data transfers are performed by the two DMA engines present in our reference architecture. To maximize parallelism, we statically assign each communication node to one of the two DMA engines according to a modified version of the worst-fit strategy, as described in Algorithm 4; in addition to trying to balance the load between the two DMAs, the strategy favors the assignment of competing communications to the same DMA.

More in detail, the algorithm greedily assigns each communication node μ_k to the DMA engine that results in the lowest projected load; this load is calculated by a cost function that augments the standard utilization with a blocking factor, C_{block}. This factor models the worst-case serialization delay for μ_k caused by any competing communication node residing on the *other* DMA. By penalizing assignments that assign competing nodes across different engines, the heuristic implicitly favors co-locating them. The complexity of this algorithm is $O(m^2 \cdot k^2)$. This estimation is pessimistic, as it assumes that all consecutive nodes of different layers are allocated to different cores. A more refined analysis could be conducted to obtain a tighter bound, but is not reported here.

Algorithm 4 DMA Allocation Algorithm

1: **Input:** Set of communication nodes $\mathcal{C}$, Maximum utilization per DMA U_{max}.
2: **Output:** Allocation of tasks to DMA_0 and DMA_1.
3: **procedure** DMA-ALLOCATE($\mathcal{M}, U_{max}$)
4: $DMA_0 \leftarrow \emptyset$, $DMA_1 \leftarrow \emptyset$ ▷ Initialize DMA assignments
5: **for all** communication task $\mu_k \in \mathcal{C}$ **do**
6: $U_0 \leftarrow$ Compute-Cost(μ_k, DMA_0, DMA_1)
7: $U_1 \leftarrow$ Compute-Cost(μ_k, DMA_1, DMA_0)
8: **if** $U_0 \leq U_1$ **and** $U_0 \leq U_{max}$ **then**
9: $DMA_0 \leftarrow DMA_0 \cup \{\mu_k\}$
10: **else if** $U_1 < U_0$ **and** $U_1 \leq U_{max}$ **then**
11: $DMA_1 \leftarrow DMA_1 \cup \{\mu_k\}$
12: **else**
13: **throw** AllocationError("Unschedulable set")
14: **end if**
15: **end for**
16: **return** DMA_0, DMA_1
17: **end procedure**
18: **function** COMPUTE-COST($\mu_k, DMA_{target}, DMA_{other}$)
19: $C_k \leftarrow$ WCET of μ_k; $T_k \leftarrow$ Period of μ_k
20: $C_{block} \leftarrow 0$ ▷ Max blocking time from competing tasks on DMA_{other}
21: **for all** $\mu_h \in DMA_{other}$ **do**
22: $o_k \leftarrow$ graph(μ_k) $\neq$ graph(μ_h)
23: $o_w \leftarrow (O_k < D_h) \wedge (D_k > O_h)$ ▷ Scheduling windows overlap
24: **if** $\mu_k \bowtie \mu_h$ **and** (o_k **or** o_w) **then**
25: $C_{block} \leftarrow \max(C_{block}$, WCET of $\mu_h)$
26: **end if**
27: **end for**
28: $U_{target} \leftarrow \sum_{\mu_j \in DMA_{target}} C_j / T_j$ ▷ Current utilization of the target DMA
29: **return** $U_{target} + (C_k + C_{block})/T_k$
30: **end function**

6.5 Scheduling Analysis

After partitioning the CNN graph, allocating threads and communication nodes to resources, and encoding precedence constraints, the final and most critical phase of the CNN-PAS framework is the formal scheduling analysis. This section presents the mathematical validation used to verify that all temporal constraints are met and that the entire task set is schedulable on the target platform.

The primary challenge lies in the heterogeneous nature of the system. We must analyze two distinct types of schedulable entities—computation threads and communication nodes—which execute on different resources (cores and DMAs) and mutually interfere with each other. A monolithic analysis is therefore insufficient.

To address this, our analysis is decoupled into two parts, mirroring the system's architecture:

- First, we analyze the schedulability of **computation threads** on each core. This is based on a preemptive Earliest Deadline First (EDF) model, which explicitly accounts for the interference caused by DMA memory transfers, treating them as a higher-priority workload.
- Second, we analyze the schedulability of **communication nodes** on the DMA engines. This analysis uses a partitioned, non-preemptive EDF model and incorporates the blocking delays that a transfer on one DMA can experience from a competing transfer on the other.

The theoretical foundation for both analyses is the Demand Bound Function (dbf), which we extend to handle tasks with offsets and to formally bound the unique sources of interference present in our platform model.

6.5.1 Scheduling algorithm for threads

To schedule the computation threads on each core, we chose the preemptive EDF scheduling algorithm. After the deadline encoding phase, each thread is assigned an offset relative to the release time of the graph, and a relative deadline. Upon the task's release, the thread is inserted in a wake-up queue; it will be activated after its offset and put into the run-queue ordered by absolute deadlines.

Threads may suffer interference from the DMA engines; as explained in Section 4, a DMA transfer has priority over thread's accessing the same memory; therefore, any access to the local memory may be stalled for the duration of the memory transfer. This interference can be modeled as a preemption from a sporadic higher priority thread.

For the sake of the analysis, we model the scheduling on core π_p as a two-level scheduler: we consider at the highest priority the communication nodes that have source or destination in σ_p^1 or in σ_p^2; the lower priority level consists of the EDF scheduler for the threads. The scheduling analysis for this type of system has already been proposed in the literature for the case of sporadic threads, for example in [97]. Here, we extend the analysis to consider threads belonging to task graphs, and in particular we will use the *demand bound criterion* together with the *request bound function* for threads with offsets.

The original demand bound analysis of [98] only considers sporadic threads. However, threads belonging to the same task-graph have offsets with respect to each other. Therefore, we use the approximated method of [99] to compute the dbf of a task-graph on a given core.

Let $\mathsf{all}(\tau_i, p)$ be the set of threads of task graph τ_i allocated on core p, and let $\mathsf{interf}(\tau_i, p)$ be the set of communications nodes of task graph τ_i that have source or destination in σ_p^1 or in σ_p^2.

A thread $\mathsf{th}_j = (C_j, D_j, T_i, \phi_j)$ belonging to task graph τ_i is characterized by a worst-case computation time, a relative deadline, an offset as computed by the encoding algorithm

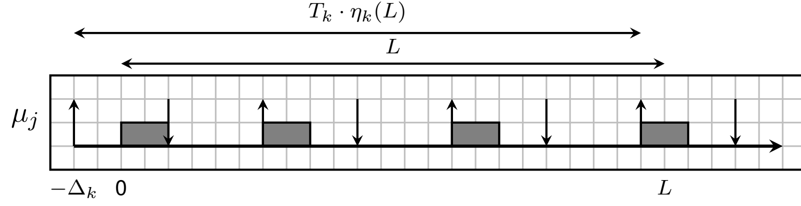


Figure 6.7: Request bound function for communication node μ_k

described in Section 6.4.1, and by a period T_i equal to the period of the corresponding task graph.

A communication node $\mu_j = (\delta_j, \mathsf{src}_j, \mathsf{dst}_j, D_j, \phi_j, T_i)$ in task graph τ_i is characterized by a worst-case transfer time δ_j, a source src_j and a destination memory dst_j, a relative deadline D_j and an offset ϕ_j computed in the encoding phase of Section 6.4.1, and a period T_i equal to the period of the task graph.

The dbf of task graph τ_i in interval L on core π_p can be computed as follows [100]:

$$\mathsf{dbf}(\tau_i, L, p) = \max_{\mathsf{th}_k \in \mathsf{all}(\tau_i, p)} \left(\sum_{\mathsf{th}_j \in \mathsf{all}(\tau_i, p)} \left(\left\lfloor \frac{L - \overline{\phi_{j,k}} - D_j}{T_i} \right\rfloor + 1 \right)_0 C_j \right) \tag{6.2}$$

where $\overline{\phi}_{k,j} = (\phi_k - \phi_j) \mod T_i$. Please refer to [100] for a proof of correctness.

We now compute a bound on the interference of communication nodes. Let us define $\Delta_k = D_k - \delta_k$. We define as $\eta_k(L)$ the maximum number of executions of μ_k that are entirely contained in interval $[0, L]$:

$$\eta_k(L) = \left\lfloor \frac{L + \Delta_k}{T_k} \right\rfloor .$$

The maximum amount of interference that message μ_k belonging to task τ_i can cause in interval $[0, L]$ is the following:

$$\mathsf{ib}_k(L) = \delta_k \cdot \eta_k(L) + \max(0, \min(\delta_k, L + \Delta_k - T_k \cdot \eta_k(L))) \tag{6.3}$$

To better understand Equation 6.3, consider the example of Figure 6.7: the interval $[0, L]$ contains $\eta_k(L) = 3$ full instances of μ_k, which contributes to the interference with $3\delta_k$, and one last instance that contributes for a fraction of the transfer time which can be computed as $\max(0, \min(L + \Delta_k - T_i\eta_k(L), \delta_k))$.

The *interference bound function* $\mathsf{ibf}(\tau_i, L, p)$ of the communication nodes belonging to task τ_i on processor π_p is computed by the following theorem.

Theorem 10

Assuming that all communication nodes are schedulable (i.e. they complete the data transfer before their respective deadlines), the interference that the communications nodes of task graph τ_i produce on the threads executing on processor π_p is upper bounded by:

$$\mathsf{ibf}(\tau_i, L, p) = \max_{\mu_k \in \mathsf{interf}(\tau_i, p)} \left(\sum_{mu_j \in \mathsf{interf}(\tau_i, p)} \mathsf{ib}_k(L - \overline{\phi}_{kj}) \right) . \tag{6.4}$$

Proof. Let $\mu_k \in \mathsf{interf}(\tau_i, p)$ be a communication node belonging to task τ_i and whose source or destination are σ_p^1 or σ_p^2. Suppose that a computation thread th_h allocated on π_p is executing

while one of the DMAs performs the data transfer μ_k. Thread th_h will suffer from interference for at most δ_k units of time in the worst case.

If μ_k is schedulable (i.e., if it always completes before its deadline), the total interference of μ_k on all threads executing on π_p in interval $[0, L]$ is never greater than $\mathsf{ib}_k(L)$: as shown in Figure 6.7, it is not possible to fit more interference in the interval if the node's deadlines are respected.

To compute the interference of all communication nodes in a task graph, we have to take into account their relative offset. We use the same techniques as in [101, 100] to estimate the alignment of the release times of the communication tasks that maximises the interference. We resume it here:

- We chose one of the communication nodes $\mu_k \mathsf{interf}(\tau_i, p)$, and we set its first release at instant $-\Delta_k$;
- We align the other communication nodes at their minimum distance from the first release of μ_k, the distance is $\overline{\phi}_{kj}$, and we sum their contribution in the interval;
- We compute the interference over all initial communication nodes μ_k, and then we take the maximum computed interference in the interval.

It is not possible to put more interfering nodes in the interval without breaking one of the following conditions:

- The schedulability of the communication nodes;
- The minimum distance between release times of the communication nodes;
- The period of the task graph.

Therefore, Equation 6.4 is an upper bound on the interference. □

Notice that Theorem 10 gives us a sufficient condition because we are overestimating the interference. In fact, a communication node μ_k which transfers data from to σ_p^1 *will not interfere* with a computation thread allocated on σ_p^2, and viceversa. However, this is not taken into account by function $\mathsf{ibf}()$, which is therefore a pessimistic bound. We pay this extra pessimism in order to keep reduce the complexity of the analysis.

Finally, for each core we have to sum the dbfs of all the threads and the ibfs of all interfering messages and check if the sum exceeds the interval, for every possible interval up to the hyperperiod.

Theorem 11

The threads allocated on core π_p are schedulable by the Earliest Deadline First algorithm if:

$$\forall L \in dead(H), \sum_{\tau_i} \mathsf{dbf}(\tau_i, L, p) + \mathsf{ibf}(\tau_i, L, p) \leq L \tag{6.5}$$

Proof. The following proof is very similar to the proof for the sufficient branch of the proof of the Demand Bound Criterion [98].

By contradiction. Suppose the theorem conditions are verified and that a deadline miss happens at time y. Let x be the latest instant prior to y where there is an idle time or a thread with deadline greater than y is active.

Therefore, by construction, in interval $[x, y]$ only threads with arrival time and deadline in the interval are executed, and there is no idle time. Since one thread missed its deadline,

the cumulative execution time of these threads is larger then $(y - x)$. The same cumulative execution time is upped bounded by Equation 6.5:

$$y - x < \sum_{\tau_i} \mathsf{dbf}(\tau_i, y - x, p) + \mathsf{ibf}(\tau_i, y - x, p) \leq y - x$$

Therefore, the theorem is proved by contradiction. □

6.5.2 Scheduling algorithm for communication nodes

The hardware platform under analysis has two DMA engines to transfer data. We reserve one of the cores to manage the two DMAs and coordinate the scheduling on the platform.

We use a *partitioned non-preemptive EDF* scheduler for the communication nodes; every DMA has its own scheduling queue of communication nodes, and every communication node is statically assigned to one of the two DMAs. At the release time r of a task graph, all the corresponding communication nodes are inserted into a waiting queue ordered by activation time $r + \phi_j$. At their activation time, they are inserted in their allocated scheduling queue which is ordered by absolute deadline $r + \phi_j + D_j$, and will be scheduled in a non-preemptive way by the corresponding DMA.

It may happen that at the scheduling time of message μ_k the other DMA engine is transferring a competing message $\mu_h \bowtie \mu_k$. In this case, the DMA is stalled until the transfer of μ_h is completed, at which time μ_k can start. Therefore, every communication node can be stalled for at most the duration of one competing message allocated to the other DMA.

Let Γ_1 be the set of communication nodes allocated on the first DMA, and let Γ_2 be the set of communication nodes allocated on the second DMA. A message $\mu_k \in \Gamma_1$ can stall for at most $\max\{\delta_h \mid \mu_h \in \mathsf{comp}(\mu_k) \cap \Gamma_2\}$. This stall time may impact all other communication nodes allocated on the same DMA, therefore we augment the worst-case transfer time of all nodes as follows:

$$\forall \mu_k \in \Gamma_1 \delta'_k = \delta_k + \max\{\delta_h \mid \mu_h \in \mathsf{comp}(\mu_k) \cap \Gamma_2\}$$

We do the same for every communication node in Γ_2.

Then, we use the demand bound function to asses the schedulability of communication nodes on every DMA. First we express the dbf of task graph τ_i for all the messages allocated on Γ_l, with $l \in= \{1, 2\}$:

$$\mathsf{dbf}(\tau_i, L, \Gamma_l) = \max_{\mu_k \in \Gamma_l \cap \tau_i} \left(\sum_{\mu_j \in \Gamma_l \cap \tau_i} \max \left(0, \left\lfloor \frac{L - \overline{\phi_{j,k}} - D_j}{T_i} \right\rfloor + 1 \right) \delta'_j \right) \tag{6.6}$$

Then, we sum all the dbfs and add the blocking time due to non-preemptive execution:

$$\forall L \in dead(H), \sum_{\tau_i} \mathsf{dbf}(\tau_i, L, \Gamma_l) + B(\Gamma_l, L) \leq L \tag{6.7}$$

where:

$$B(\Gamma_l, L) = \max_{\mu_k \in \Gamma_l} \{\delta_k | D_k > L\}$$

Theorem 12

The set of communication nodes is schedulable if Equation (6.7) is respected.

Proof. Once again, we follow the same technique as in Theorem 11.

By contradiction: assume that Equation (6.7) is respected, and a communication node misses its deadline at time y. Assume that this happens on DMA l.

Let us go back to the latest instant $x < y$ such that either there is an idle time before x or a communication node with deadline larger than y is being transferred.

In interval $[x, y]$, only communication nodes with absolute arrival time and absolute deadlines in $[x, y]$ are transferred, except at most one single communication node with deadline larger than y and absolute arrival time before x. Since a communication node misses its deadline, the total transfer time plus the waiting time for these communication nodes must be larger than $y - x$. At the same time, the total worst-case of these communication nodes is upper-bounded by Equation (6.7). Therefore:

$$y - x < \sum_{\tau_i} \mathsf{dbf}(\tau_i, y - x, \Gamma_l) + B(\Gamma_l, y - x) \leq y - x$$

This is a contradiction, hence the theorem is proved. □

6.6 Experimental evaluation

In this section, we present the experimental evaluation of the CNN-PAS framework. The primary objective of this evaluation is to validate the effectiveness of the CNN-PAS methodology in partitioning, allocating and scheduling real-time CNN tasks to meet their deadlines on a multicore microcontroller with scratchpad memories. We aim to demonstrate that our approach can successfully find schedulable configurations for complex CNNs while optimizing for performance.

The framework has been implemented in C++ as part of the open-source project `IACLIB`, available on GitLab[1]. IACLIB provides a library for the execution of CNNs on embedded systems and the framework described in this paper is a set of extensions to this library.

6.6.1 Benchmarks

CNN-PAS is evaluated using three representative CNN architectures used as benchmarks in the TinyML literature. Each model targets a different application domain and presents distinct structural characteristics.

- **ResNet-8**: a compact version of the ResNet architecture, designed for image classification tasks. Its defining feature is the use of residual blocks, where skip connections bypass one or more layers[2]. This technique helps mitigate the vanishing gradient problem, enabling the effective training of deeper networks.
- **DS-CNN**: a Depthwise Separable CNN optimized for audio keyword spotting, operating on spectrogram inputs. It replaces standard convolutions with depthwise separable convolutions[3]. The architecture is composed of a stack of these efficient blocks.
- **MCUNet**: a model architecture designed specifically for efficient inference on microcontrollers. MCUNet models are discovered through a neural architecture search framework that co-designs the network and its inference engine. We evaluate a specific instance targeting image classification.

[1] `https://gitlab.univ-nantes.fr/riviere-a-2/IACLIB`, under the `analysis/` directory.

[2] Each residual block consists of two convolutional layers and a shortcut connection ReLU activation. At the end of each block, the output is added to the block's input, skipping the connection between layers

[3] Depthwise separable convolutions consist of two stages: a depthwise convolution that applies a single filter per input channel, followed by a pointwise convolution (1×1) that combines the outputs. This factorization significantly reduces computational cost and model parameters.

Table 6.1: Summary of CNN models used in the evaluation.

Model	Primary Task	# Parameters	Operations (MACs)
ResNet-8	Image Classification	∼8K	∼15M
DS-CNN	Keyword Spotting	∼22K	∼7M
MCUNet	Image Classification	∼300K	∼80M

6.6.2 Experimental Setup

The hardware platform modeled in this work is based on the Infineon AURIX TC3XXX family of microcontrollers. To focus the analysis on scheduling and data movement, our model represents a targeted subset of the full architecture.

The platform model consists of six TriCore processors operating at a clock frequency of 300 MHz and it is equipped with two DMA engines for efficient data transfers. Each core has two local 96KB scratchpad memories. The model assumes a bus interconnect for core-to-core communication and a crossbar switch for core-to-SRAM access.

To ensure realistic timing, WCETs for all primitive tasks (e.g., convolution layers, pooling operations, and memory transfers) were empirically measured on the physical board. The experimental results presented in this paper were then obtained through a detailed simulation the CNN execution on the platform, using these measured WCET values as inputs. This approach ensures that our performance evaluation is grounded on the timing characteristics of the real hardware platform.

For computational nodes, we first identified the primitive operation for each layer type (e.g., a single 3x3 multiply-accumulate for convolution). The WCET of this primitive was measured by executing it in isolation roughly 500 times and recording the maximum cycle count. The total WCET for an entire layer was then derived from an equation that scales the primitive's WCET by the number of operations required for the layer's specific configuration. For the data transfers, we derived a linear cost model by benchmarking the DMA performance. We measured transfer times for data payloads from 32 to 2016 pixels (every pixel was encoded by one floating point, single precision value, which corresponds to 2 bytes) for both SRAM-to-SRAM and SRAM-to-RAM paths. This data allowed us to derive a predictive equation, $X = \alpha + \beta \times Y$, where X is the transfer time, Y is the number of bytes, and α and β are constants derived from the linear regression of the measured data. The key timing parameters derived from these measurements are summarized in Table 6.2. This benchmarking is not intended to evaluate the CNN performance or establish a performance baseline, but to derive reasonable WCET estimates necessary for the evaluation of our work.

Table 6.2: Measured WCET for primitive operations and data transfers on the AURIX TC3XXXs.

Category	α	β	Operation	WCET (μs)
Convolution	1.25	2.7	Single 3×3 operation	1.2
Pooling	1.7	3.2	Max-pool operation	0.9
SRAM ↔ SRAM	1.7	0.06	128 bytes transfer	3.5
SRAM ↔ DRAM	1.6	0.04	128 bytes transfer	3.2

6.6.3 Results

The framework's performance was evaluated by determining the minimum schedulable deadline for the selected benchmark CNNs. This minimum deadline was obtained under three scenarios. First, each CNN is considered executing in isolation on the hardware board to establish a baseline. Second, the CNN was executed concurrently with a synthetically generated set of periodic single node tasks imposing a background load with a maximum utilization of $U_{max} = 0.25$. The synthetic task set is composed of 10 periodic tasks, each with a period of 10 ms and a relative deadline equal to the period. This synthetic task set mimics a simple real-time application that interacts with the I/O devices, collects data from the sensors for processing by the CNN, accepts control commands, sends out commands to the actuators, or on the network, etc. The utilization bound is set to $U_{\max} = 0.25$ to account for computational tasks only, as the memory transfer overhead is analyzed and scheduled separately in our framework. For the CNN, the period is equal to the relative deadline.

Third, we evaluated the CNN running concurrently with the same synthetic task set as before, but this time we considered three instances of the same CNN executing in parallel, each with its own deadline equal to the period. This scenario represents a more challenging use case where multiple CNN-based applications run simultaneously on the same hardware platform.

In Figure 6.8 we present the relationship between minimum deadline and model complexity. The complexity of the model is quantified by the number of Multiply-Accumulate (MAC) operations, which is a more accurate measure of computational workload than the number of parameters or layers. For instance, DS-CNN requires approximately half the MACs of ResNet-8 (7×10^6 vs. 15×10^6) despite having more parameters (30k vs. 8k). This efficiency stems from its use of depthwise separable convolutions, resulting in a lower execution latency. We remark that the y-axis is in log scale.

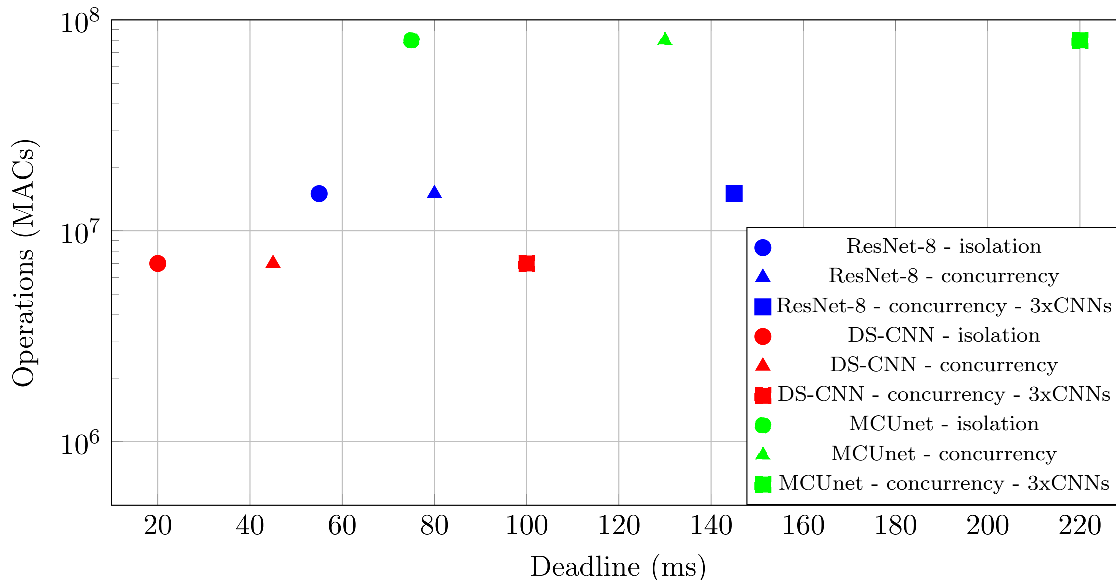


Figure 6.8: Minimum schedulable deadline for various CNNs in isolation and in concurrency as a function of their computational complexity (MACs).

Next, we evaluated the impact of the execution deadline on the framework's partitioning strategy. The number of horizontal partitions created for a given CNN is a direct result of the scheduling constraints imposed by its deadline.

Figure 6.9 illustrates this relationship, showing that the number of partitions is inversely proportional to the assigned deadline for several CNNs executing in isolation. A longer dead-

line provides the scheduler with the flexibility to merge more sequential layer operations into a single partition. For reference, the figure also shows the minimum deadline required for each CNN to be schedulable without any partitioning, where the model is a simple pipeline of layers. Our results demonstrate that partitioning achieves a speedup of 2x to 3x, depending on the CNN architecture. Ideally, the degree of parallelism would match the number of available cores. However, it is practically limited by inter-layer data dependencies and the difficulty of assigning feasible intermediate deadlines. Consequently, the number of concurrent partitions is always less than the number of available cores.

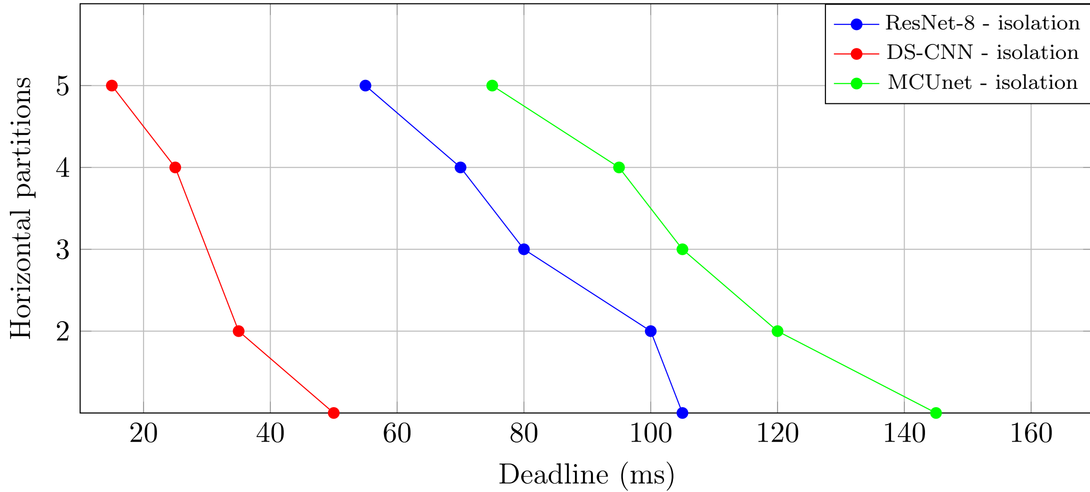


Figure 6.9: Number of horizontal partitions generated for different CNN models as a function of the assigned execution deadline.

Table 6.3 summarizes the average occupancy of each scratchpad memory, the average load (utilization) of each core, and the time to execute the CNN-PAS framework for each CNN model under various deadlines in the first setting (one single CNN executing alone). Most of the execution time of our framework is spent in the schedulability analysis, whose complexity grows with the number of threads and the number of layers. In particular, the construction of the dbfs of Equation (6.2) is the function where our program spends most of the time. We are currently working on techniques to speed up this part of the code.

Table 6.3: Average scratchpad memory occupancy, core utilization and framework execution time for different CNN models under various deadlines.

Model	**Deadline (ms)**	**Avg. Occupancy (KB)**	**Avg. Utilization %**	**Framework ex. time (s)**
ResNet-8	55	51.3	64	276
ResNet-8	80	45.5	59	205
ResNet-8	105	46.8	48	189
DS-CNN	15	16.8	27	295
DS-CNN	30	15.3	19	179
DS-CNN	55	9.4	12	132
MCUNet	75	42.5	58	495
MCUNet	105	33.2	45	354
MCUNet	120	38.4	41	301

6.6.4 Comparison vs. MicroNets

To contextualize the performance of our framework, we compare our results against MicroNets [48], a state-of-the-art family of models optimized for resource-constrained devices. MicroNets results were obtained on three different Arm Cortex-M based hardwares, small, medium and large, respectively. We decided to focus on the large hardware, which is the most comparable to our platform in terms of performance, the SRM32F767ZI microcontroller single-core @ 216 MHz, 512 KB RAM. Our evaluations are performed on an AURIX TriCore platform (6-core @ 300 MHz, 1152 KB total scratchpad memory). It is crucial to note that MicroNets benchmarks were executed using a highly optimized framework, such as TFLM, to deploy TinyML algorithms on microcontrollers, while our framework is based on IACLIB, which has been designed with portability in mind. Given the fundamental differences in processor architecture, number of cores, and software libraries, this comparison is not intended as a direct performance benchmark. Instead, its purpose is to position our results relative to an established, highly optimized solution.

The MicroNets suite [48] provides a performance breakdown for models of varying size and complexity[4]. We selected small and medium variants for three applications: Anomaly Detection (MicroNet-AD), Keyword Spotting (MicroNet-KWS), and Visual Wake Words (MicroNet-VWW). The computational load of these models, measured in Multiply-Accumulate operations (MACs), increases with size, which typically corresponds to higher accuracy.

The performance comparison is shown in Figure 6.10, which plots the operational complexity (MACs, on a log scale) against the measured inference latency. The performance numbers for MicroNets are taken from Figure 4 in [48]. The figure highlights the efficiency of our framework on the AURIX TriCore architecture (circles), which consistently outperforms the Arm Cortex-M7 baseline (triangles). Specifically, our implementation yields a speedup of 3.4x on average across all benchmark models.

As a final note, it is important to highlight that our CNN-PAS framework not only provides parallelism and performance gains, but unlike [48] it also guarantees the respect of the real-time deadlines, even in the presence of concurrent periodic real-time tasks.

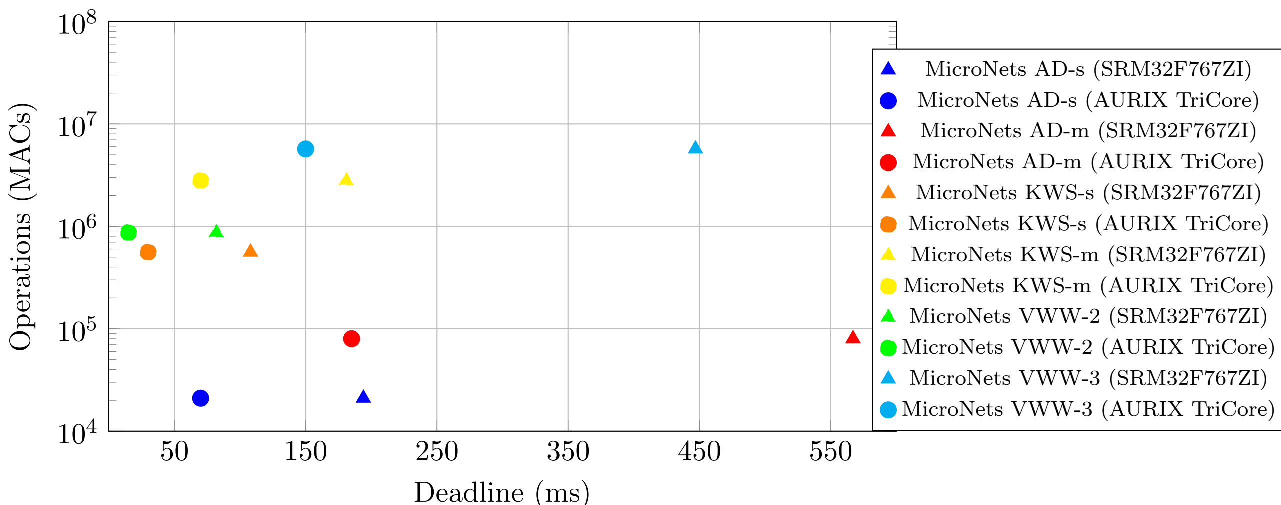


Figure 6.10: Operations vs. Deadline for our framework on an AURIX TriCore, compared with specific state-of-the-art MicroNet models executed on an Arm Cortex-M platform.

[4]These models can be found in the GitHub repository https://github.com/Arm-Examples/ML-zoo/tree/master/models

6.6.5 Applicability

The results presented in this work demonstrate the effectiveness of the proposed CNN-PAS framework for executing CNN workloads under real-time constraints on multicore micro-controllers with scratchpad memories. Although the experiments were conducted using a simulation environment, the timing parameters used in the simulations were extracted from measurements performed on the target hardware.

A full implementation on hardware with a multicore RTOS would provide the ultimate validation of the approach. However, such validation is currently limited by the absence of an RTOS that supports multicore scheduling with explicit scratchpad memory management on the AURIX family. Developing such an RTOS is part of our planned future work.

6.7 Model Instantiation from a Network Definition File

To apply the formal AECR-DAG model to a given neural network, a concrete description of the network's architecture is required. Our framework uses a structured JSON file to define the layers and their connectivity.

An example of this format is shown in Listing 6.1. The entire network is defined within a ‘network‘ block, which contains a sequence of ‘layer‘ blocks.

Listing 6.1: Example of a network definition file used as input to the IACLIB framework.

```
network {
  layer {
    name: "conv"
    type: "Convolution"
    activation: RELU
    bottom: "data"
    top: "conv_dw"
    convolution_param {
      num_output: 64
      kernel_size: 3
      stride: 2
      pad: 1
    }
  }

  layer {
    name: "conv_dw"
    type: "Convolution_dw" # Depthwise Convolution
    activation: RELU
    bottom: "conv"  # Takes input from the previous layer
    top: "conv_pw"
    convolution_param {
      kernel_size: 3
      stride: 1
      pad: 1
    }
  }

  layer {
    name: "conv_pw"
    type: "Convolution_pw" # Pointwise Convolution
    activation: RELU
    bottom: "conv_dw"  # Takes input from the depthwise layer
    top: "pool"
    convolution_param {
      num_output: 128
      kernel_size: 1        # Kernel size is always 1x1
      stride: 1
      pad: 0
    }
  }

  layer {
    name: "pool"
```

```
    type: Pooling
    activation: RELU
    bottom: "conv_pw"
    top: "fc"
    pooling_param {
      pool: AVG
      kernel_size: 2
      stride: 2
    }
  }

  layer {
    name: "fc"
    type: "InnerProduct"
    activation: SOFTMAX
    bottom: "pool"
    top: "output"
    inner_product_param {
      num_output: 12
    }
  }
}
```

Each ‘layer‘ block represents a distinct operation and is defined by several key fields:

- `name`: a unique string that serves as an identifier for the layer;
- `type`: specifies the type of operation the layer performs, such as:
  - `Convolution`;
  - `Convolution_dw` (depthwise convolution);
  - `Convolution_pw` (pointwise convolution);
  - `Pooling`;
  - `InnerProduct` (fully connected layer).
- `activation`: defines the non-linear activation function (e.g., `RELU`, `SOFTMAX`) that is applied after the primary operation.
- `bottom`: specifies the input layer(s) for this layer, which must match the `top` of a previous layer. If the layer is the first in the network, it will typically reference an input data layer (e.g., `data`);
- `top`: specifies the output layer(s) produced by this layer, which will be used as input for subsequent layers. If the layer is the last in the network, this will be the final output layer (e.g., `output`);
- Parameter Blocks: each layer type has a corresponding parameter block that specifies its unique attributes. For a convolution, this includes the number of output channels (`num_output`), `kernel_size`, `stride`, and `pad`. For an inner product layer, it specifies the number of output neurons (`num_output`). For pooling layers, it includes parameters like `pool_size`, `stride` and type of pooling, `AVG`, `MAX`, `MIN` or `GLOBAL`.

The IACLIB parser processes this file to automatically instantiate the AECR-DAG model. Each layer is translated into a set of computational nodes, and the bottom/top connections are used to establish the precedence constraints.

6.8 Conclusion

In this chapter, we introduced the CNN-PAS framework, a comprehensive methodology designed to bridge the gap between the high-level representation of CNNs and their predictable,

real-time execution on multicore embedded systems. Moving beyond the single-task optimization of the previous chapter, CNN-PAS addresses the system-level challenge of scheduling complex CNNs concurrently with other real-time tasks. We presented a complete toolchain that systematically transforms a high-level network specification into a schedulable set of threads and communication tasks through an iterative process of partitioning, allocation, precedence encoding, and formal schedulability analysis.

Our experimental evaluation demonstrated the effectiveness of this approach. We provided a clear, data-driven justification for our choice of a worst-fit allocation heuristic, showing its superiority over best-fit in exploiting multicore parallelism. The framework was successfully validated on three relevant benchmarks, ResNet-8, DS-CNN, and MCUNet, proving its capability to find schedulable configurations even when executing concurrently with other tasks. The results showed that our partitioning strategy achieves a significant speedup of 2x to 3x over a simple pipeline implementation. Furthermore, when compared against the highly-optimized MicroNets framework, CNN-PAS demonstrated competitive performance, validating it as a high-performance solution for automotive-grade hardware.

Our analysis also highlighted a key limitation and a clear direction for future work. The low average core utilization shown in Table 6.3 indicates that our current heuristic for assigning intermediate deadlines is often too conservative, preventing the system from achieving higher levels of parallelism. This presents the most significant opportunity for improvement.

We conclude this chapter by introducing IACLIB. This open-source library implements the CNN-PAS framework and serves as a simulation environment for the experiments presented in this and the previous chapter.

Part III

Conclusion and Future Work

Chapter 7

Conclusion and Future Work

This thesis addresses the critical challenge of executing CNNs within the strict temporal constraints of hard real-time embedded systems. We established the conflict between the high performance offered by modern multicore architectures and the loss of timing predictability caused by contention for shared resources. To resolve this, we adopted an architecture based on scratchpad memories and a phased Acquisition-Execution-Communication-Restitution (AECR) model to create a predictable hardware and software foundation. Upon this foundation, we built a comprehensive methodology to transform high-level CNN models into schedulable implementations.

7.1 Contributions

The primary contributions of this research are centered on developing a complete methodology for the predictable, real-time execution of CNNs on multicore embedded platforms. The work progresses from a formal theoretical model to a practical and validated scheduling framework. The specific contributions are as follows:

1. Formal task and platform model for predictable CNN execution (Chapter 4): we established a formal foundation for analyzing CNNs in a real-time context. This involved:
    - Adopting and extending the AECR-DAG task model, which explicitly separates memory-bound phases (Acquisition, Communication, Restitution) from the computational (Execution) phase to enable contention-free analysis.
    - Defining a corresponding hardware platform model based on a multicore architecture with software-managed scratchpad memories and DMA engines.
2. Optimal ILP-based mapping for single CNNs (Chapter 5): to understand the theoretical performance limits, we first addressed the problem of optimally mapping a single CNN.
    - We introduced the concept of a localized operation as the smallest schedulable unit and developed a detailed memory transfer cost model that accounts for latency penalties from non-contiguous access.
    - We formulated an Integer Linear Programming (ILP) problem that finds a provably optimal mapping of these operations to cores, minimizing end-to-end latency.
    - Through this optimal model, we discovered that an optimal mapping naturally creates contiguous horizontal partitions of the workload to align with the row-major memory layout, a key insight that directly motivated our subsequent heuristic approach.
3. CNN-PAS heuristic scheduling framework (Chapter 6): recognizing the scalability limitations of the ILP, we developed CNN-PAS, a practical, multi-phase framework for scheduling entire real-time systems containing CNNs.
    - The framework employs an iterative methodology that systematically partitions the CNN graph, allocates threads using a parallelism-aware worst-fit heuristic, models memory transfers, and encodes precedence constraints.

- We provided a complete schedulability analysis based on the Demand Bound Function (DBF), which formally verifies the timing correctness of both preemptive computational threads on cores and non-preemptive communication tasks on DMA engines, accounting for their mutual interference.

4. IACLIB software framework (Chapter 6): A significant practical outcome of this research is the development of IACLIB (Inference and Analysis of CNNs Library), an open-source software framework to implement the CNN-PAS methodology.

5. Comprehensive experimental validation (Chapter 6): The effectiveness of the CNN-PAS framework was validated through simulations using WCETs measured on a real AURIX TriCore platform. We demonstrated that CNN-PAS can find schedulable configurations for complex benchmarks like ResNet-8 and MCUNet, achieving a 2x to 3x speedup over non-parallel implementations and showing competitive performance against state-of-the-art TinyML solutions.

7.2 Future Work

The contributions presented in this thesis establish a robust foundation for predictable CNN execution, yet they also open several promising avenues for future research. Our work can be extended in three primary directions: physical implementation and validation, advanced memory management, and the enhancement of our scheduling heuristics.

A key next step is the implementation and validation of the CNN-PAS framework on the physical hardware. The current work relies on a simulation, a necessary step due to the lack of a commercial multi-core Real-Time Operating System (RTOS) that offers the control over DMA and co-scheduling required by our model. Therefore, a significant future effort involves porting a suitable open-source RTOS (such as FreeRTOS) and implementing the proposed scheduling mechanisms. This would involve developing custom DMA drivers and integrating our schedulability analysis into the kernel's admission control, allowing for a direct comparison between simulated results and real-world performance.

Furthermore, we plan to extend the framework to incorporate dynamic scratchpad memory management. Our current approach assumes a static memory allocation for all buffers, which can be inefficient. Developing mechanisms to allocate memory for task data on-demand at activation and free the space upon completion would improve memory efficiency. This would require designing and analyzing real-time memory management policies that are fast, deterministic, and avoid fragmentation, enabling the deployment of larger and more complex CNN models on the same hardware.

Finally, we aim to explore more advanced precedence encoding techniques. Our analysis showed that the current heuristic for assigning intermediate deadlines can be conservative, sometimes limiting the achievable parallelism. Future work will investigate alternative strategies, such as slack-based optimization or methods that consider the entire system's workload, to find less constrained sets of intermediate deadlines. This could unlock higher core utilization and allow the framework to find schedulable solutions for even tighter end-to-end performance requirements, further expanding the capabilities of real-time embedded systems.

7.3 Research Output

The research conducted in this thesis has contributed to the following publications and open-source software.

7.3.1 Publications

1. C. Daini, G. Lipari, H.E. Zahaf, and P.-E. Hladik, **"Optimizing CNN Inference on Multicore Scratchpad Architectures,"** in *Proceedings of the IEEE International Symposium on Real-Time Distributed Computing (ISORC)*, Toulouse, France, 2025.

2. C. Daini, G. Lipari, and H.E. Zahaf, **"Executing Real-Time CNN applications on embedded systems with scratchpad memory,"** submitted to the *Journal of Systems Architecture*.

7.3.2 Software

The software framework developed as a key component of this research is available as an open-source project.

- **IACLIB (Inference and Analysis of CNNs Library):** A comprehensive C++/C framework for the analysis, optimization, and deployment of CNNs on real-time embedded systems. The source code is available at: https://gitlab.univ-nantes.fr/riviere-a-2/IACLIB.

Bibliography

[1] Z. Zhang and J. Li, "A review of artificial intelligence in embedded systems," *Micromachines*, vol. 14, no. 5, p. 897, 2023.

[2] Y. Abadade, A. Temouden, H. Bamoumen, N. Benamar, Y. Chtouki, and A. S. Hafid, "A comprehensive survey on tinyml," *IEEE Access*, vol. 11, pp. 96 892–96 922, 2023.

[3] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Tenth international workshop on frontiers in handwriting recognition*. Suvisoft, 2006.

[4] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang *et al.*, "Tensorflow lite micro: Embedded machine learning for tinyml systems," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 800–811, 2021.

[5] "Tinyml AI inference library," https://utensor.github.io/, 2019, accessed: 2025-07-18.

[6] L. Lai, N. Suda, and V. Chandra, "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus," *arXiv preprint arXiv:1801.06601*, 2018.

[7] I. Senoussaoui, G. Lipari, H.-E. Zahaf, and M. K. Benhaoua, "Memory-processor co-scheduling of aecr-dag real-time tasks on partitioned multicore platforms with scratchpads," *Journal of Systems Architecture*, vol. 150, p. 103117, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762124000547

[8] R. Arora, A. Basu, P. Mianjy, and A. Mukherjee, "Understanding deep neural networks with rectified linear units," *arXiv preprint arXiv:1611.01491*, 2016.

[9] Z. K. Abdul and A. K. Al-Talabani, "Mel frequency cepstral coefficient and its applications: A review," *IEEE Access*, vol. 10, pp. 122 136–122 158, 2022.

[10] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1251–1258.

[11] B.-S. Hua, M.-K. Tran, and S.-K. Yeung, "Pointwise convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 984–993.

[12] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous systems," https://www.tensorflow.org/, 2015, software available from tensorflow.org.

[13] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," https://pytorch.org/, 2019, software available from pytorch.org.

[14] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[15] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: Analysis, applications, and prospects," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 12, pp. 6999–7019, 2022.

[16] K. Fukushima and S. Miyake, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.

[17] K. Fukushima, "A neural network model for a mechanism of visual pattern recognition," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 18, no. 5, pp. 802–814, 1988.

[18] D. E. Rumelhart and D. Zipser, "Feature discovery by competitive learning," *Cognitive science*, vol. 9, no. 1, pp. 75–112, 1985.

[19] S. Linnainmaa, "The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors," *Master's Thesis, Univ. Helsinki*, 1970.

[20] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[21] X. Zhang and Y. LeCun, "Character-level convolutional networks for text classification," *Advances in Neural Information Processing Systems*, vol. 28, 2015.

[22] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[23] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. PMLR, 2010, pp. 249–256.

[24] M. Ranzato, C. Poultney, S. Chopra, and Y. LeCun, "Unsupervised learning of invariant feature hierarchies with applications to object recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*. IEEE, 2007, pp. 1–8.

[25] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for?" *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[26] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, pp. 211–252, 2015.

[27] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International journal of computer vision*, vol. 88, no. 2, pp. 303–338, 2010.

[28] A. Krizhevsky, "Learning multiple layers of features from tiny images," *Technical report, University of Toronto*, 2009.

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

[30] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

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

[32] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[33] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[34] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[35] M. Sandler *et al.*, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *CVPR*, 2018.

[36] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, "Searching for mobilenetv3," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 1314–1324.

[37] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *ICLR*, 2017.

[38] H. Liu *et al.*, "Darts: Differentiable architecture search," in *ICLR*, 2019.

[39] V. J. Reddi, B. Plancher, S. Kennedy, L. Moroney, P. Warden, A. Agarwal, C. Banbury, M. Banzi, M. Bennett, B. Brown *et al.*, "Widening access to applied machine learning with tinyml," *arXiv preprint arXiv:2106.04008*, 2021.

[40] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou *et al.*, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 446–459.

[41] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov *et al.*, "Benchmarking tinyml systems: Challenges and direction," *arXiv preprint arXiv:2003.04821*, 2020.

[42] J. Lin, W.-M. Chen, Y. Lin, and S. Han, "Mcunet: Tiny deep learning on iot devices," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[43] Z. Guo, J. Chen, T. He, W. Wang, H. Abbas, and Z. Lv, "Ds-cnn: Dual-stream convolutional neural networks-based heart sound classification for wearable devices," *IEEE Transactions on Consumer Electronics*, vol. 69, no. 4, pp. 1186–1194, 2023.

[44] E. A. P. Alday, A. Gu, A. J. Shah, C. Robichaux, A.-K. I. Wong, C. Liu, F. Liu, A. B. Rad, A. Elola, S. Seyedi *et al.*, "Classification of 12-lead ecgs: the physionet/computing in cardiology challenge 2020," *Physiological measurement*, vol. 41, no. 12, p. 124003, 2020.

[45] P. Suja, S. Tripathi *et al.*, "Real-time emotion recognition from facial images using raspberry pi ii," in *2016 3rd International Conference on Signal Processing and Integrated Networks (SPIN)*. IEEE, 2016, pp. 666–670.

[46] M. Vieira, J. Silva, and P. Santos, “Low-cost real-time monitoring system for edge computing in industrial iot,” *IEEE Internet of Things Journal*, vol. 9, no. 12, pp. 9876–9890, 2022.

[47] V. Mazzia, A. Khaliq, and M. Chiaberge, “Real-time vehicle detection for autonomous driving using deep learning,” in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 3825–3831.

[48] C. Banbury, C. Zhou, I. Fedorov, R. Matas, U. Thakker, D. Gope, V. Janapa Reddi, M. Mattina, and P. Whatmough, “Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers,” *Proceedings of machine learning and systems*, vol. 3, pp. 517–532, 2021.

[49] G. Durrieu, M. Faugère, S. Girbal, D. G. Pérez, C. Pagetti, and W. Puffitsch, “Predictable flight management system implementation on a multicore processor,” in *Embedded Real Time Software (ERTS’14)*, 2014.

[50] S. Yogamani, “Efficient cnn architecture design techniques for real-time embedded systems deployment,” 2024.

[51] T. Garbay, K. Hachicha, P. Dobias, A. Pinna, K. Hocine, W. Dron, P. Lusich, I. Khalis, and B. Granado, “Zip-cnn: Design space exploration for cnn implementation within a mcu,” *ACM Transactions on Embedded Computing Systems*, vol. 24, no. 1, pp. 1–26, 2024.

[52] I. D. A. Silva, T. Carle, A. Gauffriau, and C. Pagetti, “Acetone: predictable programming framework for ml applications in safety-critical systems,” in *24th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, vol. 8, no. 2, 2022.

[53] I. D. A. Silva, “Implantation certifiable et efficace de réseaux de neurones sur des systèmes embarqués temps-réel critiques,” Ph.D. dissertation, Toulouse, ISAE, 2024.

[54] I. D. A. Silva, T. Carle, A. Gauffriau, and C. Pagetti, “Extending a predictable machine learning framework with efficient gemm-based convolution routines,” *Real-Time Systems*, vol. 59, pp. 408–437, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:261318279

[55] P. S. Juan, A. Castelló, M. F. Dolz, P. Alonso-Jordá, and E. S. Quintana-Ortí, “High performance and portable convolution operators for multicore processors,” *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 91–98, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:225079823

[56] Y. Wen, A. Anderson, V. Radu, M. F. O’Boyle, and D. Gregg, “Taso: Time and space optimization for memory-constrained dnn inference,” in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, pp. 199–208.

[57] E. Russo, M. Palesi, G. Ascia, D. Patti, S. Monteleone, and V. Catania, “Memory-aware dnn algorithm-hardware mapping via integer linear programming,” in *Proceedings of the 20th ACM International Conference on Computing Frontiers*, ser. CF ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 134–143. [Online]. Available: https://doi.org/10.1145/3587135.3592206

[58] G. C. Buttazzo, “A general view,” in *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer US, 2011, pp. 1–22.

[59] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[60] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–53, 2008.

[61] P. Puschner and A. Burns, "Guest editorial: A review of worst-case execution-time analysis," *Real-Time Systems*, vol. 18, no. 2-3, pp. 115–128, 2000.

[62] J. Goossens and R. Devillers, "General response time computation for hard real-time asynchronous periodic task sets using static schedulers," Technical Report 401, Universit e Libre de Bruxelles, Belgium, Tech. Rep., 1999.

[63] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[64] J. Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, no. 4, pp. 237–250, 1982.

[65] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software engineering journal*, vol. 8, no. 5, pp. 284–292, 1993.

[66] M. Joseph and P. Pandya, "Finding response times in a real-time system," *The Computer Journal*, vol. 29, no. 5, pp. 390–395, 1986.

[67] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," *Proceedings of the 11th real-time systems symposium*, pp. 182–190, 1990.

[68] M. L. Dertouzos and A. K. Mok, "Multiprocessor online scheduling of hard-real-time tasks," *IEEE Transactions on software engineering*, vol. 15, no. 12, pp. 1497–1506, 1989.

[69] J. Y.-T. Leung, "A new algorithm for scheduling periodic, real-time tasks," *Algorithmica*, vol. 4, no. 1, pp. 209–219, 1989.

[70] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," in *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, 1993, pp. 345–354.

[71] S. K. Baruah, J. E. Gehrke, and C. G. Plaxton, "Fast scheduling of periodic tasks on multiple resources," in *Proceedings of 9th International Parallel Processing Symposium.* IEEE, 1995, pp. 280–288.

[72] A. Davis and O. A. Burns, "A survey of the field," *Journal of Surveys*, vol. 1, pp. 1–20, 2011.

[73] B. B. Brandenburg and M. Gül, "Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations," in *2016 IEEE Real-Time Systems Symposium (RTSS)*, 2016, pp. 99–110.

[74] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers," in *2010 31st IEEE Real-Time Systems Symposium*, 2010, pp. 14–24.

[75] S. Baruah, M. Bertogna, and G. Buttazzo, *Multiprocessor scheduling for real-time systems.* Springer, 2015.

[76] B. Andersson, S. Baruah, and J. Jonsson, “Static-priority scheduling on multiprocessors,” in *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420)*, 2001, pp. 193–202.

[77] G. E. Moore *et al.*, “Cramming more components onto integrated circuits,” 1965.

[78] ——, “Progress in digital integrated electronics,” in *Electron devices meeting*, vol. 21. Washington, DC, 1975, pp. 11–13.

[79] A. Ltd. (2011) big.little technology. Accessed: 2025-08-04. [Online]. Available: https://developer.arm.com/documentation/120005/0200/

[80] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, “A survey on cache management mechanisms for real-time embedded systems,” *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, pp. 1–36, 2015.

[81] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, “The influence of processor architecture on the design and the results of wcet tools,” *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1038–1054, 2003.

[82] F. Mueller, “Timing analysis for instruction caches,” *Real-time systems*, vol. 18, no. 2, pp. 217–247, 2000.

[83] T. Lundqvist and P. Stenström, “An integrated path and timing analysis method based on cycle-level symbolic execution,” *Real-Time Systems*, vol. 17, no. 2, pp. 183–207, 1999.

[84] Y.-T. Li, S. Malik, and A. Wolfe, “Cache modeling for real-time software: Beyond direct mapped instruction caches,” in *17th IEEE Real-Time Systems Symposium.* IEEE, 1996, pp. 254–263.

[85] S. Udayakumaran and R. Barua, “Compiler-decided dynamic memory allocation for scratch-pad based embedded systems,” in *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, 2003, pp. 276–286.

[86] V. Suhendra, A. Roychoudhury, and T. Mitra, “Scratchpad allocation for concurrent embedded software,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 4, pp. 1–47, 2010.

[87] Y. Yang, M. Wang, H. Yan, Z. Shao, and M. Guo, “Dynamic scratch-pad memory management with data pipelining for embedded systems,” *Concurrency and Computation: Practice and Experience*, vol. 22, no. 13, pp. 1874–1892, 2010.

[88] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, “Survey of scheduling techniques for addressing shared resources in multicore processors,” *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, pp. 1–28, 2012.

[89] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, “Memory bandwidth management for efficient performance isolation in multi-core platforms,” *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 562–576, 2015.

[90] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, “A predictable execution model for cots-based embedded systems,” in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011, pp. 269–279.

[91] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo, "Memory-centric scheduling for multicore hard real-time systems," *Real-Time Syst.*, vol. 48, no. 6, p. 681–715, Nov. 2012. [Online]. Available: https://doi.org/10.1007/s11241-012-9158-9

[92] J. A. Nelder and R. Mead, "A Simplex Method for Function Minimization," *The Computer Journal*, vol. 7, no. 4, pp. 308–313, 01 1965. [Online]. Available: https://doi.org/10.1093/comjnl/7.4.308

[93] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2024. [Online]. Available: https://www.gurobi.com

[94] I. I. Cplex, "V12. 1: User's manual for cplex," *International Business Machines Corporation*, vol. 46, no. 53, p. 157, 2009.

[95] H. Wu, D. Zhu, and J. W. Liu, "Deadline miss ratio and utilization bound for edf scheduling on multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1482–1491, 2014.

[96] N. Serreli, G. Lipari, E. Bini *et al.*, "Deadline assignment for component-based analysis of real-time transactions," in *2nd Workshop on Compositional Real-Time Systems, Washington, DC, USA*, 2009.

[97] M. G. Harbour, J. J. G. García, and J. C. Palencia, "Response time analysis for tasks scheduled under edf within fixed priorities," Universidad de Cantabria, Tech. Rep. TR-03-01, 2003.

[98] S. Baruah, L. Rosier, and R. Howell, "Preemptively scheduling hard-real-time sporadic tasks on one processor," *Real-Time Systems*, vol. 2, no. 2, pp. 165–178, 1990.

[99] R. Pellizzoni and G. Lipari, "Feasibility analysis of real-time periodic tasks with offsets," in *11th IEEE Real Time and Embedded Technology and Applications Symposium*. IEEE, 2005, pp. 3–14.

[100] I. Senoussaoui, G. Lipari, H.-E. Zahaf, and M. K. Benhaoua, "Memory-processor co-scheduling of aecr-dag real-time tasks on partitioned multicore platforms with scratchpads," *Journal of Systems Architecture*, vol. 150, p. 103117, 2024.

[101] R. Pellizzoni and G. Lipari, "Holistic analysis of asynchronous real-time transactions with earliest deadline scheduling," *Journal of Computer and System Sciences*, vol. 73, no. 2, pp. 186–206, 2007.