

Analysis of Convolutional Neural Networks on Real-Time Multi-Core Embedded Platforms

Sarah Lounnaci*, Djamila Bendouda†, Pierre-Emmanuel Hladik‡, Claire Maiza§

*École ENSTA, Département GEII, Algiers, Algeria

†École ENSTA, Département MI, Laboratory of Innovative Technologies (LTI), Algiers, Algeria

‡Nantes Université, École Centrale Nantes, CNRS, LS2N, UMR 6004, F-44000 Nantes, France

§Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, 38000 Grenoble, France

s_lounnaci, djamila.bendouda@ensta.edu.dz Pierre-Emmanuel.hladik@ls2n.fr, claire.maiza@univ-grenoble-alpes.fr

Abstract—This article provides a bibliographical study of Convolutional Neural Networks (CNNs) architecture analysis and methods of deploying them in embedded real-time multi-core platforms. We first review the main structures of CNNs and their representation as Directed Acyclic Graphs (DAGs). We then analyse the real-time execution challenges on multi-core architectures, focusing on shared-resource interference, common arbitration mechanisms, and discuss how Worst-Case Execution Time (WCET) and Worst-Case Response Time (WCRT) can be estimated using tools like OTAWA and MIA. Next, we survey existing analytical methods for timing analysis of CNN inference, including model representation formats and toolchains. Finally, we outline our ongoing research effort developing the MIND (Model Inference Network Designer) tool to automatically extract CNN structure into a DAG compatible with the MIA timing-analysis tool, and we end this article by giving some perspectives.

Index Terms—Convolutional Neural Networks (CNNs), Directed Acyclic Graphs (DAGs), multi-core architectures, real-time embedded systems, timing interference, memory arbitration (TDMA, Round-Robin), Worst-Case Execution Time (WCET), Worst-Case Response Time (WCRT), Kalray MPPA, Artificial Intelligence (AI), static scheduling.

I. INTRODUCTION

The implementation of deep learning models, such as Convolutional Neural Networks (CNNs), in real-time embedded multi-core systems poses significant resource management and prediction issues. It is essential to examine their structure, which is frequently depicted as Directed Acyclic Graphs (DAGs), and assess their Worst-Case Execution Time (WCET) and Worst-Case Response Time (WCRT) in order to guarantee safe and effective execution. Researchers are progressively creating frameworks and methods to automate the extraction, analysis, and scheduling of these models under real-time limitations in response to these difficulties.

The objective of this study is to review and evaluate existing methods for representing and analyzing CNNs in real-time multi-core embedded systems, with a focus on timing predictability and tool support for WCET and WCRT estimation.

The remainder of this paper is organized as follows: Section II reviews the fundamentals of CNN layers and their natural representation as directed acyclic graphs (DAGs). Section III explores the challenges of real-time execution on multi-core

platforms, including WCET estimation, shared-memory contention, and interference WCRT analysis. Section IV discusses the current analytical methods (formats and frameworks) used to study CNN behavior under real-time constraints, highlighting also their limitations. Section V presents a synthesis of our current research, introducing the MIND software for automatic extraction of CNN structure into a DAG suitable for MIA-based timing analysis. Finally, Section VI concludes the paper and outlines potential directions for our actual research and even for future ones.

II. FUNDAMENTALS OF CONVOLUTIONAL NEURAL NETWORKS AND THEIR STRUCTURAL REPRESENTATION

The evolution of CNNs began with the work of LeCun et al. [1] on LeNet-5 in 1998. It was a simple architecture originally designed for specific task to recognize handwritten digits on trained small datasets like MNIST. Fast forward, CNNs became more popular due to their ability to automatically learn visual patterns which has made them a backbone of modern computer vision.

But with this success came new challenges. As CNNs grew deeper and more accurate, reaching millions of parameters and requiring billions of FLOPs to process a single image [2], their computational and memory cost increased. As a consequence, making these models run efficiently on real-time embedded devices is difficult because it is not just about reducing their size, it rather needs a deeper understanding of how they're structured and how their components interact during execution.

A. Overview of CNNs

We may think of CNNs as machines that learn to “see”. They process images, with grid-like topology, by passing them through different mathematical functions called layers as illustrated in Fig. 1.

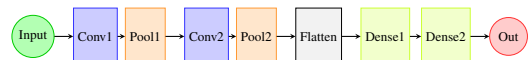


Fig. 1: Global visualization of CNN layers

These layers, known as convolutional and pooling layers, work together to detect essential patterns in images, such as edges, textures, shapes, etc. Starting from simple features and

building up to more complex ones helping the model focus on the most important parts of the image.

After this feature extraction stage, the processed data is flattened and forwarded to dense layers which are like decision-makers that will analyze the information and predict what the image contains. But before a CNN can make reliable predictions, it needs to be trained. This involves showing it thousands, sometimes millions, of labeled images so that it can learn the relationships between visual patterns and their corresponding labels. Once training is complete, the CNN becomes a powerful tool capable of analyzing new, unseen images. It processes the input through its learned layers and selects the most probable label as an output.

B. CNN Architecture

A CNN starts, most of the cases, by taking in an input in the form of a 3 dimensional tensor, which is an image with dimensions (H, W, D) , where H is the height, W the width, and D the dimension or the number of channels, usually three color channels (Red, Green, and Blue).

Mathematically, a CNN is a parameterized, composite function $f(\cdot; \theta)$ that maps an input tensor $I \in \mathbb{R}^{H \times W \times D}$ to an output O , but they are not limited to just 3D inputs. They can also handle higher-order tensors, which might include additional dimensions such as time (in videos) or spectral channels (in satellite imagery or medical scans).

The main layers of CNNs are presented in the Table I; other layers exist, but we focus on the most common ones [3].

C. From Feed-Forward Neural Network CNNs to DAG Representations

The overall architecture of CNNs maintains a strictly forward flow of data, without any cycles or loops. For this reason, CNNs are often regarded as a specific type of Feed-Forward Neural Network (FFNN) [3]. This feed-forward nature allows them to be naturally represented as Directed Acyclic Graphs (DAGs), where layers and operations form nodes connected by directed edges representing data dependencies [4].

1) *Graph-Based Structure*: Let us define an FFNN as a DAG in the following way:

$$\mathcal{G} = (\mathcal{L}, \mathcal{C}) \quad (1)$$

where:

- $\mathcal{L}$ is the set of all layers (nodes) in the network,
- $\mathcal{C} \subseteq \mathcal{L} \times \mathcal{L}$ is the set of all connections (edges),
- $(\ell_i, \ell_j) \in \mathcal{C}$, i.e., layer ℓ_i sends its output to layer ℓ_j .

2) *Predecessors and Successors*: For each layer $\ell \in \mathcal{L}$:

$$\text{Pred}(\ell) = \{ \ell' \in \mathcal{L} \mid (\ell', \ell) \in \mathcal{C} \} \quad (2)$$

$$\text{Succ}(\ell) = \{ \ell' \in \mathcal{L} \mid (\ell, \ell') \in \mathcal{C} \} \quad (3)$$

The full set of transitive predecessors, denoted $\text{Pred}^*(\ell)$, includes all layers from which data eventually flows into ℓ . Due to the acyclic nature of the graph, no layer can be its own predecessor:

$$\ell \notin \text{Pred}^*(\ell) \quad (4)$$

3) *Layer Categories*: A layer $\ell \in \mathcal{L}$ belongs to one of the following categories:

- **Input layer**: $\text{Pred}(\ell) = \emptyset$
- **Output layer**: $\text{Succ}(\ell) = \emptyset$
- **Hidden layer**: all other layers.

We let $\mathcal{L}_{\text{in}} \subset \mathcal{L}$ be the set of input layers and $\mathcal{L}_{\text{out}} \subset \mathcal{L}$ the set of output layers. Then we should have:

$$\mathcal{L}_{\text{in}} \cap \mathcal{L}_{\text{out}} = \emptyset \quad (5)$$

4) *Layer Computation Model*: Each layer ℓ performs a computation modeled by a function:

$$\phi_\ell : \mathbb{R}^{m_\ell} \rightarrow \mathbb{R}^{n_\ell} \quad (6)$$

where m_ℓ and n_ℓ represent input and output dimensions, respectively.

The function of each layer depends on its predecessors:

$$\Phi_\ell = \phi_\ell(\{\Phi_{\ell'} : \ell' \in \text{Pred}(\ell)\}) \quad (7)$$

The overall network function is:

$$\Phi_{\mathcal{N}} = \{\Phi_\ell : \ell \in \mathcal{L}_{\text{out}}\} \quad (8)$$

As an example, we consider a small CNN composed of the following layers:

- One input tensor of size $4 \times 4 \times 1$,
- A convolutional layer of $2 \times 2 \times 1$ filter, producing a $3 \times 3 \times 1$ grid of feature map nodes,
- An average pooling layer of $2 \times 2 \times 1$ pooling window, producing a $2 \times 2 \times 1$ grid of nodes,
- A Flatten operation transforming the pooling grid into a one-dimensional vector,
- A fully connected layer with three output neurons.

We represent the network as a DAG in Fig. 2.

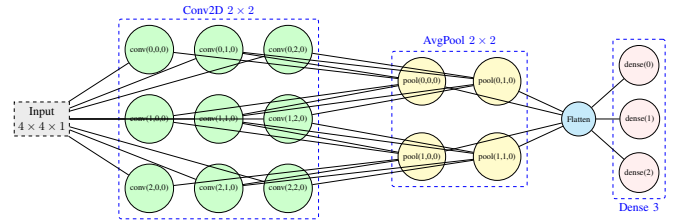


Fig. 2: Example of a CNN Represented as a DAG

Mathematically, we formally describe the DAG network as:

$$\mathcal{L} = \{\ell_{\text{in}}\} \cup \{c_{ij}\}_{i,j=0}^2 \cup \{p_{ij}\}_{i,j=0}^1 \cup \{\ell_{\text{flat}}\} \cup \{d_k\}$$

$$\mathcal{C} = \{(\ell_{\text{in}}, c_{ij})\} \cup \{(c_{ij}, p_{ij})\} \cup \{(p_{ij}, \ell_{\text{flat}})\} \cup \{(\ell_{\text{flat}}, d_k)\}$$

where c_{ij} denotes the convolution output nodes, p_{ij} the pooling output nodes, d_k the dense neurons, also:

- $\mathcal{L}_{\text{in}} = \{\ell_{\text{in}}\}$ and $\mathcal{L}_{\text{out}} = \{d_k\}$,
- Hidden layers: convolution, pooling, and flatten nodes.

This DAG represents the structured data dependencies in the CNN, where each node corresponds to a computational

TABLE I: Main CNN layers

Layer & Role	Notations	Mathematical Definition
Convolution: Local feature extraction using learnable filters.	- Input $I \in \mathbb{R}^{H \times W \times D_{in}}$ - Filters $K \in \mathbb{R}^{k_H \times k_W \times D_{in} \times D_{out}}$ - Output $O \in \mathbb{R}^{H_{out} \times W_{out} \times D_{out}}$	$O(i, j, d) = \sum_{i'=0}^{k_H-1} \sum_{j'=0}^{k_W-1} \sum_{d'=0}^{D_{in}-1} I[i+i', j+j', d'] \cdot K[i', j', d', d]$
Pooling: Reduces spatial resolution while preserving key features.	- Input $I \in \mathbb{R}^{H \times W \times D}$ - Output $O \in \mathbb{R}^{H_p \times W_p \times D}$ - Pooling Window $R_{i,j} \in \mathbb{N}^2$	<i>Max Pooling:</i> $O(i, j, d) = \max_{(m,n) \in R_{i,j}} I(m, n, d)$ <i>Average Pooling:</i> $O(i, j, d) = \frac{1}{ R_{i,j} } \sum_{(m,n) \in R_{i,j}} I(m, n, d)$
Flatten: Converts 3D feature maps into a 1D vector for Dense layers.	- Input $I \in \mathbb{R}^{H_p \times W_p \times D}$ - Output $O \in \mathbb{R}^{H_p \cdot W_p \cdot D}$	$I : \mathbb{R}^{H_p \times W_p \times D} \rightarrow O : \mathbb{R}^{H_p \cdot W_p \cdot D}$
Dense (Fully Connected): Transforms extracted features into output predictions (classification / regression).	- Input $I \in \mathbb{R}^n$ - Weights $W_g \in \mathbb{R}^{m \times n}$ - Bias $b \in \mathbb{R}^m$ - Output $O \in \mathbb{R}^m$	$O = f(W_g I + b)$ with activation functions f as ReLU, Softmax, etc.

operation, and each directed edge denotes the data flow between computations.

The DAG representation is particularly useful for real-time multicore systems deployment because it explicitly captures both data dependencies and execution order. The edges of the DAG can be interpreted as memory traffic, such as feature map transfers, making it possible to estimate memory access latency. Moreover, this representation helps identify parallelizable subgraphs, which is essential for efficient multi-core execution. We will explore real-time multicore platforms in the next section.

III. REAL-TIME EXECUTION COMPLEXITY ON MULTI-CORE EMBEDDED PLATFORMS

Real-time systems aim to respond to external events within a certain predictable period of time. The response process involves event detection, execution of the necessary computations, and delivery of the result within a specified time constraint, *i.e.*, execution deadlines, that may include the completion time alone or the start and completion times of the response. From this, two essential characteristics of real-time systems can be identified [5] :

- They must generate accurate computational results, *i.e.*, functional correctness, while interacting physically with external world;
- These computations must be completed within a specified time frame, *i.e.*, timing correctness.

It is essential to recognize that timing correctness is as crucial as functional correctness. In certain real-time systems, timing correctness may even take priority over functional correctness. However, ensuring timing correctness involves determining the Worst-Case Execution Time (WCET) of a real-time task, *i.e.*, its execution time when considered in isolation. This value is then used to compute the Worst-Case Response Time (WCRT) of each task, which includes additional delays such as conflicts due to shared resource access and interference from concurrent tasks.

A. Importance of Multi-Core in Real-Time Embedded Systems

The shift toward parallel processing has become essential, especially for handling complex and time-critical applications.

Parallel processing involves breaking down a large computational problem into smaller, independent subtasks that can be executed simultaneously across multiple cores. Fig. 3 illustrates how things work: a main task is divided into four independent subtasks, each assigned to a separate core for execution. This strategy aims to reduce execution time and improve computational efficiency.

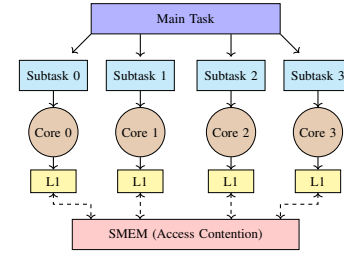


Fig. 3: Parallel execution of subtasks on a multi-core architecture

B. Common Arbitration Mechanisms in Multi-core Systems

By working with the concept of parallelism, the traditional WCET estimation methods, previously used for single-core systems, fall short in multi-core environments. This is mainly due to shared resource conflicts, such as concurrent access to Shared Memory (SMEM), as shown in Fig. 3. In multi-core processors, multiple cores may request access to the same shared caches or SMEM block simultaneously. These requests are managed by arbitration mechanisms, such as Time Division Multiple Access (TDMA) and Round Robin (RR) as shown in Table II. These policies define how and when each core gains access, and they significantly influence interference delays and, consequently, the WCET analysis.

TABLE II: Some arbitration policies for SMEM access

Policy	Advantages	Drawbacks
TDMA	• Deterministic access • High predictability for WCET • Suited for hard real-time systems	• Inefficient if cores are idle • Reserved slots go unused
RR	• Better average utilization • Fair allocation among active cores	• Less predictable WCRT • Interference varies significantly

C. Memory Hierarchy and Interference

Analyzing memory interference on modern multicore platforms is challenging because of deep hierarchies (private/shared caches, Scratch-Pad Memories (SPMs), etc.) combined with arbitration policies that vendors rarely make public. Caches boost performance but complicate WCET/WCRT estimation due to state-dependent behavior and unpredictable replacement policies, *e.g.*, FIFO, whereas SPMs provide software-managed storage, *i.e.*, the compiler or programmer explicitly decides what data is placed in the SPM. Unlike caches, this explicit placement avoids replacement uncertainty and guarantees deterministic constant-time accesses, which makes SPMs highly suitable for real-time predictability [6]–[9]. When caches are unavoidable, conservative static cache analysis can yield safe over-approximations, while SPMs enable tighter bounds through explicit data placement. At the memory-controller level, cores are served in turn by predictable arbitration mechanisms such as TDMA to ensure bounded latency and guaranteed bandwidth, thereby improving analyzability [10], [11].

D. Concept of Multicore Response Time Analysis (MRTA)

MRTA concept, originally proposed by Altmeyer et al. [12], aims to compute the Worst-Case Response Time (WCRT) of each task or subtask by considering all potential sources of execution-time interference.

Within the MRTA Framework, later introduced by Rihani et al. [13], tasks are described as collections of possible execution traces, essentially different paths the task might take during runtime. Each trace is made up of a sequence of instructions, and some of those instructions may access specific memory locations. By analyzing all these traces, the processing and memory costs a task might need can be estimated, focusing on the worst-case demand across all possible paths.

What makes this approach interesting is that it moves away from the traditional method of using fixed, context-independent WCET values. Instead, it directly calculates response times based on actual resource usage at the hardware level. In this case, the hardware is the Kalray MPPA2-256 architecture, the target platform for the analysis.

Building on this idea, a more scalable solution was proposed by de Dinechin et al. [14], who introduced a new method using an iterative time-triggered approach to more accurately estimate interference by identifying the specific time intervals during which memory accesses occur for each core. This new approach reduces the complexity level of the MRTA Framework from $\mathcal{O}(n^4)$ to $\mathcal{O}(n^2)$, allowing the proposed algorithm to be executed efficiently on large task graphs.

To compute the interference delays and the total WCRT of a task graph (DAG), we consider several factors: the WCET of each task in isolation, the mapping of tasks to cores, Worst Case Memory Access (WCMA), the bus arbitration scheme, and the release dates which are defined as the earliest time a task can start once all its inputs, *i.e.*, predecessor tasks in the DAG, are available.

We consider a DAG task set from n_0 to n_4 with predefined release dates. All time values are given in clock cycles as shown in Fig. 4.

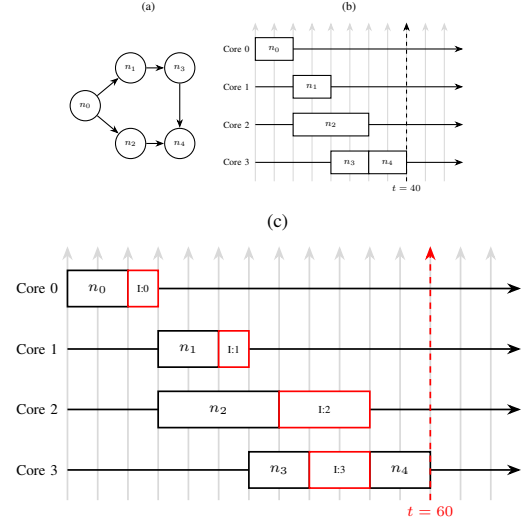


Fig. 4: (a) DAG task set, (b) Initial schedule of the DAG without considering interference, (c) Adjusted schedule of the DAG with interference impact

We observe that panel 4a is our DAG example. Tasks are triggered only after all their dependencies *i.e.*, predecessor tasks, have completed execution. For instance, task n_1 depends on data from task n_0 , meaning that n_1 cannot begin until n_0 has finished and its output is accessible. The execution model further assumes static, non-preemptive scheduling: only one task can execute on a core at any given moment, and once a task is released *i.e.*, ready to run, it begins execution immediately without interruption.

Panel 4b shows the idealized static schedule obtained from isolated WCETs, resulting in a makespan of $t = 40$.

Panel 4c illustrates the effects of shared-resource contention on the DAG schedule. After considering interference, additional stall intervals $I_0 - I_3$ appear (red boxes). Specifically, I_0 arises when n_0 competes for shared memory bandwidth with subsequent tasks; I_1 occurs as n_1 and n_2 overlap in their memory accesses; I_2 and I_3 are introduced when both producers n_2 and n_3 simultaneously attempt to write results needed by consumer n_4 . These serialized memory transactions extend the overall completion time from $t = 40$ to $t = 60$. Thus, we conclude, *parallel execution* $\Rightarrow$ *interference* $\Rightarrow$ *slowdown*.

The central challenge is not merely observing the slowdown, but *deriving sound and reasonably tight upper bounds* on these interference delays for an entire DAG under a given mapping and arbitration policy. Our analysis framework addresses this by annotating nodes with WCET, edges with WCMA/transfer costs, then applying the updated MRTA concept through MIA tool (discussed in the next subsection) to determine pessimistic interference windows consistent with the chosen memory

arbitration model in Kalray architectures, thereby ensuring that the resulting WCRT remains predictable.

The main advantage of this deterministic scheduling scheme is that it minimizes runtime overhead by relying on static, table-driven execution rather than dynamic runtime engines. For this reason, we avoid RTOS (Real-Time Operating Systems) environments, since even under static schedules execution may still be perturbed by Interrupt Service Routines (ISRs), hardware Interrupt ReQuests (IRQs), non-exclusive use of cores, or coarse preemption mechanisms. This choice is primarily motivated by predictability concerns, as static scheduling enables analysis at the granularity of tasks and communication events [15]. In this article, we abstract from such effects and assume a dedicated, contention-free execution model, focusing instead on worst-case timing behavior induced by memory and communication interference.

E. Framework for Time Predictability in Real-Time Systems on Many-Core Architectures

Ensuring time predictability is a critical challenge when designing a real-time system on multi/many-core architectures. Existing tools such as OTAWA and MIA provide essential support in analyzing timing uncertainties, and we summarize these two tools in Table III.

TABLE III: Comparison of OTAWA and MIA tools

Tool	Focus	Key Features
OTAWA (Open ToolBox for Adaptive WCET Analysis) [16]	WCET estimation of single tasks on single-core platforms	• Modular static-analysis framework developed by TRACES at IRT, Univ. of Toulouse • Constructs Control Flow Graphs (CFGs) from binary codes • Detects loops and models hardware features (caches, pipelines) • Provides tight WCET bounds across multiple architectures
MIA (Multi-core Interference Analysis) [17]	Interference-aware WCRT analysis on multi-core platforms	• Developed by VERIMAG (formerly MRTA framework) • Estimates contention on shared resources (buses, memory) • Operates on DAGs with an initial task mapping • Computes updated release dates and WCRT through interference quantification • Supports schedulability verification under deadlines

In this following section, we will review analytical methods that have been formally developed to evaluate CNNs in real-time contexts. These approaches are designed to provide timing guarantees by taking into account the structural characteristics of CNNs.

IV. EXISTING ANALYTICAL METHODS FOR CNNs IN EMBEDDED REAL-TIME CONTEXTS

The first major challenge in analyzing CNNs for embedded real-time systems lies in the preservation of *computational semantics* across tools and platforms. A CNN trained in a high-level framework such as TensorFlow or PyTorch may behave differently when deployed on hardware, due to differences in how mathematical operations are implemented. For example, convolution operations can vary in terms of padding strategy, matrix traversal order, and even rounding behavior. These subtle differences, often negligible in general applications, can significantly affect predictability and correctness in real-time systems. Also, semantic compatibility remains a difficult

problem, especially for frameworks that convert models into low-level code for embedded targets.

Another significant challenge is *timing predictability*, i.e., the ability to estimate the WCET of a CNN model running on a multi-core platform. CNN inference engines typically rely on dynamic memory allocation, making them unsuitable for real-time domains. These engines are optimized for high accuracy and fast throughput rather than timing determinism.

1) *Model Representation Formats*: Despite the growing adoption of CNNs across embedded systems, there is still no standardized or universally accepted model format within the machine learning community. Most popular Machine Learning (ML) frameworks such as TensorFlow and PyTorch rely on custom, opaque (black-box) formats, often based on Google's Protocol Buffers. These formats are binary, not human-readable, and require external tooling for model parsing and interpretation. For example, TensorFlow/Keras supports the .h5 format based on HDF5 serialization standard that allows models to be saved with architecture, weights, and training configuration intact. In contrast, the transition from Caffe to Caffe2 (parts of PyTorch), earlier models became unsupported without proper migration tools [18], highlighting the lack of a consistent formal specification for syntax reusability.

We should also keep in mind that popular execution environments rely on dynamic interpreters, device drivers, and memory managers that allocate and free buffers at runtime. For instance, PyTorch uses a caching GPU memory allocator to speed up allocations and deallocations, and TensorFlow's profiling guides emphasize host/device scheduling, kernel launches, and pipeline effects that are workload and platform dependent. While effective for throughput on servers, these dynamic mechanisms hinder tight WCET and WCRT guarantees on embedded platforms.

A practical mitigation on microcontrollers and bare-metal targets is to remove dynamic allocation at inference time and operate from a statically provisioned memory arena. TensorFlow Lite Micro (TFLM) exemplifies this design: all tensors and scratch buffers are placed in a single, application-provided tensor arena, with no malloc/free during inference. This design trades flexibility for predictability and is compatible with static, table-driven scheduling.

A complementary direction is to remove the heavy-weight framework at runtime via Ahead-Of-Time (AOT) compilation and code generation. Compiler stacks such as TVM/MicroTVM [19] generate C code and statically allocate buffers. Intermediate Representation Execution Environment (IREE) similarly compiles AOT models into target-specific executables with explicit scheduling logic, scaling down to small footprints [20]. These approaches substantially reduce runtime variability (allocators, interpreters, driver jitter), thereby tightening WCET/WCRT bounds.

2) *Frameworks and Toolchains*: Some approaches for ML applications, e.g., ACETONE framework [21], attempt to generate, while preserving model semantics, equivalent low-level C code from the trained FFNN for single core target devices. This transformation enables static analysis tools to

be applied for WCET estimation as execution paths become analyzable and memory access patterns more explicit, *i.e.*, semantic traceability. However, even with such transformations, challenges persist due to irregular multiple structures of CNN workloads and varying layer complexities.

To address part of this challenge, a proposition of extending ACETONE framework has been introduced recently to optimize convolution routines based on General Matrix-Matrix Multiplication (GEMM) to improve execution efficiency [22].

Nevertheless, we should note that current WCET analysis tools are often designed for sequential code only and do not scale to highly parallel compute-intensive tasks such as CNN inference. As a result, evaluating execution time under multi-core interference remains an open and active research area.

V. DISCUSSION AND RESEARCH PERSPECTIVE

This survey is part of our ongoing research on improving temporal analysis of CNN inference on multi-/many-core systems. Our objective is to automatically extract the inference structure of a CNN and represent it as a DAG compatible with the MIA tool for real-time analysis, building on the PhD work of Iryna De Albuquerque [23]. To this end, we are developing MIND (Model Inference Network Designer), a Java-based framework that performs this CNN-to-DAG translation. The high-level project pipeline is shown in Fig. 5.



Fig. 5: End-to-end tool-chain of our ongoing project

Given a CNN model description in a `.py` or `.json` file as input, MIND extracts the model topology: its layers, connections, etc., without including weight values. The resulting DAG is then exported in `.yaml` format to ensure compatibility with the MIA tool, which calculates WCRT value.

To construct the DAG, MIND applies a systematic parsing process. Each CNN layer is analyzed to extract its information, which is then converted into nodes representing atomic computational tasks. Directed edges are added to preserve data dependencies between layers and to quantify transfer volumes. This ensures that the resulting structure is directed, acyclic, and analyzable by the MIA tool.

After DAG construction, tasks are assigned to cores using a deterministic allocation strategy, distributing them in an RR manner. In this prototype, no priority scheme is applied, since all tasks are considered equally important. This mapping step is essential for timing analysis, as it fixes the execution order and avoids non-deterministic effects.

Regarding the chosen architecture, we covered both the Kalray Bostan MPPA2-256 and Coolidge MPPA3-80. We calculated WCET, WCMA, and data-transfer costs based on CNN operations and on the cycle costs of arithmetic operations reported in Kalray’s technical documentation, considering FP32/INT32, FP16/INT16, and INT8 precisions (with FP32 being the upper bound supported by Kalray) [24], [25]. The

cycles are scaled to the frequencies of the Bostan and Coolidge architectures (in normal mode, excluding turbo mode).

Architectural parameters such as inter-cluster latency, SMEM size and bus width directly influence DAG annotation and are all taken into account in the MIND software. Inter- and intra-cluster transfers are explicitly modeled by adding latency only for inter-cluster communications. The SMEM size of each cluster is also considered in DDR recharges, since all parameters and hyperparameters are assumed to reside initially in DDR; their transfer into the SRAM, *i.e.*, SMEM, of each cluster is constrained by the available memory capacity of Bostan and Coolidge (2 MB per cluster in Bostan for 16 clusters, and 4 MB per cluster in Coolidge for 5 clusters). Bus-width assumptions are also integrated, since precise values for transfer buses are not available in Kalray technical documentation: we assume 16 bytes/cycle for Coolidge and 8 bytes/cycle for Bostan, while also neglecting caches for simplicity.

Finally, for Coolidge, the presence of the Tensor Co-Processor Accelerator (TCA) is taken into consideration, as it accelerates arithmetic computations. In this way, the generated DAG is not only structural but also architecture-aware.

Beyond architecture-specific considerations, MIND supports CNNs composed of Convolution and Pooling Layer (CPL) zones as well as Multi-Layer Perceptron (MLP) zones consisting of Flatten and Dense layers, under the restriction that the CNNs are FFNNs. Networks that are not FFNNs, such as RNNs, LSTMs, or Transformers, are not supported, since their recurrent or attention-based dependencies conflict with the acyclicity requirement of DAG semantics. Lightweight TinyML models are also supported, if they operate as FFNNs. We only consider already trained models: any CNN that behaves as an FFNN during the execution phase can be parsed by MIND and represented as a DAG for timing analysis.

It should be noted that our current MIND software can process a wide range of CNNs, including real benchmarks such as LeNet, VGG, and ResNet, but only when composed of the supported layers presented earlier in this article. The first version MINDv1 does not yet support more recent or exotic layers. We can also construct DAGs for large and deep CNNs; however, the bottleneck arises not in the CNN-to-DAG conversion within MIND but in the downstream WCRT computation within MIA. As networks grow deeper and heavier, the DAG size (the number of nodes and edges) increases significantly, inflating the interference calculation and causing performance issues in the MIA tool. In our experiments, the single-threaded MIA program implementation required several days (at least ~ 5 days for VGG16, for example) and very large RAM resources to analyze a full heavyweight CNN inference. Finer-grain DAG representations further enlarge the graph and increase the waiting time for MIA to produce a final WCRT result. As perspectives, scalability on the MIA side could be improved by parallelizing interference iterations or by redeveloping the tool in a compiled language such as C language to accelerate execution.

Fig. 6 summarizes the CNN-to-DAG workflow in MIND as discussed above.

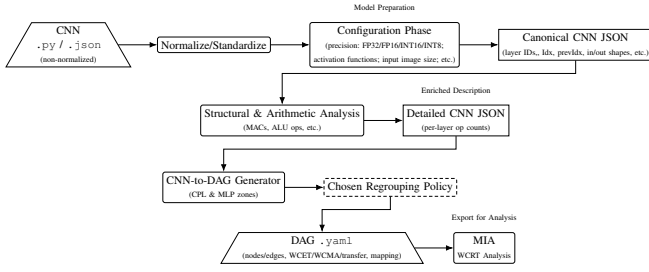


Fig. 6: Detailed MIND pipeline for CNN-to-DAG conversion

As shown in the figure, MIND first normalizes raw .py/.json descriptions and validates the graph, then applies user configurations (data precision, activations, input size, etc.) to produce a canonical JSON with explicit layer identifiers and indexes, predecessor references, and computed input/output shapes for all layers. A structural/arithmetic pass then derives per-layer costs (MACs, ALU operations, etc.). The CNN is subsequently transformed into a DAG partitioned into CPL and MLP zones, after which the user selects regrouping policies that control node granularity and edge structure. Finally, the architecture-annotated DAG (including WCET, WCMA, transfer, and mapping fields) is exported in .yaml format, ready to be processed by the MIA tool for WCRT analysis.

VI. CONCLUSION AND FUTURE DIRECTIONS

This paper reviewed key challenges in deploying CNNs on real-time multi/many-core embedded systems and introduced our ongoing work on MIND, an automated CNN-to-DAG extractor integrated with the MIA timing-analysis tool. At this stage, we have not yet validated on real Kalray hardware as direct access to these platforms was not available during our study; this will be addressed in future work.

While Bostan and Coolidge are the primary targets, the MIND framework was designed with portability in mind. By updating its backbone program, MIND could be adapted to other platforms. However, this extension is conditional on the availability of detailed architectural specifications. GPUs, for example, expose massively parallel compute units and the majority of their detailed architectural specifications are not publicly available, making the extension of MIND to GPUs potentially unrealistic. FPGAs and ASICs, by contrast, are not core-based architectures; they rely on configurable logic or dedicated accelerators rather than general-purpose processing elements. As a result, extending MIND to these platforms would require reformulating mapping and cost modeling steps, which lie outside the scope of this first version.

Beyond portability, another perspective is the adaptation of MIND to heterogeneous hardware. Many modern processors combine different classes of cores, e.g., high-performance accelerators, and introduce multi-level memory hierarchies, e.g., caches. In such cases, DAG nodes could be annotated with core or memory affinity, ensuring that computational tasks are mapped to the most suitable processing units and that data transfers reflect the correct hierarchy. MIND already accounts

for a form of heterogeneity through the integration of TCA co-processor into each core of the Coolidge architecture. However, extending MIND to support more complex heterogeneous platforms remains future work for its later versions.

The topic we address is inherently complex, combining neural-network structure, multi-core interference, and timing verification. This paper should be seen as a first step, with MIND providing a novel starting point for systematic, architecture-aware DAG extraction for real-time analysis.

REFERENCES

- [1] Y. LeCun *et al.*, “Gradient-based learning applied to document recognition,” *Proc. of the IEEE*, 1998.
- [2] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proc. of the IEEE conference on CVPR*, 2016.
- [3] I. Goodfellow *et al.*, *Deep Learning*. MIT Press, 2016.
- [4] A. Gauffriau, I. D. A. Silva, and C. Pagetti, “Formal description of ml models for unambiguous implementation,” *arXiv preprint*, 2023.
- [5] Q. Li and C. Yao, *Real-time concepts for embedded systems*. CRC press, 2003.
- [6] J. Reineke *et al.*, “Timing predictability of cache replacement policies,” *Real-Time Systems*, vol. 37, no. 2, pp. 99–122, 2007.
- [7] M. Lv *et al.*, “A survey on static cache analysis for real-time systems,” *LITES*, vol. 3, no. 1, pp. 05:1–05:48, 2016.
- [8] R. Banakar *et al.*, “Scratchpad memory: A design alternative for cache on-chip memory in embedded systems,” *Proc. of the CODES*, 2002.
- [9] Y. L. Liu *et al.*, “Scratchpad memory architectures and allocation algorithms for real-time and low-power embedded systems: A survey,” *Journal of CSE*, vol. 9, no. 2, pp. 81–97, 2015.
- [10] B. A. Kesson, K. Goossens, and M. Ringhofer, “Predator: A predictable sdram memory controller,” *Proc. of CODES+ISSS*, pp. 251–256, 2007.
- [11] J. Reineke *et al.*, “Pret dram controller: Bank privatization for predictability and temporal isolation,” *Proc. of the IEEE RTAS*, 2011.
- [12] S. Altmeyer, R. I. Davis, L. Indrusiak, C. Maiza, V. Nelis, and J. Reineke, “A generic and compositional framework for multicore response time analysis,” in *Proc. of the 23rd International Conference on RTNS*, 2015.
- [13] H. Rihani, M. Moy, C. Maiza, R. I. Davis, and S. Altmeyer, “Response time analysis of synchronous data flow programs on a many-core processor,” in *Proc. of the 24th International Conference on RTNS*, 2016.
- [14] M. D. de Dinechin, M. Schuh, M. Moy, and C. Maiza, “Scaling up the memory interference analysis for hard real-time many-core systems,” in *DATE Conference & Exhibition*. IEEE, 2020.
- [15] I. D. A. Silva, T. Carle, A. Gauffriau, and C. Pagetti, “Automatic predictable c code generation of machine learning models for avionics systems,” *ETR*, vol. 20, p. 5, 2021.
- [16] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, “OTAWA: An open toolbox for adaptive wcet analysis,” in *IFIP International Workshop on STEUS*. Springer, 2010.
- [17] M. Schuh, “Implantation sûre d’applications temps-réel critiques sur plateforme pluri-coeur,” Ph.D. dissertation, Univ. Grenoble Alpes, 2022.
- [18] Y. Jia *et al.*, “Caffe: Convolutional architecture for fast feature embedding,” *arXiv:1408.5093*, 2014.
- [19] C. Liu *et al.*, “Deploying machine learning models to ahead-of-time compiled c runtimes with microtvm,” *arXiv:2304.04842*, 2023.
- [20] H. I.-C. Liu *et al.*, “Tinyiree: An ml execution environment for small footprints,” *arXiv:2205.14479*, 2022.
- [21] I. D. A. Silva, T. Carle, A. Gauffriau, and C. Pagetti, “Acetone: predictable programming framework for ml applications in safety-critical systems,” in *24th ECRTS*, 2022.
- [22] C. Daini, G. Lipari, H. Zahaf, and P.-E. Hladik, “Optimizing CNN Inference on Multicore Scratchpad Architectures,” in *Proceedings IEEE ISORC 2025*, Toulouse, France, May 2025.
- [23] 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.
- [24] B. Dupont de Dinechin, “Kalray mppa: Engineering a manycore processor platform for mission-critical applications,” Keynote slides at MCSoc 2016, Kalray SA, 2016, cTO, Kalray.
- [25] —, “Securing the kalray mppa manycore architecture,” Presentation slides, Kalray SA, Dec. 2018, cTO, Kalray.