

Deterministic Superscalar AI Hardware Accelerator for Dependable Embedded Real-Time Systems

DISSERTATION

zur Erlangung des Grades eines Doktors
der Ingenieurwissenschaften

vorgelegt von

M.Sc. Yosab Bebawy



eingereicht bei der Naturwissenschaftlich-Technischen Fakultät
der Universität Siegen

Siegen 2026

Betreuer und erster Gutachter
Prof. Dr.-Ing. Roman Obermaisser
Universität Siegen

Zweiter Gutachter
Prof. Dr.-Ing. Bing Li
Technische Universität Ilmenau

Tag der mündlichen Prüfung
06.10.2026

Declaration of Authorship

I, M. Sc. Yosab BEBAWY, declare that this thesis titled, “Deterministic Superscalar AI Hardware Accelerator for Dependable Embedded Real-Time Systems” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

“Every day is a challenge, and every great success is built on small successes achieved each day. In the end, it’s the small successes that truly count.”

Yosab Bebawy

UNIVERSITÄT SIEGEN

Abstract

Fakultät IV
Chair for Embedded Systems

Doctor of Engineering

**Deterministic Superscalar AI Hardware Accelerator for Dependable Embedded
Real-Time Systems**

by M. Sc. Yosab BEBAWY

0.1 English Version

Safety-critical real-time systems that rely on AI-based object classification and detection for critical decision-making require strict guarantees of reliability and timing correctness, as even minor deviations in behavior can lead to catastrophic consequences. Integrating AI hardware accelerators into such real-time environments therefore demands not only high computational throughput but also predictable and analyzable execution behavior. Conventional accelerators, such as the Versatile Tensor Accelerator (VTA), meet performance requirements but fall short in predictability due to their event-driven execution model and limited timing control. This variability introduces significant risks in scenarios where timely and deterministic inference is essential. To address these limitations, this work adopts a time-triggered execution paradigm for AI acceleration. It first introduces the Time-Triggered VTA (TT-VTA), which replaces the non-deterministic behavior of the original VTA with a deterministic counterpart based on statically defined, pattern-based timing schedules. These schedules are generated offline at design time using a scheduler with a built-in simulator for the VTA architecture. The scheduler further integrates additional simulation frameworks: DRAMSim2 is used to model memory-related instructions, while a cycle-accurate simulator captures compute-bound operations. Together, these components enable precise timing analysis, deterministic resource utilization, and predictable execution behavior. Evaluation with a CNN classification model shows that the TT-VTA preserves classification accuracy while ensuring predictable execution, however with a slight increase in inference latency (53 ms compared to 51 ms for conventional VTA), indicating room for further optimization. Building on this foundation, the Superscalar Time-Triggered VTA (STT-VTA) is proposed to improve performance without sacrificing determinism. STT-VTA extends the architecture by exploiting parallelism through multiple coordinated accelerator functional-units, whose configuration is determined at compile time via an enhanced architecture management and scheduling framework. This framework analyzes the computational structure needed for the target AI model and generates optimized schedules that maintain strict timing guarantees while increasing throughput. Experimental results demonstrate that the STT-VTA achieves the same classification accuracy as both TT-VTA and conventional VTA, while reducing inference latency by 17-41% compared to the TT-VTA. With the STT-VTA, another challenge arose in managing memory-intensive workloads, which introduce performance bottlenecks. To address this, the work further incorporates deterministic caching and pre-fetching strategies, leading to the Superscalar Time-Triggered VTA with Caching and Pre-fetching (STA-CP). Unlike conventional dynamic approaches, these mechanisms are integrated directly into the static scheduling process. At compile time, all tensor operations and memory transactions are fully specified, enabling predictable data movement and efficient overlap between computation and memory communication. By eliminating redundant memory accesses and hiding memory latency within a fully analyzable execution model, STA-CP achieves improved throughput while preserving strict temporal determinism. Experimental evaluation on different CNN classification models demonstrates up to a 30% reduction in deterministic inference latency compared to TT-VTA. Overall, the proposed STA-CP architecture demonstrates that the integration of time-triggered execution, controlled parallelism, and deterministic memory-management techniques can simultaneously achieve high performance and predictable execution behavior. These characteristics are fundamental for deploying AI accelerators in safety-critical real-time systems, where both computational correctness and strict timing guarantees are essential.

0.2 German Version

Sicherheitskritische Echtzeitsysteme, die für entscheidungsrelevante Aufgaben auf KI-basierte Objekterkennung und -klassifikation angewiesen sind, erfordern strikte Garantien hinsichtlich Zuverlässigkeit und zeitlicher Korrektheit, da selbst geringfügige Verhaltensabweichungen katastrophale Folgen haben können. Die Integration von KI-Hardwarebeschleunigern in solche Echtzeitumgebungen verlangt daher nicht nur eine hohe Rechenleistung, sondern auch ein vorhersehbares und analytisch erfassbares Ausführungsverhalten. Konventionelle Beschleuniger, wie der Versatile Tensor Accelerator (VTA), erfüllen zwar die Leistungsanforderungen, weisen jedoch aufgrund ihres ereignisgesteuerten Ausführungsmodells und der begrenzten zeitlichen Kontrollierbarkeit eine eingeschränkte Vorhersagbarkeit auf. Diese Variabilität stellt ein erhebliches Risiko in Szenarien dar, in denen eine zeitgerechte und deterministische Inferenz essenziell ist. Zur Adressierung dieser Limitationen verfolgt diese Arbeit daher ein zeitausgelöstes Ausführungsparadigma für KI-Beschleunigung. Zunächst wird der Time-Triggered VTA (TT-VTA) eingeführt, der das nicht-deterministische Verhalten des ursprünglichen VTA durch eine deterministische Variante ersetzt, die auf statisch definierten, musterbasierten Zeitschemata basiert. Diese Schedules werden offline im Entwurfszeitpunkt mithilfe eines Schedulers mit integriertem Simulator für die VTA-Architektur erzeugt. Der Scheduler integriert zudem zusätzliche Simulationsframeworks: DRAMSim2 wird zur Modellierung speicherbezogener Instruktionen eingesetzt, während ein zyklusgenauer Simulator rechenintensive Operationen erfasst. Zusammengenommen ermöglichen diese Komponenten eine präzise Zeitanalyse, eine deterministische Ressourcennutzung sowie ein vorhersagbares Ausführungsverhalten. Die Evaluation mit einem CNN-Klassifikationsmodell zeigt, dass der TT-VTA die Klassifikationsgenauigkeit beibehält und gleichzeitig ein vorhersagbares Ausführungsverhalten gewährleistet, jedoch mit einer leichten Erhöhung der Inferenzlatenz (53 ms gegenüber 51 ms beim konventionellen VTA), was Optimierungspotenzial aufzeigt. Aufbauend auf dieser Grundlage wird der Superscalar Time-Triggered VTA (STT-VTA) vorgeschlagen, um die Leistung zu steigern, ohne die Deterministik zu beeinträchtigen. Der STT-VTA erweitert die Architektur durch die Ausnutzung von Parallelität mittels mehrerer koordinierter Beschleuniger-Funktionseinheiten, deren Konfiguration zur Compile-Zeit durch ein erweitertes Architektur- und Scheduling-Framework bestimmt wird. Dieses Framework analysiert die für das Ziel-KI-Modell erforderliche Rechenstruktur und erzeugt optimierte Schedules, die strikte Zeitgarantien einhalten und gleichzeitig den Durchsatz erhöhen. Die experimentellen Ergebnisse zeigen, dass der STT-VTA dieselbe Klassifikationsgenauigkeit wie sowohl TT-VTA als auch der konventionelle VTA erreicht, während die Inferenzlatenz im Vergleich zum TT-VTA um 17–41% reduziert wird. Im Kontext des STT-VTA trat jedoch eine weitere Herausforderung bei der Verarbeitung speicherintensiver Workloads auf, die zu Performance-Engpässen führen. Zur Lösung dieses Problems werden deterministische Caching- und Pre-fetching-Strategien integriert, was zum Superscalar Time-Triggered VTA mit Caching und Pre-fetching (STA-CP) führt. Im Gegensatz zu konventionellen dynamischen Ansätzen sind diese Mechanismen direkt in den statischen Scheduling-Prozess eingebettet. Bereits zur Compile-Zeit werden alle Tensoroperationen und Speichertransaktionen vollständig spezifiziert, wodurch eine vorhersagbare Datenbewegung sowie eine effiziente Überlappung von Rechen- und Speicheroperationen ermöglicht werden. Durch die Eliminierung redundanter Speicherzugriffe und das Verbergen von Speicherlatenzen innerhalb eines vollständig analysierbaren Ausführungsmodells erreicht STA-CP

einen verbesserten Durchsatz bei gleichzeitiger Wahrung strikter zeitlicher Deterministik. Die experimentelle Evaluierung verschiedener CNN-Klassifikationsmodelle zeigt eine Reduktion der deterministischen Inferenzlatenz um bis zu 30% im Vergleich zum TT-VTA. Insgesamt zeigen die vorgeschlagenen STA-CP-Architekturen, dass die Kombination aus zeitausgelöster Ausführung, kontrollierter Parallelität und deterministischen Speicherverwaltungstechniken gleichzeitig hohe Leistung und vorhersagbares Ausführungsverhalten ermöglichen kann. Diese Eigenschaften sind grundlegend für den Einsatz von KI-Beschleunigern in sicherheitskritischen Echtzeitsystemen, in denen sowohl rechnerische Korrektheit als auch strikte Zeitgarantien essenziell sind.

Acknowledgements

I would like to express my sincere gratitude to the Head of the Chair for Embedded Systems, Prof. Dr.-Ing. Roman Obermaisser, for his continuous support, guidance, and insightful feedback throughout the course of this research. His expertise, constructive discussions, and encouragement have been instrumental in the completion of this thesis and have significantly contributed to my academic and professional development. Additionally, I would like to express my sincere gratitude to Prof. Dr.-Ing. Bing Li for his valuable support throughout my research.

I am particularly grateful for the opportunity to conduct this work under their supervision, as well as for the trust, academic freedom, and responsibility they provided during the research process. Their commitment to scientific rigor and excellence has been a constant source of motivation and inspiration.

I would also like to thank all colleagues and members of the chair for the supportive and collaborative working environment, as well as for the valuable discussions and technical assistance throughout this research journey.

Furthermore, I gratefully acknowledge the financial support that made this work possible. This research was supported by the EdgeAI-Trust project, funded within the Chips Joint Undertaking (Chips JU), a public-private partnership under the European Union's Horizon Europe research and innovation program in collaboration with National Authorities, under grant agreement number 101139892. Additional support was provided by the EcoMobility project under Grant 101112306, and by the German Federal Ministry of Education and Research (BMFTR) under Grant agreement 16MEE0316. Further support was received from the Deutsche Forschungsgemeinschaft (DFG) under grant agreement OB 384/13-1.

Contents

Declaration of Authorship	iii
Abstract	vii
0.1 English Version	viii
0.2 German Version	ix
Acknowledgements	xi
1 Introduction	1
1.1 Problem Statement	1
1.2 Objective	3
1.3 Thesis Structure	5
1.4 List of Publications	7
2 Basic Concepts and Literature Review	9
2.1 Requirements of AI Accelerators for Safety-Critical Systems	10
2.2 AI Frameworks and Compilation Ecosystems	11
2.3 AI Accelerator Architectures	12
2.3.1 General-Purpose AI Accelerators	12
2.3.2 Model-Specific FPGA Acceleration	15
2.4 Inference Parallelization and Workload Distribution	15
2.5 Memory Systems and Latency-Hiding Techniques	16
2.6 Time-Triggered and Deterministic Computing Systems	17
2.7 DRAM Timing Simulation and WCET-Oriented Memory Modeling	19
2.8 FPGA Toolchains and Hardware Design Frameworks	21
2.9 Comparative Overview and Research Gap Analysis	21
3 Development Process of Deterministic AI Hardware Accelerator	25
3.1 Step 1: AI Model Development and Training	25
3.2 Step 2: TVM/VTA Compilation and Instruction Generation	27
3.3 Step 3: Behavioral Simulation and Timing Extraction	27
3.4 Step 4: Deterministic Scheduling and Architecture Configuration	28
3.5 Step 5: Architecture Generation and Hardware Adaptation	29
3.6 Step 6: FPGA Synthesis and Hardware Realization	30
3.7 Step 7: Deployment and Runtime Execution	30
4 Time-Triggered VTA (TT-VTA)	31
4.1 System Model of Time-Triggered VTA	31
4.2 Time-Triggered Dispatcher	32
4.3 TT-VTA scheduler	34
4.4 TT-VTA Evaluation	38
4.5 Discussion on TT-VTA	41

5	Superscalar Time-Triggered VTA (STT-VTA)	43
5.1	System Model of Superscalar Time-Triggered VTA	43
5.1.1	Functional Requirements	43
5.1.2	Design Requirements	44
5.2	STT-VTA scheduler	45
5.3	STT-VTA Evaluation	50
5.4	Discussion on STT-VTA	53
6	Deterministic Caching and Pre-fetching for STT-VTA (STA-CP)	55
6.1	System Model of STA-CP	55
6.1.1	Functional Modules and Dispatching Semantics	56
6.1.2	Cache-Aware Memory Access Semantics	57
6.1.3	Scheduling Semantics	58
6.1.4	Synthesis and Execution Semantics	58
6.2	STA-CP Scheduler	59
6.2.1	Stage 1 — Data Dependency Analysis and Layer-Based Cache Management	60
6.2.2	Stage 2 - Scheduling	63
6.3	STA-CP Evaluation	69
6.3.1	Experimental Setup	69
6.3.2	Tested Models and STA-CP Configurations	69
6.3.3	Measurement Methodology	71
6.3.4	Resource Utilization	72
6.3.5	Inference Latency Results	73
6.4	STA-CP Evaluation Beyond Resource-Constrained Configurations . . .	74
6.5	Discussion on STA-CP	76
7	Timing Analysis of STA-CP	79
7.1	Overview and Motivation	79
7.2	Formal Definition of WCET	79
7.3	Instruction-Level Timing Analysis	80
7.4	Module-Level WCET Composition	81
7.5	Impact of Deterministic Scheduling	81
7.6	Memory System Contribution to WCET	82
7.6.1	Deterministic Cache Behavior	82
7.6.2	Static Pre-fetching	83
7.6.3	Contention-Free Memory Access	83
7.7	Effect of Superscalar Scaling on WCET	83
7.8	Buffering and WCET Stability	84
7.9	End-to-End WCET Expression	84
7.10	Discussion	84
8	WCET Validation of the STA-CP	87
8.1	Validation Objectives	87
8.2	Relation to the Timing Analysis Model	87
8.3	Experimental Validation Setup	88
8.4	WCET Accuracy Validation	88
8.5	Determinism Validation	89
8.6	Composability Validation	91
8.7	System-Level Latency Considerations	91
8.8	Discussion	91

9	Discussion and Conclusion	93
10	Outlook and Future Research Directions	97
A	WCET Analysis and Evaluation of DRAMSim2	101
A.1	Overview	101
A.2	DRAM Memory System Model	101
A.3	Worst-Case Memory Latency Estimation	102
A.4	WCET Estimation Methodology	102
A.5	Interpretation of DRAMSim2 Timing Traces	103
A.6	Integration with the Proposed Scheduling Framework	103
A.7	Limitations	104

List of Figures

1.1	Thesis Structure	6
2.1	Overview of TVM Compiler Stack [67]	12
2.2	(a) Logged Instructions from TVM, (b) Instructions dependency Graph	13
2.3	VTA Instruction Patterns [20]	13
2.4	HW Architecture of the "Versatile Tensor Accelerator (VTA) [19, 20]" .	14
3.1	Development Toolchain for the Proposed Time-Triggered AI Accelerators	26
4.1	TT-VTA Architecture	32
4.2	Scheduling Pattern	33
4.3	Time-Triggered Dispatcher Architecture	33
4.4	Scheduler with built-in simulation "TT-VTA Scheduler"	34
4.5	Test Cases with Different Interference Frequencies & Data-Sizes	40
4.6	Resource utilization of VTA and TT-VTA	40
5.1	Superscalar Time-triggered VTA (STT-VTA) Architecture	44
5.2	VTA Instruction Patterns including Metadata	44
5.3	Xilinx Tool Framework	45
5.4	Scheduler with built-in simulation "STT-VTA Scheduler"	46
5.5	Resource utilization of STT-VTA with different number of Compute Units	51
5.6	100 inference executions under DRAM interference (16 kHz interfer- ence frequency and 1024-byte data size)	52
6.1	Superscalar Time-Triggered Versatile Tensor Accelerator with Cache and Pre-fetching (STA-CP) Architecture	56
6.2	STT-VTA & STA-CP: Instruction Patterns including Metadata	57
6.3	Scheduler with built-in simulation "STA-CP Scheduler"	59
6.4	Data Dependency Analysis for Caching	60
6.5	Cache Utilization	70
6.6	Resources Utilization of STA-CP Configurations	73
6.7	ResNet-101 Inference Latency under Different STA-C and STA-CP Configurations	75

List of Tables

2.1	Comparison of DRAM Memory Simulation Frameworks	20
2.2	Comparative Overview of SotA- and Proposed-Acceleration Architectures	24
4.1	Conventional VTA & TT-VTA Test Cases	38
5.1	Models tested (without DRAM interference)	50
6.1	Different STA-CP configurations	70
6.2	$T_{Overhead}$ w.r.t Tested Models	71
6.3	Inference latency ($T_{Inference}$) for tested models across STA-CP configurations	73
8.1	Determinism Validation of STA-CP Compared to Conventional VTA . .	89
9.1	Conclusion Matrix: Positioning of STA-CP Against Existing Approaches	96

List of Abbreviations

AI	Artificial Intelligence
FPGA	Field Programmable Gate Array
VTA	Versatile Tensor Accelerator
TT	Time-Triggered
TT-VTA	Time-Triggered Versatile Tensor Accelerator
STT-VTA or STA	Superscalar Time-Triggered Versatile Tensor Accelerator
STA-C	Superscalar Time-Triggered VTA with Caching
STA-CP	Superscalar Time-Triggered VTA with Caching and Pre-fetching
CNN	Convolution Neural Network
DNN	Deep Neural Network
ML	Machine Learning
XAI	Explainable AI
WCET	Worst Case Execution Time
DRAM	Dynamic Random Access Memory
BRAM	Block Random Access Memory
LUT	Look-Up Tables
FF	Flip-Flop
DSP	Digital Signal Processing
TVM	Tensor Virtual Machine
YOLO	You Only Look Once
OPCODE	Operational Code
PID	Pending Instruction Detector
IP	Intellectual Property
ID	Identifier
CPU	Central Processing Unit
GPU	Graphical Processing Unit
LD	Load Unit
CMP	Compute Unit
ST	Store Unit

This work is lovingly dedicated to my wonderful wife, “Meray,” and to my dear Mom and Dad. Your unwavering presence, love, and support have been my steady foundation through every step of this journey. I am deeply grateful for your patience, encouragement, and belief in me—especially in moments when I needed it most. Thank you for always being there, and for making everything I do feel meaningful.

Chapter 1

Introduction

1.1 Problem Statement

Artificial Intelligence (AI) is increasingly prevalent across various application domains due to its impressive performance in industrial technologies, including autonomous vehicles, aerospace, robotics, and medical applications. The deployment of AI models in such safety-critical systems has rapidly expanded due to their unmatched capabilities in data processing, automation, and decision-making [1]. Leveraging the predictive ability of AI models, particularly Deep Neural Networks (DNNs), these systems have demonstrated significant advantages in identifying patterns, making complex classifications, and managing intricate tasks. However, unlike traditional systems, AI and ML models pose unique challenges in safety-critical applications [2], where predictability, reliability, and transparency are paramount to ensure human safety and regulatory compliance [3, 4].

To address these challenges, a range of technical solutions, methodologies, and standards has been proposed to facilitate the development of reliable and trustworthy AI models. Explainable Artificial Intelligence (XAI) techniques [5], for example, are employed to improve the interpretability and validation of model outputs by providing insight into the underlying decision-making processes. To mitigate vulnerabilities to adversarial inputs, robust training and testing approaches have been developed to enhance model resilience against unexpected or manipulated input and training data [6]. In addition, formal verification methods, including runtime monitoring [7] and model checking [8], are applied to ensure that models comply with predefined safety requirements during operation. Complementary frameworks, such as Safety of the Intended Functionality (SOTIF) [9] and model-based testing, further support the verification and validation of AI systems operating in dynamic real-world environments.

At the same time, industrial safety standards have evolved to address the growing integration of AI and ML technologies, introducing rigorous safety measures tailored to these systems. Standards such as ISO 26262 for automotive functional safety, ISO/PAS 21448 (SOTIF) [9] for managing limitations in AI-based automotive systems, and DO-178C for aviation software assurance have increasingly expanded their scope to incorporate ML-related considerations. These standards establish structured processes for the design, development, verification, and validation of AI and ML components, helping to ensure compliance with stringent safety requirements for deployment in safety-critical applications. Likewise, standards such as IEC 61508 for industrial functional safety and ISO 13485 for medical device quality and reliability define requirements for deterministic, fail-safe operation and comprehensive system validation, thereby supporting the suitability of AI-enabled components for high-assurance environments.

However, ensuring that a trained AI model is inherently trustworthy and achieves high predictive accuracy is not sufficient to guarantee correct and predictable behavior when deployed within safety-critical systems. In practice, reliability must be considered at the system level, where both the model and its execution environment jointly determine overall behavior under operational constraints.

A representative example can be found in autonomous driving systems, where multiple software applications execute concurrently on a shared computing platform [10]. While AI-based perception functions perform object detection, classification, and lane recognition, other safety-critical tasks such as vehicle control, localization, sensor fusion, and communication services simultaneously compete for computational and memory resources. As a result, contention for shared resources, particularly memory bandwidth and communication buses, can introduce significant delays and timing variability in AI inference execution. Such delays may lead to outdated perception information being delivered to the decision-making and control layers, potentially affecting the vehicle's ability to react within the required safety margins. Similar challenges arise in other safety-critical domains, including industrial automation, avionics, and medical systems, where predictable execution behavior and bounded response times are essential.

In this context, the underlying hardware platforms on which AI models are executed introduce additional challenges in achieving the real-time performance, reliability, and determinism required in safety-critical applications. Modern high-performance computing (HPC) systems, for instance, commonly employ heterogeneous architectures comprising CPUs and GPUs to exploit substantial parallelism. This is particularly advantageous for compute-intensive AI workloads such as inference and deep learning training, as it enables efficient processing of large-scale datasets [11]. However, despite these benefits, such platforms are typically energy-intensive and often struggle to satisfy strict real-time requirements due to overheads and latency introduced by dynamic task scheduling, inter-node communication, and distributed memory management [12, 13].

In contrast, embedded hardware platforms increasingly integrate multiple computational resources—including CPUs, AI engines, and AI accelerators—within highly constrained system architectures [4]. Given their limited computational and energy budgets, these systems require extensive sharing of critical resources such as memory subsystems, interconnect buses, and input/output devices [13]. Consequently, resource contention and execution interference become more prevalent [14], leading to non-determinism and variability in inference execution times—an effect that is particularly problematic in safety-critical environments.

Addressing these architectural challenges is therefore essential to improving the reliability and predictability of AI-enabled embedded systems in safety-critical domains. One promising approach is the use of dedicated AI hardware accelerators to support and expedite inference execution on silicon platforms such as FPGAs, alongside general-purpose CPUs. By offloading computationally intensive AI workloads, these accelerators allow CPUs to execute other tasks concurrently, improving overall system efficiency. However, to satisfy real-time constraints, such accelerators must also provide deterministic execution behaviour, including both spatial and temporal isolation capabilities [15], thereby contributing to system-level safety guarantees. This introduces key design requirements for AI hardware accelerators in safety-critical systems:

1. **Spatial requirements:** Hardware accelerators should ensure spatial isolation [15], such that concurrent processes or tasks do not interfere with one

another. This can be achieved through resource partitioning, dedicated memory regions, and isolated execution units. By enforcing spatial separation, interference between tasks is minimised [16], thereby improving system reliability.

2. **Temporal requirements:** Temporal determinism is essential for safety-critical systems [17] that require real-time capabilities. Hardware accelerators should be able to guarantee that tasks are completed within specified time frames. This can be achieved through time-triggered architectures, where tasks are scheduled to execute at precise intervals, and by utilizing real-time operating systems that manage task execution based on priorities and deadlines. Ensuring temporal determinism helps in maintaining the predictability [18] of the system, which is essential for making timely decisions.

Therefore, the following research questions are addressed in this thesis:

- **Q1: How can AI hardware accelerators be designed to provide deterministic and WCET-analyzable inference execution for safety-critical real-time systems?**

This question investigates how time-triggered execution models, static scheduling, and predictable resource management can be used to replace the non-deterministic behaviour commonly found in conventional AI accelerators.

- **Q2: How can the performance of AI hardware accelerators be improved through parallelism without affecting determinism?**

This question explores how architectural design choices such as superscalar parallelism, deterministic memory management, and model-specific specialization influence throughput, resource utilisation, and system adaptability.

- **Q3: How can memory interference be eliminated while hiding memory latency and maintaining high inference performance?**

This question focuses on mitigating timing latencies caused by shared-memory utilisation and investigates how AI hardware accelerator architectures can benefit from static latency-hiding strategies, such as deterministic caching and static pre-fetching.

1.2 Objective

In this research, our primary focus revolves around the exploration and further advancements of an AI hardware accelerator known as the "Versatile Tensor Accelerator" (VTA) [19, 20]. Although this accelerator already enhances the speed of inference operations, it operates in an event-triggered fashion, lacking predictability and determinism during execution, particularly when dealing with shared resources like Dynamic Random Access Memories (DRAMs). A notable challenge arises from potential memory interference when the VTA attempts simultaneous read and write operations on the DRAM, while another API is performing operations on the same memory.

Our objective is to improve the functionalities of the VTA architecture, specifically, by introducing the concept of time-triggered inference, denoted as TT-VTA. This improvement aims to enhance the predictability and determinism of the design while mitigating interference in shared resources. The key element of this improvement is the development and integration of a time-triggered dispatcher (TT-dispatcher) into the VTA architecture. The TT-dispatcher relies on timing schedules to orchestrate the

execution of tensor operations (i.e., instructions) associated with the CNN models (e.g., ResNet and YOLO families).

To derive the timing schedules, we developed a dedicated Scheduler with built-in simulator. This scheduler generates precise design-time execution schedules that are essential for enabling deterministic execution of tensor operations within the TT-VTA architecture at run time. Consequently, the objective of this work is to provide an innovative solution that transforms the VTA into a time-triggered architecture, thereby improving predictability, determinism, and resource management.

Although the proposed TT-VTA already enhances the predictability and determinism of inference operations, its performance can be further optimized by introducing pipelining [21, 22]. We focused on enabling superscalar pipelining for the Versatile Tensor Accelerator (VTA) [19, 20]. Therefore, we aimed to optimize the inference execution process by extending the scheduler with built-in simulator to analyze tensor operations within a given AI model. This extension identifies potential pipelining optimizations and, based on the analysis, determines the number of additional VTA functional units (i.e., VTA modules) required to achieve faster inference than a single (TT-)VTA architecture. This approach enables the selection of an optimized STT-VTA architecture tailored to the analyzed model. Additionally, the scheduler generates timing schedules for the identified STT-VTA modules, supporting a synchronized and coordinated inference execution process at run time. Meanwhile, the STT-VTA adheres to stringent safety requirements for safety-critical real-time systems such as eliminating interference from shared resources to minimize timing variability during inference execution.

However, the STT-VTA architecture still faces inherent limitations related to memory subsystem inefficiency. Since intermediate feature maps and partial results must be repeatedly exchanged with the FPGA's off-chip DRAM, the system's throughput is bounded by the external memory latency. Although the time-triggered model of the STT-VTA eliminates nondeterminism, it also enforces serialized data movement from/to the off-chip memory to preserve isolation, leading to underutilization of its tensor cores (i.e., compute units). Consequently, STT-VTA struggles to scale performance for models with heavy feature-map reuse or high activation bandwidth requirements.

In general-purpose high-performance computing systems, caching and pre-fetching have long been the dominant techniques for hiding memory latency, exploiting spatial and temporal locality, and improving effective bandwidth utilisation. By keeping frequently accessed data closer to the processing units and proactively anticipating future memory accesses, these mechanisms can significantly improve throughput and energy efficiency. However, conventional cache hierarchies typically rely on dynamic policies, such as least-recently-used (LRU) replacement and speculative pre-fetching based on runtime behaviour and predicted memory access patterns. As a result, these mechanisms inherently introduce non-deterministic access latencies and potential interference among concurrent threads or processes. This lack of temporal predictability makes conventional caching approaches unsuitable for real-time execution models that require formally verifiable timing guarantees [23, 24, 25].

This work challenges the traditional view that caching and pre-fetching must inherently be non-deterministic. We argue that when these techniques are reimaged within a time-triggered execution framework, they can retain their performance benefits while achieving complete predictability. The key insight is that if every cache access, replacement, and pre-fetch operation is scheduled at compile time—rather than dynamically decided at runtime—then the utilization and the behavior of the entire memory subsystem can become analyzable and deterministic.

To realize this concept, we propose the Superscalar Time-Triggered Versatile Tensor Accelerator with Caching and Pre-fetching (STA-CP), a deterministic architecture that extends the STT-VTA by integrating on-chip cache hierarchies and predictive pre-fetching mechanisms as core architectural components. Unlike conventional accelerator designs that rely on dynamic runtime heuristics, STA-CP employs statically defined, time-triggered cache and memory operations. Consequently, all data transfers and cache accesses are executed according to predetermined schedules synchronized with the global execution timeline, thereby ensuring predictable and temporally deterministic system behaviour.

Consequently, STA-CP integrates two complementary memory management strategies:

1. **Deterministic Caching:** On-chip BRAM is organized as a statically managed cache, where data placement and reuse schedules are determined entirely at compile time. This deterministic cache holds intermediate feature maps and frequently reused tensors, eliminating redundant DRAM transactions and significantly reducing off-chip bandwidth requirements. Because all cache operations are predefined, their timing behavior is analyzable and free of runtime uncertainty.
2. **Static Pre-fetching:** Data required by future tensor cores (i.e., compute units) within the STA-CP are proactively fetched from DRAM/BRAM according to compile-time analysis of the execution graph. Each pre-fetch transaction is assigned a timestamp aligned with known memory-idle intervals, allowing data transfer and computation to overlap deterministically. This eliminates pipeline stalls and ensures continuous utilization of compute units without compromising temporal isolation.

By unifying these mechanisms within a superscalar time-triggered execution model, STA-CP achieves a balance previously thought unattainable: high computational throughput with full temporal determinism. The design effectively bridges the gap between the performance-oriented practices of high-performance computing and the analyzability requirements of real-time embedded systems.

1.3 Thesis Structure

Figure 1.1 illustrates the overall structure and logical progression of this thesis. The organization follows a stepwise development process that begins with the theoretical foundations and state-of-the-art background, and then incrementally evolves toward the design, optimization, analysis, and validation of deterministic AI hardware accelerators for safety-critical real-time systems.

Chapter 1 introduces the motivation, research challenges, objectives, and contributions of the thesis. In particular, the chapter discusses the growing need for deterministic and predictable AI acceleration in safety-critical systems and outlines the research questions addressed throughout the thesis.

Chapter 2 presents the fundamental concepts and a comprehensive state-of-the-art analysis through an extensive literature review. The chapter discusses existing AI accelerator architectures, deterministic and time-triggered computing systems, memory-management techniques, superscalar execution approaches, caching and pre-fetching strategies, as well as relevant FPGA and AI software frameworks. Furthermore, the chapter identifies the major limitations of existing approaches regarding predictability, timing determinism, resource contention, and WCET analyzability.

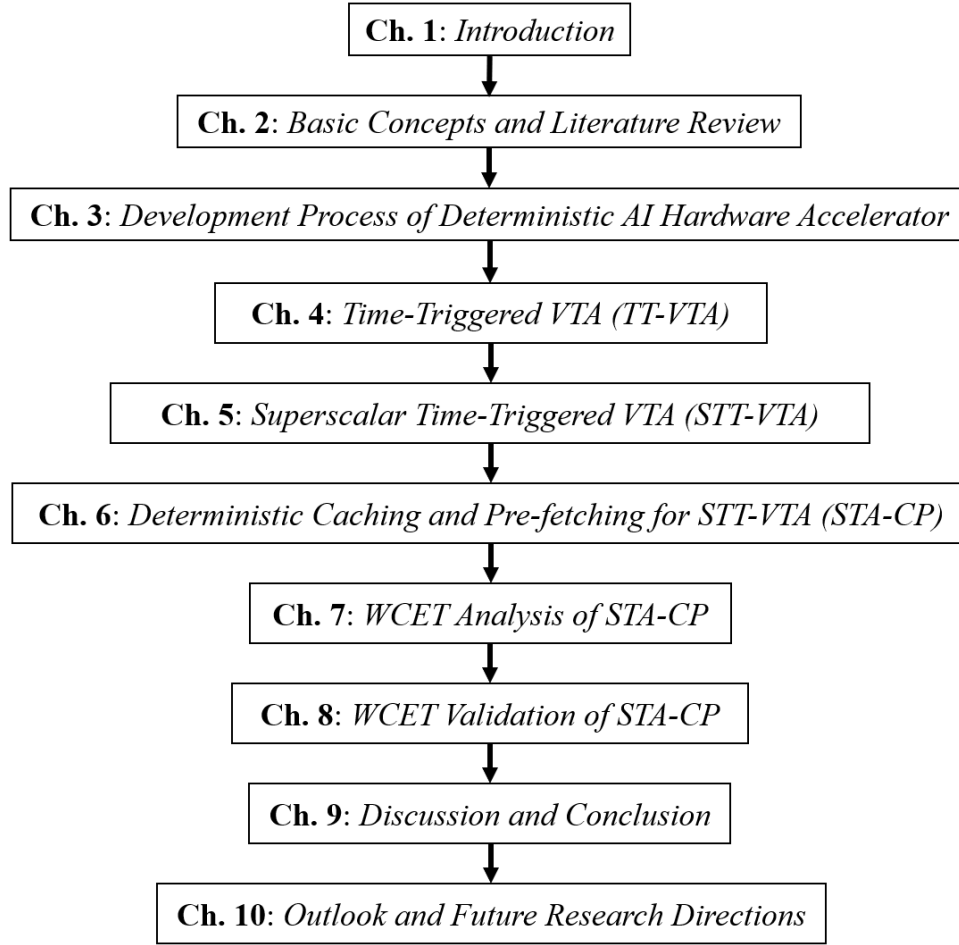


FIGURE 1.1: Thesis Structure

Based on this analysis, the research gap motivating the proposed architectures is established.

Chapter 3 introduces the complete development process of the proposed deterministic AI hardware accelerators. The chapter explains the end-to-end toolchain and design flow used throughout the thesis, including AI-model generation, instruction extraction using Apache TVM/VTA, cycle-accurate timing analysis, DRAM simulation, static scheduling, architecture generation, FPGA synthesis, and deployment.

Chapter 4 presents the first proposed architecture, namely the *Time-Triggered Versatile Tensor Accelerator (TT-VTA)*. This chapter introduces the transformation of the original event-triggered VTA into a deterministic time-triggered accelerator through the integration of the Time-Triggered Dispatcher and statically generated timing schedules. Furthermore, the chapter introduces the first version of the Scheduler with Built-in Simulation (i.e., TT-VTA Scheduler) and demonstrates how deterministic scheduling enables predictable inference execution and temporal isolation of shared memory resources.

Chapter 5 extends the TT-VTA architecture toward controlled superscalar execution through the introduction of the *Superscalar Time-Triggered VTA (STT-VTA)* architecture. The chapter presents the second version of the Scheduler with Built-in Simulation (i.e., STT-VTA Scheduler), which supports architecture specialization, metadata adaptation, and controlled parallel execution of tensor operations. In addition, the chapter investigates the effects of superscalar execution on inference latency,

resource utilization, and deterministic execution behavior.

Chapter 6 introduces the final architecture proposed in this thesis, namely the *Superscalar Time-Triggered VTA with Deterministic Caching and Pre-fetching (STA-CP)*. This chapter extends the STT-VTA architecture by integrating deterministic cache management and schedule-aware pre-fetching mechanisms. The chapter further introduces the third version of the Scheduler with Built-in Simulation (i.e., STA-CP Scheduler), which performs deterministic cache allocation, pre-fetch scheduling, memory-access optimization, and architecture adaptation according to the analyzed AI model. Moreover, the chapter evaluates the impact of deterministic caching and pre-fetching on inference latency, memory utilization, and execution efficiency.

Chapter 7 presents the formal *Timing Analysis of STA-CP*. This chapter develops the analytical framework used to derive the Worst-Case Execution Time of the proposed STA-CP architecture. It formalizes the timing semantics of the scheduler, instruction execution model, memory-access behavior, deterministic cache operations, and pre-fetching mechanisms. Furthermore, the chapter derives the mathematical formulations required to analytically determine the WCET of the architecture under deterministic execution constraints.

Chapter 8 focuses on the *WCET Validation of STA-CP*. The chapter validates the analytical WCET model through simulation and hardware experimentation on FPGA platforms. In particular, it compares the analytically derived WCET values against measured execution latencies obtained from real hardware executions and evaluates the accuracy, determinism, and composability of the proposed architecture.

Chapter 9 concludes the thesis, summarizes the achieved contributions, and evaluates the overall research outcomes with respect to the proposed research questions.

Finally, **Chapter 10** outlines potential directions for future research on deterministic AI acceleration.

As illustrated in Figure 1.1, the thesis follows a progressive architectural evolution. Each chapter incrementally extends the previous architecture by introducing new deterministic execution and optimization mechanisms. Starting from the deterministic execution model of TT-VTA, the work evolves toward superscalar parallelism in STT-VTA and finally toward deterministic caching and pre-fetching in STA-CP. The subsequent WCET-analysis and validation chapters then establish the analytical and experimental foundations required to demonstrate the predictability and determinism of the final architecture for safety-critical real-time systems.

1.4 List of Publications

1. **Y. Bebawy**, M. -C. Schmidt, H. Ahmadian, A. M. Ezekiel and R. Obermaisser, "Time-Triggered Inference on FPGAs," 2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA), Padova, Italy, 2024, pp. 1-7, doi: 10.1109/ETFA61755.2024.10710675.
2. A. M. Ezekiel, **Y. Bebawy** and R. Obermaisser, "Enhancing Neural Network Predictability through Simulation-Based Analysis of Memory Accesses for Time-Triggered Tensor Acceleration," 2024 10th International Conference on Control, Decision and Information Technologies (CoDIT), Vallette, Malta, 2024, pp. 2839-2844, doi: 10.1109/CoDIT62066.2024.10708250.
3. **Y. Bebawy**, A. Micheal Ezekiel and R. Obermaisser, "Superscalar Time-Triggered Versatile-Tensor Accelerator," in IEEE Transactions on Computer-Aided Design

- of Integrated Circuits and Systems, vol. 44, no. 7, pp. 2503-2515, July 2025, doi: 10.1109/TCAD.2025.3528355.
4. **Y. Bebawy**, A. Alhumaidi, R. Obermaisser, "Deterministic Caching and Prefetching for High-Performance Superscalar Time-Triggered Tensor Accelerators", in ACM Transactions on Embedded Computing Systems, 2026. (Status: under review)
 5. A. Alhumaidi, **Y. Bebawy**, R. Obermaisser, "Deterministic Inference on Distributed AI Accelerators Interconnected by TSN", 21 April 2026, PREPRINT (Version 1) available at Research Square [<https://doi.org/10.21203/rs.3.rs-9246074/v1>]
 6. T. Strathmann, G. Hake, H. Guissouma, C. P. Hohl, **Y. Bebawy**, S. V. Maelen and A. Koerner, "Project Overview for Step-Up!CPS - Process, Methods and Technologies for Updating Safety-critical Cyber-physical Systems," 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 2021, pp. 1326-1329, doi: 10.23919/DATE51398.2021.9474032.
 7. **Y. Bebawy** et al., "Incremental Contract-based Verification of Software Updates for Safety-Critical Cyber-Physical Systems," 2020 International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, NV, USA, 2020, pp. 1708-1714, doi: 10.1109/CSCI51800.2020.00318.
 8. G. Ehmen, B. Koopmann, **Y. Bebawy** and P. Ittershagen, "Measurement-based Online Verification of Timing Properties in Distributed Systems," 2020 International Conference on Omni-layer Intelligent Systems (COINS), Barcelona, Spain, 2020, pp. 1-6, doi: 10.1109/COINS49042.2020.9191647.
 9. B. Sorkhpour, R. Obermaisser and **Y. Bebawy**, "Optimization of Frequency-Scaling in Time-Triggered Multi-Core Architectures using Scenario-Based Meta-Scheduling," AmE 2019 - Automotive meets Electronics; 10th GMM-Symposium, Dortmund, Germany, 2019, pp. 1-10.
 10. Fakih, M., Grüttner, K., Schreiner, S., Seyyedi, R., Azkarate-Askasua, M., Onaindia, P., Poggi, T., Romero, N. G., Quesada Gonzalez, E., Sundström, T., Peiró Frasset, S., Balbastre, P., Mohammadat, T., Öberg, J., **Bebawy, Y.**, Obermaisser, R., Maleki, A., Lenz, A., & Graham, D. (2019). Experimental Evaluation of SAFEPOWER Architecture for Safe and Power-Efficient Mixed-Criticality Systems. *Journal of Low Power Electronics and Applications*, 9(1), 12. <https://doi.org/10.3390/jlpea9010012>
 11. Obermaisser, R., Ahmadian, H., Maleki, A., **Bebawy, Y.**, Lenz, A., & Sorkhpour, B. (2019). Adaptive Time-Triggered Multi-Core Architecture. *Designs*, 3(1), 7. <https://doi.org/10.3390/designs3010007>

Chapter 2

Basic Concepts and Literature Review

The rapid advancement of Artificial Intelligence (AI), particularly Deep Neural Networks (DNNs), has significantly transformed modern computing systems and enabled major technological progress across a broad range of domains, including autonomous driving [10], robotics [26], industrial automation [27], avionics [28], healthcare [29], and cyber-physical systems [30, 31]. Modern AI models provide highly accurate capabilities for object detection, classification, localization, semantic understanding, sensor fusion, prediction, and autonomous decision making [32, 33, 34, 35]. Consequently, AI inference is increasingly being integrated into safety-critical systems, where execution correctness, timing predictability, and bounded response latency become fundamental requirements.

Despite the substantial computational capabilities achieved by contemporary AI accelerators, deploying an AI model within safety-critical real-time systems remains a major challenge. Most existing AI acceleration platforms have been primarily designed for cloud computing, high-performance computing, and edge-inference environments, where the dominant optimization objectives are throughput maximization, average-case latency reduction, and energy efficiency [23, 36, 37, 38]. In contrast, safety-critical systems impose fundamentally different requirements, including deterministic execution behavior, predictable resource utilization, temporal isolation, analyzable Worst-Case Execution Time (WCET) and bounded latency [39, 40, 41, 42].

Most contemporary AI accelerator architectures achieve high performance through runtime scheduling, dynamic arbitration, adaptive caching, speculative pre-fetching, and dynamically managed memory hierarchies [23, 43, 44]. While these techniques significantly improve average-case performance, they simultaneously introduce timing variability, execution nondeterminism, and contention over shared resources. Consequently, such systems become difficult to analyze formally and unsuitable for applications requiring strict timing guarantees [45, 46, 47].

This chapter reviews the current state of the art in AI acceleration and positions the contributions of this thesis within the broader research landscape. The discussion begins by outlining the requirements imposed by safety-critical AI systems, followed by a review of existing AI software ecosystems and accelerator architectures. Subsequently, memory-optimization techniques, deterministic computing approaches, DRAM timing simulation frameworks, and FPGA development toolchains are discussed. Finally, the chapter concludes with a comparative analysis highlighting the research gaps addressed by the proposed TT-VTA, STT-VTA, and STA-CP architectures.

2.1 Requirements of AI Accelerators for Safety-Critical Systems

Traditional AI accelerators are predominantly designed for throughput-oriented environments such as cloud infrastructures, data centers, and consumer-oriented edge devices [23, 36, 48]. In such systems, occasional timing variability is often acceptable provided that overall throughput remains sufficiently high. However, executing AI inference within safety-critical embedded systems introduces significantly stricter requirements that extend beyond conventional performance metrics [39, 40].

One of the most important requirements is **deterministic execution behavior**. Determinism refers to the ability of a system to exhibit identical temporal behavior when executing identical workloads under identical conditions. In safety-critical systems, deterministic execution is essential because delayed or unpredictable responses may directly compromise system safety. Standards such as ISO-26262 for automotive systems, DO-178C for avionics, IEC-61508 for industrial automation, and EN-50128 for railway systems explicitly require predictable execution behavior and analyzable timing guarantees [49, 50, 51, 52].

Unfortunately, modern AI accelerators inherently introduce timing variability due to several architectural characteristics. These include runtime instruction scheduling, dynamic memory arbitration, shared-resource contention, adaptive cache replacement policies, speculative pre-fetching, and dynamically resolved dependencies [23, 44]. Several studies have identified shared-memory interference and bus contention as major sources of timing unpredictability in heterogeneous embedded systems [46, 47, 53, 54, 55]. Since AI inference workloads heavily depend on continuous movement of tensors, feature maps, activations, and weights between memory hierarchies, memory contention becomes one of the dominant factors affecting inference predictability [48, 24].

Another major requirement is **WCET analyzability**. Safety-critical systems require formal or empirical upper bounds on execution latency to ensure that deadlines are always satisfied [42]. Traditional Worst-Case Execution Time (WCET) analysis techniques, which rely on static, predictable behavior for simple CPU pipelines, are difficult to apply to AI accelerators because such accelerators have complex architectures and rely heavily on dynamic execution models, adaptive memory systems, and runtime-managed resources [56, 57, 45, 41]. Consequently, several researchers advocated time-triggered execution models and static scheduling techniques as mechanisms for improving predictability and analyzability [39, 40, 45].

Power efficiency also constitutes an important requirement in embedded and cyber-physical systems. CPUs and GPUs generally provide high computational throughput but often consume considerable amounts of energy due to speculative execution, complex control logic, multi-threading, and large memory hierarchies [58]. FPGA-based accelerators, in contrast, provide improved energy efficiency through customizable datapaths, dedicated hardware specialization, and reduced control overhead [19, 59, 60]. Consequently, FPGA-based acceleration has emerged as a highly attractive solution for embedded AI inference [61].

Finally, **scalability** and **flexibility** remain fundamental design considerations. AI models continuously evolve, requiring accelerator architectures to support multiple neural-network structures, tensor dimensions, and computational requirements [30, 31]. Balancing flexibility, performance, hardware utilization, and deterministic execution therefore represents one of the central challenges in modern AI accelerator research.

2.2 AI Frameworks and Compilation Ecosystems

Modern Artificial Intelligence (AI) development is largely driven by high-level software frameworks such as TensorFlow [62], PyTorch [63], Keras [64] and others. These frameworks provide comprehensive environments for designing, training, and deploying Machine Learning (ML) and Deep Learning (DL) models by abstracting low-level implementation complexity. Their widespread adoption is primarily attributed to efficient tensor computation, automatic differentiation, extensive software ecosystems, and support for heterogeneous hardware acceleration.

At their core, these frameworks rely on tensor-based computation models and computational graphs to represent neural network operations. TensorFlow employs a graph-oriented execution model that combines eager execution with graph optimization mechanisms through `tf.function`, enabling both development flexibility and optimized deployment. In contrast, PyTorch follows a dynamic define-by-run paradigm, where computation graphs are constructed during execution, simplifying debugging and rapid prototyping. Keras provides a higher-level declarative interface integrated within TensorFlow, enabling modular and user-friendly neural network development while leveraging TensorFlow's execution backend.

A major strength of these frameworks is their ability to exploit heterogeneous computing platforms, including CPUs, GPUs, TPUs, and FPGA-based accelerators. Through backend libraries such as CUDA [65] and cuDNN [66], they support parallel execution, operator fusion, memory reuse, and runtime scheduling optimizations to maximize throughput and resource utilization. In addition, their ecosystems provide extensive tooling for distributed training, mixed-precision computation, hyperparameter optimization, model compression, and deployment on edge platforms through technologies such as TensorFlow Lite, TensorRT, and TorchScript.

Despite these advantages, mainstream AI frameworks are primarily optimized for average-case performance rather than deterministic execution. Runtime scheduling, asynchronous kernel launches, dynamic memory allocation, and adaptive caching mechanisms introduce variability in execution timing and memory behavior. Such nondeterministic execution models are acceptable in cloud and high-performance computing environments but pose significant challenges in safety-critical real-time systems, where bounded latency and predictable behavior are mandatory [4, 17]. Moreover, contention over shared resources such as GPUs, memory buses, and caches can further degrade temporal predictability and complicate Worst-Case Execution Time (WCET) analysis [14, 13].

Consequently, while AI frameworks are highly effective for model development and deployment, they do not inherently satisfy the strict temporal and determinism requirements of safety-critical systems. This limitation motivates the need for specialized compilation and execution pipelines capable of transforming high-level AI models into statically analyzable execution patterns. Such transformations typically involve static scheduling, deterministic memory management, elimination of runtime variability, and mapping onto dedicated hardware accelerators with predictable execution semantics.

In this context, frameworks such as TensorFlow and PyTorch serve primarily as front-end model specification environments, while deterministic accelerator architectures act as predictable back-end execution engines. Bridging this gap is essential for enabling WCET-analyzable AI inference in real-time embedded systems.

2.3 AI Accelerator Architectures

2.3.1 General-Purpose AI Accelerators

General-purpose AI accelerators attempt to support a wide range of neural-network models using reusable hardware structures and configurable execution pipelines. Unlike model-specific accelerators, these architectures emphasize flexibility and adaptability [61].

One of the most prominent open-source AI accelerators is the Versatile Tensor Accelerator (VTA) [61, 20], which serves as the FPGA backend of Apache TVM [67]. Apache TVM [67], that stands for Apache "Tensor Virtual Machine", is an open-source deep learning compiler stack designed to bridge the gap between the productivity of high-level AI frameworks and the performance efficiency of hardware accelerators. As illustrated in Figure 2.1, TVM provides a unified compilation infrastructure capable of translating models developed in mainstream machine learning frameworks into optimized executable representations for heterogeneous hardware platforms, including CPUs, GPUs, and domain-specific AI accelerators such as the Versatile Tensor Accelerator (VTA) [19, 20]. Owing to its modular architecture and cross-platform support, TVM has become a widely adopted framework for AI deployment across cloud, edge, and embedded systems.

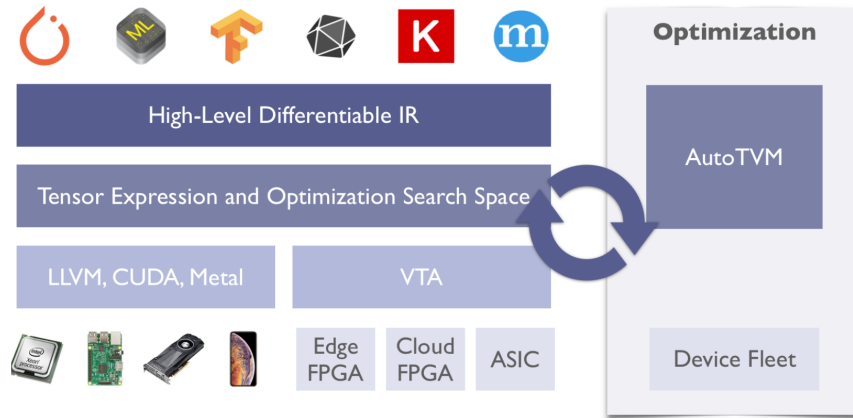


FIGURE 2.1: Overview of TVM Compiler Stack [67]

The TVM compiler stack integrates several optimization stages, including graph-level transformations, operator fusion, memory optimization, and target-specific code generation. In addition, TVM incorporates auto-tuning mechanisms that automatically explore optimal hardware specific configurations to improve execution efficiency on a given target platform. These capabilities enable TVM to generate optimized inference pipelines while abstracting much of the low-level hardware complexity from developers.

A key component of the TVM ecosystem is the Versatile Tensor Accelerator (VTA), illustrated in Figure 2.4. VTA is a configurable tensor-processing accelerator designed primarily for FPGA-based platforms and tightly integrated into the TVM compilation flow. It provides specialized hardware support for tensor operations commonly used in deep neural network inference, such as matrix multiplication, accumulation, and data movement.

In this work, Apache TVM and its low-level integration with VTA are utilized to capture and analyze instruction-level execution behavior during AI inference. By enabling TVM's debug mode, the compiler produces detailed execution traces

containing the generated VTA instructions in binary format, as shown in Figure 2.2-a. These instructions follow the structured encoding format illustrated in Figure 2.3, where each instruction contains the information required to coordinate execution across the accelerator modules.

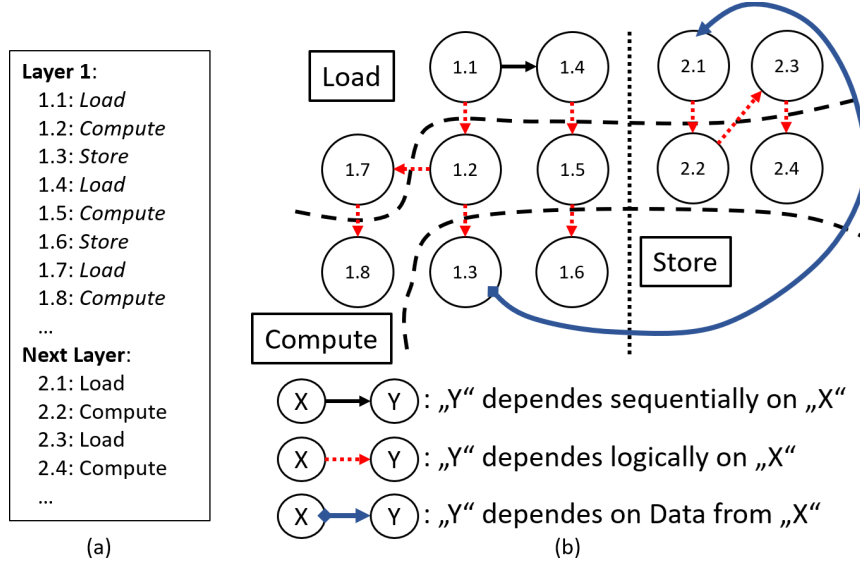


FIGURE 2.2: (a) Logged Instructions from TVM, (b) Instructions dependency Graph

Each instruction pattern includes an operation code (OPCODE), which specifies the instruction type (e.g., load, compute, or store). Dependencies between instructions are classified into two forms: explicit control dependencies, derived from Dependency FLAGS (red dashed arrows in Figure 2.2-b), indicating whether an instruction must wait for predecessors or trigger successors; and implicit data dependencies, inferred from memory access patterns (blue solid arrows). For example, instruction “2.1: Load” depends on data produced by “1.3: Store.” These relationships can be visualized in a model graph that captures execution flow and data movement during inference.

For memory instructions (Load/Store), the pattern specifies DRAM access details such as addresses, iteration counts, and strides. While for compute instructions (e.g., GEMM, ALU), it embeds MICRO-Operations (Figure 2.3), providing fine-grained directives for matrix computations.

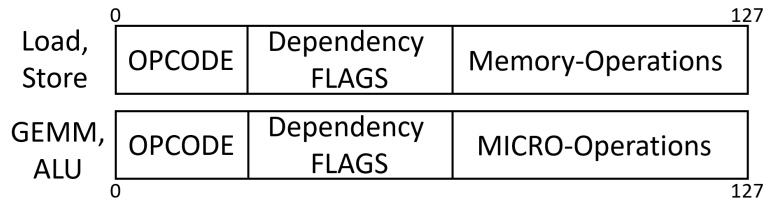


FIGURE 2.3: VTA Instruction Patterns [20]

For the execution of such instructions, the VTA architecture comprises several dedicated hardware modules that collectively support hardware-accelerated inference execution, including:

1. **Fetch Module:** Sorts the fetched instructions from the DRAM into the instruction queues of the respective modules based on the instruction type, distinguished by its OPCODE pattern, see Figure 2.3.

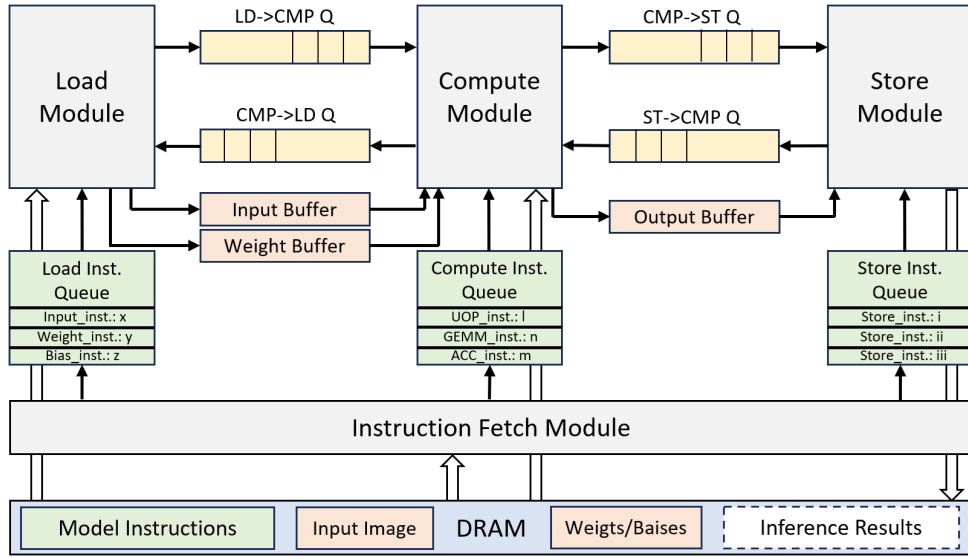


FIGURE 2.4: HW Architecture of the "Versatile Tensor Accelerator (VTA) [19, 20]"

2. **Load Module:** Loads input data, weights, and biases from the DRAM according to the load instructions, then writes them to the designated input and weight buffers, preparing them for processing by the Compute Module.
3. **Compute Module:** Retrieves input, weight, and bias data, performs the required matrix-multiplication operations based on the MICRO-Operations information, and writes the intermediate-/final-inference results to the output buffer.
4. **Store Module:** Retrieves the results from the output buffer and stores them in the DRAM according to the memory locations as specified by the store instructions.

Although the VTA architecture supports dynamic execution of multiple neural-network models through software reconfiguration and compiler-managed execution flows, its execution behavior remains event-triggered and dynamically managed. The synchronization mechanisms based on inter-module dependency queues, together with shared-memory arbitration, introduce timing variability and non-deterministic execution behavior [68]. As a result, providing deterministic timing guarantees and performing accurate Worst-Case Execution Time (WCET) analysis become significantly more challenging.

Another influential accelerator is Google's Tensor Processing Unit (TPU) [23, 69]. TPUs employ large systolic arrays optimized for tensor and matrix computations while integrating unified buffers, dynamic memory hierarchies, and compiler-managed execution pipelines. Similarly, GPU-based accelerators such as NVIDIA Tensor Cores [70, 71] exploit massive thread-level parallelism, shared memories, and runtime scheduling to maximize throughput.

NVDLA [43, 72] represents another open-source DNN accelerator architecture targeting embedded AI inference. NVDLA integrates configurable convolution engines, shared buffers, and scalable processing pipelines. Nevertheless, similar to other modern accelerators, execution scheduling and memory management remain dynamically controlled.

Overall, existing general-purpose accelerators prioritize throughput and flexibility but provide limited support for deterministic execution and analyzable timing behavior.

2.3.2 Model-Specific FPGA Acceleration

Another major research direction focuses on synthesizing neural networks directly into dedicated hardware circuits. Unlike general-purpose accelerators, these approaches optimize hardware specifically for individual AI models [59, 73].

FINN [59] represents one of the most widely known frameworks for Binarized Neural Networks (BNNs). By reducing weights and activations to binary representations, FINN significantly simplifies neural computations and enables highly efficient FPGA implementations with low power consumption and high throughput. However, these benefits come at the cost of reduced inference accuracy [74]. Furthermore, each model modification requires complete hardware recompilation, thereby limiting flexibility.

LogicNets [73] similarly transforms quantized neural networks into truth-table representations synthesized directly into FPGA logic. Although this approach achieves high performance, hardware resource utilization grows exponentially with neuron fan-in, thereby limiting scalability for larger and more complex models.

Other frameworks such as hls4ml [75] and DNNWeaver [76] employ High-Level Synthesis (HLS) to automatically generate FPGA implementations from neural-network descriptions. These approaches simplify deployment and accelerate design-space exploration but remain largely model-specific.

In contrast, the TT-VTA architecture proposed in this thesis aims to preserve flexibility while simultaneously enabling deterministic execution behavior. Rather than generating dedicated hardware per model, TT-VTA supports multiple AI models through statically generated timing schedules and predictable resource management.

2.4 Inference Parallelization and Workload Distribution

Optimizing AI inference time is typically achieved through AI model optimization techniques [77] as well as through partitioning AI workloads across separate processing elements, whether on the same embedded platform [78] or distributed among end systems and the cloud [79]. For instance, The authors of BranchyNet [80] proposed an inference partitioning strategy that optimizes execution time by (1) identifying the shortest branch in the neural network model capable of producing inference results with acceptable accuracy, and (2) executing the initial layers of the DNN-shortest branch on a local computational platform, while offloading subsequent layers to the cloud. This approach aims to balance the delay associated with local platforms, which often have limited computational resources, against the communication latencies incurred when sending data to and from the cloud. While their approach accelerates inference, it may do so at the expense of model accuracy, a trade-off that could be unacceptable for safety-critical applications requiring both timely and precise outcomes. In these domains, sacrificing accuracy to reduce latency can lead to suboptimal or unsafe decisions, which is particularly problematic in fields like autonomous driving, where high reliability and exacting accuracy standards are critical. Therefore, solutions aimed at such applications must achieve both low-latency inference and uncompromised accuracy to meet stringent safety requirements and regulatory standards.

Another approach to address partitioning issues for real-time DNNs involves developing schedulers to distribute AI workloads between CPUs and GPUs on the same computing platform, thus enhancing inference speed through effective scheduling [81]. However, executing AI inference on distributed computational nodes, such as CPUs and GPU threads, introduces contention for shared resources like memory [46] and buses [47], leading to significant inference delays on computing platforms. For such systems, memory access contention issues can often be mitigated using a hypervisor [54] or middleware [55]. These tools enable efficient scheduling and arbitration of memory access requests, allowing the system to coordinate and prioritize memory usage among various processes and cores. This level of resource management is made possible by software layers that can dynamically manage access, ensuring each process or core gets appropriate memory bandwidth and utilization's slot when required [53]. However, they also introduce several performance drawbacks, particularly concerning timing and latency as every memory request must go through their layers. Thus, affecting and degrading the performance of the system and increasing the inference time.

The proposed STT-VTA architecture addresses these limitations by enabling deterministic superscalar execution. Instead of relying on runtime scheduling and arbitration, STT-VTA employs compile-time generated schedules that enable parallel execution while preserving predictable timing behavior.

2.5 Memory Systems and Latency-Hiding Techniques

Efficient and predictable memory access remains a cornerstone of high-performance AI acceleration, particularly in real-time and resource-constrained embedded systems. Deep learning inference workloads are typically memory-bound, as the movement of feature maps, activations, and weights often dominates both execution latency and energy consumption [36, 23, 24]. Over the past decade, extensive research has focused on optimizing data movement through caching, buffering, and pre-fetching mechanisms. However, most of these approaches have been designed for throughput-oriented platforms such as GPUs, TPUs, or cloud accelerators [71, 23], where the goal is to maximize average-case performance rather than provide bounded execution times.

For example, architectures such as Google's TPU [23] and NVIDIA's Tensor Cores integrate multi-level cache hierarchies and shared memory systems to exploit spatial and temporal reuse of tensor data. Likewise, domain-specific accelerators such as Eyeriss [36, 48, 60], ShiDianNao [38], MAERI [37], and NVDLA [43, 72] employ sophisticated dataflows (e.g., row-stationary or weight-stationary mappings) and local scratchpads to minimize DRAM bandwidth pressure.

These designs achieve impressive throughput and energy efficiency for edge and data-center workloads, but they are governed by dynamic runtime behavior—for example, least-recently-used (LRU) or pseudo-LRU replacement policies—and therefore exhibit non-deterministic access latency. Cache hits, evictions, and coherence transactions depend on instantaneous workloads and input data distributions. Consequently, despite their efficiency, such systems cannot guarantee worst-case execution time (WCET) bounds or temporal isolation, limiting their applicability in safety-critical systems.

Recent research has sought to improve cache predictability through compiler-assisted or scratchpad-based management [82, 83, 84]. These techniques partially mitigate nondeterminism but still rely on dynamic hardware policies.

In contrast, the proposed STA-CP architecture solution adopts a fundamentally different approach: a time-triggered caching paradigm. All cache-related operations—data placement, refill, and eviction—are determined statically at compile time and encoded in the global time schedule. The cache behaves as a deterministic, temporally partitioned memory structure rather than an adaptive one. This design retains the benefits of data reuse typical of high-performance accelerators while ensuring that access latencies are statically bounded.

On the other hand, pre-fetching is another long-established strategy to mitigate memory latency in high-performance architectures [85, 86]. Traditional stride-based, stream-based, and correlation-based pre-fetchers mechanisms leverage runtime prediction of memory access patterns to bring data closer to processing units ahead of time [87].

In AI accelerators, these techniques have been adapted to exploit model-level or layer-level predictability. For instance, frameworks such as DeepCache [24], ROMANet [25], and DNPU [88] pre-load feature maps or weight blocks based on dependency graphs derived from network architectures. Compiler-assisted approaches like Tetris [89] and NeuroStream [90] attempt to overlap data movement with computation through static scheduling heuristics. These solutions significantly improve throughput on dynamic or batched workloads, but their speculative or demand-based pre-fetching introduces timing variance and interference—both incompatible with deterministic, time-triggered execution models.

The STA-CP architecture introduces a static, schedule-aware pre-fetching mechanism. Rather than relying on runtime speculation, each pre-fetch operation is precisely scheduled at compile time, aligned with idle memory slots identified through dependency analysis. This ensures that pre-fetches neither compete with ongoing transfers nor cause resource contention. As a result, tensor cores are continuously fed with ready data, avoiding stalls while preserving strict temporal isolation and WCET analyzability.

2.6 Time-Triggered and Deterministic Computing Systems

Deterministic and time-triggered computing systems have long been recognized as essential foundations for safety-critical real-time applications. In such systems, guaranteeing predictable timing behavior is often more important than maximizing average-case performance, since timing violations may lead to catastrophic system failures. Consequently, considerable research has focused on eliminating runtime uncertainty through static scheduling, composable architectures, predictable communication, and temporally isolated resource management.

One of the most influential paradigms in this domain is the time-triggered execution model, originally introduced and formalized by Kopetz [39]. In time-triggered systems, all activities are initiated according to globally synchronized schedules generated offline at design time. This eliminates runtime arbitration and nondeterministic event-driven interactions, enabling deterministic execution and analyzable timing behavior. Time-triggered architectures have been widely adopted in safety-critical distributed systems through protocols and standards such as TTP/C [91], and Time-Triggered Ethernet (TTEthernet) [39, 92]. These systems demonstrate how static scheduling and temporal partitioning can guarantee bounded communication latency and interference-free execution.

Obermaisser significantly contributed to the advancement of time-triggered and distributed embedded systems through extensive research on predictable communication, composable architectures, and deterministic system integration [40, 93, 94]. His work investigated the integration of distributed time-triggered systems, globally synchronized architectures, fault-tolerant communication infrastructures, and deterministic middleware for embedded real-time applications. In particular, Obermaisser emphasized the importance of globally synchronized execution and temporally isolated resource management for guaranteeing predictable system behavior in distributed safety-critical systems.

Furthermore, Obermaisser's research explored deterministic execution within mixed-criticality and distributed embedded platforms, addressing challenges related to timing composability, communication predictability, and distributed scheduling [40, 94]. These concepts strongly align with the architectural objectives pursued in this thesis, where deterministic scheduling and temporal isolation are extended from distributed embedded systems toward AI hardware acceleration.

Another important direction in deterministic computing systems is the concept of composability. Goossens et al. [95, 96, 97] introduced composable architectures and predictable Networks-on-Chip (NoCs) capable of providing temporal isolation between concurrently executing applications. Their work investigated predictable communication infrastructures, time-division multiplexing, arbitration-free communication mechanisms, and composable System-on-Chip (SoC) architectures. The main objective of composability is to ensure that the timing behavior of one subsystem remains unaffected by the activities of other subsystems sharing the same hardware resources.

The notion of composability is particularly important in modern heterogeneous computing platforms where multiple processing elements compete for shared resources such as memory buses, interconnects, and DRAM systems. Goossens demonstrated that uncontrolled resource sharing introduces timing interference and unpredictability, making Worst-Case Execution Time (WCET) analysis increasingly difficult. To address these issues, composable architectures enforce temporal partitioning and predictable communication schemes that eliminate interference among independent tasks.

Research on deterministic processors and predictable execution architectures further extended these concepts toward computation itself. The PRET (Precision Timed) architecture proposed by Edwards and Lee [41] introduced timing-predictable processors designed specifically for real-time systems. PRET eliminates several sources of timing unpredictability commonly found in conventional processors, such as speculative execution and dynamic caching behavior. Instead, it promotes explicit timing control and deterministic instruction execution.

Similarly, Pellizzoni et al. [45] proposed the PREDATOR memory controller architecture, which focuses on predictable DRAM arbitration for multicore systems. Their work demonstrated how conventional memory systems introduce substantial timing interference due to dynamically scheduled memory requests and contention on shared buses. By statically managing memory access patterns, predictable memory controllers can significantly improve temporal isolation and WCET analyzability.

The deterministic concepts introduced in these prior works strongly influenced the architectural design principles adopted in this thesis. However, unlike conventional real-time processors, communication systems, or composable NoCs, AI accelerators introduce additional challenges related to tensor-level parallelism, dynamic memory reuse, irregular computation graphs, and high-bandwidth data movement. Therefore, the architectures proposed in this thesis extend the principles of deterministic and

time-triggered computing toward AI acceleration. The proposed TT-VTA, STT-VTA, and STA-CP architectures statically schedule tensor operations, DRAM accesses, cache operations, and pre-fetching activities at compile time. This eliminates runtime arbitration and minimizes timing variability during inference execution.

More specifically, TT-VTA introduces globally synchronized time-triggered execution into the VTA architecture, replacing event-triggered dependency management with static scheduling. STT-VTA extends this concept further by enabling deterministic superscalar execution while preserving temporal isolation among shared resources. Finally, STA-CP integrates deterministic caching and schedule-aware pre-fetching mechanisms, enabling latency hiding and memory optimization without introducing runtime nondeterminism.

Consequently, the proposed architectures can be viewed as an extension of classical time-triggered and composable-system principles into the domain of AI hardware acceleration. By combining deterministic scheduling, predictable memory behavior, temporal isolation, and static execution planning, the proposed architectures enable AI inference execution that is analyzable, composable, and suitable for safety-critical real-time systems.

2.7 DRAM Timing Simulation and WCET-Oriented Memory Modeling

Accurate modeling of off-chip memory behavior is essential for evaluating modern AI accelerators, particularly in safety-critical real-time systems where memory latency and contention directly affect timing predictability and Worst-Case Execution Time estimation. Since DRAM accesses frequently dominate inference latency and energy consumption in deep learning workloads [23, 36], several DRAM simulation frameworks have been developed to analyze memory timing behavior, scheduling policies, and contention effects prior to hardware deployment.

Among the most widely adopted frameworks are DRAMSim2 [98], DRAMSim3 [99], and DRAMSys [100]. Although all three provide DRAM timing simulation capabilities, they differ considerably in abstraction level, modeling complexity, extensibility, and suitability for deterministic timing analysis as illustrated in Table 2.1.

DRAMSim2 is a lightweight cycle-accurate DRAM simulator that models low-level timing behavior, including read/write-request scheduling, row-buffer management, bank conflicts, and memory-controller arbitration. Its modular C++ implementation and relatively compact codebase make it highly suitable for architectural exploration and custom accelerator integration. In particular, *DRAMSim2* provides a practical balance between timing fidelity and implementation simplicity, exposing fine-grained timing information while remaining efficient enough for iterative design-space exploration and deterministic timing analysis.

DRAMSim3 extends *DRAMSim2* with support for modern DRAM technologies such as DDR4, LPDDR4, and HBM, while also introducing enhanced power modeling and advanced multi-channel memory support. However, these improvements come at the cost of increased simulator complexity, configuration overhead, and integration effort.

DRAMSys, in contrast, adopts a SystemC/TLM-based system-level simulation methodology. It is primarily designed for full-system co-simulation and heterogeneous platform exploration. While this high-level abstraction facilitates interoperability with virtual platforms, it also reduces visibility into low-level timing details that are essential for precise interference analysis and WCET estimation.

TABLE 2.1: Comparison of DRAM Memory Simulation Frameworks

Feature	DRAMSim2 [98]	DRAMSim3 [99]	DRAMSys [100]
Simulation Level	Cycle-accurate	Cycle-accurate	System-level / TLM-based
Implementation Style	C++	C++	SystemC
Timing Fidelity	High	Very High	Medium-High
Low-Level DRAM Visibility	High	High	Moderate
Supported DRAM Standards	DDR3 / Basic DDR4	DDR4, LPDDR4, HBM, etc.	Multiple via TLM models
Memory Controller Modeling	Detailed	Advanced	Abstracted
Power Modeling	Basic	Enhanced	Moderate
Multi-Channel Support	Limited	Advanced	Supported
Integration Complexity	Low	High	High
Simulation Overhead	Low	Medium-High	High
Ease of Customization	High	Moderate	Moderate
Suitability for FPGA/Accelerator Studies	High	Moderate	Moderate
Suitability for WCET Analysis	High	Moderate	Limited
Deterministic Timing Analysis	Well-suited	Partially suited	Limited suitability
Primary Research Focus	Timing analysis and architectural exploration	Modern DRAM architecture eval- uation	Full-system co- simulation

For the objectives of this thesis, the primary focus is not the evaluation of the latest DRAM technologies, but rather the analysis and control of memory-access behavior within deterministic AI accelerator architectures. Consequently, the selected simulator must provide analyzable timing behavior while remaining sufficiently lightweight for integration into the proposed static scheduling framework.

In this context, DRAMSim2 represents the most suitable choice. Its timing model captures critical DRAM behaviors such as scheduling delays, row-buffer conflicts, and arbitration latency, all of which directly influence deterministic execution analysis and WCET estimation. Furthermore, DRAMSim2 satisfies the key requirements of this work through its:

- **Cycle-accurate timing model**, enabling fine-grained analysis of DRAM access latency and memory-interference behavior,
- **Lightweight and modular architecture**, facilitating integration with all versions of the proposed scheduler and simulation framework,
- **Transparent memory scheduling behavior**, supporting deterministic evaluation of memory-access timing and WCET estimation,
- **Low simulation overhead**, enabling efficient exploration of multiple scheduling strategies, DRAM architectures, and timing configurations.

Accordingly, DRAMSim2 is integrated into the proposed offline scheduling framework to model memory-related instructions and estimate worst-case memory latency during compilation. A detailed discussion of the DRAMSim2-based WCET modeling methodology, including memory-latency decomposition, worst-case latency derivation, and interpretation of DRAMSim2 timing traces, is presented in Appendix A.

2.8 FPGA Toolchains and Hardware Design Frameworks

The development of FPGA-based AI accelerators for safety-critical real-time systems requires a robust and integrated toolchain capable of supporting the complete hardware design lifecycle, including modeling, verification, synthesis, implementation, timing analysis, and deployment.

In this work, the Xilinx Vivado Design Suite [101] is employed as the primary development environment for implementing and validating the proposed TT-VTA, STT-VTA, and STA-CP architectures.

Vivado provides a comprehensive FPGA development framework supporting RTL design entry, High-Level Synthesis (HLS), behavioral simulation, logic synthesis, placement and routing, static timing analysis, and bitstream generation for FPGA deployment.

Behavioral simulation constitutes a fundamental stage in validating the proposed time-triggered accelerator architectures. Vivado enables cycle-accurate functional simulation prior to synthesis, allowing detailed observation of signal transitions, execution ordering, module synchronization, and memory transactions. Furthermore, behavioral simulation is not only used for functional validation, but also for determining the cycle-level execution latency of both memory-bound and compute-bound instructions. The collected timing information is subsequently utilized by the proposed scheduler to construct deterministic execution schedules and estimate overall inference latency and WCET.

Following functional validation, Vivado synthesizes the RTL design into FPGA primitives such as Look-Up Tables (LUTs), Flip-Flops (FFs), Block RAMs (BRAMs), and Digital Signal Processing (DSP) slices. This stage enables quantitative evaluation of hardware utilization, scalability, and architectural efficiency.

After synthesis, Vivado performs placement and routing of the generated netlist onto the physical FPGA fabric. Vivado further provides advanced timing-analysis capabilities, including Static Timing Analysis (STA), clock-domain analysis, and timing-closure optimization. These capabilities are particularly critical for time-triggered architectures because statically generated schedules assume fixed execution latencies.

Overall, Vivado serves as the primary hardware development and verification framework throughout this thesis. By bridging the gap between architectural concepts and physically deployable hardware implementations, Vivado provides the necessary infrastructure for developing deterministic AI accelerators targeting safety-critical real-time systems.

2.9 Comparative Overview and Research Gap Analysis

The literature reviewed throughout this chapter demonstrates that existing AI accelerator architectures primarily focus on maximizing throughput, improving energy efficiency, and reducing average-case inference latency. Although these objectives are essential for cloud computing, edge AI, and high-performance embedded systems, they are often achieved through dynamically managed execution models, adaptive memory hierarchies, speculative pre-fetching mechanisms, and runtime arbitration policies. Consequently, most existing accelerators inherently exhibit timing variability, resource contention, and non-deterministic execution behavior, which significantly complicate Worst-Case Execution Time (WCET) analysis and limit their applicability in safety-critical real-time systems.

Table 2.2 summarizes the characteristics of the reviewed state-of-the-art architectures and contrasts them with the proposed TT-VTA, STT-VTA, and STA-CP architectures. The comparison highlights several important trade-offs among throughput, determinism, flexibility, power efficiency, scalability, and memory-management strategies.

The comparative analysis therefore reveals a fundamental research gap in current AI accelerator architectures. Existing approaches generally prioritize either:

- throughput and computational efficiency,
- energy optimization,
- or flexibility and scalability,

while deterministic execution behavior and temporal predictability remain largely unaddressed.

More specifically, none of the reviewed architectures simultaneously provide:

- deterministic inference execution,
- explicit WCET analyzability,
- scalable superscalar parallelism,
- deterministic memory management,
- interference-free resource utilization,
- and schedule-aware memory-latency hiding.

The architectures proposed in this thesis address these points through a unified time-triggered execution paradigm that combines static scheduling, deterministic resource management, superscalar execution, deterministic caching, and schedule-aware pre-fetching.

The TT-VTA architecture introduces deterministic time-triggered inference execution by replacing the event-triggered dependency-management model of the conventional VTA with statically generated schedules and deterministic DRAM-access coordination. This enables predictable execution timing and eliminates uncontrolled memory interference.

The STT-VTA architecture extends this concept by introducing deterministic superscalar execution. Through the integration of multiple Compute modules and statically coordinated execution schedules, STT-VTA enables parallel tensor processing while preserving deterministic execution behavior and temporal isolation of shared resources.

Finally, the STA-CP architecture further extends the deterministic execution model through the integration of deterministic caching and schedule-aware pre-fetching mechanisms. Unlike conventional adaptive cache hierarchies and speculative pre-fetchers, the STA-CP architecture statically determines all cache-management and pre-fetching operations at compile time. This enables memory-latency hiding and improved pipeline utilization while preserving deterministic timing behavior and analyzable execution semantics.

Consequently, the proposed TT-VTA, STT-VTA, and STA-CP architectures establish a transition from conventional throughput-oriented AI acceleration toward deterministic and WCET-analyzable AI inference suitable for deployment in safety-critical

real-time systems. By combining predictable scheduling, deterministic memory behavior, interference-free resource management, and scalable tensor parallelism, the proposed architectures provide a unified framework for enabling dependable AI acceleration in environments where strict timing guarantees are mandatory.

TABLE 2.2: Comparative Overview of SotA- and Proposed-Acceleration Architectures

Architecture	Target Platform	Throughput	Determinism	Power	Scalability & Flexibility	Model Accuracy	Memory Optimization Strategy
FINN [59]	FPGA	High	No	Low	Low	Reduced	Static dataflow / local buffers
LogicNets [73]	FPGA	High	No	Low	Low	Reduced	Truth-table based local memory
BranchyNet [80]	CPU/GPU + Cloud	High	No	High	Medium	Acceptable	Dynamic workload partitioning
Google TPU [23]	ASIC	Very High	No	Medium	Medium	Preserved	Dynamic multi-level cache hierarchy
Eyeriss [36, 60]	ASIC	High	No	Low	Medium	Preserved	Row-stationary dataflow with scratchpads
MAERI [37]	ASIC	High	No	Medium	Medium	Preserved	Reconfigurable on-chip network and buffers
NVDLA [43]	ASIC/FPGA	High	No	Medium	Medium	Preserved	Dynamic caching and buffering
DeepCache [24]	Mobile GPU/CPU	High	No	Medium	High	Slightly Reduced	Dynamic runtime caching
ROMANet [25]	DNN Accelerators	High	Partial	Low	Medium	Preserved	Reuse-aware DRAM optimization
DNPU [88]	FPGA/ASIC	High	No	Low	Medium	Preserved	Dynamic pre-fetching
TT-VTA (Proposed)	FPGA	Medium	Yes	Low	High	Preserved	Compile-time, statically scheduled (DRAM-operations)
STT-VTA (Proposed)	FPGA	High	Yes	Low	High	Preserved	Compile-time, statically scheduled (DRAM-operations)
STA-CP (Proposed)	FPGA	High	Yes	Low	High	Preserved	Compile-time, statically scheduled (DRAM-, Cache-, and pre-fetching-operations)

Chapter 3

Development Process of Deterministic AI Hardware Accelerator

This chapter presents the overall development methodology employed throughout this thesis for transforming the conventional event-triggered Versatile Tensor Accelerator (VTA) into deterministic time-triggered AI accelerator architectures suitable for safety-critical real-time systems. Unlike the subsequent architecture-specific chapters, which focus on the hardware architectures and models, the underlying scheduling algorithms, and their experimental evaluation, this chapter provides a comprehensive overview of the integrated development flow and toolchain that underpin their realization. In particular, it describes the systematic process used to analyze AI models, derive deterministic execution schedules, generate architecture configurations, synthesize FPGA implementations, and deploy the resulting accelerator architectures.

The complete development workflow is illustrated in Figure 3.1. The process begins with the generation and training of AI models using conventional deep-learning frameworks (i.e., Step 1) and concludes with the deployment and runtime execution of the synthesized accelerator architecture on FPGA hardware (i.e., Step 7). A central component of this workflow is the proposed Scheduler with Built-in Simulation (i.e., Step 4), which performs instruction-level analysis, timing extraction, deterministic scheduling, and architecture-specific configuration generation required for FPGA synthesis and deployment.

As shown in Figure 3.1, the workflow is divided into two primary phases. The first phase, referred to as the *Analysis, Design, and Development Phase*, focuses on AI-model analysis, instruction extraction, timing simulation, deterministic scheduling, and FPGA-architecture generation. The second phase, referred to as the *Testing and Deployment Phase*, focuses on deploying the generated architecture onto FPGA hardware and validating its execution behavior using real inference workloads and runtime input data.

Overall, the complete workflow consists of six major development steps (i.e., Steps 1–6), each contributing to the realization of deterministic and predictable AI acceleration, followed by a verification and validation step (i.e., Step 7) for evaluating the developed accelerator architecture.

3.1 Step 1: AI Model Development and Training

The development process begins with the creation and training of AI models using high-level deep-learning frameworks such as TensorFlow [62] or PyTorch [63]. These

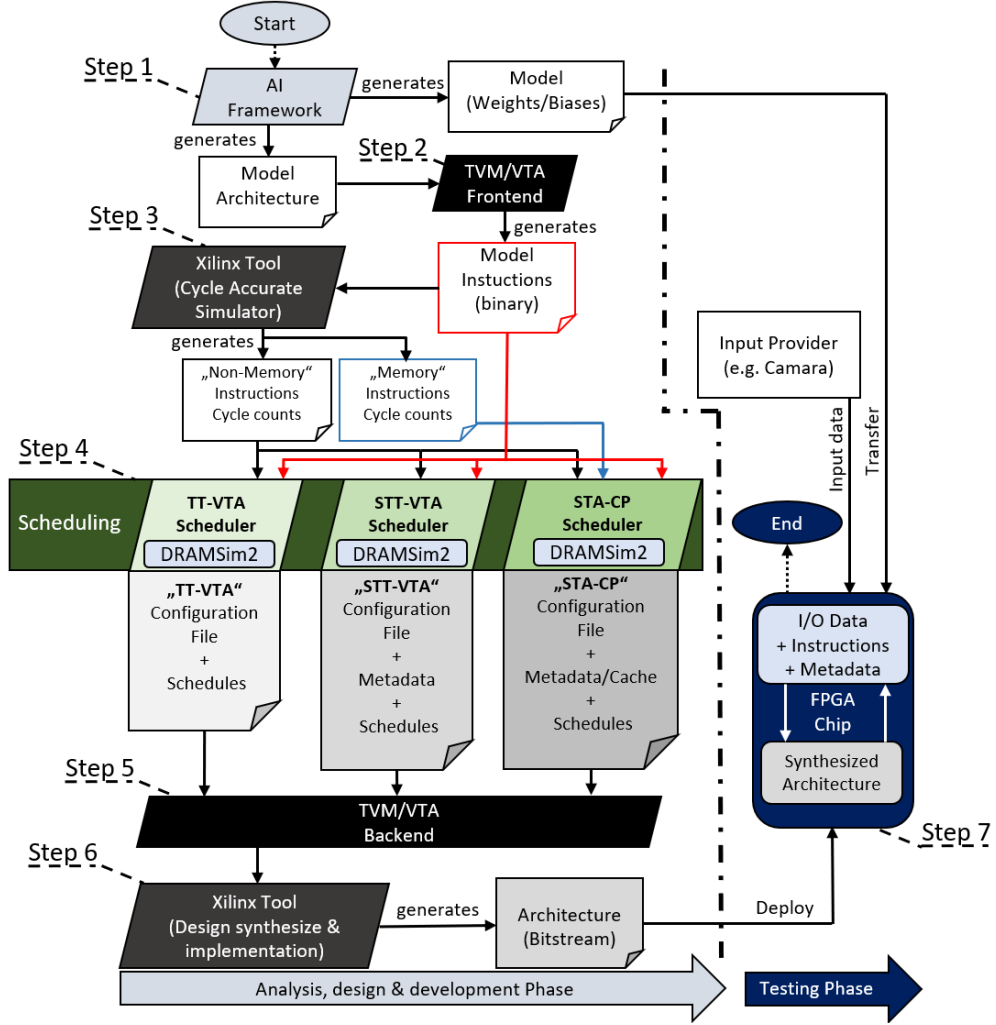


FIGURE 3.1: Development Toolchain for the Proposed Time-Triggered AI Accelerators

frameworks provide powerful abstractions for constructing neural-network architectures, training them using large datasets, and generating optimized parameter sets consisting of weights and biases.

At this stage, the model designer focuses exclusively on the functional behavior and inference accuracy of the neural-network model. The model architecture defines the structure of the neural network, including the organization of layers, tensor dimensions, activation functions, convolution operations, and inter-layer dependencies. Simultaneously, the training process generates the optimized weights and biases that parameterize the model and determine its inference behavior.

Two primary outputs are therefore generated during this stage. The first output is the *model architecture*, which formally describes the neural-network graph and the sequence of tensor operations required during inference execution. The second output consists of the trained model parameters, namely the weights and biases that will later be transferred to the FPGA accelerator during runtime inference execution.

It is important to emphasize that the focus of this thesis is not on AI-model training, optimization, or improving model accuracy. Instead, the neural-network models are assumed to be already trained, validated, and optimized within their respective AI frameworks, and are therefore treated solely as inputs to the proposed

development workflow. Consequently, the primary objective of this work is to enable deterministic and predictable inference execution on specialized AI hardware accelerators while preserving the accuracy and functional behavior of the original neural-network models.

Once the neural-network architecture has been finalized, the model graph is forwarded to the compiler infrastructure responsible for translating the high-level AI representation into low-level tensor instructions executable by the target accelerator architecture.

3.2 Step 2: TVM/VTA Compilation and Instruction Generation

Following model generation, the neural-network architecture is processed using the Apache TVM frontend [67]. The TVM compiler parses the neural-network graph generated by TensorFlow or PyTorch and transforms the model into a sequence of tensor operations compatible with the Versatile Tensor Accelerator (VTA) [19, 20].

During this stage, TVM performs several important model optimizations, including graph-level transformations, tensor-operation fusion, operator optimization, and hardware-aware code generation. These optimizations aim to improve execution efficiency while preserving the semantic correctness of the original AI model.

The output of this stage is a sequence of VTA-compatible instructions represented in binary format. These instructions describe all operations required during inference execution, including memory transfers, tensor computations, synchronization dependencies, and result-storage operations.

The generated instruction stream primarily consists of three instruction categories. Load instructions are responsible for transferring input tensors, weights, and biases from DRAM into local accelerator buffers. Compute instructions describe tensor-processing operations such as matrix multiplication and accumulation. Store instructions transfer intermediate or final inference results back to memory.

3.3 Step 3: Behavioral Simulation and Timing Extraction

One of the fundamental requirements for deterministic AI acceleration is the ability to accurately estimate the execution latency of all accelerator instructions before deployment. Since the proposed architectures rely entirely on statically generated schedules, precise cycle-level timing information must be extracted during the design phase.

To achieve this objective, the proposed workflow integrates two complementary simulation environments: the Xilinx cycle-accurate behavioral simulator and DRAM-Sim2 memory simulator. Together, these simulation frameworks enable accurate modeling of both compute-bound and memory-bound execution behavior.

The Xilinx cycle-accurate simulator is used to perform RTL-level behavioral simulation of the VTA architecture prior to hardware synthesis. During this stage, the generated instruction stream is utilized as a testbench input to emulate the execution sequence of the target AI model. This simulation enables detailed observation of instruction execution ordering, signal transitions, module synchronization, and communication behavior between the functional units (i.e., modules) of the accelerator.

Behavioral simulation serves several important purposes. First, it validates the functional correctness of the generated instruction stream and verifies proper synchronization between the accelerator modules. Second, it enables cycle-level tracing of tensor operations and inter-module interactions. Most importantly, it allows extraction of the execution latency of non-memory instructions, including tensor-computation and arithmetic operations as well as on-chip memory accesses latencies.

However, behavioral simulation alone cannot accurately represent the latency of external DRAM operations. Since DRAM exists outside the FPGA fabric, several important timing characteristics cannot be faithfully modeled at RTL level. These include memory-controller arbitration, row-buffer conflicts, bank activation delays, refresh operations, and bus-transfer latency.

To address this limitation, the workflow integrates DRAMSim2 [98], a cycle-accurate DRAM simulator capable of modeling realistic off-chip memory behavior. DRAMSim2 accurately estimates the latency of all memory-related instructions executed by the accelerator, including feature-map, input, weight, and bias transfers as well as intermediate-result storage operations.

The integration of both simulation environments provides a complete timing characterization of the target AI model. The extracted timing information subsequently forms the basis for deterministic scheduling, architecture configuration, pipeline analysis, resource-contention analysis, and WCET estimation.

3.4 Step 4: Deterministic Scheduling and Architecture Configuration

The Scheduler with Built-in Simulation represents the central component of the proposed development framework. It receives the generated instruction stream together with the cycle-level timing information extracted during the previous simulation stage. Using this information, the scheduler emulates the execution behavior of the target AI model while simultaneously generating deterministic timing schedules and architecture-specific configuration files.

The **TT-VTA scheduler** [102] focuses on deterministic execution planning for the TT-VTA architecture. It analyzes instruction dependencies, determines instruction-triggering instances, and generates static schedules defining the exact execution instants of the Load, Compute, and Store modules. In addition, the scheduler determines DRAM-access locking windows required for interference-free memory execution.

The **STT-VTA scheduler** [103] extends this functionality to support deterministic superscalar execution within the STT-VTA architecture. During this stage, the scheduler analyzes the instruction stream to identify pending instructions caused by busy functional units. Using this information, the scheduler determines the number of compute modules required to improve inference throughput while preserving deterministic timing behavior. Furthermore, the scheduler redistributes instructions among the functional units and updates instruction metadata to support concurrent execution paths.

The **STA-CP scheduler** [104] further extends the functionality by integrating deterministic caching and schedule-aware pre-fetching mechanisms for the STA-CP architecture. At this stage, the scheduler performs memory-dependency analysis, cacheability analysis, cache-size estimation, pre-fetch candidate selection, and cache-metadata generation. Unlike conventional dynamic cache hierarchies and speculative

pre-fetchers, all caching and pre-fetching operations are statically determined during compile time. This enables memory-latency hiding while preserving temporal determinism and analyzable execution behavior.

3.5 Step 5: Architecture Generation and Hardware Adaptation

Once the scheduling and simulation stages are completed, the generated timing schedules, instruction metadata, and architecture-configuration files are forwarded to the TVM/VTA backend for architecture generation and hardware adaptation. At this stage, the accelerator architecture is automatically modified according to the configuration requirements determined by the Scheduler with Built-in Simulation.

This phase represents the transition from analytical scheduling and timing analysis toward a physically realizable hardware architecture that can subsequently be synthesized and deployed on FPGA platforms. The generated configuration files define the required number and types of functional units, the organization of memory subsystem, the dispatching infrastructure, and the communication topology between the accelerator modules. Consequently, the architecture becomes automatically specialized according to the execution characteristics and resource requirements of the analyzed AI model.

Based on these configuration files, the TVM/VTA backend modifies the original VTA architecture by adapting the high-level C++ descriptions that define the functionality and execution behavior of the individual accelerator components, including Fetch, Load, Compute, Store modules to facilitate the consideration of metadata. In parallel, the `architecture_design.tcl` file is automatically updated to reflect the generated architectural configuration. This includes modifying the interconnect structure, instantiating additional functional units, integrating TT-dispatcher pipelines, configuring cache memories and intermediate buffers, and adapting the communication topology between the accelerator modules. The generated TCL script therefore represents the complete structural specification of the target accelerator architecture and serves as the primary automation interface for the FPGA implementation flow.

The performed architectural modifications depend on the selected accelerator variant and the optimization decisions generated during scheduling and simulation. For the TT-VTA architecture, the backend integrates the Time-Triggered (TT) Dispatcher pipelines and removes the original synchronization-based dependency queues used in the conventional event-triggered VTA architecture. This transformation enables accelerator operations to be executed according to statically generated timing schedules rather than runtime synchronization events.

For the STT-VTA architecture, the backend further extends the design through the instantiation of multiple compute modules to support deterministic superscalar execution. In addition, the instruction-routing infrastructure, module identifiers, and inter-module communication mechanisms are adapted to support parallel execution and deterministic data movement between the functional units.

For the STA-CP architecture, the backend additionally integrates on-chip cache memory and additional intermediate buffers between modules to support the pre-fetching strategy. These extensions include cache-aware memory-routing logic, additional communication buffers, and the metadata support required for cache management and schedule-aware pre-fetching operations.

3.6 Step 6: FPGA Synthesis and Hardware Realization

Following architecture generation and adaptation, the produced hardware descriptions of the individual modules and the `architecture_design.tcl` files are forwarded to the Xilinx Vivado Design Suite [101, 105], which serves as the primary FPGA development and synthesis framework throughout this thesis. The generated accelerator modules are represented using high-level C++ descriptions that define the functionality and execution behavior of the individual architectural components. These behavioral models are processed using Vivado High-Level Synthesis (HLS), which automatically translates them into synthesizable Register-Transfer Level (RTL) implementations suitable for FPGA deployment.

Subsequently, the adapted `architecture_design.tcl` file is utilized by the Vivado toolchain to automate the implementation of the generated architecture. The TCL-based automation framework controls module integration, IP generation and configuration, interconnect creation, and the instantiation of the required communication and memory infrastructures according to the architecture generated during the previous stage. Vivado then performs logic synthesis to map the generated accelerator architecture onto FPGA resources such as Look-Up Tables (LUTs), Flip-Flops (FFs), Block RAMs (BRAMs), and DSP slices. This stage enables quantitative evaluation of hardware-resource utilization and verifies whether the generated architecture remains within the available resource limits of the target FPGA platform.

After synthesis, Vivado performs placement and routing, during which the synthesized logic is physically mapped onto the FPGA fabric. Finally, Vivado generates the FPGA bitstream required for deployment onto the target FPGA platform. The resulting bitstream represents the complete hardware realization of the generated deterministic AI accelerator architecture and is subsequently used for runtime deployment and experimental validation.

3.7 Step 7: Deployment and Runtime Execution

Following synthesis and hardware implementation, the generated FPGA bitstream is deployed onto the target FPGA platform. Afterwards, the trained model parameters (i.e., weights and biases), model instructions, and their corresponding metadata are transferred to the DRAM of the FPGA, while the timing schedules are transferred directly to the accelerator's dispatching pipelines of the respective modules.

During operation, input data is transferred to the accelerator through an external interface connected to a host PC, which serves as an image provider and emulates the functionality of a camera source. Afterwards, the accelerator is activated and autonomously executes inference according to the statically generated execution schedules. As a result, the proposed architectures provide predictable inference behavior, deterministic memory access, interference-free execution, and WCET-analyzable timing semantics suitable for safety-critical real-time systems.

Chapter 4

Time-Triggered VTA (TT-VTA)

This chapter provides a comprehensive overview of the system model of the Time-Triggered VTA (TT-VTA) architecture, together with the scheduler featuring a built-in simulator (TT-VTA Scheduler) used to generate the static timing schedules required for deterministic execution. Section 4.1 introduces the integration of the Time-Triggered (TT) dispatcher with the VTA functional modules, forming the foundation of the TT-VTA architecture. Section 4.2 describes the hardware intellectual property (IP) of the TT-dispatcher, which enables the inference execution based on precomputed timing schedules within the accelerator. Section 4.3 presents the TT-VTA scheduler, developed to emulate the execution behavior of the VTA architecture and generate timing schedules for the TT-VTA hardware. In Section 4.4, the proposed TT-VTA architecture is evaluated to demonstrate its capability to provide deterministic and predictable AI inference execution. Finally, Section 4.5 concludes the chapter by discussing the key observations, limitations, and potential directions for further architectural improvements.

4.1 System Model of Time-Triggered VTA

To mitigate the limitations inherent in the conventional VTA, we extended the VTA architecture by integrating a TT-dispatcher into the design, as in Figure 4.1.

This integration aims to enhance the determinism of the VTA architecture by ensuring predictable inference execution, bounded latency, and interference-free memory access. To achieve this, the original VTA dependency queues and their associated interfaces were removed, as illustrated in Figure 4.1, in contrast to the baseline VTA architecture shown in Figure 2.4. Instead, dedicated slave AXI interfaces were introduced for each functional module and connected to the master AXI interfaces of the proposed Time-Triggered (TT) dispatcher.

The TT-dispatcher centrally coordinates the execution of all functional units according to statically generated timing schedules, thereby eliminating dynamic arbitration and runtime scheduling uncertainties. In addition, the TT-dispatcher manages access to shared resources, particularly the external off-chip DRAM, by coordinating communication with other system entities such as the CPU executing a software API. Access to DRAM is granted to the CPU only during predefined time windows in which no memory transactions are scheduled for the TT-VTA architecture. This temporal partitioning mechanism prevents memory contention and ensures predictable execution behavior across the entire system.

Our approach employs a dedicated scheduling pattern, illustrated in Figure 4.2, which defines the temporal execution behavior of the TT-VTA architecture. Each schedule entry contains the following fields:

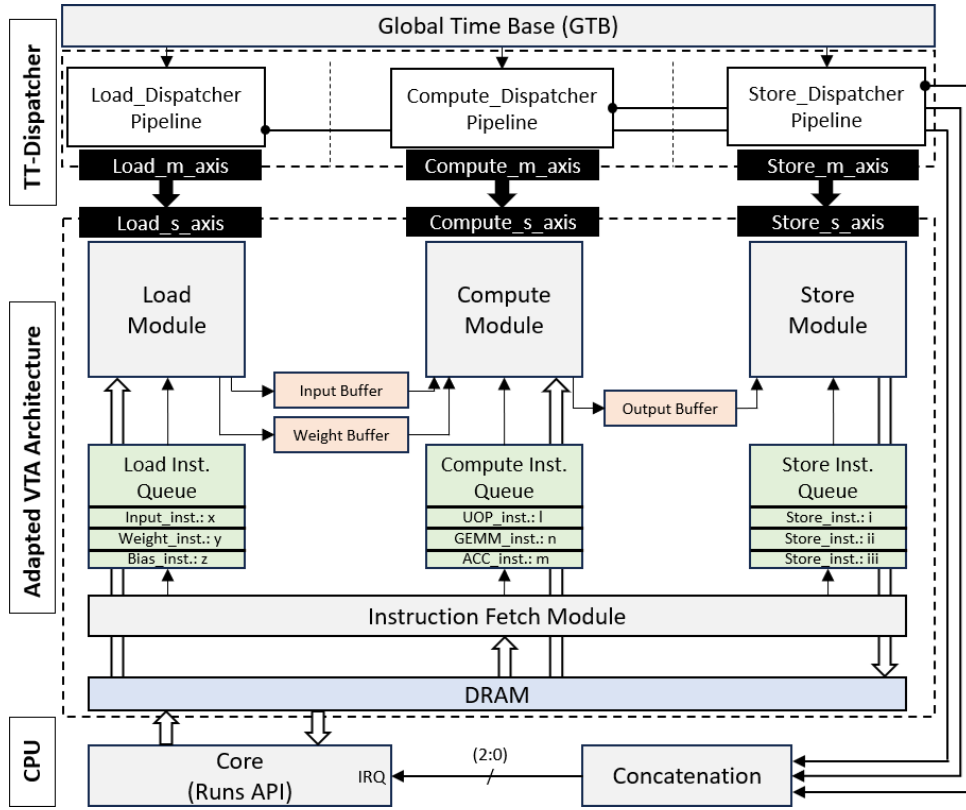


FIGURE 4.1: TT-VTA Architecture

- **NEXT Entry:** points to the next logical entry within the scheduling sequence, enabling sequential or cyclic schedule traversal.
- **PHASE:** specifies the exact global timing instance at which the corresponding functional module is triggered to execute the pending instruction.
- **RESERVED:** allocates 7 bits for future architectural extensions and additional scheduling metadata.
- **DRAM Lock:** indicates the accessibility status of the shared DRAM for external entities such as the CPU:
 - **1:** the DRAM is reserved exclusively for TT-VTA memory operations.
 - **0:** the DRAM is available for access by other system entities.

The generated schedules are organized as linked-list scheduling structures, providing flexibility for both single-shot and periodic execution of tensor operations. This structure enables deterministic repetition of inference workloads while supporting adaptable execution patterns for different application requirements.

4.2 Time-Triggered Dispatcher

The Time-Triggered Dispatcher (TT-dispatcher), illustrated in Figure 4.3, is a proprietary hardware intellectual property (IP) developed in VHDL to enable the execution of statically generated timing schedules within the TT-VTA architecture on FPGA platforms. The dispatcher acts as the central coordination unit responsible for temporally orchestrating all functional modules according to predefined execution schedules.

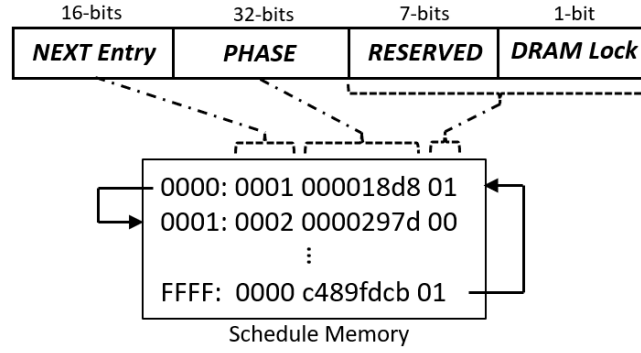


FIGURE 4.2: Scheduling Pattern

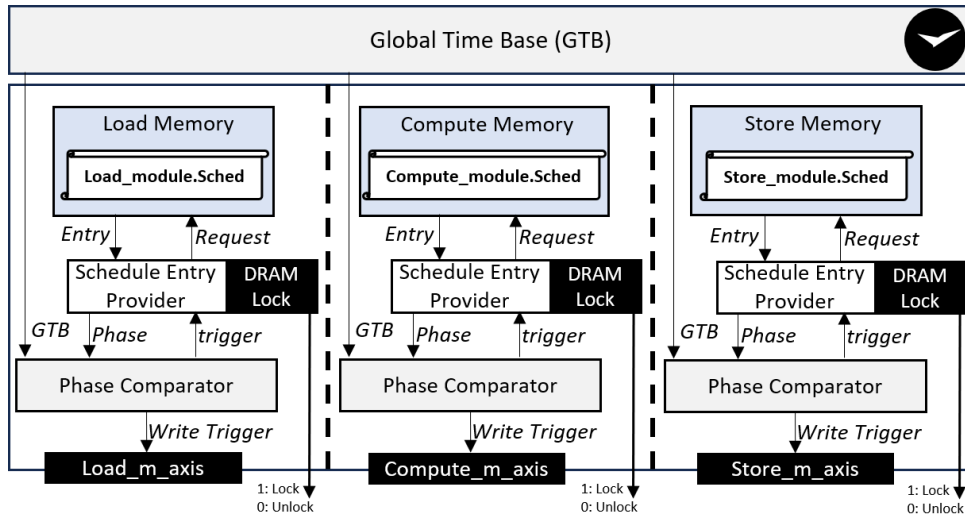


FIGURE 4.3: Time-Triggered Dispatcher Architecture

The TT-dispatcher consists of three identical scheduling pipelines, each dedicated to controlling a specific functional module within the architecture. All pipelines operate synchronously using a shared global timing reference referred to as the *Global Time Base (GTB)* counter.

Each scheduling pipeline comprises the following components:

- **Memory:** stores the generated timing schedule associated with the corresponding functional module.
- **Schedule Entry Provider:** retrieves schedule entries from memory, extracts the phase information, and forwards it to the phase comparator. In addition, the provider extracts the *DRAM Lock* bit and forwards it to the CPU interrupt interface (IRQ) upon receiving a trigger notification from the phase comparator indicating that the execution of the current module is imminent, thereby informing the CPU about the availability status of the shared DRAM. Once the current schedule entry has been executed, the provider loads the subsequent entry using the *NEXT Entry* field.
- **Phase Comparator:** compares the scheduled phase value against the current Global Time Base (GTB) counter value. When both values match, the corresponding functional module is triggered for execution.

- **Master AXI-Stream Interface (m_axis):** initiates AXI-Stream write transactions toward the slave AXI-Stream interface of the connected functional module, thereby signaling instruction execution.

By relying on globally synchronized schedules and deterministic triggering mechanisms, the TT-dispatcher eliminates runtime arbitration and dynamic scheduling overheads, ensuring predictable execution timing and interference-free coordination among functional modules.

4.3 TT-VTA scheduler

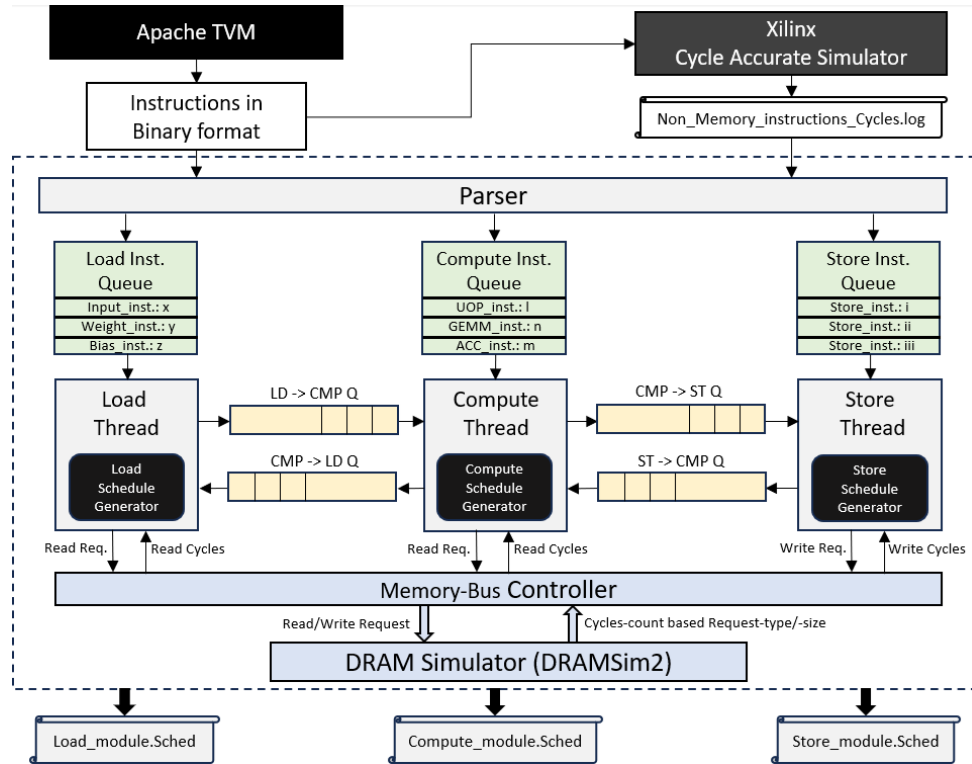


FIGURE 4.4: Scheduler with built-in simulation "TT-VTA Scheduler"

In this work, we developed a C++-based scheduler with built-in simulator derived from the VTA architecture, illustrated in Figure 4.4. The simulator serves two primary purposes: (1) emulating the execution behavior and sequencing of tensor and memory instructions based on the instruction patterns shown in Figure 2.3, and (2) concurrently generating precise module-oriented timing schedules for the corresponding functional units within the architecture.

By leveraging the capabilities of Apache TVM, the instructions executed by the VTA during AI inference can be captured and logged. Enabling the debug mode within the TVM compiler generates a text file containing the binary representation of the executed instructions, referred to as the *Instructions in Binary Format*. These instruction binaries are subsequently used as a testbench input for the Xilinx behavioral simulator to emulate the execution of the conventional VTA architecture. This process enables the extraction and recording of cycle counts associated with instruction execution across the different VTA functional modules.

In our methodology, VTA instructions are classified into two categories: (1) memory-related and non-memory-related instructions, and (2) dependent and independent instructions. The latter classification is discussed in detail in the context of the proposed simulator architecture. Initially, however, the focus is placed on distinguishing between memory and non-memory operations.

As previously discussed in section 3.3, Xilinx behavioral simulation is employed to capture the cycle counts associated with instruction execution. Nevertheless, the behavioral simulator cannot accurately model memory-related instructions, since it does not incorporate the timing behavior of physical off-chip DRAM during simulation. Instead, Block RAM (BRAM) is used as a virtual memory solely to validate the correctness of instruction sequencing and functional behavior within the VTA architecture.

Consequently, the cycle counts obtained from behavioral simulation are accurate only for non-memory-related instructions, such as tensor computation and on-chip operations. In contrast, the timing behavior of memory-related instructions cannot be reliably represented under this setup, as BRAM access characteristics differ significantly from those of external DRAM.

To address this limitation, cycle counts for memory-related instructions are obtained using the cycle-accurate DRAM simulator DRAMSim2, which is integrated into the proposed TT-VTA scheduler. This hybrid simulation approach enables accurate modeling of both computation-bound and memory-bound operations, thereby providing precise timing information required for deterministic scheduling and WCET-oriented analysis.

At this stage, all executable model instructions, together with the corresponding timing information for computation-bound instructions, have been collected to enable the TT-VTA scheduler to accurately emulate the instruction execution sequence of a given AI model. In the following, we present the distinct phases of the simulation process, see Figure 4.4:

1. **Configuration Phase:** Before the simulation begins, a simulation configuration file is set up with a single VTA architecture.
2. **Initialization Phase:** The simulator instantiates several components based on the simulation configuration file. These components include module threads, instruction queues, and dependency queues which are essential for orchestrating the execution flow.
3. **Pre-Simulation Phase:** The Parser customizes the instructions extracted from the text file by incorporating the logged cycle counts from the log file for the non-memory instructions. Simultaneously, it serves as the fetch module, segregating and arranging the different instructions into distinct vector queues associated with their respective modules threads, based on the instruction OPCODE information according to the instruction.
4. **Simulation Phase:** The Load, Compute, and Store threads operate according to the logic detailed in Algorithm 1. These threads replicate the parallelism inherent in the respective modules within the VTA architecture. They retrieve their corresponding instructions from the designated vector queues, while also analyzing the individual instruction patterns. In case an instruction specifies a dependency, these threads await the requisite trigger before executing the queued instruction. Conversely, they proactively push a dependency if instructed to do so. Moreover, these threads initiate the DRAMSim2, supplying

Algorithm 1 : TT-VTA → Modules Threads Logic

```

1: module_schedule.open(/module_schedule.sched)
2: for  $c = 0; c \leq inst\_queue.size(); c++$  do
3:   inst=inst_queue(c)
4:   prev_inst=inst_queue(c-1)
5:   if  $inst.pop = True$  then
6:     Wait until (!pop_queue.empty())
7:     dep_inst=pop_queue.read()
8:     inst.start_instant=dep_inst.finish_instant
9:   else
10:    inst.start_instant=prev_inst.finish_instant
11:   end if
12:   if  $inst.type = memory$  then
13:     if  $mem\_controller$  is blocked then
14:       Wait until ( $mem\_controller.free()$ )
15:       inst.start_instant= $mem\_controller.freed\_instant$ 
16:     end if
17:     mem_cycles=DRAM(type=read/write,offset,size={y,x,stride})
18:     inst.finish_instant=inst.start_instant+mem_cycles
19:   else
20:     inst.finish_instant=inst.start_instant+inst.logged_cycles
21:   end if
22:   module_schedule.append(inst.start_instant)
23:   if  $inst.push = True$  then
24:     push_queue.write(inst)
25:   end if
26: end for
27: module_schedule.close()

```

the memory transaction characteristics, including the instruction type, data size for read or write operations, and memory offset. Subsequently, they await the reporting of the cycle counts essential for the execution of the instruction on the physical hardware. Ultimately, these threads possess the capability to generate comprehensive timing schedules for all instructions, seamlessly integrating them into dedicated schedule files. Before proceeding into the detailed explanation, it is essential to clarify some key terms that will be used throughout the discussion to enhance understanding of the logic.

Declarations:

- **inst**: denotes the instruction currently under evaluation
- **inst_queue**: represents the storage location for all instructions associated with the respective module
- **inst.pop**: indicates that the current instruction depends on another instruction from a different module
- **inst.push**: signifies that another instruction in a different module is dependent on the current one
- **pop/_push_queues**: constitute intermediary mutex queues employed to synchronize the operational sequence between threads, effectively emulating the dependency queues within the actual VTA architecture.
- **c**: is an instruction counter

Modules Threads logic as in Algorithm 1:

- (a) **(line 1)**: The algorithm for module threads initiates by opening the module schedule file.
- (b) **(lines 2-4)**: For each instruction in the queue, the current instruction and its predecessor instruction of the same type are identified.
- (c) **(line 5-6)**: If the pop flag of the current instruction is true (i.e., current instruction depends on another instruction from a different module), it waits until the source module's pop queue is not empty (i.e., dependency is satisfied).
- (d) **(lines 7-8)**: It then retrieves the dependent instruction and the current instruction's start instant is set to the dependent instruction's finish instant.
- (e) **(line 10)**: In case the pop flag is false (i.e., current instruction does not depend on any other instructions), the current instruction's start instant is set to the predecessor instruction's finish instant (i.e., sequential scheduling).
- (f) **(lines 12-15)**: For memory-type instructions, if the memory controller is blocked, it waits until the memory controller is free, setting the start instant accordingly.

Note: The simulator ensures sequential scheduling of memory instructions (i.e. load then store, or store then load). This approach is essential because the TT-VTA architecture effectively mitigates DRAM interference from concurrent load and store operations during inference execution.

- (g) **(line 17):** For memory instructions, The threads initiate the DRAMSim2, supplying the memory transaction characteristics, including the instruction type, data size for read or write operations, and memory offset. Subsequently, they await the reporting of the cycle counts essential for the execution of the instruction on the physical hardware.
 - (h) **(line 18):** The memory cycles are, then, added to the start instant to determine the finish instant of the instruction.
 - (i) **(line 20):** For non-memory instructions, the finish instant is computed by the addition of the start instant and the logged cycles that are collected before the simulation from the Xilinx cycle-accurate simulator.
 - (j) **(line 22):** The start instant of the instruction is then appended to the module schedule.
 - (k) **(lines 23-24):** If the push flag is true, the current instruction is pushed to the respective module's push queue.
 - (l) **(lines 26-27):** The loop iterates for each instruction, and the module schedule is closed at the end.
5. **Post-Simulation Phase:** The TT-dispatching pipelines are instantiated, while the original dependency queues are removed, as illustrated in Figure 4.1. Finally, the `architecture_design.tcl` file is adapted to reflect the updated architectural configuration, including the modified interconnect structure and the integration of the TT-dispatcher pipelines. The TCL file is then utilized by the Xilinx Vivado toolchain to automatically synthesize, implement the design architecture accordingly.

Finally, the TT-VTA architecture is synthesized and implemented using Vivado-Xilinx tools, producing a bitstream file that can be used to configure the FPGA upon deployment. At this stage, the TT-VTA architecture is ready for testing and validation.

4.4 TT-VTA Evaluation

We conducted a comparative evaluation on a Xilinx Zynq UltraScale+ ZCU104 FPGA platform by deploying a convolutional neural network based on the ResNet-18 classification model on both the conventional VTA and the proposed TT-VTA. Using the same model and identical input images across both architectures, the objective was to validate the performance of TT-VTA and assess its ability to maintain deterministic execution by avoiding interference while still meeting predefined inference timing constraints.

For this evaluation, we defined two test scenarios, as summarized in Table 4.1: one in which DRAM interference is introduced by a non-safety-critical CPU-based application, and another without such interference. The interfering workload is implemented as an API executing on the CPU that accesses shared memory (i.e., DRAM) concurrently with the inference execution.

Test-Cases	VTA	TT-VTA
(1) Without CPU interference	51ms	53.2ms
(2) With CPU Interference	77.03ms	53.2ms

TABLE 4.1: Conventional VTA & TT-VTA Test Cases

- **Test-Case 1: Without CPU Interference**

During the evaluation, the conventional VTA successfully classified all input images, achieving an average inference latency of approximately 51ms per sample. The proposed TT-VTA architecture produced identical classification results across all evaluated categories, thereby preserving the functional correctness and inference accuracy of the original VTA. However, the TT-VTA exhibited a slightly higher average inference latency of approximately 53.2ms per sample.

This increase in latency was expected and primarily results from the deterministic execution mechanisms introduced by the TT-VTA architecture. Specifically, the generated execution schedules are constructed using Worst-Case Execution Time (WCET)-oriented timing assumptions for the individual instructions, ensuring predictable and interference-free execution. In addition, the integration of the Time-Triggered Dispatcher (TT-dispatcher) introduces a controlled scheduling overhead associated with module synchronization and time-triggered instruction dispatching. Although these mechanisms slightly increase inference latency, they provide deterministic execution behavior and timing predictability.

- **Test-Case 2: With CPU Interference**

However, when the API introduced interference at varying frequencies ranging from 16Hz to 16kHz during different evaluation scenarios by concurrently issuing DRAM read/write operations with data sizes of 256, 512, and 1024 bytes while the VTA and TT-VTA were executing inference tasks, the conventional VTA exhibited significant execution-time variability, as illustrated in Figure 4.5. Under these interference conditions, the average inference latency of the conventional VTA increased from 51.14ms up to 77.03ms per sample, depending on the interference intensity and memory-access pattern.

In contrast, the TT-VTA maintained a constant average inference latency of 53.2ms across all evaluated scenarios, demonstrating stable and deterministic execution behavior despite the presence of concurrent memory traffic. This robustness is achieved through the time-triggered scheduling mechanism integrated within the TT-VTA architecture. Specifically, the TT-dispatcher explicitly informs the CPU about predefined DRAM-access windows during which non-safety-critical tasks are permitted to access shared memory resources. Outside these windows, DRAM access is restricted to prioritize inference-related operations executed by the TT-VTA. Consequently, memory contention and interference are effectively eliminated, ensuring predictable inference timing and temporal isolation.

These results underscore the effectiveness of the proposed approach, demonstrating that the TT-VTA architecture achieves predictable and deterministic execution behavior under varying interference conditions.

In addition to timing predictability, it is equally important to evaluate the hardware cost associated with introducing deterministic execution mechanisms. Therefore, Figure 4.6 compares the resource utilization of the TT-VTA architecture against the baseline VTA implementation on the Xilinx ZCU104 platform. The figure reports the percentage of utilized FPGA resources, including Look-Up Tables (LUTs), Flip-Flops (FFs), Block RAMs (BRAMs), and Digital Signal Processors (DSPs).

The Xilinx ZCU104 evaluation board provides a maximum of 230,400 LUTs, 460,800 FFs, 11Mb of on-chip Block RAM (BRAM), and 1,728 DSP slices, which

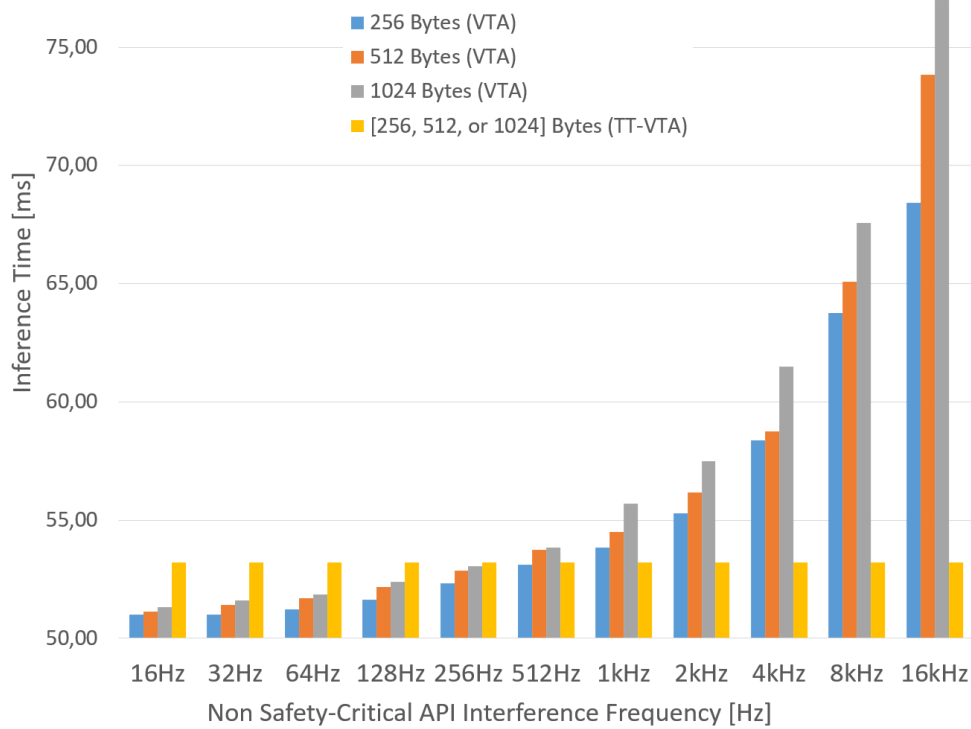


FIGURE 4.5: Test Cases with Different Interference Frequencies & Data-Sizes

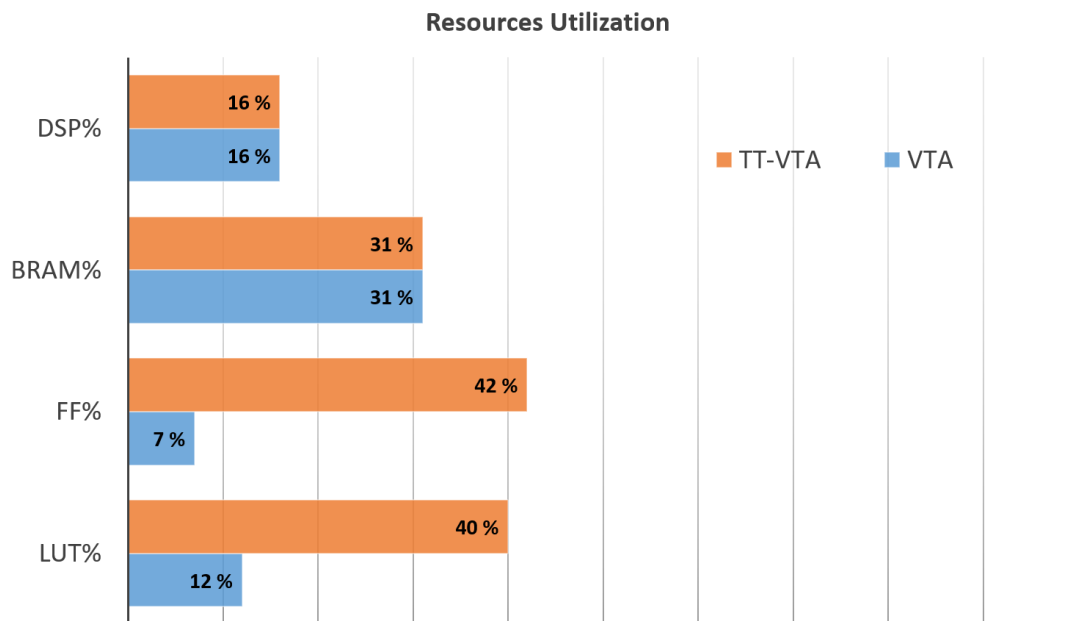


FIGURE 4.6: Resource utilization of VTA and TT-VTA

define the upper hardware-resource limits available for implementing the proposed architectures.

As expected, the TT-VTA architecture exhibits higher resource utilization compared to the conventional VTA architecture. This increase is primarily attributed to the integration of the TT-Dispatcher pipelines, together with their associated scheduling

memories and synchronization logic, including the Global Time Base (GTB) module. These additional hardware components are essential for enabling deterministic time-triggered inference execution and coordinated module activation within the architecture. Nevertheless, the introduced hardware overhead remains acceptable when considering the substantial improvements achieved in temporal determinism, execution predictability, and interference-free operation provided by the TT-VTA architecture.

4.5 Discussion on TT-VTA

In this chapter, we presented a methodology and supporting toolchain for transforming the conventional event-triggered VTA architecture into a deterministic time-triggered AI hardware accelerator, referred to as the TT-VTA. This transformation is achieved through the integration of statically generated timing schedules produced by the TT-VTA scheduler, together with the incorporation of the Time-Triggered Dispatcher (TT-dispatcher) into the VTA architecture.

The experimental results demonstrate that, although the TT-VTA introduces a modest increase in both inference latency and hardware resource utilization compared to the original event-triggered VTA, it provides significant advantages for safety-critical real-time systems. In particular, the proposed architecture ensures predictable execution behavior, deterministic resource utilization, and temporal isolation of shared resources. These properties effectively mitigate memory contention, execution-time variability, and interference-related delays, thereby enabling reliable and analyzable AI inference execution. Consequently, the TT-VTA can be integrated into safety-critical platforms with reduced integration complexity while providing stronger timing guarantees and improved system predictability.

Despite these advantages, optimizing inference latency remains an important objective. Therefore, the subsequent chapters extend the proposed architecture by introducing controlled parallel execution, deterministic caching, and static pre-fetching mechanisms. These enhancements aim to further improve inference performance while preserving the deterministic execution guarantees established by the TT-VTA architecture.

Chapter 5

Superscalar Time-Triggered VTA (STT-VTA)

This chapter provides a comprehensive overview of the system model of the Superscalar Time-Triggered VTA (STT-VTA) architecture, together with the scheduler featuring a built-in simulator (STT-VTA scheduler) used to generate the configuration file and the timing schedules specific to the STT-VTA. In Section 5.1, we describe the compilation and development process for the STT-VTA architecture, emphasizing the functional and design requirements essential to its development. It is important to note that the dynamic configuration of the STT-VTA is model-driven, adapting to the specific AI model in use. Section 5.2 details the upgrades made to the scheduler, which enables (1) the generation of the configuration files that support the automatic development of the STT-VTA, (2) the adaptation of model's instructions metadata, and (3) the creation of timing schedules for the functional units within the STT-VTA. In Section 5.3, the proposed STT-VTA architecture is evaluated to demonstrate its capability to provide deterministic and optimized AI inference execution latency. Finally, Section 5.4 concludes the chapter by discussing the key observations, limitations, and potential directions for further architectural improvements.

5.1 System Model of Superscalar Time-Triggered VTA

In this work, we enhance the performance of the Time-Triggered VTA (TT-VTA) architecture by introducing the Superscalar Time-Triggered VTA (STT-VTA). The STT-VTA, as depicted in Figure 5.1, is intended to incorporate multiple VTA modules into a single architecture, thereby supporting the parallel execution of model instructions. However, Integrating multiple modules within the architecture necessitates addressing additional functional- and design-requirements.

5.1.1 Functional Requirements

The fetch module must not only rely on the instruction type (i.e. OPCODE), to distribute the retrieved instructions from the DRAM to their respective modules, but also consider the targeted modules IDs. Meanwhile, the Load, Compute, and Store modules must be aware of their source and destination modules, as the STT-VTA architecture requires cross-module interactions.

These requirements are met by extending the instruction patterns, see Figure 5.2, to include additional metadata containing information about the module ID where the instruction is executed, the source module ID from which the instruction expects input data, and the destination module ID to which the output data must be transferred.

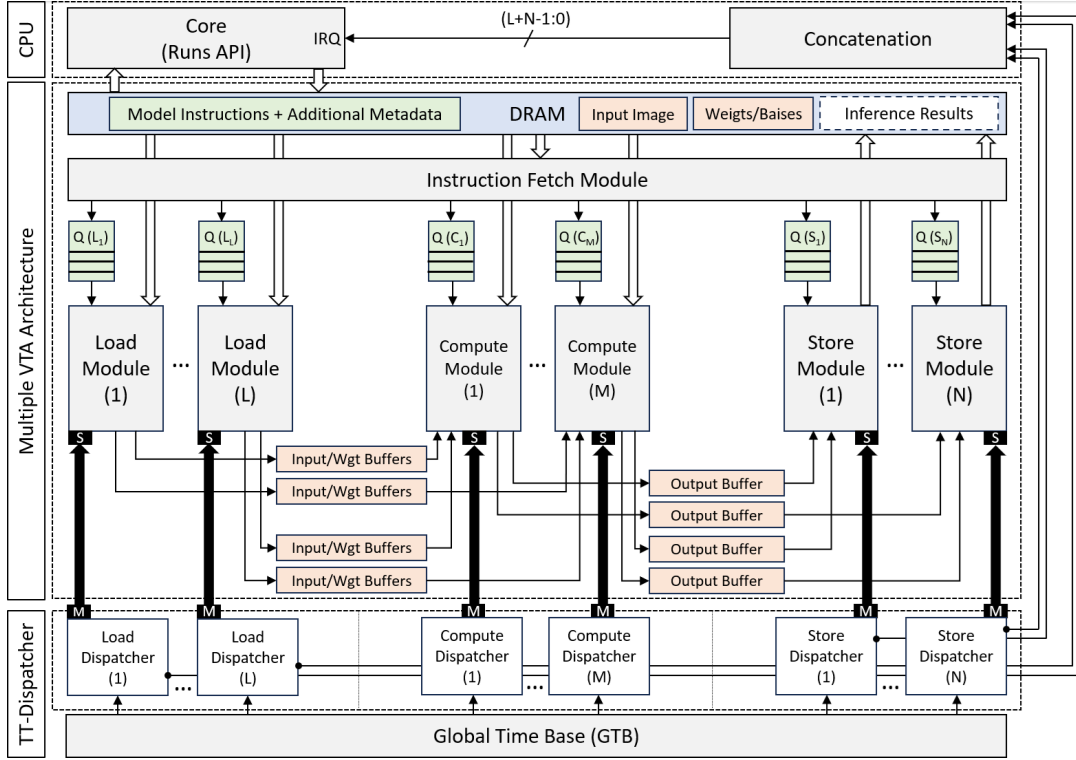


FIGURE 5.1: Superscalar Time-triggered VTA (STT-VTA) Architecture

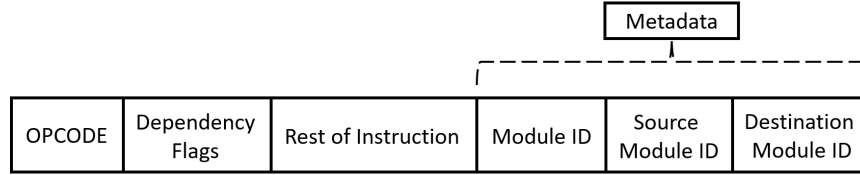


FIGURE 5.2: VTA Instruction Patterns including Metadata

5.1.2 Design Requirements

Additionally, The inclusion of STT-VTA modules are highly dependent on the AI model analysis performed by the STT-VTA scheduler, which is detailed in section 5.2. This analysis is crucial as it determines the optimal configurations and interactions of the multiple modules within the final STT-VTA architecture to ensure efficient parallel executions of instructions.

Upon completion of the scheduling process, the TVM/VTA backend automatically adapts the accelerator architecture according to the selected design configuration. For STT-VTA, this includes generating the required number of functional units, configuring the interconnect structure. The generated architecture is then compiled and synthesized using the Xilinx Vivado framework.

To support this process, the VTA compiler backend provides the Xilinx toolchain with two primary design artifacts, illustrated in Figure 5.3:

- **C++-based module descriptions:** specifying the functional behavior of the accelerator modules,
- **TCL-based architecture scripts:** defining module interconnectivity and automating the FPGA design flow.

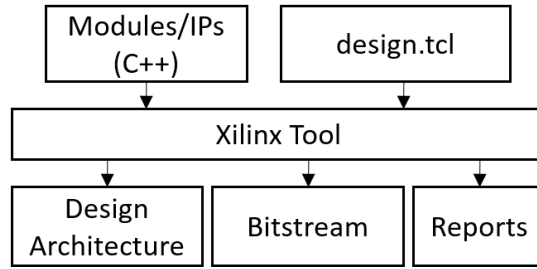


FIGURE 5.3: Xilinx Tool Framework

The behavior of the accelerator modules—including Fetch, Load, Compute, Store, and TT-Dispatcher modules—is specified using high-level C++ descriptions. These descriptions are processed through Xilinx High-Level Synthesis (HLS), which automatically translates the behavioral models into synthesizable RTL implementations suitable for FPGA deployment.

In parallel, the *architecture_design.tcl* script is utilized to automate synthesis, implementation, placement and routing, timing analysis, bitstream generation, and resource reporting within Vivado. TCL (Tool Command Language) additionally enables automatic customization of the generated architecture according to the selected FPGA platform and the architectural parameters produced by the scheduler.

Consequently, the Superscalar Time-Triggered VTA (STT-VTA) system architecture, illustrated in Figure 5.1, undergoes dynamic development and compilation tailored to the specific demands of AI models. This highlights the architecture’s configurability and underscores the practical utility of the approach in optimizing hardware accelerators for diverse AI applications.

5.2 STT-VTA scheduler

We have significantly enhanced the functionality of our scheduler to support the realization of the STT-VTA. This enhancement enables the scheduler to detect potential instruction delays during simulation—specifically identifying any pending instructions caused by the busy state of the modules responsible for processing them. By anticipating these delays, we can optimize the architecture to avoid such issues during the actual runtime of the STT-VTA. This capability is crucial for determining the optimal number of STT-VTA modules required to expedite inference, thereby ensuring efficient and timely execution of AI workloads.

The improved scheduler functionality incorporates two additional components, illustrated in Figure 5.4:

1. **Pending Instruction Detector (PID):** The modules threads are adapted to detect pending instructions in their respective instruction queues, thus notifying the Configuration Manager about their findings.
2. **Configuration Manager:** When a thread alerts the Configuration Manager about a pending instruction, the manager initiates the necessary actions, as shown in Algorithm 2, to handle the instruction and update the simulation’s configuration file. This process starts by terminating the simulation (line 2) and then modifying the metadata for affected instructions (lines 3-6). Specifically, the module ID of the pending instruction is incremented, redirecting it to another module of the same type for the next simulation cycle. Since a new

module will now be part of the simulation, the metadata of the related instructions must be updated to ensure other modules direct data appropriately. Thus, the update includes modifying (1) the destination module ID for dependent instructions and (2) adjusting the source module ID for subsequent instructions. Finally, the Configuration Manager updates the configuration file (line 7), specifying the required number of module threads and their inter-connectivity (i.e., interdependencies through dependency queues), thus instantiating them for the next iteration. The simulation is then reinitialized to apply these changes (line 8).

Algorithm 2 Configuration_Manager Thread

```

1: Configuration_Manager(pending_inst, dep_inst, sub_inst_id){
2:   simulation.terminate()
3:   subsequent_inst=instructions_list[sub_inst_id]
4:   pending_inst.module_id+=1
5:   dep_inst.destination_module_id=pending_inst.module_id
6:   subsequent_inst.source_module_id=pending_inst.module_id
7:   update_sim_config_file()
8:   simulation.run()}
  
```

In the following, we describe the different phases of the simulation and scheduling process implemented in the STT-VTA scheduler, which illustrated in Figure 5.4.

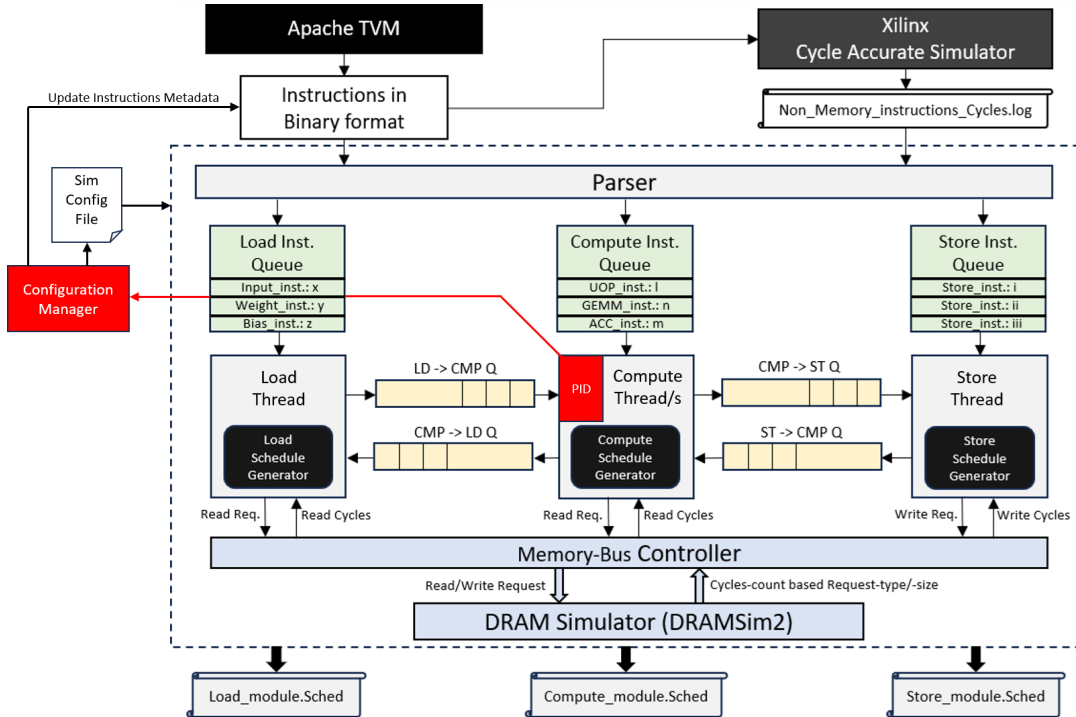


FIGURE 5.4: Scheduler with built-in simulation "STT-VTA Scheduler"

1. **Configuration Phase:** Before the simulation begins, the simulation configuration file is set up with a single VTA architecture, and all additional metadata for the instructions are initialized to default values (i.e., module ID=1, source module ID=1, and destination module ID=1). It is important to note that the configuration file as well as the additional metadata for the instructions will be

updated iteratively based on the findings of the Pending Instruction Detector and the Configuration Manager during subsequent phases of the simulation.

2. **Initialization Phase:** The simulator instantiates several components based on the simulation configuration file. These components include module threads, instruction queues, and dependency queues which are essential for orchestrating the execution flow.
3. **Pre-Simulation Phase:** The Parser segregates and arranges the instructions into distinct instruction queues associated with their respective modules threads, based on the instruction OPCODE and its respective module ID information according to the instruction metadata.
4. **Simulation Phase:** The Load, Compute, and Store threads operate according to the logic detailed in Algorithm 3. Before proceeding into the detailed explanation, it is essential to clarify some key terms that will be used throughout the discussion to enhance understanding of the logic.

Declarations:

- **inst:** denotes the instruction currently under evaluation
- **inst_queue:** represents the storage location for all instructions associated with the respective module
- **inst.pop:** indicates that the current instruction depends on another instruction from a different module
- **inst.push:** signifies that another instruction in a different module is dependent on the current one
- **pop/_push_queues:** constitute intermediary mutex queues employed to synchronize the operational sequence between threads, effectively emulating the dependency queues within the actual VTA architecture.
- **c:** is an instruction counter

Modules Threads logic as in Algorithm 3:

- (a) **(line 1):** The algorithm for module threads initiates by opening the module schedule file.
- (b) **(lines 2-4):** For each instruction in the queue, the current instruction and its predecessor instruction of the same type are identified.
- (c) **(line 5-6):** If the pop flag of the current instruction is true (i.e., current instruction depends on another instruction from a different module), it waits until the source module's pop queue is not empty (i.e., dependency is satisfied).
- (d) **(lines 7-8):** It then retrieves the dependent instruction and checks if its finish instant is greater than or equal to the predecessor instruction's finish instant on the same module.
- (e) **(line 9):** If true, the current instruction's start instant is set to the dependent instruction's finish instant.
- (f) **(lines 11-12):** otherwise, if the instruction is not memory type, the Configuration Manager is called to handle the pending instruction. Consequently,

Algorithm 3 : STT-VTA → Modules Threads Logic

```

1: module_schedule.open(/module_schedule.sched)
2: for  $c = 0; c \leq inst\_queue.size(); c++$  do
3:   inst=inst_queue(c)
4:   prev_inst=inst_queue(c-1)
5:   if  $inst.pop = True$  then
6:     Wait until ( $!pop\_queue[inst.source\_module\_id].empty()$ )
7:     dep_inst = pop_queue[inst.source_module_id].read()
8:     if  $dep\_inst.finish\_instant \geq prev\_inst.finish\_instant$  then
9:       inst.start_instant=dep_inst.finish_instant
10:    else
11:      if  $inst.type \neq memory$  then
12:        Configuration_Manager(inst, dep_inst, inst.sub_inst_id)
13:      end if
14:    end if
15:  else
16:    inst.start_instant=prev_inst.finish_instant
17:  end if
18:  if  $inst.type = memory$  then
19:    if  $mem\_controller$  is blocked then
20:      Wait until ( $mem\_controller.free()$ )
21:      inst.start_instant=mem_controller.freed_instant
22:    end if
23:    mem_cycles=DRAM(type=read/write,offset,size={y,x,stride})
24:    inst.finish_instant=inst.start_instant+mem_cycles
25:  else
26:    inst.finish_instant=inst.start_instant+inst.logged_cycles
27:  end if
28:  module_schedule.append(inst.start_instant)
29:  if  $inst.push = True$  then
30:    push_queue[inst.destination_module_id].write(inst)
31:  end if
32: end for
33: module_schedule.close()

```

the simulation terminates at this stage, and the next iteration begins from the initialization phase immediately after reconfiguring the simulation configuration file.

Note: The simulator prioritizes non-memory-related instructions to optimize inference time by sorting and scheduling them on parallel compute modules.

- (g) **(line 16):** If the pop flag is false (i.e., current instruction does not depend on any other instructions), the current instruction's start instant is set to the predecessor instruction's finish instant (i.e., sequential scheduling).
- (h) **(lines 19-21):** For memory-type instructions, if the memory controller is blocked, it waits until the memory controller is free, setting the start instant accordingly.

Note: The simulator ensures sequential scheduling of memory instructions (i.e. load then store, load then load, or store then store). This approach is essential because the STT-VTA architecture effectively mitigates DRAM interference from concurrent load(s) and/or store(s) operations during inference execution. Thus, this eliminates the need for multiple load or store modules. Instead, they are considered as singular entities within the architecture.

- (i) **(line 23):** For memory instructions, The threads initiate the DRAMSim2, supplying the memory transaction characteristics, including the instruction type, data size for read or write operations, and memory offset. Subsequently, they await the reporting of the cycle counts essential for the execution of the instruction on the physical hardware.
 - (j) **(line 24):** The memory cycles are added to the start instant to determine the finish instant of the instruction.
 - (k) **(line 26):** For non-memory instructions, the finish instant is computed by the addition of the start instant and the logged cycles that are collected before the simulation from the cycle-accurate simulator from Xilinx tool, see Figure 3.1.
 - (l) **(line 28):** The start instant of the instruction is then appended to the module schedule.
 - (m) **(lines 29-30):** If the push flag is true, the current instruction is pushed to the destination module's push queue.
 - (n) **(lines 32-33):** The loop iterates for each instruction, and the module schedule is closed at the end.
5. **Post-Simulation Phase:** The information stored in the simulation configuration file, along with the instructions and their metadata, are extracted and utilized to reconfigure the modules algorithms and architecture_design.tcl files. The reconfiguration process is summarized as follows:
- (a) **TT-Dispatcher pipelines:** According to the gathered information, dispatching pipelines are instantiated. This ensures that the newly required dispatching pipelines within the TT-Dispatcher align with the number of modules in the STT-VTA architecture, see Figure 5.1.
 - (b) **STT-VTA Modules:** The inclusion of new modules in the architecture requires the instantiation of intermediate memories between themselves

and their new source and destination modules, the creation of new interfaces for the dispatching pipelines, and the establishment of corresponding instruction queues. Additionally, the module algorithms are updated to account for the source and destination information stored in the instructions metadata.

- (c) **Fetch Module:** The fetch module interfaces are updated and connected to the instruction queues of all instantiated modules to arrange the instructions into the appropriate instruction queues based on the module information stored in the metadata.

Finally, the STT-VTA architecture is synthesized and implemented using Vivado-Xilinx tools, producing a bitstream file that can be used to configure the FPGA upon deployment. At this stage, the STT-VTA architecture is ready for testing and validation.

5.3 STT-VTA Evaluation

To evaluate the effectiveness of the proposed STT-VTA architecture and validate the developed approach, a comparative analysis was conducted between the STT-VTA, TT-VTA, conventional VTA, and the ARM Cortex-A53 processor. All experiments were performed on a Xilinx Zynq UltraScale+ ZCU104 FPGA platform.

A representative set of AI models, including ResNet-18, ResNet-34, ResNet-50, YOLOv3-tiny, YOLOv3-full, and YOLOv4, was deployed on the ARM Cortex-A53, conventional VTA, and TT-VTA architectures to establish baseline inference performance measurements. Subsequently, the proposed STT-VTA development flow, presented in Figure 3.1, was applied to generate and configure the STT-VTA architecture for each model. The same AI models and input datasets were then executed on the STT-VTA architecture, and the resulting inference latencies were recorded to enable a direct comparison across all evaluated platforms, as summarized in Table 5.1.

TABLE 5.1: Models tested (without DRAM interference)

	ARM (A53)	VTA	TT-VTA	STT-VTA		
<i>Model Type</i>	Avg. Inference latency [ms]	Avg. Inference latency [ms]	Inference latency [ms]	Inference latency [ms]	Inference latency reduction compared to TT-VTA [%]	Resource Utilization increase Compared to TT-VTA [%]
ResNet-18	93.5	51	53.2	42.5	20%	15%
ResNet-34	166.4	91.9	95.04	76.1	20%	15%
ResNet-50	339.7	188.6	192.8	127.2	34%	29%
YOLOv3-tiny	201.3	105.05	109.5	90.9	17%	15%
YOLOv3-full	321.2	170	179.9	120.5	33%	29%
YOLOv4	264.9	139.8	143.3	84.5	41%	44%

The experimental results demonstrate that the STT-VTA architecture consistently outperforms the ARM Cortex-A53, conventional VTA, and TT-VTA architectures in terms of inference latency. However, these performance gains are achieved at the cost of increased FPGA resource utilization, as illustrated in Figure 5.5.

For relatively moderate AI workloads, such as ResNet-18, ResNet-34, and YOLOv3-tiny, the Scheduler with Built-in Simulation (v2) determined that integrating one additional Compute module (i.e., a total of two Compute modules within the STT-VTA architecture) was sufficient to achieve meaningful acceleration. Experimental evaluation confirmed an inference-latency reduction of approximately 17–20% compared to the TT-VTA baseline. According to the Xilinx synthesis reports, this architectural extension introduced an additional FPGA LUT utilization overhead of approximately 15%.

For more computationally intensive models, including ResNet-50, YOLOv3-full, and YOLOv4, additional Compute modules were required to achieve substantial performance improvements. In the case of ResNet-50 and YOLOv3-full, the scheduler instantiated two additional Compute modules (i.e., three Compute modules in total), resulting in inference-latency reductions of approximately 33–34% compared to TT-VTA. For YOLOv4, the architecture required three additional Compute modules (i.e., four Compute modules in total), achieving an inference-latency reduction of approximately 41%. These performance improvements were accompanied by increased FPGA resource utilization, with LUT usage increasing by approximately 29% for ResNet-50 and YOLOv3-full, and by approximately 44% for YOLOv4, relative to the TT-VTA architecture.

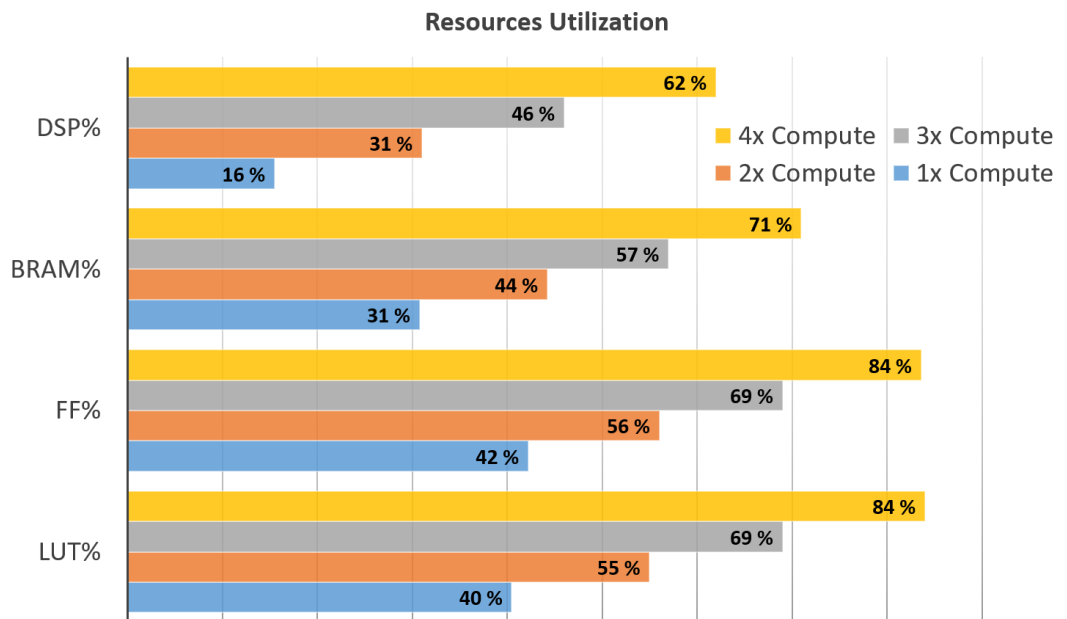


FIGURE 5.5: Resource utilization of STT-VTA with different number of Compute Units

These results highlight the trade-off between inference acceleration and hardware-resource consumption within the STT-VTA architecture. While superscalar execution significantly improves inference latency, especially for computationally demanding AI models, the corresponding increase in FPGA resource utilization must be carefully considered when targeting resource-constrained or power-sensitive embedded platforms.

To further evaluate the robustness of the proposed architecture, the STT-VTA was tested under DRAM-interference conditions alongside the ARM Cortex-A53, conventional VTA, and TT-VTA architectures. The objective of this evaluation was

to assess the ability of the STT-VTA architecture to maintain deterministic and low-latency inference execution in the presence of shared-memory contention.

The interference scenario was generated using an external API executing on a separate processor core, continuously issuing high-frequency DRAM read/write transactions to emulate a non-safety-critical application competing for shared-memory resources during inference execution. The interference frequency was configured to 16kHz with memory transactions of 1024-byte data blocks.

For each architecture and AI model, 100 inference executions were performed to evaluate execution consistency and timing stability. Figure 5.6 presents the resulting inference-latency distributions and standard deviations.

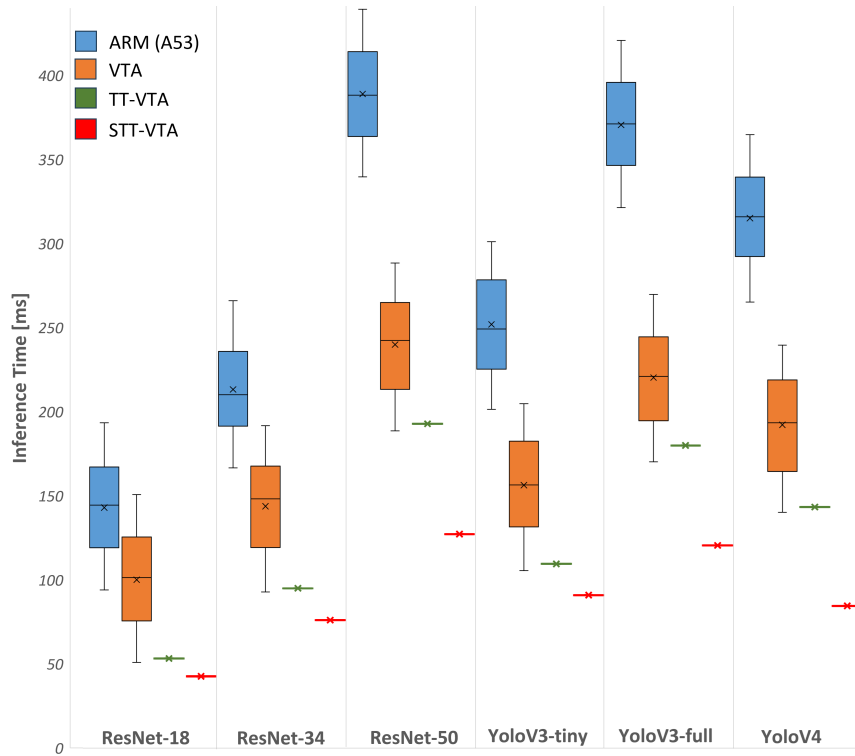


FIGURE 5.6: 100 inference executions under DRAM interference (16 kHz interference frequency and 1024-byte data size)

The results demonstrate that the ARM Cortex-A53 and conventional VTA architectures experienced substantial performance degradation under DRAM interference. Both architectures suffered from unpredictable execution delays caused by memory contention and DRAM busy states. The conventional VTA, in particular, exhibited high sensitivity to interference due to its lack of deterministic memory-access coordination mechanisms.

In contrast, both TT-VTA and STT-VTA maintained stable and deterministic inference latencies across all experimental runs despite the presence of high-frequency DRAM interference. This resilience is achieved through the coordinated DRAM-access management mechanisms integrated within the time-triggered architectures. Specifically, the TT-dispatcher enables the architectures to inform the CPU about time intervals during which DRAM access is reserved for inference-critical operations. During these protected intervals, non-critical DRAM accesses are blocked, thereby eliminating contention and ensuring uninterrupted memory availability for the accelerator.

These results demonstrate that the STT-VTA architecture not only achieves significantly lower inference latency through superscalar execution, but also preserves deterministic timing behavior and interference-free execution under shared-memory contention. Consequently, the proposed architecture is particularly suitable for safety-critical real-time AI systems that require both high-performance inference acceleration and strict timing predictability.

5.4 Discussion on STT-VTA

In this chapter, we presented an approach and an associated toolchain for enabling the deployment of AI models within safety-critical real-time systems while simultaneously achieving accelerated inference execution and deterministic timing behavior. The proposed methodology focuses on the design and configuration of AI hardware accelerators capable of balancing high-performance inference execution with the stringent timing and predictability requirements of embedded safety-critical platforms. In this work, it is assumed that the AI models are already properly trained and validated within their respective software frameworks, allowing the focus to remain on optimizing and controlling inference execution at the hardware-acceleration level.

To achieve these objectives, the proposed approach leverages the STT-VTA (Superscalar Time-Triggered Versatile Tensor Accelerator) architecture, which extends the TT-VTA through superscalar execution and parallel processing of inference instructions. Experimental results demonstrated that the STT-VTA achieves inference-latency reductions ranging from 19% to 41%, as summarized in Table 5.1. In addition to improving performance, the architecture preserves deterministic execution behavior by providing precise control over execution timing and eliminating interference from non-safety-critical operations during inference execution.

These characteristics make the STT-VTA particularly suitable for safety-critical domains such as automotive, aerospace, and medical systems, where unpredictable AI inference behavior and execution-time variability may directly compromise system safety. The presented results demonstrate that complex AI models can be efficiently accelerated on embedded FPGA platforms while maintaining the deterministic and predictable execution behavior required for real-time safety-critical applications.

Although the STT-VTA introduces increased FPGA resource utilization due to the integration of additional Compute modules, the resulting architectural flexibility provides significant advantages. Depending on the complexity of the deployed AI model, the FPGA Look-Up Table (LUT) utilization ranges from approximately 55% to 84%, as illustrated in Figure 5.5. Nevertheless, the architecture supports deterministic sequential or periodic execution of multiple AI models on the same FPGA platform, provided that the deployed models require an equivalent maximum number of functional units.

For example, the architecture can support deterministic inference switching between models such as ResNet-18 for image classification followed by YOLOv3-tiny for object detection, or between larger models such as ResNet-50 and YOLOv3-full. This capability is achieved by preloading the corresponding model instructions and generated timing schedules into the STT-VTA architecture. Consequently, the STT-VTA provides a flexible and reusable acceleration framework capable of supporting multiple inference workloads with minimal architectural reconfiguration while preserving predictable timing behavior. This capability is particularly beneficial for dynamic embedded systems requiring multiple AI functionalities, such as combined

classification and object-detection pipelines operating within the same real-time environment.

Despite the achieved performance improvements, further optimization of inference latency remains an important research direction. In particular, experimental observations revealed that external DRAM access remains a fundamental performance bottleneck within the STT-VTA architecture. Consequently, the following chapter extends the proposed architecture through the integration of deterministic memory caching and data pre-fetching mechanisms.

The proposed caching mechanism enables memory-dependent modules to store and retrieve intermediate tensors, feature maps, and computation results using on-chip BRAM resources rather than repeatedly accessing external DRAM. This approach reduces the frequency of DRAM read/write operations, minimizes memory-access blocking delays, and improves DRAM availability for other system entities.

In addition, the proposed pre-fetching mechanism enables input data, weights, and bias parameters to be proactively loaded from DRAM during predetermined time windows in which the memory subsystem is not occupied by critical inference operations. By statically scheduling these memory transfers, the architecture further reduces memory-access latency and mitigates DRAM contention.

Furthermore, potential interference affecting the caching and pre-fetching subsystems is managed using deterministic scheduling principles similar to those applied for DRAM-access arbitration. This ensures predictable and interference-free operation across all memory-related architectural components.

Together, these enhancements aim to significantly reduce the dependency on external DRAM and enable efficient utilization of multiple memory-oriented functional units, including Load and Store modules. However, these improvements are expected to introduce additional FPGA resource overhead, which is analyzed and evaluated in the following chapter.

Chapter 6

Deterministic Caching and Pre-fetching for STT-VTA (STA-CP)

This chapter presents a comprehensive overview of the system model of the Superscalar Time-Triggered VTA with Caching and Pre-fetching (STA-CP) architecture, together with the scheduler featuring a built-in simulator (STA-CP Scheduler). Unlike the previous chapters, this chapter adopts a more formal presentation style by introducing the semantic definitions, scheduling rules, and architectural notations underlying the development of the STA-CP architecture and its associated scheduling framework.

Section 6.1 introduces the STA-CP architecture and its development methodology, highlighting the functional and architectural requirements that guided its design. Particular emphasis is placed on the model-driven nature of the STA-CP architecture, where the static hardware configuration is generated at compile time according to the specific computational and memory requirements of the analyzed AI model.

Section 6.2 presents the architecture and scheduling methodology of the STA-CP scheduler. This section details the scheduling stages, dependency analysis mechanisms, and scheduling rules responsible for: (1) generating the configuration files required for the automated construction of the STA-CP architecture, (2) adapting and extending instruction metadata to support deterministic caching and pre-fetching operations, and (3) generating precise timing schedules for all functional modules within the architecture.

In Section 6.3, the proposed STA-CP architecture is evaluated to demonstrate its capability to provide both deterministic and optimized AI inference execution with reduced memory-access latency and improved resource utilization. Finally, Section 6.5 concludes the chapter by discussing the key findings, architectural trade-offs, current limitations, and potential directions for future enhancements of the proposed architecture and scheduling framework.

6.1 System Model of STA-CP

The Superscalar Time-Triggered Versatile Tensor Accelerator with Cache and Pre-fetching (STA-CP) extends the foundational STT-VTA architecture, addressing the performance limitations arising from frequent and latency-sensitive DRAM accesses. The STA-CP architecture integrates an on-chip, BRAM-based deterministic caching mechanism and a static data pre-fetching strategy, while fully preserving the temporal guarantees of the time-triggered execution model. Before discussing the development process of the STA-CP in detail, we first introduce the general notations and definitions used throughout this chapter.

General Notation

- Module $m \in \mathcal{M} = \{L, C, S\}$. Where: $L \implies Load$, $C \implies Compute$, $S \implies Store$.
- Intermediate data buffer $dB_{src,dst} \in BUFF_{interm.} = \{dB_{L,C}, dB_{C,S}\}$.
- Module dispatcher $d_m \in \mathcal{D} = \{d_L, d_C, d_S\}$.
- Module schedule $Sched_m \in Schedule = \{Sched_L, Sched_C, Sched_S\}$.
- Global Time Base (GTB) is a global counter mechanism that synchronizes all dispatchers and enforces deterministic time-triggered execution, where the frequency of the GTB is $\mathcal{T}$.
- Instruction set $\mathcal{I} = \mathcal{I}_L \cup \mathcal{I}_C \cup \mathcal{I}_S$ for a given AI model, where each $i \in \mathcal{I}$ is appended with metadata μ_i .
- Instruction queue $iq_m \in IQ = \{iq_L, iq_C, iq_S\}$ is used to hold the instructions of the respective module m .

6.1.1 Functional Modules and Dispatching Semantics

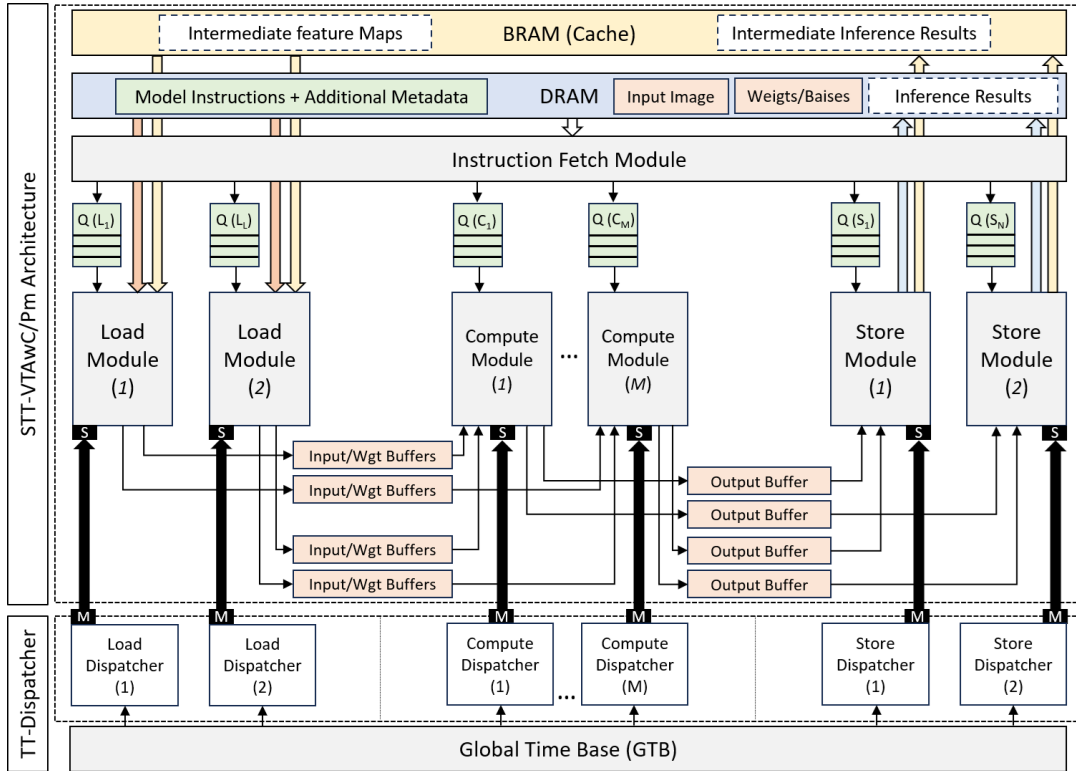


FIGURE 6.1: Superscalar Time-Triggered Versatile Tensor Accelerator with Cache and Pre-fetching (STA-CP) Architecture

At STA-CP core as in Figure 6.1, the architecture retains the modular structure of STT-VTA, consisting of Load modules, Compute modules, and Store modules, each managed by dedicated dispatchers under the supervision of a global time base (GTB). Instructions for these modules are distributed by the Instruction Fetch Module, which reads the Instruction set $\mathcal{I}$ from DRAM (M_{DRAM}) and distributes each instruction

$i \in \mathcal{I}$ into the respective module instruction queue iq_m . In the STA-CP, each functional module m executes exactly one instruction $i \in iq_m$ at every triggering instant from its dispatcher d_m . Formally, the dispatching operation of module m can be expressed as:

$$d_m(\text{Sched}_m, t) \Rightarrow i^{\text{start}}(t) \in iq_m,$$

where Sched_m denotes the precomputed time schedule for module m , and $t \in \mathcal{T}$ represents the current global time instant.

6.1.2 Cache-Aware Memory Access Semantics

A central extension of STA-CP is the integration of an on-chip BRAM-based cache "BRAM", which holds intermediate feature maps and inference results. By reusing cached data whenever possible, the architecture minimizes redundant DRAM operations and improves throughput.

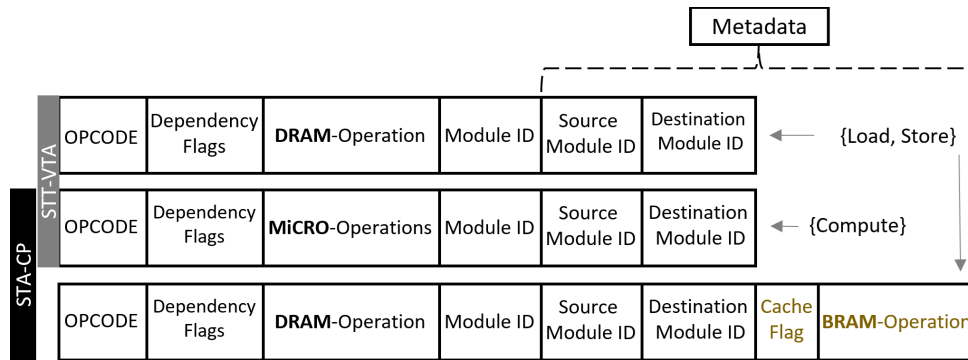


FIGURE 6.2: STT-VTA & STA-CP: Instruction Patterns including Metadata

The instruction pattern used for the STA-CP architecture comprising the following fields (i.e., see Figure 6.2):

1. Main Fields:

- OPCODE (3 bits):** Encodes the operation type, specifying whether the instruction corresponds to a Load, Compute, or Store operation.
- Dependency Flags (4 bits):** Capture inter-instruction data and execution dependencies, ensuring temporal ordering and synchronization across modules.
- Operation Field (115 bits):** Describes the operation-specific parameters. For memory instructions, this field specifies DRAM transaction parameters including DRAM offset and region of the write/read operations, whereas for compute instructions, it defines arithmetic micro-operations such as tensor multiplication, convolution, or accumulation.

2. Metadata Fields (μ_i):

- Module ID (2 bits):** Identifies the functional modules responsible for executing the instruction in case there are multiple modules of the same type available in the architecture (i.e., superscalar architecture).
- Source and Destination Module IDs (2 bits each):** Define the producer-consumer relationship between modules, enabling deterministic data routing within the superscalar pipeline.

- (c) **Cache Flag (1 bit):** Indicates whether the instruction operates on cacheable data within the on-chip BRAM cache. When set to 1, the module targets the cache memory and avoiding off-chip DRAM access.

$$\text{Target}(i) = \begin{cases} \text{BRAM}, & \text{if } \mu_i.\text{CacheFlag} = 1 \\ \text{DRAM}, & \text{otherwise} \end{cases}$$

- (d) **BRAM Operation Field (115 bits):** Mirrors the DRAM operation field but encodes parameters specific to BRAM transactions.

$$\text{BRAM}_{\text{Operation}} = \{\text{BRAM_offset}, \text{BRAM_range}\}$$

- (e) **Reserved Field (12 bits):** Allocated for future architectural extensions.

By encoding this extended metadata into the Load and Store instructions, the STA-CP's memory modules become cache-aware—dynamically selecting the appropriate memory target (BRAM or DRAM) based on each instruction's cache flag.

6.1.3 Scheduling Semantics

During scheduling process (detailed in Section 6.2), the STA-CP scheduler with built-in simulator analyzes the instruction stream $\mathcal{I}$ to:

1. Determine the appropriate set of functional modules $\mathcal{M}' = \{L^{(x)}, C^{(y)}, S^{(z)}\}$, where $x, y, z \in \mathbb{N} \geq 1$ are the number of instances of the corresponding module for the superscalar architecture, including:
 - (a) their Instruction queues $IQ' = \{iq_L^{(x)}, iq_C^{(y)}, iq_S^{(z)}\}$,
 - (b) their dispatching pipelines $\mathcal{D}' = \{d_L^{(x)}, d_C^{(y)}, d_S^{(z)}\}$, and
 - (c) the sizes of their intermediate data buffers $\text{BUFF}'_{\text{interm.}} = \{dB_{L,C}^{(x*y)}, dB_{C,S}^{(y*z)}\}$ to allow cross-interconnections between modules.
2. Adapt the metadata μ_i of instruction $i \in \mathcal{I}$, which lead to a new instruction set $\mathcal{I}'$.
3. Calculate required cache size (i.e., *BRAM* size)
4. Generate the module schedule " Sched_m " that defines the starting time instance "Start(i)" and finish time instance "Finish(i)" for all instructions in $\mathcal{I}$.

6.1.4 Synthesis and Execution Semantics

When the scheduling phase is concluded, the VTA compiler backend and Xilinx toolchain synthesize the specialized configuration of STA-CP. The synthesis function is defined as:

$$\mathcal{F}_{\text{syn}} : (\mathcal{M}', IQ', \mathcal{D}', \text{BRAM}, \text{Schedule}, \mathcal{T}, \text{BUFF}'_{\text{interm.}}) \rightarrow \text{Bitstream}_{\text{STA-CP}}$$

which generates the hardware bitstream that configures the STA-CP system for a specific workload.

Finally, the execution of the architecture can be represented by:

Execute : (Input = {Image, Weights, Biases}, $\mathcal{I}'$, Bitstream_{STA-CP}) $\rightarrow$

Deterministic Inference Latency and Output

ensuring that, when a given instruction set $\mathcal{I}'$ is executed on the deployed STA-CP architecture, its behavior—and resulting output—remain both temporally predictable and deterministically reproducible.

6.2 STA-CP Scheduler

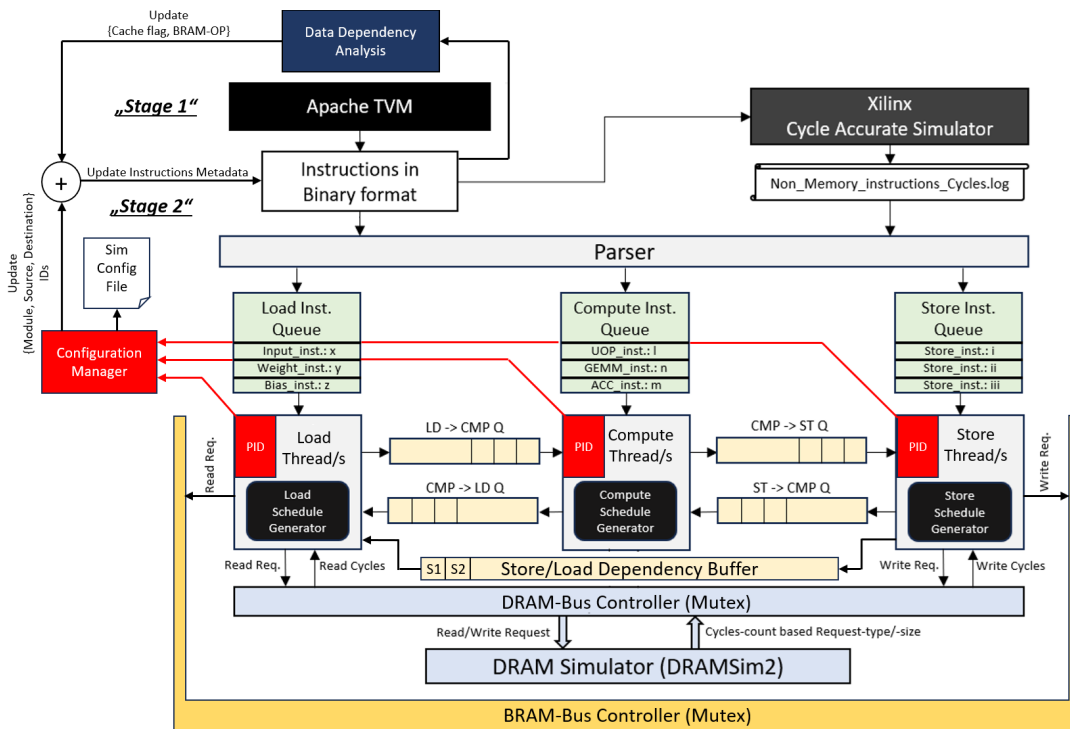


FIGURE 6.3: Scheduler with built-in simulation "STA-CP Scheduler"

In this section, we focus on the STA-CP scheduler (i.e., illustrated in Figure 6.3), which has been developed to support the extended STA-CP architecture. This version performs deeper analysis of model instructions to determine their data dependencies and to classify memory operations—including load and store instructions—between BRAM (cache) and DRAM. It also identifies load instructions eligible for pre-fetching by the load modules, and determines the optimal cache memory size as well as the appropriate dimensions of intermediate buffers (input, weight, and output buffers) between functional units. At the same time, version 3 adapts instructions metadata to guarantee consistent and synchronized execution across functional units.

The scheduler, shown in Figure 6.3, is structured into two primary stages, where:

- **Stage 1:** focuses on analyzing data dependencies among memory instructions (i.e., both loads and stores instructions) to identify caching opportunities and to determine the optimal cache size.

- **Stage 2:** plays an important role in selecting load instructions eligible for pre-fetching, optimizing the architecture for a given AI model, and generating timing schedules for the individual modules.

6.2.1 Stage 1 — Data Dependency Analysis and Layer-Based Cache Management

The first stage of the simulation framework performs a comprehensive data dependency analysis to uncover store-load relationships that exhibit temporal data reuse and are therefore good candidates for caching. Identifying these dependencies early in the simulation process is essential for enabling predictable memory management and effective exploitation of on-chip BRAM.

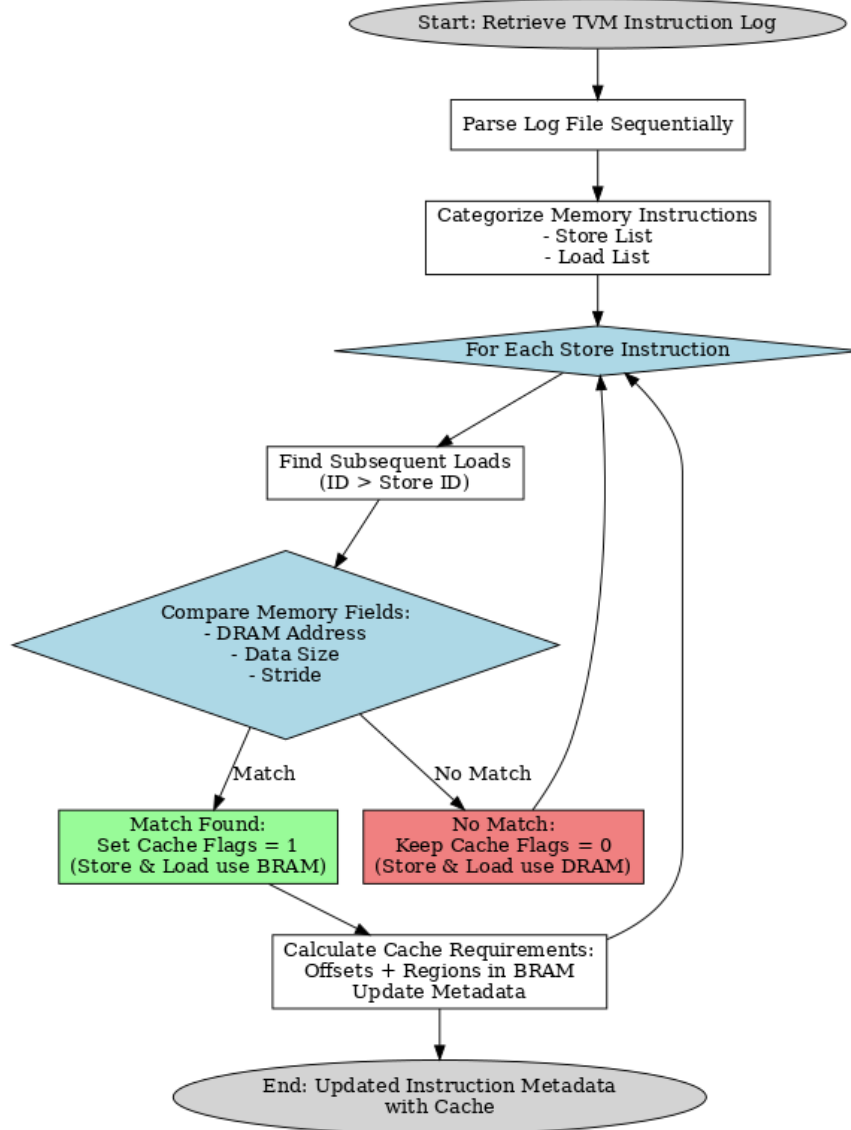


FIGURE 6.4: Data Dependency Analysis for Caching

This stage, as illustrated in Figure 6.4, begins by retrieving the instruction trace generated by the TVM compiler, which is stored in a log file. The simulator sequentially parses this file and categorizes only the memory instructions into two separate lists:

1. A store list, which contains all store-type instructions.
2. A load list, which contains all load-type instructions.

Once the memory instructions are categorized, the simulator performs a systematic dependency analysis by iterating over both lists. To ensure causal consistency, each store instruction $s \in \mathcal{I}_S$ is analyzed against subsequent load instructions $l \in \mathcal{I}_L$ that appear later in the trace (i.e., $\text{ID}(l) > \text{ID}(s)$), where instruction's ID is a tuple (i.e., Layer + program counter). If both instructions operate on the same memory address, stride, and have equal data size, they are considered to form a valid memory dependency ($\text{MemDep}(s, l) = 1$). These dependencies indicate potential data reuse opportunities, then the complete set of such active cacheable pairs is defined as:

$$\mathcal{A}_{\text{cache}} = \{(s, l) \mid s \in \mathcal{I}_S, l \in \mathcal{I}_L, \text{MemDep}(s, l) = 1\}$$

Once cacheable pairs are identified, the simulator proceeds to determine the specific cache memory requirements. Since the data sizes of both store and load instructions are known, the simulator employs an incremental accumulation method to calculate the total amount of data that must be cached. This process allows the simulator to derive precise BRAM offsets and allocate appropriate memory regions within the cache. These calculated offsets and regions are then used to update the BRAM-operation fields in the metadata of the store and load instructions, ensuring that cached data is correctly mapped and retrievable during execution.

Unlike generic instruction-level cache models, this analysis is performed at the layer-granularity of a given AI model. Since most AI workloads—such as CNNs or transformers—execute in well-defined stages, modeling cache reuse at the layer granularity better reflects real temporal locality and simplifies predictability analysis. Therefore, for each valid store-load pair (s, l) , the simulator identifies the first and last layers during which the corresponding data remains required. Thus, the corresponding reuse window can be defined as:

$$[\text{Layer}(s), \text{Layer}(l)]$$

where $\text{Layer}(s)$ and $\text{Layer}(l)$ are the layer IDs of the instruction s and l , respectively.

Within this interval, the data produced by an instruction s (i.e., Data_s) must remain resident in the on-chip BRAM to guarantee deterministic reuse by l . After the final reuse layer completes, the data becomes obsolete and may be safely evicted.

Additionally, since on-chip BRAM capacity is limited, the simulator enforces a strict capacity constraint:

$$\sum_{(s,l) \in \mathcal{A}_{\text{cache}}} \text{Size}(\text{Data}_s) \leq \text{CacheSize}$$

Whenever the inclusion of a new cacheable pair (s, l) would exceed the available BRAM capacity within its reuse window, the pair is excluded from caching to preserve deterministic resource bounds and the pair proceeds using direct DRAM access instead of on-chip caching.

Conversely, if a cacheable pair is successfully admitted to the on-chip cache, its data must remain resident until the end of its reuse window, after which it becomes eligible for eviction at $\text{Layer}(l)$. To manage this process predictably and efficiently, the proposed framework maintains a dynamic set of free BRAM regions, allowing fine-grained and deterministic reuse of space released through eviction.

Let $\mathcal{F}$ denote the ordered set of currently free BRAM address space, where b_i and e_i are the first and last free addresses spaces respectively:

$$\mathcal{F} = \{[b_i, e_i] \mid 0 \leq b_i < e_i < \text{CacheSize}\}.$$

Initially, $\mathcal{F} = \{[0, \text{CacheSize}]\}$, representing the entire BRAM address space. When allocating space for a new cacheable pair (s, l) , the allocator deterministically selects the first region sufficiently large to accommodate its data:

$$\exists [b_i, e_i] \in \mathcal{F} \text{ such that } e_i - b_i \geq \text{Size}(\text{Data}_s).$$

The pair is then assigned to the lowest available address range within that region and offset:

$$\text{Region}(s, l) = [b_i, b_i + \text{Size}(\text{Data}_s)], \quad \text{Off}(s, l) = b_i.$$

After allocation, the occupied range is removed from $\mathcal{F}$, and any remaining portion of the interval is retained as free space. This deterministic *first-fit* allocation policy ensures consistent cache behavior across runs.

Once a pair's reuse window expires (i.e., the last reuse layer has completed), the corresponding data is evicted. Where, the eviction updates the active cache set, as well as the free-space map by merging adjacent free intervals:

$$\mathcal{A}_{\text{cache}} = \mathcal{A}_{\text{cache}} \setminus \{(s, l)\},$$

$$\mathcal{F} = \text{Merge}(\mathcal{F}, \text{Region}(s, l))$$

The adopted merging strategy enables deterministic mid-memory reuse within the BRAM address space. Whenever a previously allocated region becomes available, it can be reassigned to a new cacheable pair, provided that the free region satisfies the required storage capacity constraints.

It is important to note that this memory-management policy may introduce internal fragmentation within the BRAM. Nevertheless, the primary objective of the STA-CP architecture is to minimize memory-access latency and preserve deterministic execution behavior, rather than maximizing BRAM utilization efficiency alone. Although more sophisticated memory-management mechanisms, such as paging [106], could reduce fragmentation, they would also introduce additional address-translation overhead, increase BRAM access frequency, and complicate timing predictability. Such effects directly conflict with the low-latency and temporal-determinism requirements targeted by the proposed architecture.

Finally, each instruction is annotated with extended metadata to capture cache-related attributes as:

$$\mu_i = (\text{module}_{ID}, \text{src}_{ID}, \text{dst}_{ID}, 1, \text{Off}(s, l), \text{Region}(s, l))$$

where the update is applied to both store s and load l in the pair (s, l) .

Consequently, Stage 1 provides the analytical groundwork for deterministic caching within the Superscalar Time-Triggered Tensor Accelerator (STA-CP). By coupling fine-grained data dependency analysis with layer-based cache management and interval-aware allocation, this stage ensures that every cache decision is predictable, analyzable, and efficiently utilizes available cache memory. Importantly, it supports deterministic reuse of freed memory regions—including those in the middle of the address space—thus maintaining high resource utilization without compromising temporal determinism of reuse windows.

6.2.2 Stage 2 - Scheduling

In the following, we outline the distinct phases of the scheduler and its simulation process, as depicted in Figure 6.3. The simulator operates as a set of concurrent software threads, each serving as a lightweight abstraction of the corresponding hardware module in the STA-CP architecture. This simulation framework enables a detailed analysis of the execution behavior of the instruction stream associated with a given AI model, ensuring accurate modeling of module interactions, dependencies, and timing characteristics. Each phase is described operationally and expressed formally to define the simulation state-transition semantics.

1. **Configuration Phase:** Before scheduling and simulation process begins, a configuration file *Config* is initialized with the following parameters:

$$Config = \langle T, M_{\text{cache}}, M_{\text{dram}}, IQ, DQ, B_{\text{SL}}, \text{pre-fetch_Flag} \rangle$$

where:

- T is a set of module threads, where $T_{m(ID)} \in T = \{T_{\text{load}}^{(x)}, T_{\text{compute}}^{(y)}, T_{\text{store}}^{(z)}\}$, $x, y, z \in \mathbb{N} \geq 1$, and ID is a thread identifier;
- M_{cache} and M_{dram} are mutual-exclusion locks for BRAM and DRAM controllers;
- Instruction Queues (IQ): $iq_{m(ID)} \in IQ = \{iq_{\text{load}}^{(x)}, iq_{\text{compute}}^{(y)}, iq_{\text{store}}^{(z)}\}$;
- Dependency Queues (DQ):
 $\forall T_{m(ID)} \in T, \exists dq_{m(ID)} \in DQ = \{(dq_{\text{src}(ID)}, dq_{\text{dst}(ID)})\}$,
 where $\text{src}(ID)$ & $\text{dst}(ID)$ are, respectively, the source and destination module threads with the identifier ID ;
- B_{SL} is a Store/Load Dependency Buffer: A unidirectional buffer that is instantiated between the store and load threads, enabling load threads to track executed store instructions and their finishing time instants. This information is essential for determining whether a load instruction is eligible for pre-fetching, thus scheduled earlier or must wait for a (subset-) data dependency from a store instruction;
- $\text{pre-fetch_Flag} \in \{\text{True}, \text{False}\}$ controls early load scheduling, if set to *True* by the user;

Additionally the metadata of the instruction set $\mathcal{I}$ are initialized as follows:

$\forall i \in I$, which include metadata μ_i

if i is **Memory-instruction**, then $\mu_i = (\text{module_ID}, \text{src_ID}, \text{dst_ID}, \text{CacheFlag}, \text{BRAM}_{\text{Operation}})$, initialized as $(1, 1, 1, 0/1, \text{off}, \text{range})$ - where CacheFlag and $\text{BRAM}_{\text{Operation}}$ are results from Stage 1 (i.e., see Section 6.2.1), and if i is **non-Memory-instruction**, then $\mu_i = (\text{module_ID}, \text{src_ID}, \text{dst_ID})$, initialized as $(1, 1, 1)$.

2. **Pre-Simulation Phase** The Parser sorts the instructions $\mathcal{I}$ into the queue $iq_{m(ID)}$ of each module thread $T_{m(ID)}$, based on the instruction OPCODE (i.e., instruction type), and module ID from the instruction metadata (i.e., see Figure 6.2). Additionally, Let the initial simulation state be defined as:

$$\Sigma := \langle Config, \mathcal{I} \rangle.$$

3. **Schedule generation and Simulation Phase** The primary objective of this phase is to generate precise timing schedules for all active modules within the STA-CP architecture. For each module $m(ID)$, the schedule $Sched_{m(ID)}$ represents an ordered sequence of start timing instances corresponding to its assigned instructions $i \in iq_{m(ID)}$. Accordingly, the simulation process determines both the start and finish instances of every instruction executed across the module threads.

Declarations: For an instruction i :

- $i.pop \in \{\text{True}, \text{False}\}$, if true $\rightarrow$ current instruction depends on another module's instruction,
- $i.push \in \{\text{True}, \text{False}\}$, if true $\rightarrow$ another module's instruction depends on the current one,
- $pred(i)$ = immediate predecessor of i in $iq_{m(ID)}$,
- $Start(i)$ and $Finish(i)$ are the scheduled start and finish timing instances.

Operational Rules: Each module thread $T_{m(ID)}$ operates concurrently, adhering to the execution semantics defined by Rules (1–5):

- (a) **Rule 1 (Deterministic Instruction Sequencing per Module-Thread):** Each thread $T_{m(ID)}$ continues to process and schedule every single instruction $i \in iq_{m(ID)}$, while preserving strict temporal ordering:

$$\forall i, j \in iq_{m(ID)}, i < j \implies Finish(i) \leq Start(j).$$

- (b) **Rule 2 (Scheduling Start instance) $\implies Start(i)$:**

- i. **Condition 2.1 (Scheduling-based Sequential Operation):** If the pop flag $i.pop = \text{False}$ (i.e., instruction i does not depend on any other instruction), the current instruction's start instant is scheduled directly after its predecessor instruction's finish instant.

$$Start(i) := Finish(pred(i)).$$

- ii. **Condition 2.2 (Scheduling-based dependency):** If the pop flag $i.pop = \text{True}$ (indicating that instruction i depends on another instruction j from a different source module), the dependent instruction j and its scheduled finish instance are retrieved from the corresponding source queue $dq_{src(ID)}$, where $ID = \mu_i.src_ID$. The module thread then checks whether the finish time of j is greater than or equal to that of the predecessor instruction within the same module.

$$\text{if } Finish(j) \geq Finish(pred(i)), \text{ then } Start(i) := Finish(j), \\ \text{else } i \text{ is marked as a pending instruction}$$

- iii. **Condition 2.3 (Pending Instruction Handling):** If $i.pop = \text{True}$ and i is pending because $Finish(pred(i)) > Finish(j)$, then the configuration manager is called to handle the pending instruction:

$$(Config, \mathcal{I}) \xrightarrow{\text{ConfigMgr}(PID(i))} (Config', \mathcal{I}')$$

This means, terminating the current scheduling and simulation round and restarting with an updated configuration $Config'$ and updated instructions $\mathcal{I}'$.

An instruction is classified as *pending* when it is ready for execution by its designated module but cannot proceed due to that module being occupied. To mitigate such stalls, the simulator employs the **Pending Instruction Detector (PID)** and the **Configuration Manager** to dynamically scale the architecture by instantiating additional module threads as needed. Thus, the PID is instantiated for all available threads, not only compute threads as in [103]. Consequently, each module thread is capable of detecting pending instructions in its instruction queue and notifying the Configuration Manager of any findings. Whereas, the Configuration Manager initiates a reconfiguration procedure to handle the pending instruction and update the simulation configuration file ($Config$). Equations (6.1)–(6.4) collectively define the reconfiguration workflow managed by the Configuration Manager. Upon detecting a pending instruction, the manager suspends the ongoing simulation, reallocates the instruction to a newly instantiated module of the same type, and propagates the corresponding dependency updates across all affected instruction metadata. Specifically, the module ID of the pending instruction is incremented (6.1), while metadata fields of dependent (i.e., destination Module ID) and successor (i.e., source Module ID) instructions are updated to preserve correct data routing and synchronization (6.2). The configuration file is then revised using the Equations in (6.3) to reflect the newly added module-thread instance ($T_{m(ID)}$) and its inter-thread dependencies ($dq_{m(ID)}$) as well as its dedicated instruction queue ($iq_{m(ID)}$). Finally, the simulator is reinitialized with the updated configuration state (6.4), ensuring a seamless continuation of the simulation under the expanded architectural setup.

(1) Termination:

Stop current simulation cycle;

(2) Assign instruction i to another module thread:

$$\mu_i.\text{module_ID} := \mu_i.\text{module_ID} + 1; \quad (6.1)$$

(3) Δ -Metadata Propagation:

$$\forall j \in \mathcal{I}, \begin{cases} \mu_j.\text{src_ID} = \mu_i.\text{module_ID} & \text{if } j \text{ depends on } i, \\ \mu_j.\text{dst_ID} = \mu_i.\text{module_ID} & \text{if } i \text{ depends on } j, \end{cases} \quad (6.2)$$

(4) Configuration Update:

$$\begin{aligned} & \text{if } T_{m(\mu_i.\text{module_ID})} \nexists \text{ in } \text{Config}.T, \text{ then} \{ \\ & \quad \text{Config}'.T := \text{Config}.T \cup \{T_{m(\mu_i.\text{module_ID})}\}; \\ & \quad \text{Config}'.DQ := \text{Config}.DQ \cup \{dq_{m(\mu_i.\text{module_ID})}\}; \\ & \quad \text{Config}'.IQ := \text{Config}.IQ \cup \{iq_{m(\mu_i.\text{module_ID})}\}; \end{aligned} \quad (6.3)$$

(5) Reinitialization:

$$\Sigma' := \langle \text{Config}', \mathcal{I}' \rangle. \quad (6.4)$$

This process ensures that all pending instructions are resolved, inter-instruction dependencies are correctly realigned, and actual modules of the STA-CP architecture are efficiently utilized during inference execution after being deployed on the hardware.

- iv. **Condition 2.4 (Scheduling-based Pre-Fetching):** Let i_l and i_s denote a load and store instructions respectively. If pre-fetching option enabled in the configuration file $\text{Config.pre_fetch_Flag} = \text{True}$ and $\text{type}(i_s) = \text{Store}$, then:

$$\forall i_s \in iq_{\text{store}(ID)}, B_{\text{SL}} := B_{\text{SL}} \cup (i_s, \text{Finish}(i_s))$$

For each store instruction i_s , the store thread ($T_{\text{store}(ID)}$) pushes the instruction and its scheduled finish instant into the Store/Load Dependency buffer B_{SL} . This information enables the load thread ($T_{\text{load}(ID)}$) to correctly schedule their load instructions by accounting for potential overlaps or intersections with preceding store operations.

Additionally, if Pre-fetching option enabled $\text{Config.pre_fetch_Flag} = \text{True}$ and $\text{type}(i_l) = \text{Load}$, then:

$$\text{Start}(i_l) := \begin{cases} \text{Finish}(i_s) & \text{if } \exists i_s \in B_{\text{SL}} : \text{Overlap}(i_l, i_s), \\ \text{Finish}(\text{pred}(i_l)) & \text{otherwise.} \end{cases}$$

where B_{SL} is a Store/Load Dependency buffer, and i_s denotes a store instruction whose memory operations intersect with those of the load instruction, either in DRAM or BRAM. If no intersection exists, the load instruction becomes eligible for pre-fetching, and its start instant is scheduled directly after the completion of its predecessor instruction within the load module. It should be noted, however, that this preliminary scheduling may be adjusted in accordance with **Condition 2.5**, which ensures that all pre-fetching operations are placed exclusively within memory idle intervals. This constraint prevents any potential interference with ongoing memory transactions, thereby preserving deterministic and contention-free memory behavior.

- v. **Condition 2.5 (Scheduling-based Memory Interference avoidance):** For memory-type instructions $\text{type}(i) \in \{\text{Load}, \text{Store}\}$, regardless of whether the cache flag is set to True or False, the thread first checks the

availability of the corresponding memory controller (i.e., the mutual-exclusion lock M_{memory}). If the controller is busy, the instruction waits until it becomes free, after which its start instant is assigned accordingly:

if $\text{ControllerBusy}(M_{\text{memory}}) \Rightarrow$ wait until free,

then, $\text{Start}(i) := \begin{cases} \text{Finish}(\text{last}(M_{\text{cache}})) & \text{if } \mu_i.\text{CacheFlag} = \text{True}, \\ \text{Finish}(\text{last}(M_{\text{dram}})) & \text{otherwise.} \end{cases}$

(c) **Rule 3 (Identifying Finish instance) $\Rightarrow$ Finish(i):**

- i. **Condition 3.1 (DRAM Transactions):** If $\mu_i.\text{CacheFlag} = \text{False}$ and $\text{type}(i) \in \{\text{Load}, \text{Store}\}$:

$\text{Finish}(i) := \text{Start}(i) + \text{DRAMSim2}(i),$

where: $\text{DRAMSim2}(i)$ represents the simulated DRAM access latency corresponding to the i -th read or write instruction. For memory instructions targeting DRAM, DRAMSim2 [98] is utilized to model and evaluate their transaction timing. This approach is necessary because DRAM is treated as an external peripheral on FPGA boards and therefore is not accurately represented within Xilinx's Cycle-Accurate Simulator. To capture realistic timing behavior, DRAMSim2 is used to estimate the precise cycle count associated with each memory instruction under analysis. Formally, this can be expressed as:

$\text{DRAMSim2}(i) := \text{sim}(i.\text{Address}, i.\text{DataSize}, i.\text{OperationType})$

- ii. **Condition 3.2 (BRAM Transactions or Non-Memory Instructions):** If $(\mu_i.\text{CacheFlag} = \text{True} \text{ and } \text{type}(i) \in \{\text{Load}, \text{Store}\})$ or $\text{type}(i) = \text{Compute}$:

$\text{Finish}(i) := \text{Start}(i) + \text{cycles}(i).$

Where: $\text{cycles}(i)$ are obtained in advance from the Xilinx's Cycle-Accurate Simulator (referring to Figure 3.1). This is due to the fact that, during behavioral simulation, BRAM-memory is already used as an alternative to off-chip memory (DRAM).

- iii. **Condition 3.3 (Push-Notification to other Threads):** If $i.\text{push} = \text{True}$, then the instruction i and its scheduled finished instance $\text{Finish}(i)$ are pushed to the destination module's queue

$dq_{\text{dst}(ID)} := dq_{\text{dst}(ID)} \cup (i, \text{Finish}(i)), \text{ where } ID = \mu_i.\text{dst_ID}.$

- (d) **Rule 4 (Module Schedule Generation) $\Rightarrow$ Sched_m:** The start instant of the instruction is then appended to the module schedule.

$\text{Schedule} : \text{Sched}_{m(ID)}.append(\text{Start}(i))$

Note: When the scheduling and simulation process is concluded as well as the STA-CP architecture is deployed on the hardware (i.e., FPGA), the schedule of module m is then utilized by the dispatcher pipeline d_m to trigger the module to execute each instruction $i \in iq_{m(ID)}$ whenever the global time counter matches the instruction's scheduled start instant,

proceeding sequentially until the entire queue has been processed.

- (e) **Rule 5 (Identifying Interm. src/dst data buffer size)** $\implies dB_{src,dst}$: This rule addresses the impact of architectural scaling and the introduction of pre-fetching on intermediate data buffering. When pre-fetching is enabled, load modules retrieve data earlier than its consumption time, which accelerates buffer fill rates. Without adequate buffering, this behavior may result in overwriting or corruption of unprocessed data. To prevent such hazards and ensure deterministic data flow, the intermediate buffer size between any source (data producer) and destination (data consumer) modules must be carefully analyzed and dimensioned.

Formally, let $dB_{src(ID),dst(ID)}$ denote the intermediate buffer capacity between a source module $src(ID)$ and a destination module $dst(ID)$. For each instruction $i \in iq_m(ID)$, let $DataSet(i)$ represent the volume of data produced by i , and let j denote the corresponding consuming instruction in the destination module, thus:

$$\forall dB_{src(ID),dst(ID)} \in BUFF_{interm}.$$

The evolution of the buffer occupancy is governed by:

$$dB_{src(ID),dst(ID)} := \begin{cases} dB_{src(ID),dst(ID)} + DataSet(i), & \text{if } i.\text{push} = \text{True} \wedge \text{type}(i) \in \{\text{Load}, \text{Compute}\} \text{ at } \text{Finish}(i), \\ dB_{src(ID),dst(ID)} - DataSet(i), & \text{if } j.\text{pop} = \text{True} \wedge \text{type}(j) \in \{\text{Compute}, \text{Store}\} \text{ at } \text{Start}(j). \end{cases}$$

Each data block produced by instruction i must remain valid within the intermediate buffer during its logical lifetime:

$$[\text{Finish}(i), \text{Start}(j)],$$

representing the interval between data production and its subsequent consumption.

At any scheduled instant t , the instantaneous buffer occupancy is given by:

$$dB_{src(ID),dst(ID)}(t) = \left(dB_{src(ID),dst(ID)} \mid \text{Finish}(i) \leq t < \text{Start}(j) \right),$$

while the maximum buffer requirement observed throughout the simulation is:

$$dB_{src(ID),dst(ID)}^{max} = \max_t dB_{src(ID),dst(ID)}(t).$$

This formalization enables the simulator to derive precise buffer capacity requirements that guarantee contention-free communication between modules within the STA-CP architecture, even under aggressive pre-fetching and parallel execution conditions.

4. **Post-Simulation Phase** In this phase, the configuration file, instruction meta-data, intermediate buffer sizes, and Cache memory size information are used to configure the STA-CP architecture, referring to Section 6.1.4. Finally, the STA-CP architecture is synthesized and implemented using Vivado-Xilinx tools (see Figure 3.1), generating a bitstream file that can be used to configure the

FPGA upon deployment. At this stage, the architecture is ready for testing and validation.

6.3 STA-CP Evaluation

For hardware validation, the STA-CP architecture was implemented and deployed on a Xilinx Zynq UltraScale+ ZCU104 platform, featuring a ZU7EV SoC with a quad-core ARM Cortex-A53 and programmable logic (PL). The accelerator resides entirely in the PL fabric, while the processing system (PS) hosts the TVM runtime and evaluation scripts for AI applications (i.e., CNN models for object classification).

6.3.1 Experimental Setup

- **Toolchain:** Xilinx Vivado 2023.1 was employed for STA-CP logic synthesis, placement, routing, and bitstream generation, after which the resulting bitstream was deployed to the programmable logic (PL) fabric using TVM's evaluation script.
- **Clocking and Interfaces:** All STA-CP modules operate at 333MHz within a single global clock domain. Communication between the processing system (PS) and the accelerator is established via AXI4-Lite for control signaling and AXI4-Full for bidirectional data transfers between the programmable logic (PL) and DRAM. An additional AXI4-Full interface facilitates data movement between modules that access the on-chip cache memory (BRAM), while standalone-buffer connections are employed for intermediate data transfers among internal modules.
- **Memory Constraints:** The ZCU104 evaluation board provides a total of 11Mb of on-chip Block RAM (BRAM), distributed across 312 blocks, each with a capacity of $36\text{Kb}/8 = 4.5\text{KB}$. This configuration yields a maximum of approximately 1.375MB of on-chip memory available within the programmable logic (PL) fabric. In addition, a 512MB DDR4 serves as an external DRAM for the PL. It is important to emphasize that the on-chip memory (i.e., BRAM) on FPGA designs typically supports access latencies several times lower—or in many cases an order of magnitude lower—than external DRAM accesses due to shorter paths, lack of refresh overhead, and simpler control logic [107].
- **Other Resource Constraints:** The ZCU104 evaluation board provides the following maximum hardware resources: 230,400 look-up tables (LUTs), 460,800 flip-flops (FFs), and 1,728 digital signal processors (DSPs). These values define the upper limits available for implementing the STA-CP architecture on the platform.
- **Configuration Loading:** During initialization, the processing system (PS) loads the instructions and their associated metadata into the accelerator's instruction memory region located in DRAM. Once initialized, the accelerator autonomously executes the scheduled inference under the coordination of the Global Time Base (GTB).

6.3.2 Tested Models and STA-CP Configurations

Several Convolutional Neural Network models were used during testing the proposed architecture: (1) ResNet-18, (2) ResNet-34, (3) ResNet-50, and (4) ResNet-101.

These models represent large feed-forward networks with extensive feature-map reuse. All models were quantized to INT8 using TVM’s quantization-aware flow to ensure compatibility with the STA-CP’s modules.

TABLE 6.1: Different STA-CP configurations

Config.	Nr. of Load/s	Nr. of Compute/s	Nr. of Store/s	Peak DRAM Utilization	Peak BRAM Utilization	Intermediate Buffers
STA(1)	1	1	1	4.36MB	-	144KB
STA(2)	1	2	1	4.36MB	-	144KB
STA-C	2	2	2	3.85MB	512KB	144KB
STA-CP	2	2	2	3.85MB	512KB	432KB = 144KB + 288KB

Four configurations were used for comparison as illustrated in Table 6.1. In this configurations, STA(1) corresponds to the TT-VTA architecture equipped with a single compute module, while STA(2) represents the STT-VTA architecture featuring two compute modules for parallel tensor execution. The STA-C configuration extends the STT-VTA design by integrating an on-chip cache (up to 512 KB) within the programmable logic (PL) fabric alongside its modules. Finally, the STA-CP configuration extends the STA-C design by enabling static data pre-fetching alongside the existing cache memory. To support early data fetching, an additional 288KB of intermediate buffer space was introduced. This expansion was necessary because pre-fetching accelerates data arrival into intermediate buffers, which could otherwise lead to data overwrite or corruption if unprocessed data were prematurely replaced. The sizes of the cache memory and intermediate buffers are derived analytically by the scheduler during Stages 1 and 2 of the compilation process, as detailed in Section 6.2. The 512 KB cache size represents the maximum allowable Block RAM (BRAM) utilization for this cache, as it brings the BRAM usage on the ZCU104 platform to approximately 97% of the available capacity. Therefore, we constrained the cache size to 512KB.

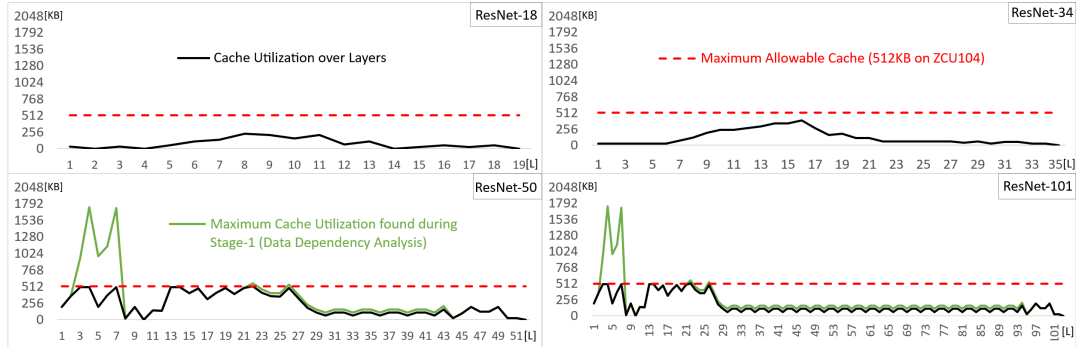


FIGURE 6.5: Cache Utilization

Figure 6.5 shows the layer-wise cache utilization for the ResNet models. The red dashed line indicates the maximum on-chip cache capacity of the ZCU104 platform (512KB), which was provided to the scheduler in Stage 1 to account for Block-RAM limitations. The black curve represents the actual cache utilization during inference under the Block-RAM constraints, while the green curve depicts the maximum cache demand estimated from the data-dependency analysis of all identified cachable pairs, assuming unlimited Block-RAM availability.

For smaller networks such as ResNet-18 and ResNet-34, cache utilization remains well below the hardware limit across all layers. This indicates all identified

cachable pairs fit comfortably within the available BRAM. Consequently, this allowed deterministic reuse without any cache overflow.

In contrast, the deeper networks, ResNet-50 and ResNet-101, exhibit substantially higher and more variable cache utilization, particularly in the early layers. These fluctuations arise from larger tensor dimensions and overlapping data dependencies during the initial convolutional stages. The green traces indicate the peak cache requirements, which in some layers exceed the 512KB limit. Consequently, during Stage 1 of the data-dependency analysis, only cachable pairs that achieve high hit rates and fit within the available on-chip cache during their reuse window are selected for caching. Pairs that exceed the Block-RAM capacity are excluded from on-chip caching and are instead mapped to DRAM.

For instance, in both ResNet-50 and ResNet-101, cache utilization in layer 22 approaches the maximum on-chip cache capacity. Consequently, certain data that would ideally be cached are excluded. These data are subsequently required in layer 44 of ResNet-50 and layer 95 of ResNet-101. Because their hit rates were low and caching them would have exceeded the available cache capacity during their reuse windows, they were not stored in the on-chip cache. This accounts for the small gap observed between the green and black curves between layers 22–44 in ResNet-50 and layers 22–95 in ResNet-101.

6.3.3 Measurement Methodology

1. Latency Measurement:

- (a) **Inference latency @ compile time:** During compile time, the scheduler not only generates timing schedules for the STA-CP modules but also performs an analytical estimation of the expected inference latency for the given model. These schedules correspond solely to hardware-level execution; therefore, the analysis assumes that the model instructions and input data are already resident in DRAM, and that all instructions have been distributed to the instruction queues of the respective modules. As a result, any data transfer overhead is excluded from this latency estimation. The total inference time can thus be expressed as:

$$T_{Inference}(Model) = T_{STA-CP}(sim) + T_{Overhead}(HW)$$

TABLE 6.2: $T_{Overhead}$ w.r.t Tested Models

Models	Nr. of Layers	$T_{Overhead}$ [ms]
ResNet-18	19	13.7
ResNet-34	35	21.8
ResNet-50	52	45.1
ResNet-101	103	69.1

Where. $T_{Overhead}$ represents the latency associated with transferring input data and model instructions from the processing system (PS) to DRAM, fetching instructions from DRAM and their categorization and distribution into the instruction queues of individual modules via the Fetch module

prior to inference execution on the programmable logic (PL), and transferring inference results from DRAM back to the PS upon completion. For all models the $T_{Overhead}$ is illustrated in Table 6.2.

- (b) **Inference latency @ run time:** For each evaluated model, the execution duration (i.e., inference latency) was measured using both: (1) the internal timing mechanism of the $TVM_{runtime}$ script and (2) hardware-level timing based on the Global Time Base GTB state before and after inference execution on the accelerator architecture. The comparison revealed that the inference latencies measured on the programmable logic (PL) were consistently aligned with the anticipated inference latency estimated by the scheduler at compile time and were uniformly lower than those reported by the time-measuring-function within $TVM_{runtime}$. Formally, the PL-level inference latency is defined as:

$$T_{Inference(PL)}(model) = GTB_{After} - GTB_{Before} = T_{STA-CP}(sim),$$

while the relationship between the two measurement methods can be expressed as:

$$T_{Inference(PL)}(model) < T_{Inference(TVM_{Runtime})}(model).$$

The inference latencies presented in Table 6.3 correspond to measurements obtained from the $TVM_{runtime}$. These results were selected to demonstrate the effectiveness of the proposed approach under a full-system evaluation scenario—that is, encompassing the complete inference pipeline from input provisioning to the retrieval of inference results.

2. **Resource Utilization:** Post-implementation reports from Vivado were analyzed to extract resource utilization metrics including look-up tables (LUT), flip-flops (FF), block-ram (BRAM), and digital signal processor (DSP) usage.

6.3.4 Resource Utilization

The resource utilization of the tested architectures on the ZCU104 platform is summarized in Figure 6.6. It reports the percentage of the FPGA resources consumed, including look-up tables (LUTs), flip-flops (FFs), block RAM (BRAM), and digital signal processors (DSPs).

- **STA(1):** With a single compute module, STA(1) consumes 40% of LUTs, 42% of FFs, 31% of BRAM, and 16% of DSPs. This configuration represents the TT-VTA implementation, that achieve deterministic inference execution with modest resource usage.
- **STA(2):** Doubling the number of compute modules resulted in resource utilization rising to 55% LUTs, 56% FFs, 44% BRAM, and 31% DSPs. The notable increase in DSP consumption corresponds to the integration of an additional compute module, which effectively enabled parallel tensor operations and contributed to a substantial reduction in overall inference latency.
- **STA-C:** The integration of a 512KB on-chip cache memory further increases resource utilization, most notably in BRAM, which rises to 82%. LUT and FF usage also grow to 76% and 81%, respectively, reflecting the added control logic and cache management infrastructure, including additional load and store

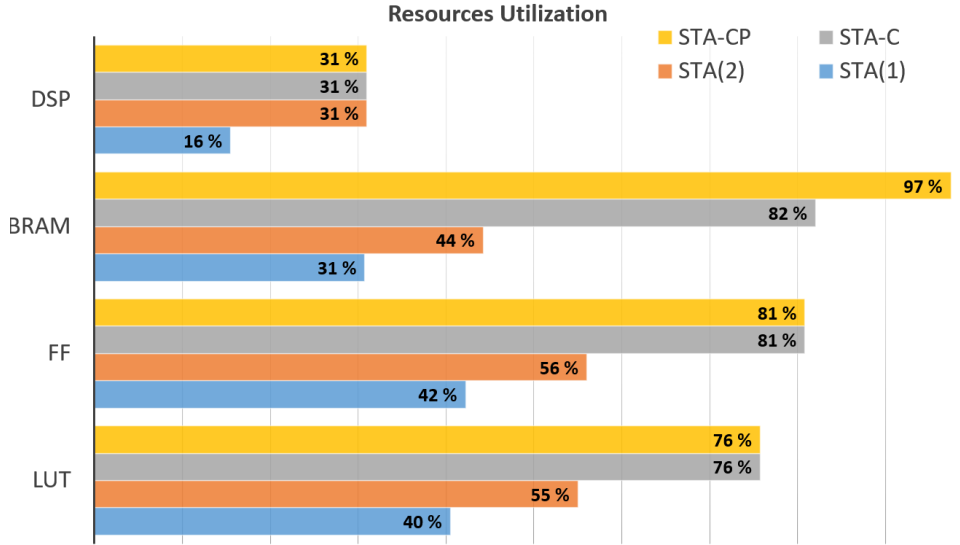


FIGURE 6.6: Resources Utilization of STA-CP Configurations

modules. DSP utilization remains constant at 31%, confirming that no further compute modules were introduced beyond those in STA(2).

- **STA-CP:** Enabling both caching and pre-fetching drives BRAM utilization to its peak at 97%, primarily due to the enlarged intermediate buffer size of 432KB required to support proactive data loading. LUT and FF utilization remain unchanged at 76% and 81%, respectively, indicating that pre-fetching strategy primarily adds memory control overhead without requiring any additional logic overhead. Similarly, DSP usage remains steady at 31%, confirming that compute parallelism remains constant across configurations.

6.3.5 Inference Latency Results

TABLE 6.3: Inference latency ($T_{\text{Inference}}$) for tested models across STA-CP configurations

Models	STA(1) [ms]	STA(2) [ms]	STA-C [ms]	STA-CP [ms]
ResNet-18	53.2	42.5	41.8	37.7
ResNet-34	95.0	76.1	75.0	66.2
ResNet-50	192.8	160.0	143.3	134.3
ResNet-101	310.6	257.7	235.5	216.4

Table 6.3 summarizes the inference latencies for all tested ResNet models across the STA-CP configurations. STA(1), corresponding to TT-VTA with a single compute module, establishes the baseline latency for each model. By introducing a second compute module in STA(2), inference times are reduced by 17–20% relative to STA(1), demonstrating the benefits of parallel execution of tensor operations across multiple compute modules.

In STA-C, deterministic caching further lowers latency by 1.4–10% compared to STA(2), as repeated accesses to DRAM are eliminated and frequently reused intermediate data are retained in on-chip memory. The STA-CP configuration, which

combines caching with static data pre-fetching, achieves the lowest inference times, providing an additional 6–12% improvement over STA-C.

Overall, STA-CP reduces latency by approximately 30% relative to STA(1) for all models, highlighting the cumulative impact of parallelism, caching, and pre-fetching. These results demonstrate the effectiveness of the proposed caching and pre-fetching strategies within different workloads. For example we have seen that some models benefit more from caching as they have high data dependencies, particularly for deeper networks with extensive reuse of intermediate tensors, such as ResNet-50 and ResNet-101. We have also seen models with little data dependency like ResNet-18 and ResNet-34 which benefited more from pre-fetching. However, the combined results is at the end the same using STA-CP architecture which can achieve approximately 30% reduction of inference latency of all models. This validate the effectiveness of the STA-CP.

Overall, the STA-CP architecture delivers a cumulative latency reduction of approximately 30% compared to STA(1) across all evaluated models, clearly demonstrating the synergistic effect of superscalar parallelism, deterministic cache utilization, and static data pre-fetching. The evaluation reveals a balanced and predictable performance gain across the different model structures: (i) deeper networks with strong data dependencies, such as ResNet-50 and ResNet-101, benefit primarily from data caching due to extensive intermediate tensor reuse, while shallower models like ResNet-18 and ResNet-34 benefit more from early data pre-fetching, which mitigates data stalls and improves pipeline continuity. Collectively, these results validate the architectural soundness of STA-CP, confirming its ability to sustain high throughput, efficient memory utilization, and fully deterministic execution across a range of deep learning workloads.

6.4 STA-CP Evaluation Beyond Resource-Constrained Configurations

In this section, we evaluate the impact of deterministic caching and pre-fetching on inference latency under multiple architectural configurations. The analysis compares cache-only configurations, denoted as STA-C, against configurations that integrate both caching and pre-fetching, denoted as STA-CP. In addition, the number of Compute units and the allocated cache capacity are varied to investigate their combined influence on architectural performance and scalability.

Unlike the previous evaluations, the Scheduler with Built-in Simulation was intentionally constrained from searching for the globally optimal architecture for the target AI model. Instead, the following architectural constraint was enforced throughout all experiments:

$$\text{No. of Load/Store Units} = \min(\text{No. of Compute Units}, \text{No. of Available Memories})$$

Under this setup, the memory subsystem was restricted to a single DRAM interface and a single Cache memory instance, while the number of Compute units was varied across three configurations consisting of 1x, 2x, and 4x Compute units. This controlled configuration enables systematic evaluation of the interaction between computational parallelism, cache capacity, and memory-access optimization mechanisms.

ResNet-101 was selected as the benchmark model due to its high computational complexity, deep network structure, and substantial memory footprint. The model consists of approximately 40,000 instructions, making it an appropriate workload for

stressing the memory subsystem and evaluating the scalability of the proposed architecture. Its intensive computation and frequent memory transactions provide clear visibility into the effects of cache utilization, pre-fetching efficiency, and functional-unit scaling.

Figure 6.7 illustrates the inference latency obtained for different STA-C and STA-CP configurations. The x-axis represents the allocated cache size, while the y-axis shows the measured inference latency in milliseconds. The evaluated configurations contain one (1xC), two (2xC), and four (4xC) Compute units.

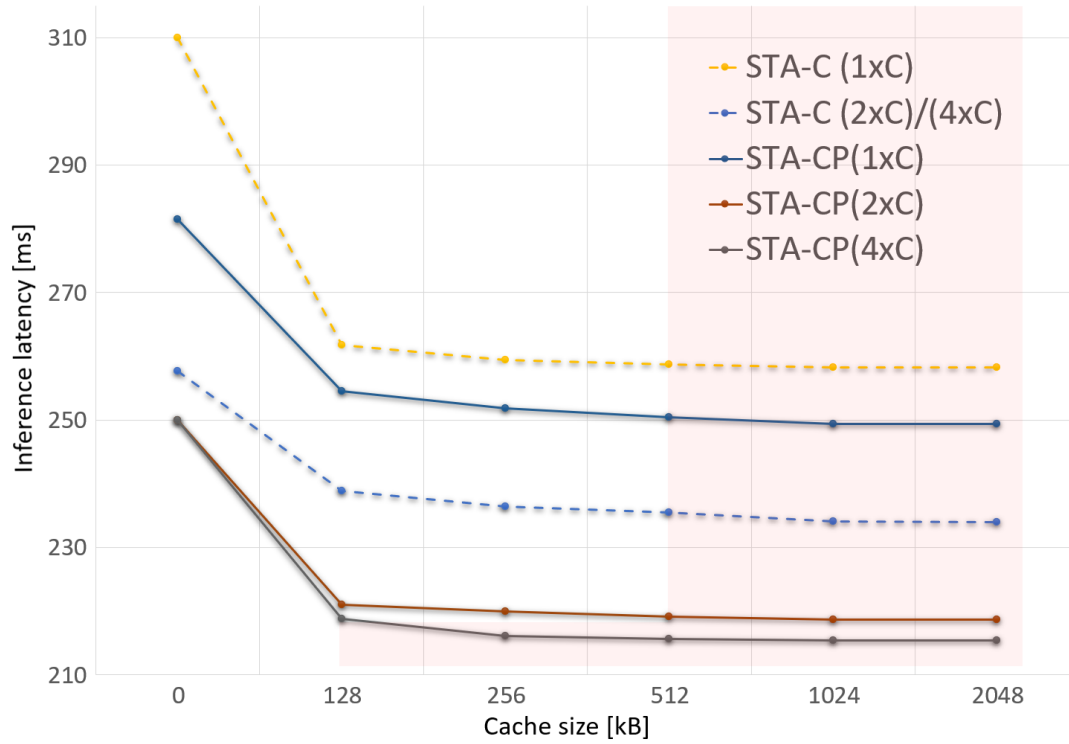


FIGURE 6.7: ResNet-101 Inference Latency under Different STA-C and STA-CP Configurations

It is important to note that the shaded red regions in Figure 6.7 correspond to configurations that exceed the available hardware resources of the target FPGA platform. Consequently, these configurations were validated exclusively through the Scheduler with Built-in Simulation and were not physically synthesized or deployed on the FPGA. These simulation-based evaluations were included to analyze the scalability behavior of the proposed architecture beyond the practical hardware limitations of the evaluated platform and to provide insight into the potential benefits achievable on larger FPGA devices.

The results presented in Section 6.3.5 demonstrate that the STA(2) architecture without caching or pre-fetching remains strongly constrained by DRAM-access latency. Although increasing the number of Compute units improves performance compared to the single-Compute-unit configuration (i.e., STA(1)), the achieved gains remain limited because the additional computational resources cannot be efficiently utilized while memory operations continue to stall execution. This behavior confirms that shared DRAM access constitutes a major performance bottleneck within the architecture.

When deterministic caching is enabled (STA-C), a noticeable reduction in inference latency is observed across all configurations. Even relatively small cache capacities

significantly improve performance by storing frequently reused tensors and intermediate feature maps within BRAM resources, thereby reducing the frequency of external DRAM accesses. For example, the STA-C configuration with one Compute unit reduces inference latency from approximately 310ms to nearly 261ms once a cache size of 128KB is introduced. Similar improvements can also be observed for the configurations containing two and four Compute units.

As cache capacity increases, the performance improvement gradually stabilizes. This behavior indicates that the working set of the ResNet-101 workload becomes sufficiently accommodated within the available cache space, thereby reducing the marginal benefit of allocating additional BRAM capacity beyond approximately 1MB of cache memory.

The largest performance improvements are achieved when deterministic pre-fetching is integrated alongside caching in the STA-CP architecture. In this configuration, data transfers from DRAM and BRAM are statically scheduled ahead of execution and overlapped with computation phases. Consequently, Compute units can access the required data with significantly reduced waiting times, minimizing execution stalls and improving overall pipeline utilization. The performance improvements become particularly evident in the configurations containing multiple Compute units. For example, the STA-CP (2×C) and STA-CP (4×C) architectures, incorporating two and four Compute units, respectively, can theoretically achieve inference latencies of approximately 216ms and 214ms. A similar latency improvement is observed for the single-Compute-unit configuration.

6.5 Discussion on STA-CP

This chapter presented the Superscalar Time-Triggered Versatile Tensor Accelerator with Caching and Pre-fetching (STA-CP), a novel deterministic acceleration framework that extends the time-triggered paradigm to enable predictable memory orchestration. By transforming caching and pre-fetching—traditionally dynamic and heuristic-driven mechanisms—into statically scheduled, compile-time operations, STA-CP enables both high-performance inference execution and analytically tractable temporal behaviour.

Experimental evaluation on the Xilinx Zynq UltraScale+ ZCU104 FPGA platform demonstrates that STA-CP achieves up to a 30% reduction in inference latency across diverse convolutional neural network workloads. These improvements are primarily attributed to deterministic overlap of computation and memory access, reduced DRAM contention, and efficient reuse of intermediate data within the on-chip cache hierarchy.

Beyond these performance gains, STA-CP introduces the broader design principle of memory-aware determinism, demonstrating that predictable memory management and real-time analyzability can be achieved without compromising throughput. By co-optimizing computation and memory scheduling within a unified time-triggered framework, the proposed architecture advances the state of the art in deterministic AI acceleration. The corresponding worst-case execution time (WCET) analysis and validation of STA-CP are provided in Chapter 7 (Timing Analysis for STA-CP) and Chapter 8 (WCET Validation for STA-CP), respectively, ensuring formal grounding of its timing guarantees.

Finally, future work will explore extensions toward multi-model and mixed-criticality execution, as well as adaptive yet deterministic caching strategies capable of optimizing data reuse under dynamic workload conditions. Collectively, these

directions further position STA-CP as a foundation for next-generation, certifiable AI systems in safety-critical domains.

Chapter 7

Timing Analysis of STA-CP

7.1 Overview and Motivation

Ensuring bounded and predictable execution latency is a fundamental requirement for deploying AI accelerators in real-time and safety-critical systems. In contrast to conventional throughput-oriented AI accelerators, which primarily optimize average-case performance, the proposed *Superscalar Time-Triggered Versatile Tensor Accelerator with Cache and Pre-fetching* (STA-CP) is explicitly designed to guarantee deterministic execution behavior and analyzable timing semantics. This determinism enables rigorous *Worst-Case Execution Time* (WCET) analysis, which is essential for system certification and temporal verification in domains such as autonomous driving, avionics, robotics, and industrial automation.

Traditional AI accelerator architectures rely heavily on runtime scheduling, speculative execution, adaptive cache policies, and dynamic arbitration of shared resources. Although these techniques improve throughput and resource utilization, they introduce timing variability and execution nondeterminism, making accurate WCET estimation extremely difficult. In contrast, the STA-CP architecture eliminates these sources of unpredictability through a fully static execution model in which all operations are analyzed, scheduled, and coordinated at compile time.

The proposed architecture achieves WCET analyzability through several tightly integrated mechanisms. First, all execution activities are coordinated through a *Global Time Base* (GTB), enabling a unified and synchronized time-triggered execution across the entire accelerator architecture. Second, instruction execution schedules are generated statically using the Scheduler with Built-in Simulation (i.e., STA-CP Scheduler), ensuring that the start and completion instances of all instructions are known before deployment. Third, memory behavior is made deterministic through compile-time cache management and schedule-aware pre-fetching, thereby eliminating runtime cache misses and speculative memory operations. Finally, all shared-resource contention is removed through schedule-aware arbitration and temporal partitioning of memory transactions.

As a result, the execution timeline of the accelerator becomes fully deterministic and analytically tractable. Consequently, the WCET of the accelerator can be computed analytically rather than estimated probabilistically through runtime profiling or statistical analysis. This property represents one of the key contributions of the proposed STA-CP architecture.

7.2 Formal Definition of WCET

Let $\mathcal{I}'$ denote the scheduled instruction set after metadata enrichment, scheduling, cache adaptation, and pre-fetching analysis. The WCET of the STA-CP accelerator

executing a given AI model is formally defined as:

$$WCET = \max_{i \in I'} (\text{Finish}(i))$$

where $\text{Finish}(i)$ represents the statically determined completion time of instruction i .

Since all instruction start times and execution durations are determined during scheduling, the WCET corresponds to the final completion event across all functional units within the accelerator architecture. In other words, the WCET represents the latest execution timestamp in the globally synchronized execution schedule generated by the scheduler.

7.3 Instruction-Level Timing Analysis

The timing analysis process begins at the instruction level, where the execution latency of each instruction is individually determined according to its type and memory behavior. Since the proposed architecture contains both computation-bound and memory-bound operations, different timing-analysis models are utilized depending on the instruction characteristics.

1. Compute and BRAM-Based Instructions For compute instructions and memory operations targeting on-chip BRAM resources, the completion time is defined as:

$$\text{Finish}(i) = \text{Start}(i) + \text{cycles}(i)$$

where $\text{cycles}(i)$ denotes the execution latency obtained through cycle-accurate behavioral simulation using the Xilinx simulation environment.

These instructions exhibit highly deterministic timing behavior because BRAM access latency and computational execution latency remain fixed once the architecture is synthesized. Consequently, the execution latency of these operations can be accurately determined at design time.

2. DRAM-Based Memory Instructions For memory instructions targeting external DRAM, the completion time is determined using DRAMSim2:

$$\text{Finish}(i) = \text{Start}(i) + \text{DRAMSim2}(i)$$

where $\text{DRAMSim2}(i)$ represents the deterministic upper-bound latency of the corresponding DRAM transaction.

The use of DRAMSim2 enables accurate modeling of external-memory behavior, including row activations, bank conflicts, refresh delays, and memory-access timing constraints. However, unlike conventional architectures where DRAM contention introduces runtime variability, the proposed STA-CP architecture eliminates dynamic memory contention through statically scheduled memory accesses. Consequently, the DRAM timing behavior remains deterministic and analyzable.

The WCET can therefore be reformulated as:

$$WCET = \max_{i \in I'} (\text{Start}(i) + C(i))$$

where:

$$C(i) = \begin{cases} \text{cycles}(i), & \text{if } i \text{ is compute or BRAM-based} \\ \text{DRAMSim2}(i), & \text{otherwise} \end{cases}$$

This formulation highlights that the total execution time of each instruction is composed of its statically scheduled start time and its deterministic execution latency.

7.4 Module-Level WCET Composition

Due to the superscalar nature of the STA-CP architecture, execution is distributed across multiple functional-unit instances:

$$\mathcal{M}' = \{L^{(x)}, C^{(y)}, S^{(z)}\}$$

where:

- $L^{(x)}$ denotes Load modules,
- $C^{(y)}$ denotes Compute modules,
- $S^{(z)}$ denotes Store modules.

Each module instance maintains its own statically generated execution schedule:

$$\text{Sched}_{m(ID)} = \{\text{Start}(i_1), \text{Start}(i_2), \dots\}$$

The WCET contribution of an individual module is therefore defined as:

$$\text{WCET}_{m(ID)} = \max_{i \in iq_{m(ID)}} \text{Finish}(i)$$

where $iq_{m(ID)}$ represents the instruction queue assigned to module $m(ID)$.

The global WCET of the accelerator corresponds to the maximum completion time across all module instances:

$$\text{WCET} = \max_{m(ID) \in \mathcal{M}'} \text{WCET}_{m(ID)}$$

This modular timing formulation enables scalable WCET analysis even when multiple functional units are instantiated within the architecture.

7.5 Impact of Deterministic Scheduling

The deterministic execution behavior of the STA-CP architecture is fundamentally enabled through static scheduling performed by the Scheduler with Built-in Simulation. During scheduling, all dependencies, execution orders, and memory transactions are resolved prior to deployment. Thus, The scheduler guarantees:

- static dependency resolution,
- contention-free execution,
- deterministic instruction ordering,
- and globally synchronized execution timing.

For intra-module execution ordering, the scheduler enforces:

$$\forall i, j \in iq_m(ID), i < j \Rightarrow \text{Finish}(i) \leq \text{Start}(j)$$

This condition guarantees strictly sequential execution within each functional unit and prevents overlapping operations that could introduce timing uncertainty.

Furthermore, instruction start times are defined as:

$$\text{Start}(i) = \max(\text{Finish}(\text{pred}(i)), \text{Finish}(j))$$

where:

- $\text{pred}(i)$ represents the predecessor instruction,
- j represents the latest conflicting operation.

Consequently, every instruction start time becomes a deterministic function of:

- predecessor completion times,
- dependency resolution,
- and memory-availability constraints.

This deterministic scheduling model eliminates runtime variability and enables exact timing analysis.

7.6 Memory System Contribution to WCET

In deep-learning workloads, memory latency often dominates overall inference execution time. Consequently, deterministic memory behavior is essential for achieving analyzable WCET bounds.

7.6.1 Deterministic Cache Behavior

The STA-CP architecture introduces deterministic cache management through compile-time cache allocation and statically scheduled cache operations.

The memory target of each instruction is determined during scheduling:

$$\text{Target}(i) = \begin{cases} \text{BRAM}, & \mu_i.\text{CacheFlag} = 1 \\ \text{DRAM}, & \text{otherwise} \end{cases}$$

Since all cache-placement and cache-eviction decisions are resolved statically:

- runtime cache misses are eliminated,
- dynamic replacement policies are removed,
- memory-access latency becomes constant and predictable.

This behavior fundamentally differs from conventional cache hierarchies that rely on adaptive runtime replacement strategies such as LRU or pseudo-LRU.

7.6.2 Static Pre-fetching

The STA-CP architecture further improves memory behavior through deterministic pre-fetching. With which, instructions are inserted only during statically identified idle memory intervals:

$$\text{Start}(i_l) \leq \text{Start}_{\text{demand}}(i_l)$$

As a result:

- data becomes available before computation begins,
- memory stalls are minimized,
- and execution overlap is improved.

Importantly, because pre-fetching is statically scheduled rather than speculative, it introduces no runtime interference or timing variability.

7.6.3 Contention-Free Memory Access

To eliminate runtime contention, memory transactions are temporally partitioned during scheduling:

$$\text{Start}(i) = \text{Finish}(\text{last}(M_{\text{memory}}))$$

This condition guarantees:

- compile-time resolution of mutual exclusion,
- elimination of runtime queuing delays,
- and deterministic memory-access timing.

Consequently, DRAM behavior becomes predictable and suitable for WCET analysis.

7.7 Effect of Superscalar Scaling on WCET

The STA-CP architecture supports deterministic superscalar execution through the instantiation of additional functional units.

The Configuration Manager dynamically introduces additional module instances during scheduling whenever bottlenecks or pending instructions are detected:

$$\exists T_{m(ID)} \Rightarrow \text{No pending instruction remains}$$

This superscalar scaling enables:

- parallel execution of tensor operations,
- reduction of the critical execution path,
- elimination of statically detectable bottlenecks,
- and improved pipeline utilization.

Importantly, all scaling decisions occur before deployment. Therefore, superscalar execution does not introduce runtime nondeterminism.

7.8 Buffering and WCET Stability

Intermediate buffering structures are essential for maintaining continuous pipeline execution and preventing runtime stalls caused by data dependencies.

The required buffer size is defined as:

$$dB_{src(ID),dst(ID)}^{max} = \max_t dB_{src(ID),dst(ID)}(t)$$

This guarantees:

- no data overwrites,
- no buffer underflow,
- no runtime back-pressure,
- and uninterrupted pipeline execution.

Consequently, buffering removes another major source of execution-time variability and contributes to WCET stability.

7.9 End-to-End WCET Expression

Combining all timing constraints and execution dependencies, the complete WCET formulation of the STA-CP architecture becomes:

$$WCET = \max_{i \in \mathcal{I}'} (\max (\text{Finish}(\text{pred}(i)), \text{Finish}(\text{dep}(i)), \text{Finish}(\text{mem}(i))) + C(i))$$

where:

- $\text{pred}(i)$ denotes the predecessor instruction within the same module,
- $\text{dep}(i)$ denotes inter-module dependencies,
- $\text{mem}(i)$ denotes the latest memory operation on the same memory controller,
- and $C(i)$ denotes the execution latency of instruction i .

This expression captures the complete timing semantics of the STA-CP architecture and formally demonstrates that all execution behavior remains analyzable at compile time.

7.10 Discussion

The STA-CP architecture enables precise WCET analysis because all major sources of runtime nondeterminism are removed through static scheduling and deterministic resource management.

More specifically, the architecture provides:

- **Static Scheduling:** all execution events are known before deployment,
- **Deterministic Memory Behavior:** cache and DRAM accesses are statically controlled,

- **Contention Elimination:** memory operations are temporally partitioned,
- **Predictable Parallelism:** superscalar scaling is resolved statically,
- **Stable Buffering Semantics:** pipeline execution remains uninterrupted.

As a result, the proposed STA-CP architecture enables predictable and WCET-analyzable inference execution under the modeled and controlled STA-CP execution assumptions, while still supporting superscalar execution and deterministic memory-latency optimization. These properties make the architecture a strong candidate for real-time and safety-critical AI systems that require both high computational performance and strict timing guarantees.

Chapter 8

WCET Validation of the STA-CP

This chapter validates the Worst-Case Execution Time (WCET) analysis developed in Chapter 7 for the proposed *Superscalar Time-Triggered Versatile Tensor Accelerator with Caching and Pre-fetching (STA-CP)* architecture. The validation focuses on three timing-related properties: *WCET accuracy*, *determinism*, and *composability*. In this context, accuracy refers exclusively to the correspondence between the WCET predicted by the Scheduler with Built-in Simulation and the execution latency measured on FPGA hardware. It does not refer to the classification accuracy of the neural-network models.

The purpose of this chapter is to demonstrate that the STA-CP architecture provides an experimentally verifiable timing model. More specifically, the evaluation investigates whether the statically derived WCET remains valid after synthesis and deployment, whether repeated executions exhibit identical timing behavior, and whether these guarantees are preserved across different models and architecture configurations.

8.1 Validation Objectives

The first objective is **WCET accuracy**. This evaluates whether the execution latency predicted at compile time by the Scheduler with Built-in Simulation (i.e., STA-CP Scheduler) corresponds to the measured latency of the deployed FPGA implementation. The predicted latency is denoted as $T_{\text{STA-CP}}(\text{sim})$ and represents the WCET obtained from the static execution schedule.

The second objective is **determinism**. This evaluates whether repeated executions of the same model under the same architectural configuration produce identical execution times. Since STA-CP relies on static scheduling, deterministic cache management, and schedule-aware pre-fetching, no runtime arbitration or adaptive behavior should affect the execution latency.

The third objective is **composability**. This evaluates whether the WCET model remains valid when the workload or architecture configuration changes. STA-CP supports different numbers of load-, compute-, and store-modules, cache sizes, and pre-fetching configurations. Therefore, the validation must confirm that architectural scaling does not introduce timing unpredictability.

Together, these objectives validate the main claim of this thesis: STA-CP provides a deterministic and composable AI acceleration architecture whose WCET can be predicted at compile time and verified experimentally on hardware.

8.2 Relation to the Timing Analysis Model

As defined in Chapter 7, the WCET of a model executed on STA-CP is computed as:

$$T_{\text{WCET}}(\text{Model}) = \max_{i \in \mathcal{I}} \text{Finish}(i)$$

where $\mathcal{I}$ denotes the scheduled instruction set and $\text{Finish}(i)$ denotes the statically determined completion time of instruction i .

The scheduler derives these completion times using instruction-dependency analysis, cycle-accurate timing for compute-bound operations as well as for on-chip memory operations, and DRAMSim2-based timing for off-chip memory accesses. Therefore, the WCET generated during scheduling is defined as:

$$T_{\text{WCET}} \equiv T_{\text{STA-CP}}(\text{sim})$$

Since STA-CP eliminates runtime arbitration, dynamic scheduling, adaptive cache replacement, and speculative memory behavior, the measured execution time is expected to match the statically generated WCET.

8.3 Experimental Validation Setup

The validation was conducted on a Xilinx Zynq UltraScale+ ZCU104 FPGA platform. The STA-CP accelerator was deployed in the Programmable Logic (PL), while the Processing System (PS) was used for execution control, data transfer, and measurement.

Several ResNet models were used, including ResNet-18, ResNet-34, ResNet-50, and ResNet-101. These models provide different instruction-stream lengths, dependency structures, and memory-access intensities, making them suitable for validating the timing model across workloads of increasing complexity.

For each model and architecture configuration, two timing values were collected. The first value is the predicted WCET generated by the Scheduler with Built-in Simulation. The second value is the measured execution latency obtained on the FPGA using the Global Time Base (GTB):

$$T_{\text{Inference(PL)}} = GTB_{\text{After}} - GTB_{\text{Before}}$$

where GTB_{Before} is captured immediately before inference execution and GTB_{After} is captured after inference completion. This measurement captures the execution latency inside the programmable logic and therefore reflects the timing behavior of the deployed STA-CP accelerator itself.

8.4 WCET Accuracy Validation

WCET accuracy is validated by comparing the predicted WCET with the measured execution latency on FPGA hardware. The expected relationship is:

$$T_{\text{Inference(PL)}}(\text{Model}) = T_{\text{WCET}}(\text{Model})$$

This equality indicates that the scheduler-generated timing model accurately predicts the execution latency of the deployed accelerator.

In conventional AI accelerators, such correspondence is difficult to achieve because runtime scheduling, cache variability, memory contention, and arbitration delays may alter execution timing. Consequently, WCET values are usually conservative upper bounds rather than exact timing predictions.

In STA-CP, all instruction execution times, memory accesses, cache operations, and pre-fetching operations are statically scheduled. Therefore, no runtime event

can extend execution beyond the generated schedule. The validation of this equality confirms that the proposed WCET-analysis framework accurately represents the hardware execution behavior.

8.5 Determinism Validation

Determinism is validated by repeatedly executing the same model under the same architectural configuration and observing whether the measured execution latency remains constant across executions. Since the STA-CP architecture relies entirely on statically generated schedules, deterministic cache management, and schedule-aware pre-fetching, the execution behavior is expected to remain temporally stable and free from runtime-induced timing variability.

For n repeated executions, deterministic behavior requires:

$$T_{\text{Inference(PL)}}^{(1)} = T_{\text{Inference(PL)}}^{(2)} = \dots = T_{\text{Inference(PL)}}^{(n)}$$

Equivalently, the execution-time variance should approach zero:

$$\sigma_T^2 \approx 0$$

where:

$$\sigma = \sqrt{\frac{1}{n} \sum_{k=1}^n (T^{(k)} - \mu_T)^2}$$

where:

$$\mu_T = \frac{1}{n} \sum_{k=1}^n T^{(k)}$$

- σ is the standard deviation,
- n is the number of executions,
- $T^{(k)}$ is the measured inference latency of execution k ,
- μ_T is the mean inference latency.

To experimentally validate this property, multiple inference executions were performed for each evaluated model and architectural configuration. Table 8.1 summarizes representative measurements obtained from 100 consecutive inference executions of the ResNet-50 model on the deployed FPGA platform.

TABLE 8.1: Determinism Validation of STA-CP Compared to Conventional VTA

Architecture	Mean Latency [ms]	Min [ms]	Max [ms]	Std. Dev. [ms]	Jitter [ms]
VTA	188.6	181.9	197.4	4.82	15.5
STA(1) = TT-VTA	192.8	192.78	192.82	0.02	0.04
STA(2) = STT-VTA	160.0	159.98	160.02	0.02	0.04
STA-CP	134.3	134.29	134.31	0.01	0.02

The results demonstrate that the conventional event-triggered VTA architecture exhibits noticeable timing variability caused primarily by runtime memory contention,

dynamically varying DRAM-access latency, and event-driven synchronization behavior. In contrast, the proposed deterministic architectures significantly reduce execution-time variance.

In particular, the STA-CP architecture exhibits nearly identical execution latency across all repeated runs, with a measured standard deviation of approximately

$$\sigma = 0.01 \text{ ms} = 10 \mu\text{s}.$$

Although a non-zero variation is observed, its magnitude is negligible relative to the overall inference latency of the evaluated models. For example, for an STA-CP inference latency of 134.3 ms , the relative standard deviation is

$$\frac{\sigma}{\mu_T} \times 100 = \frac{0.01}{134.3} \times 100 \approx 0.0074\%.$$

Thus, the observed variation corresponds to less than one-hundredth of one percent of the total execution time, indicating highly stable execution behavior across repeated runs.

To further assess the significance of this variation, the FPGA platform operates at a clock frequency of 333 MHz , corresponding to a clock period of

$$T_{clk} = \frac{1}{f_{clk}} = \frac{1}{333 \times 10^6} \approx 3.0 \text{ ns}.$$

Consequently, the measured standard deviation corresponds to approximately

$$\frac{10 \mu\text{s}}{3.0 \text{ ns}} \approx 3333$$

clock cycles. While this value may appear noticeable when expressed in clock cycles, it remains negligible compared to the total number of cycles required for inference execution, which is on the order of tens of millions of cycles.

Furthermore, the remaining variation originates primarily from the measurement infrastructure rather than from the STA-CP architecture itself. More specifically, timer-resolution limitations, Global Time Base (GTB) sampling granularity, synchronization overheads between the Processing System (PS) and the Programmable Logic (PL), and software overhead associated with reading performance counters can introduce minor deviations into the recorded measurements. Since these effects occur outside the accelerator execution path, they do not represent architectural nondeterminism.

Consequently, the experimental results confirm that the STA-CP architecture exhibits deterministic execution behavior which validates that:

- instruction start times remain fixed across executions,
- DRAM accesses occur within statically allocated execution windows,
- cache behavior remains deterministic,
- pre-fetching operations execute according to predefined schedules,
- and runtime arbitration is completely eliminated.

Therefore, the proposed architecture provides a temporally stable execution environment with analyzable timing characteristics, making it suitable for WCET-based verification and deployment in safety-critical real-time systems that require strong guarantees on execution predictability and bounded response times.

8.6 Composability Validation

Composability is validated by evaluating whether the WCET model remains accurate and deterministic across different models and architectural configurations. Let the set of evaluated configurations be:

$$Config_j \in \mathcal{C}$$

where each configuration may be represented as:

$$Config_j = (C_j, L_j, S_j, Cache_j, Prefetch_j)$$

with C_j , L_j , and S_j denoting the number of Compute, Load, and Store modules, respectively, and $Cache_j$ and $Prefetch_j$ representing the cache and pre-fetching configuration.

For each model and configuration, the scheduler generates an independent execution schedule:

$$Sched(Model, Config_j)$$

The corresponding WCET is:

$$T_{WCET}(Model, Config_j) = \max_{i \in \mathcal{I}(Model, Config_j)} Finish(i)$$

Composability requires that the measured latency matches the predicted WCET for each evaluated configuration:

$$T_{Inference(PL)}(Model, Config_j) = T_{WCET}(Model, Config_j)$$

This relationship demonstrates that architectural changes, such as increasing the number of Compute modules or modifying the cache and pre-fetching configuration, do not invalidate the timing model. Instead, all timing effects introduced by architectural scaling are captured during scheduling.

8.7 System-Level Latency Considerations

The WCET validation in this chapter focuses on accelerator execution within the programmable logic. However, complete end-to-end inference latency also includes system-level overheads. Therefore, the total inference latency can be expressed as:

$$T_{Inference} = T_{WCET} + T_{Overhead}$$

where $T_{Overhead}$ includes data transfer between the Processing System and DRAM, instruction loading, accelerator initialization, and result retrieval.

These overheads are external to the deterministic STA-CP execution engine and therefore do not affect the correctness of the WCET model. Nevertheless, they remain relevant for complete system-level response-time analysis.

8.8 Discussion

The validation criteria and results demonstrate that the STA-CP architecture provides an exact and experimentally verifiable timing model. WCET accuracy confirms that

the scheduler-generated timing prediction corresponds to the measured hardware execution latency. Determinism confirms that repeated executions exhibit stable timing behavior without observable jitter. Composability confirms that these timing properties remain valid across different workloads and architectural configurations.

This behavior is made possible by the combination of static scheduling, time-triggered execution, deterministic memory management, deterministic caching, and schedule-aware pre-fetching. Unlike conventional AI accelerators, where WCET estimation is often pessimistic due to runtime variability, STA-CP enables precise timing guarantees because the runtime execution follows the statically generated schedule.

Chapter 9

Discussion and Conclusion

In conclusion, this thesis presented a comprehensive hardware–software co-design methodology for enabling deterministic and analyzable AI inference execution in safety-critical real-time systems. The central contribution is the development of the *Superscalar Time-Triggered Versatile Tensor Accelerator with Cache and Pre-fetching (STA-CP)*, a novel AI accelerator architecture that combines time-triggered execution, static scheduling, deterministic memory management, and model-driven architectural specialization to provide predictable and WCET-analyzable inference execution.

Unlike conventional AI accelerators, which primarily optimize average-case throughput through dynamic scheduling and runtime arbitration, the proposed STA-CP architecture adopts a fundamentally different execution paradigm. All tensor operations, memory transactions, cache accesses, and pre-fetching activities are statically analyzed, scheduled, and coordinated at compile time. By eliminating runtime arbitration and dynamically competing memory accesses, the proposed architecture removes major sources of timing variability and execution nondeterminism that commonly exist in modern AI accelerators.

Through the progression of TT-VTA, STT-VTA, and finally STA-CP, this thesis systematically addressed the three research questions defined at the beginning of this work.

Regarding Q1: “How can AI hardware accelerators be designed to provide deterministic and WCET-analyzable inference execution for safety-critical real-time systems?”, this thesis demonstrated that deterministic AI acceleration can be achieved through the integration of:

- Time-triggered execution models,
- Static instruction scheduling,
- Compile-time dependency analysis,
- Deterministic resource management,
- And globally synchronized execution control.

The proposed TT-VTA architecture introduced the transition from conventional event-triggered execution toward a fully time-triggered execution model. The integration of the TT-Dispatcher and the Global Time Base (GTB) enabled all functional units to execute according to statically generated schedules with predefined start and completion times. Consequently, inference execution became fully deterministic and reproducible across executions.

Furthermore, this thesis introduced a WCET estimation methodology that combines cycle-accurate behavioral simulation of compute instructions with DRAM

latency analysis using DRAMSim2. The resulting framework enables accurate prediction and validation of execution latency for both compute-bound and memory-bound operations. Experimental evaluations demonstrated close agreement between statically predicted execution times and measured hardware execution latencies, validating the correctness and reliability of the proposed timing model.

Regarding Q2: “How can the performance of AI hardware accelerators be improved through parallelism without affecting determinism?”, this thesis demonstrated that deterministic superscalar parallelism can significantly accelerate inference execution while preserving predictability. The proposed STT-VTA architecture extended the TT-VTA by introducing multiple statically scheduled functional units capable of parallel execution.

The Scheduler with Built-in Simulation dynamically analyzes model dependencies, instruction-level parallelism, and memory constraints to determine the optimal number of functional units required for a given AI model. This model-driven design methodology enables architecture specialization at compile time while maintaining deterministic execution semantics.

Experimental results demonstrated substantial inference-latency improvements ranging from approximately 17% to 41% compared to TT-VTA, depending on the complexity of the evaluated AI model and the number of instantiated Compute units. More importantly, these improvements were achieved without introducing runtime scheduling variability or interference, thereby preserving the deterministic execution guarantees required in safety-critical environments.

Regarding Q3: “How can memory interference be eliminated while hiding memory latency and maintaining high inference performance?”, this thesis introduced deterministic caching and static pre-fetching mechanisms through the STA-CP architecture.

Unlike conventional dynamic cache systems that rely on runtime replacement policies such as LRU, the proposed deterministic cache operates entirely under compile-time control. Cache allocation, reuse, eviction, and memory placement are statically determined during scheduling, eliminating unpredictable cache behavior and ensuring timing determinism.

In addition, the proposed schedule-aware pre-fetching mechanism statically inserts memory-transfer operations into predefined idle intervals of the memory subsystem. This enables data to be loaded ahead of execution without introducing runtime contention or interference. By overlapping memory transfers with computation, the architecture significantly reduces compute stalls caused by DRAM latency.

Experimental evaluations demonstrated that deterministic caching and pre-fetching substantially alleviate the DRAM bottleneck, improve pipeline utilization, and enable efficient scaling with additional Compute units. The STA-CP architecture achieved major inference-latency reductions approximately 30% while preserving deterministic execution behavior, even under external DRAM interference scenarios.

Beyond the architectural contributions, this thesis also introduced a fully integrated development and deployment toolchain built upon the Scheduler with built-in simulator, Apache TVM, customized VTA backends, DRAMSim2, and the Xilinx Vivado framework. This toolchain automates:

- Instruction extraction and dependency analysis,
- Timing-aware scheduling,
- Architecture specialization,
- Functional-unit generation,

- Deterministic cache allocation,
- Schedule generation,
- Hardware synthesis,
- And FPGA deployment.

This automated workflow enables scalable development of deterministic AI accelerators while significantly reducing manual integration complexity.

The practical significance of this work is particularly relevant for safety-critical application domains such as autonomous driving, industrial automation, avionics, robotics, and medical systems, where bounded latency, and predictable timing behavior and execution are mandatory requirements. The proposed STA-CP architecture demonstrates that AI inference acceleration and real-time determinism are not mutually exclusive objectives, but can instead be jointly achieved through careful architectural co-design and static system analysis.

Nevertheless, the proposed approach involves deliberate trade-offs. The reliance on compile-time scheduling and static architectural configuration reduces runtime flexibility and may lead to conservative resource utilization in certain scenarios. Furthermore, scalability remains constrained by available FPGA resources, BRAM capacity, and memory bandwidth limitations. However, these trade-offs are necessary to guarantee temporal determinism, and WCET-analyzability that are fundamentally required in safety-critical systems.

Finally, Table 9.1 summarizes the positioning of the proposed STA-CP architecture relative to conventional AI accelerators and throughput-oriented optimized systems. The comparison highlights the fundamental distinction of STA-CP: rather than optimizing solely for peak throughput or average-case efficiency, the architecture prioritizes deterministic execution, WCET analyzability, and contention-free memory behavior, and certification-oriented design principles. The table further illustrates how the proposed architecture combines competitive inference performance with predictable execution behavior, thereby addressing the core limitations of existing AI accelerator approaches in real-time environments.

Overall, this thesis advances the state of the art in AI hardware acceleration by introducing a deterministic, time-triggered execution paradigm for AI inference. The proposed STA-CP architecture demonstrates that predictable, interference-free, and WCET-analyzable AI acceleration is achievable through the integration of static scheduling, deterministic memory management, superscalar parallelism, and schedule-aware caching and pre-fetching. By bridging the gap between high-performance AI acceleration and real-time system requirements, this work establishes a strong foundation for the future deployment of AI in safety-critical embedded systems.

TABLE 9.1: Conclusion Matrix: Positioning of STA-CP Against Existing Approaches

Aspect	Conventional Accelerators (e.g., FINN, VTA)	Optimized (Caches, GPU/TPU)	Systems (Pre-fetching,)	Proposed STA-CP Architecture
Execution Model	Dynamic scheduling, run-time arbitration	Dynamic, throughput-oriented execution		Time-triggered scheduling (compile-time defined)
Determinism	Not guaranteed (timing varies per run)	Non-deterministic due to cache/memory behavior		Fully deterministic execution (zero timing jitter)
WCET Analyzability	Not supported	Not supported (average-case optimized)		Explicit WCET derivation and validation
Memory Behavior	Runtime-managed, contention-prone	Cache-based, dynamic replacement (e.g., LRU)		Statically scheduled, contention-free memory access
Cache Design	Limited or absent	Dynamic (LRU, pseudo-LRU)		Deterministic cache with compile-time allocation and eviction
Pre-fetching	Limited/static	Speculative and runtime-based		Schedule-aware, conflict-free pre-fetching
Performance Objective	Peak throughput (model-specific optimization)	High average throughput and efficiency		Predictable performance with competitive efficiency
Flexibility	Requires recompilation per model	High runtime adaptability		Model-driven (compile-time adaptation)
Scalability Limits	Resource growth (e.g., LUT explosion)	Memory bandwidth and contention		Bounded by memory bandwidth and BRAM, explicitly modeled
Real-Time Suitability	Limited	Not suitable (no guarantees)		Designed for real-time and safety-critical systems
Certification Readiness	Not applicable	Not applicable		Suitable for certifiable systems (predictable timing)
Trade-offs	High performance, low predictability	High efficiency, high variability		Reduced flexibility in exchange for determinism and WCET guarantees

Chapter 10

Outlook and Future Research Directions

The research presented in this thesis establishes a systematic foundation for deterministic and WCET-analyzable AI acceleration in safety-critical real-time systems. Through the integration of time-triggered execution semantics, static scheduling methodologies, controlled superscalar parallelism, deterministic caching, and schedule-aware pre-fetching, the proposed TT-VTA, STT-VTA, and STA-CP architectures demonstrate that high-performance AI inference and strict temporal predictability can be achieved simultaneously within FPGA-based accelerator platforms.

Beyond the architectural contributions themselves, this work introduces a broader design methodology for predictable AI computing infrastructures. As AI systems continue to evolve toward increasingly autonomous, interconnected, and safety-critical applications, the need for deterministic accelerator architectures with analyzable execution behavior is expected to become significantly more important. The concepts introduced in this thesis therefore open several strategic directions for future research and technological development.

Evolution Towards Next-Generation AI Models

Future AI systems are expected to increasingly rely on computationally intensive architectures such as Vision Transformers (ViTs) [108], Large Language Models (LLMs) [109], diffusion models [110], Graph Neural Networks (GNNs) [111], and multi-modal foundation models [112]. Compared to conventional CNN-based inference, these models introduce substantially more complex execution characteristics, including irregular tensor dependencies, dynamic attention mechanisms, significantly larger parameter spaces, and increased memory bandwidth demands.

The deterministic execution and architecture-specialization concepts proposed in this thesis provide a suitable foundation for future predictable accelerators targeting such workloads. In particular, future research may investigate deterministic scheduling mechanisms for attention-based computation, predictable tensor-stream orchestration, statically analyzable model partitioning, and scalable deterministic memory hierarchies capable of supporting large-scale AI models while preserving temporal guarantees.

Such developments may enable the deployment of advanced generative and reasoning AI systems within safety-critical domains where bounded latency, predictable execution, and formal timing analysis remain mandatory requirements.

Distributed and Networked Deterministic AI Systems

Future intelligent systems will increasingly operate as distributed AI infrastructures spanning edge devices, embedded platforms, cloud environments, and cyber-physical systems [113]. Autonomous vehicles, industrial automation systems, collaborative robotics, and smart infrastructures are expected to rely on distributed AI inference executed cooperatively across multiple heterogeneous computing nodes.

The architectural principles introduced in this thesis can be extended toward distributed deterministic AI acceleration frameworks in which accelerator nodes, tensor exchanges, and execution pipelines are globally synchronized through time-triggered communication mechanisms. In particular, future work may investigate:

- distributed WCET-aware inference scheduling,
- deterministic tensor communication over Time-Sensitive Networks (TSN) [114],
- synchronized multi-node accelerator orchestration,
- and end-to-end timing analysis for distributed AI pipelines.

Such research directions may contribute toward the realization of predictable large-scale edge-AI infrastructures capable of supporting real-time cooperative intelligence with formally analyzable temporal behavior.

Fault-Tolerant and Predictable Adaptive Architectures

Although the architectures proposed in this thesis intentionally rely on statically generated execution schedules to preserve analyzability and temporal determinism, future safety-critical AI systems will increasingly require advanced fault-tolerance capabilities to ensure reliable operation under hardware faults, transient errors, resource degradation, and unpredictable environmental conditions.

A particularly important future research direction therefore involves the development of fault-tolerant adaptive AI accelerators that can detect, predict, and mitigate runtime anomalies while preserving deterministic execution guarantees and WCET analyzability.

Such architectures could combine real-time monitoring, predictive fault analysis, and proactive scheduling mechanisms to identify abnormal system behavior before timing violations or critical failures occur. By tightly integrating fault-detection and adaptation mechanisms with statically analyzable execution schedules, future systems may enable deterministic runtime recovery and controlled system reconfiguration while preserving predictable inference execution and bounded timing behavior.

This approach may enable future AI accelerators to maintain both high computational performance and strict temporal guarantees even under degraded operating conditions. Potential future research directions include:

- deterministic fault-detection and isolation mechanisms,
- proactive fault-aware scheduling,
- predictable hardware redundancy and recovery strategies,
- runtime monitoring for timing and resource anomalies,

- deterministic workload migration across accelerator units,
- and safe runtime reconfiguration under fault conditions.

Such capabilities may become increasingly important for next-generation autonomous vehicles, industrial automation systems, aerospace platforms, medical devices, and distributed cyber-physical systems, where continuous operation, predictable recovery behavior, and strict timing guarantees are essential requirements. Consequently, integrating fault tolerance with deterministic AI acceleration represents a promising step toward future dependable and certifiable intelligent systems.

Final Perspective

Overall, this thesis demonstrates that temporal predictability, WCET analyzability, and high computational performance are not inherently conflicting objectives in AI acceleration. Instead, through systematic co-design of execution semantics, scheduling methodologies, memory-management mechanisms, and accelerator architectures, it is possible to construct AI acceleration platforms that simultaneously achieve high throughput and predictable execution behavior.

The TT-VTA, STT-VTA, and STA-CP architectures therefore represent more than isolated architectural optimizations; they establish a broader architectural paradigm for predictable AI computing. As AI technologies continue to become increasingly integrated into safety-critical and real-time domains, the concepts and methodologies introduced in this thesis are expected to contribute toward the realization of future trustworthy, and predictable AI execution infrastructures.

Appendix A

WCET Analysis and Evaluation of DRAMSim2

A.1 Overview

Accurate modeling of memory-access latency is essential for estimating the Worst-Case Execution Time (WCET) of AI accelerator architectures, particularly in safety-critical real-time systems where timing predictability is mandatory. Since off-chip DRAM accesses frequently dominate inference latency in deep-learning workloads, memory behavior must be analyzed at a fine-grained timing level to ensure deterministic execution and bounded response times.

In this thesis, *DRAMSim2* is employed as the primary memory simulation framework for modeling DRAM timing behavior and evaluating memory-access latency. Although *DRAMSim2* does not directly compute formal WCET bounds, it provides cycle-accurate timing traces that enable empirical derivation of worst-case memory latency under different scheduling and contention scenarios. These timing measurements are subsequently integrated into the proposed scheduler and simulation framework to estimate deterministic inference latency and WCET for TT-VTA, STT-VTA, and STA-CP.

A.2 DRAM Memory System Model

DRAMSim2 models the behavior of the memory subsystem at cycle level, including memory-controller scheduling, command arbitration, row-buffer management, bank conflicts, and DRAM timing constraints. Based on this model, the total latency of a memory request can be decomposed into three primary components:

$$L_{mem} = L_{queue} + L_{sched} + L_{dram} \quad (A.1)$$

where:

- L_{queue} represents the queuing delay incurred while waiting in the memory-controller request queue,
- L_{sched} denotes the latency introduced by request scheduling and arbitration policies within the memory controller,
- L_{dram} corresponds to the intrinsic DRAM-access latency associated with executing the memory transaction.

The DRAM-access latency itself depends on several timing constraints imposed by the DRAM protocol and memory-state transitions:

$$L_{dram} = t_{RCD} + t_{CAS} + t_{RP} + \Delta_{conflict} + \Delta_{refresh} \quad (A.2)$$

where:

- t_{RCD} is the Row-to-Column Delay,
- t_{CAS} is the Column Access Strobe latency,
- t_{RP} is the Row Precharge delay,
- $\Delta_{conflict}$ represents additional latency caused by row-buffer misses and bank conflicts,
- $\Delta_{refresh}$ denotes delays introduced by DRAM refresh operations.

This decomposition enables fine-grained analysis of memory latency sources and provides visibility into interference behavior affecting deterministic execution.

A.3 Worst-Case Memory Latency Estimation

The worst-case memory-access latency is defined as the maximum observed or estimated delay for a DRAM transaction under a given memory configuration and workload pattern:

$$L_{mem}^{WC} = \max (L_{queue} + L_{sched} + L_{dram}) \quad (A.3)$$

Within this work, the worst-case latency is derived empirically using DRAMSim2 simulation traces generated under stress-test conditions and high-contention memory-access scenarios. Two complementary approaches are employed:

- Observing the maximum latency across all recorded DRAMSim2 timing traces,
- Constructing adversarial memory-access patterns designed to maximize bank conflicts, row-buffer misses, and scheduling contention.

Because the proposed accelerator architectures eliminate uncontrolled runtime arbitration and statically schedule memory transactions, the variability of memory latency is substantially reduced compared to conventional dynamic systems. Consequently, the derived latency bounds provide stable and predictable timing estimates suitable for deterministic scheduling and WCET-oriented analysis.

A.4 WCET Estimation Methodology

The overall WCET of the accelerator execution can be approximated by combining the execution latency of compute-bound instructions with the worst-case latency contribution of memory-bound operations. Formally, the WCET can be expressed as:

$$WCET = WCET_{compute} + N_{mem} \cdot L_{mem}^{WC} \quad (A.4)$$

where:

- $WCET_{compute}$ represents the execution latency of compute-bound operations assuming local or cache-resident data accesses,

- N_{mem} denotes the number of off-chip DRAM accesses,
- L_{mem}^{WC} is the worst-case DRAM-access latency derived from DRAMSim2.

In the proposed framework, cycle counts for compute-bound instructions are obtained through cycle-accurate hardware simulation, while memory-related instruction latencies are derived using DRAMSim2. These timing measurements are then integrated into the scheduler to generate deterministic execution schedules and estimate the expected inference latency of the accelerator architectures.

For the STA-CP architecture, deterministic caching and static pre-fetching further reduce the number of off-chip memory accesses and minimize contention effects, thereby improving both average-case performance and worst-case timing predictability.

A.5 Interpretation of DRAMSim2 Timing Traces

DRAMSim2 generates detailed per-request timing traces that expose the issue cycle, completion cycle, and total latency of each memory transaction. A simplified example is shown below:

Request	Issue Cycle	Completion Cycle	Latency
R1	100	165	65
R2	101	250	149
R3	102	180	78

The worst-case observed latency is therefore:

$$L_{max} = \max(65, 149, 78) = 149 \text{ cycles} \quad (\text{A.5})$$

This value is subsequently used as an empirical upper bound for DRAM-access latency under the evaluated configuration. By repeating this process across different scheduling strategies, DRAM configurations, and workload patterns, the scheduler can derive stable timing bounds for deterministic execution analysis.

A.6 Integration with the Proposed Scheduling Framework

The DRAMSim2 timing model is tightly integrated into the proposed Scheduler with Built-in Simulation introduced throughout this thesis. During compile time, the scheduler forwards memory-related instructions to DRAMSim2 to obtain cycle-level latency estimates for DRAM transactions. Simultaneously, cycle-accurate hardware simulation is used to evaluate compute-bound operations and on-chip memory accesses.

The collected timing information is then combined to:

- Construct static execution schedules,
- Estimate deterministic inference latency,
- Detect potential memory-contention scenarios,
- Evaluate cache-placement and pre-fetching strategies,

- Derive WCET-oriented timing bounds for the proposed accelerator architectures.

This integrated workflow enables deterministic coordination of computation and memory communication, which is fundamental for achieving predictable execution behavior in TT-VTA, STT-VTA, and STA-CP.

A.7 Limitations

Although DRAMSim2 provides cycle-accurate memory timing simulation, several limitations remain:

- DRAMSim2 provides empirical timing bounds rather than mathematically formal WCET guarantees,
- The quality of WCET estimation depends on the completeness of the evaluated stress-test and contention scenarios,
- Real-world systems may introduce additional interference sources, such as DMA traffic, multicore contention, or external peripherals,
- Newer DRAM standards and advanced memory-controller features are modeled more comprehensively in newer simulators such as DRAMSim3.

Nevertheless, for the objectives of this thesis, DRAMSim2 provides an effective balance between timing fidelity, integration simplicity, transparency, and simulation efficiency. Its cycle-accurate timing model is sufficient for analyzing deterministic memory behavior and deriving stable latency bounds for the proposed time-triggered AI accelerator architectures.

Bibliography

- [1] Perez-Cerrolaza and et al. "Artificial Intelligence for Safety-Critical Systems in Industrial and Transportation Domains: A Survey". In: *ACM Comput. Surv.* 56.7 (Apr. 2024). ISSN: 0360-0300. DOI: [10.1145/3626314](https://doi.org/10.1145/3626314). URL: <https://doi.org/10.1145/3626314>.
- [2] Ana Pereira and Carsten Thomas. "Challenges of Machine Learning Applied to Safety-Critical Cyber-Physical Systems". In: *Machine Learning and Knowledge Extraction* 2.4 (2020), pp. 579–602. ISSN: 2504-4990. DOI: [10.3390/make2040031](https://www.mdpi.com/2504-4990/2/4/31). URL: <https://www.mdpi.com/2504-4990/2/4/31>.
- [3] A. M. Ezekiel, D. Onwuchekwa, and R. Obermaisser. "Optimization of the Versatile Tensor Accelerator (VTA) Load Module in a Time-Triggered Memory Access". In: *2023 26th Euromicro Conference on Digital System Design (DSD)*. 2023, pp. 146–152. DOI: [10.1109/DSD60849.2023.00030](https://doi.org/10.1109/DSD60849.2023.00030).
- [4] Giorgio Buttazzo. "Can We Trust AI-Powered Real-Time Embedded Systems?" In: *Third Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2022)*. Ed. by Marko Bertogna, Federico Terraneo, and Federico Reghenzani. Vol. 98. Open Access Series in Informatics (OASICS). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 1:1–1:14. ISBN: 978-3-95977-221-1. DOI: [10.4230/OASICS.NG-RES.2022.1](https://drops.dagstuhl.de/entities/document/10.4230/OASICS.NG-RES.2022.1). URL: <https://drops.dagstuhl.de/entities/document/10.4230/OASICS.NG-RES.2022.1>.
- [5] David Gunning and David Aha. "DARPA's Explainable Artificial Intelligence (XAI) Program". In: *AI Magazine* 40.2 (June 2019), pp. 44–58. DOI: [10.1609/aimag.v40i2.2850](https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/2850). URL: <https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/2850>.
- [6] Yanik Dabke, Dr Gadgil, and Viyan Dabke. "Techniques to Detect and Mitigate Adversarial Attacks through Augmentation of Training Data: A Systematic Review". In: Oct. 2023. DOI: [10.36227/techrxiv.24208722.v2](https://doi.org/10.36227/techrxiv.24208722.v2).
- [7] Seyyed Ahmad Javadi et al. "Monitoring AI Services for Misuse". In: *Proceedings of the 2021 AAAI/ACM Conference on AI, Ethics, and Society*. AIES '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 597–607. ISBN: 9781450384735. DOI: [10.1145/3461702.3462566](https://doi.org/10.1145/3461702.3462566). URL: <https://doi.org/10.1145/3461702.3462566>.
- [8] Tomás Grimm, Djones Lettnin, and Michael Hübner. "A Survey on Formal Verification Techniques for Safety-Critical Systems-on-Chip". In: *Electronics* 7.6 (2018). ISSN: 2079-9292. DOI: [10.3390/electronics7060081](https://www.mdpi.com/2079-9292/7/6/81). URL: <https://www.mdpi.com/2079-9292/7/6/81>.
- [9] ISO Pas 21448. "Road Vehicles-Safety of the Intended Functionality". In: *International Organization for Standardization* (2019).
- [10] Peter Schlicht. "AI in the Automotive Industry: How AI is Changing the Automotive World". In: May 2023, pp. 257–265. ISBN: 978-3-658-40231-0. DOI: [10.1007/978-3-658-40232-7_29](https://doi.org/10.1007/978-3-658-40232-7_29).

- [11] J. Duarte et al. "FPGA-Accelerated Machine Learning Inference as a Service for Particle Physics Computing". In: *Comput Softw Big Sci* 3, 13. 2019. DOI: <https://doi.org/10.1007/s41781-019-0027-2>.
- [12] Carlos Boneti et al. "A Dynamic Scheduler for Balancing HPC Applications". In: *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 2008, pp. 1–12. DOI: [10.1109/SC.2008.5217785](https://doi.org/10.1109/SC.2008.5217785).
- [13] Jon Perez-Cerrolaza et al. "GPU Devices for Safety-Critical Systems: A Survey". In: *ACM Comput. Surv.* 55.7 (Dec. 2022). ISSN: 0360-0300. DOI: [10.1145/3549526](https://doi.org/10.1145/3549526). URL: <https://doi.org/10.1145/3549526>.
- [14] Ignacio Sañudo Olmedo, Nicola Capodieci, and Roberto Cavicchioli. "A Perspective on Safety and Real-Time issues for GPU Accelerated ADAS". In: *IECON 2018-44th Annual Conference of the IEEE Industrial Electronics Society*. IEEE. 2018, pp. 4071–4077.
- [15] Paolo Rech. "Artificial Neural Networks for Space and Safety-Critical Applications: Reliability Issues and Potential Solutions". In: *IEEE Transactions on Nuclear Science* 71.4 (2024), pp. 377–404. DOI: [10.1109/TNS.2024.3349956](https://doi.org/10.1109/TNS.2024.3349956).
- [16] R. Hilbrich and L. Dieudonné. "Deploying Safety-Critical Applications on Complex Avionics Hardware Architectures". In: *Journal of Software Engineering and Applications* 6 (2013), pp. 229–235. DOI: [10.4236/jsea.2013.65028](https://doi.org/10.4236/jsea.2013.65028).
- [17] Fawad Riasat Raja, David Chen, and René Hexel. "A Flexible Communication Protocol With Guaranteed Determinism for Distributed, Safety-Critical Real-Time Systems". In: *IEEE Access* 10 (2022), pp. 48049–48070. DOI: [10.1109/ACCESS.2022.3170102](https://doi.org/10.1109/ACCESS.2022.3170102).
- [18] Sanjoy Baruah. "Predictability Issues in Mixed-Criticality Real-Time Systems". In: *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*. Ed. by Marten Lohstroh, Patricia Derler, and Marjan Sirjani. Cham: Springer International Publishing, 2018, pp. 77–87. ISBN: 978-3-319-95246-8. DOI: [10.1007/978-3-319-95246-8_5](https://doi.org/10.1007/978-3-319-95246-8_5). URL: https://doi.org/10.1007/978-3-319-95246-8_5.
- [19] Thierry Moreau and et al. "Leveraging the VTA-TVM Hardware-Software Stack for FPGA Acceleration of 8-Bit ResNet-18 Inference". In: *Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-Designing Pareto-Efficient Deep Learning*. ReQuEST '18. Williamsburg, VA, USA: Association for Computing Machinery, 2018. ISBN: 9781450359238. DOI: [10.1145/3229762.3229766](https://doi.org/10.1145/3229762.3229766). URL: <https://doi.org/10.1145/3229762.3229766>.
- [20] Thierry Moreau and et al. "VTA: An Open Hardware-Software Stack for Deep Learning". In: *ArXiv abs/1807.04188* (2018). URL: <https://api.semanticscholar.org/CorpusID:49672003>.
- [21] Udit Gupta et al. "RecPipe: Co-designing Models and Hardware to Jointly Optimize Recommendation Quality and Performance". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 870–884. ISBN: 9781450385572. DOI: [10.1145/3466752.3480127](https://doi.org/10.1145/3466752.3480127). URL: <https://doi.org/10.1145/3466752.3480127>.

- [22] Feng Liu et al. "CGPA: Coarse-Grained Pipelined Accelerators". In: *Proceedings of the 51st Annual Design Automation Conference*. DAC '14. San Francisco, CA, USA: Association for Computing Machinery, 2014, pp. 1–6. ISBN: 9781450327305. DOI: [10.1145/2593069.2593105](https://doi.org/10.1145/2593069.2593105). URL: <https://doi.org/10.1145/2593069.2593105>.
- [23] Norman P. Jouppi and et al. *In-Datacenter Performance Analysis of a Tensor Processing Unit*. 2017. arXiv: [1704.04760](https://arxiv.org/abs/1704.04760) [cs.AR]. URL: <https://arxiv.org/abs/1704.04760>.
- [24] Mengwei Xu and et al. "DeepCache: Principled Cache for Mobile Deep Vision". In: *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*. MobiCom '18. New Delhi, India: Association for Computing Machinery, 2018, pp. 129–144. ISBN: 9781450359030. DOI: [10.1145/3241539.3241563](https://doi.org/10.1145/3241539.3241563). URL: <https://doi.org/10.1145/3241539.3241563>.
- [25] Rachmad Vidya Wicaksana Putra and et al. "ROMANet: Fine-Grained Reuse-Driven Off-Chip Memory Access Management and Data Organization for Deep Neural Network Accelerators". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 29.4 (2021), pp. 702–715. DOI: [10.1109/TVLSI.2021.3060509](https://doi.org/10.1109/TVLSI.2021.3060509).
- [26] Gurjeet Singh, Vijay Kumar Banga, and Thaweesak Yingthawornsuk. "Artificial Intelligence and Industrial Robot". In: *2022 16th International Conference on Signal-Image Technology Internet-Based Systems (SITIS)*. 2022, pp. 622–625. DOI: [10.1109/SITIS57111.2022.00098](https://doi.org/10.1109/SITIS57111.2022.00098).
- [27] A. Bandhana and J. Vokřínek. "AI-Driven Manufacturing: Surveying for Industry 4.0 and Beyond". In: *Springer: Oper. Res. Forum* 6 6.145 (2025). URL: <https://doi.org/10.1007/s43069-025-00554-6>.
- [28] G. Demir, S. Moslem, and S. Duleba. "Artificial Intelligence in Aviation Safety: Systematic Review and Biometric Analysis". In: *Springer: Int J Comput Intell Syst* 17.279 (2024). URL: <https://doi.org/10.1007/s44196-024-00671-w>.
- [29] Erarda Vuka, Lourdes Ruiz Salvador, and Esmeralda Kadena. "AI in Healthcare: Applications, Challenges and Opportunities". In: *2024 IEEE 28th International Conference on Intelligent Engineering Systems (INES)*. 2024, pp. 000231–000234. DOI: [10.1109/INES63318.2024.10629126](https://doi.org/10.1109/INES63318.2024.10629126).
- [30] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning". In: *Nature* 512 (May 2015), pp. 436–444. DOI: <https://doi.org/10.1038/nature14539>.
- [31] Jeffrey Heaton. "Ian Goodfellow, Yoshua Bengio, and Aaron Courville: Deep learning: The MIT Press, 2016, 800 pp, ISBN: 0262035618". In: *Genetic Programming and Evolvable Machines* 19 (Oct. 2017). DOI: [10.1007/s10710-017-9314-z](https://doi.org/10.1007/s10710-017-9314-z).
- [32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Commun. ACM* 60.6 (May 2017), pp. 84–90. ISSN: 0001-0782. DOI: [10.1145/3065386](https://doi.org/10.1145/3065386). URL: <https://doi.org/10.1145/3065386>.
- [33] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: [10.1109/CVPR.2016.90](https://doi.org/10.1109/CVPR.2016.90).
- [34] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 779–788. DOI: [10.1109/CVPR.2016.91](https://doi.org/10.1109/CVPR.2016.91).

- [35] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-yuan Liao. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. Apr. 2020. DOI: [10.48550/arXiv.2004.10934](https://doi.org/10.48550/arXiv.2004.10934).
- [36] Yu-Hsin Chen and et al. "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks". In: Feb. 2016. DOI: [10.1109/ISSCC.2016.7418007](https://doi.org/10.1109/ISSCC.2016.7418007).
- [37] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. "MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects". In: *SIGPLAN Not.* 53.2 (Mar. 2018), pp. 461–475. ISSN: 0362-1340. DOI: [10.1145/3296957.3173176](https://doi.org/10.1145/3296957.3173176). URL: <https://doi.org/10.1145/3296957.3173176>.
- [38] Zidong Du and et al. "ShiDianNao: Shifting Vision Processing Closer to the Sensor". In: June 2015. DOI: [10.1145/2749469.2750389](https://doi.org/10.1145/2749469.2750389).
- [39] Hermann Kopetz and Wilfried Steiner. "Real-Time Scheduling". In: *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Cham: Springer International Publishing, 2022, pp. 247–267. ISBN: 978-3-031-11992-7. DOI: [10.1007/978-3-031-11992-7_10](https://doi.org/10.1007/978-3-031-11992-7_10). URL: https://doi.org/10.1007/978-3-031-11992-7_10.
- [40] R. Obermaisser. "Time-Triggered Communication (1st ed.)". In: CRC Press, Inc. Subs. of Times Mirror 2000 Corporate Blvd. NW Boca Raton, FL United States, 2011. ISBN: 978-1-4398-4661-2. URL: <https://doi.org/10.1201/9781315217512>.
- [41] Stephen A. Edwards and Edward A. Lee. "The case for the precision timed (PRET) machine". In: *Proceedings of the 44th Annual Design Automation Conference*. DAC '07. San Diego, California: Association for Computing Machinery, 2007, pp. 264–265. ISBN: 9781595936271. DOI: [10.1145/1278480.1278545](https://doi.org/10.1145/1278480.1278545). URL: <https://doi.org/10.1145/1278480.1278545>.
- [42] Reinhard Wilhelm et al. "The worst-case execution-time problem—overview of methods and survey of tools". In: *ACM Trans. Embed. Comput. Syst.* 7.3 (May 2008). ISSN: 1539-9087. DOI: [10.1145/1347375.1347389](https://doi.org/10.1145/1347375.1347389). URL: <https://doi.org/10.1145/1347375.1347389>.
- [43] Xu Lyu, Weigong Zhang, and Junwen Wang. "A Comprehensive Review on the Runtime of NVidia Deep Learning Accelerator". In: *Proceedings of the 5th International Conference on Algorithms, Computing and Systems*. ICACS '21. Xi'an, China: Association for Computing Machinery, 2022, pp. 88–93. ISBN: 9781450385084. DOI: [10.1145/3490700.3490715](https://doi.org/10.1145/3490700.3490715). URL: <https://doi.org/10.1145/3490700.3490715>.
- [44] Rodrigo Huerta et al. "Dissecting and Modeling the Architecture of Modern GPU Cores". In: *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*. MICRO '25. Association for Computing Machinery, 2025, pp. 369–384. ISBN: 9798400715730. DOI: [10.1145/3725843.3756041](https://doi.org/10.1145/3725843.3756041). URL: <https://doi.org/10.1145/3725843.3756041>.
- [45] Marco Paolieri et al. "An Analyzable Memory Controller for Hard Real-Time CMPs". In: *IEEE Embedded Systems Letters* 1.4 (2009), pp. 86–90. DOI: [10.1109/LES.2010.2041634](https://doi.org/10.1109/LES.2010.2041634).

- [46] Roberto Cavicchioli, Nicola Capodieci, and Marko Bertogna. "Memory Interference Characterization between CPU Cores and Integrated GPUs in Mixed-Criticality Platforms". In: *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2017, pp. 1–10. DOI: [10.1109/ETFA.2017.8247615](https://doi.org/10.1109/ETFA.2017.8247615).
- [47] Francesco Restuccia et al. "Is Your Bus Arbiter Really Fair? Restoring Fairness in AXI Interconnects for FPGA SoCs". In: *ACM Transactions on Embedded Computing Systems* 18 (Oct. 2019), pp. 1–22. DOI: [10.1145/3358183](https://doi.org/10.1145/3358183).
- [48] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 367–379. DOI: [10.1109/ISCA.2016.40](https://doi.org/10.1109/ISCA.2016.40).
- [49] ISO. *ISO-26262: Road vehicles – Functional safety*. Tech. rep. ISO 26262:2018. Geneva, CH: International Organization for Standardization, 2018.
- [50] RTCA SC-205. *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*. Tech. rep. DO-178C. Washington, D.C.: RTCA, Inc., Dec. 2011.
- [51] International Electrotechnical Commission. *IEC-61508:Functional safety of electrical/electronic/programmable electronic safety-related systems*. International Electrotechnical Commission. Geneva, Switzerland, 2010. URL: <https://webstore.iec.ch/en/publication/5515>.
- [52] Ciro Donnarumma et al. "EN-50128 Certification-Oriented Design of a Safety-Critical Hard Real-Time Kernel". In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2019, pp. 314–317. DOI: [10.1109/ISSREW.2019.00090](https://doi.org/10.1109/ISSREW.2019.00090).
- [53] Konstantinos Kontodimas et al. "Analysis and Evaluation of I/O Hypervisor Scheduling". In: *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. 2015, pp. 45–54. DOI: [10.1109/UCC.2015.19](https://doi.org/10.1109/UCC.2015.19).
- [54] Esmail Asyabi, Mohsen Sharifi, and Azer Bestavros. "ppXen: A hypervisor CPU scheduler for mitigating performance variability in virtualized clouds". In: *Future Generation Computer Systems* 83 (2018), pp. 75–84. ISSN: 0167-739X. DOI: <https://doi.org/10.1016/j.future.2018.01.015>. URL: <https://www.sciencedirect.com/science/article/pii/S0167739X17310555>.
- [55] Hong Seong Park. "Real-Time Scheduling Method for Middleware of Industrial Automation Devices". In: *Scheduling Problems*. Ed. by Rodrigo da Rosa Righi. Rijeka: IntechOpen, 2019. Chap. 2. DOI: [10.5772/intechopen.86769](https://doi.org/10.5772/intechopen.86769). URL: <https://doi.org/10.5772/intechopen.86769>.
- [56] Georg Seifert. "Static Analysis Methodologies for WCET Calculating with Asynchronous IO". In: *Software Engineering 2022 Workshops*. Bonn: Gesellschaft für Informatik e.V., 2022, pp. 115–127. DOI: [10.18420/se2022-ws-14](https://doi.org/10.18420/se2022-ws-14).
- [57] Zehong Zheng, Hongbiao Liu, and Bo Liu. "Fat Block: Narrowing the Boundary of pWCET Analysis on GPU". In: *Proceedings of the 2026 3rd International Conference on Autonomous Driving and Intelligent Sensing Technology*. ADIST '26. Association for Computing Machinery, 2026, pp. 110–118. ISBN: 9798400722165. DOI: [10.1145/3807211.3807227](https://doi.org/10.1145/3807211.3807227). URL: <https://doi.org/10.1145/3807211.3807227>.

- [58] Hyunbin Park and Shiho Kim. "Chapter Three - Hardware Accelerator Systems for Artificial Intelligence and Machine Learning". In: *Hardware Accelerator Systems for Artificial Intelligence and Machine Learning*. Ed. by Shiho Kim and Ganesh Chandra Deka. Vol. 122. Advances in Computers. Elsevier, 2021, pp. 51–95. DOI: <https://doi.org/10.1016/bs.adcom.2020.11.005>. URL: <https://www.sciencedirect.com/science/article/pii/S0065245820300929>.
- [59] Michaela Blott et al. "FINN- R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks". In: *ACM Transactions on Reconfigurable Technology and Systems* 11 (Dec. 2018), pp. 1–23. DOI: [10.1145/3242897](https://doi.org/10.1145/3242897).
- [60] Yu-Hsin Chen et al. "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (2019), pp. 292–308. DOI: [10.1109/JETCAS.2019.2910232](https://doi.org/10.1109/JETCAS.2019.2910232).
- [61] Thierry Moreau et al. "A Hardware–Software Blueprint for Flexible Deep Learning Specialization". In: *IEEE Micro* 39.5 (2019), pp. 8–16. DOI: [10.1109/MM.2019.2928962](https://doi.org/10.1109/MM.2019.2928962).
- [62] Martín Abadi et al. "TensorFlow: a system for large-scale machine learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 265–283. ISBN: 9781931971331.
- [63] Adam Paszke et al. "PyTorch: an imperative style, high-performance deep learning library". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.
- [64] Antonio Gulli and Sujit Pal. *Deep Learning with Keras*. Packt Publishing, 2017. ISBN: 1787128423.
- [65] Michael Garland. "Parallel computing with CUDA". In: *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 2010, pp. 1–1. DOI: [10.1109/IPDPS.2010.5470378](https://doi.org/10.1109/IPDPS.2010.5470378).
- [66] Marc Jorda, Pedro Valero-Lara, and Antonio Peña. "Performance Evaluation of cuDNN Convolution Algorithms on NVIDIA Volta GPUs". In: *IEEE Access* PP (May 2019), pp. 1–1. DOI: [10.1109/ACCESS.2019.2918851](https://doi.org/10.1109/ACCESS.2019.2918851).
- [67] Tianqi Chen and et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 579–594. ISBN: 9781931971478.
- [68] Aniebiet Micheal Ezekiel, Yosab Bebawy, and Roman Obermaisser. "Enhancing Neural Network Predictability through Simulation-Based Analysis of Memory Accesses for Time-Triggered Tensor Acceleration". In: *2024 10th International Conference on Control, Decision and Information Technologies (CoDIT)*. 2024, pp. 2839–2844. DOI: [10.1109/CoDIT62066.2024.10708250](https://doi.org/10.1109/CoDIT62066.2024.10708250).
- [69] Norman P. Jouppi et al. "Ten lessons from three generations shaped Google's TPUv4i". In: *Proceedings of the 48th Annual International Symposium on Computer Architecture*. ISCA '21. Virtual Event, Spain: IEEE Press, 2021, pp. 1–14. ISBN: 9781450390866. DOI: [10.1109/ISCA52012.2021.00010](https://doi.org/10.1109/ISCA52012.2021.00010). URL: <https://doi.org/10.1109/ISCA52012.2021.00010>.

- [70] Stefano Markidis et al. "NVIDIA Tensor Core Programmability, Performance Precision". In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2018, pp. 522–531. DOI: [10.1109/IPDPSW.2018.00091](https://doi.org/10.1109/IPDPSW.2018.00091).
- [71] Adrian M. Caulfield and et al. "A Cloud-Scale Acceleration Architecture". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016, pp. 1–13. DOI: [10.1109/MICRO.2016.7783710](https://doi.org/10.1109/MICRO.2016.7783710).
- [72] Alessandro Veronesi, Milos Krstic, and Davide Bertozzi. "Cross-Layer Hardware/Software Assessment of the Open-Source NVDLA Configurable Deep Learning Accelerator". In: *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)*. 2020, pp. 58–63. DOI: [10.1109/VLSI-SOC46417.2020.9344109](https://doi.org/10.1109/VLSI-SOC46417.2020.9344109).
- [73] Yaman Umuroglu et al. "High-Throughput DNN Inference with LogicNets". In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2020, pp. 238–238. DOI: [10.1109/FCCM48280.2020.00071](https://doi.org/10.1109/FCCM48280.2020.00071).
- [74] Jeahack Lee et al. "Implementing Binarized Neural Network Processor on FPGA-Based Platform". In: *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. 2022, pp. 469–471. DOI: [10.1109/AICAS54282.2022.9869997](https://doi.org/10.1109/AICAS54282.2022.9869997).
- [75] Farah Fahim. "hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices". In: Fermi National Accelerator Laboratory (FNAL), Batavia, IL (United States). Mar. 2021. URL: <https://www.osti.gov/biblio/1824149>.
- [76] Hardik Sharma et al. "From high-level deep neural models to FPGAs". In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-49. Taipei, Taiwan: IEEE Press, 2016.
- [77] Xinlei Liu. "Model Optimization Techniques for Embedded Artificial Intelligence". In: *2021 2nd International Conference on Computing and Data Science (CDS)*. 2021, pp. 1–6. DOI: [10.1109/CDS52072.2021.00008](https://doi.org/10.1109/CDS52072.2021.00008).
- [78] Surat Teerapittayanon, Bradley McDanel, and H.T. Kung. "BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks". In: Dec. 2016, pp. 2464–2469. DOI: [10.1109/ICPR.2016.7900006](https://doi.org/10.1109/ICPR.2016.7900006).
- [79] Sorin Grigorescu et al. "Cloud2Edge Elastic AI Framework for Prototyping and Deployment of AI Inference Engines in Autonomous Vehicles". In: *Sensors* 20.19 (Sept. 2020), p. 5450. ISSN: 1424-8220. DOI: [10.3390/s20195450](https://doi.org/10.3390/s20195450). URL: <http://dx.doi.org/10.3390/s20195450>.
- [80] Roberto G. Pacheco and Rodrigo S. Couto. "Inference Time Optimization Using BranchyNet Partitioning". In: *2020 IEEE Symposium on Computers and Communications (ISCC)*. 2020, pp. 1–6. DOI: [10.1109/ISCC50000.2020.9219647](https://doi.org/10.1109/ISCC50000.2020.9219647).
- [81] Hao Li, Joseph K. Ng, and Tarek Abdelzaher. "Enabling Real-Time AI Inference on Mobile Devices via GPU-CPU Collaborative Execution". In: *2022 IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2022, pp. 195–204. DOI: [10.1109/RTCSA55878.2022.00027](https://doi.org/10.1109/RTCSA55878.2022.00027).

- [82] Haozhen Jia, Xin Chen, and Daxing Dong. "FPGA-Based Implementation and Quantization of Convolutional Neural Networks". In: *Proceedings of the 2025 3rd International Conference on Communication Networks and Machine Learning*. CNML '25. Association for Computing Machinery, 2025, pp. 389–393. ISBN: 9798400713231. DOI: [10.1145/3728199.3728263](https://doi.org/10.1145/3728199.3728263). URL: <https://doi.org/10.1145/3728199.3728263>.
- [83] Clement Farabet et al. "CNP: An FPGA-based processor for Convolutional Networks". In: *2009 International Conference on Field Programmable Logic and Applications*. 2009, pp. 32–37. DOI: [10.1109/FPL.2009.5272559](https://doi.org/10.1109/FPL.2009.5272559).
- [84] Mengxiao Liu et al. "High Performance Memory Management for a Multi-core Architecture". In: *2009 Ninth IEEE International Conference on Computer and Information Technology*. Vol. 1. 2009, pp. 63–68. DOI: [10.1109/CIT.2009.120](https://doi.org/10.1109/CIT.2009.120).
- [85] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128119055.
- [86] Doug Joseph and Dirk Grunwald. "Prefetching using Markov Predictors". In: *Proceedings of the 24th Annual International Symposium on Computer Architecture*. ISCA '97. Denver, Colorado, USA: Association for Computing Machinery, 1997, pp. 252–263. ISBN: 0897919017. DOI: [10.1145/264107.264207](https://doi.org/10.1145/264107.264207). URL: <https://doi.org/10.1145/264107.264207>.
- [87] Steven P. Vanderwiel and David J. Lilja. "Data prefetch mechanisms". In: *ACM Comput. Surv.* 32.2 (June 2000), pp. 174–199. ISSN: 0360-0300. DOI: [10.1145/358923.358939](https://doi.org/10.1145/358923.358939). URL: <https://doi.org/10.1145/358923.358939>.
- [88] Dongjoo Shin et al. "DNPU: An Energy-Efficient Deep-Learning Processor with Heterogeneous Multi-Core Architecture". In: *IEEE Micro* 38 (Sept. 2018), pp. 85–93. DOI: [10.1109/MM.2018.053631145](https://doi.org/10.1109/MM.2018.053631145).
- [89] Mingyu Gao et al. "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory". In: *SIGARCH Comput. Archit. News* 45.1 (Apr. 2017), pp. 751–764. ISSN: 0163-5964. DOI: [10.1145/3093337.3037702](https://doi.org/10.1145/3093337.3037702). URL: <https://doi.org/10.1145/3093337.3037702>.
- [90] Erfan Azarkhish et al. "Neurostream: Scalable and Energy Efficient Deep Learning with Smart Memory Cubes". In: *IEEE Transactions on Parallel and Distributed Systems* 29.2 (2018), pp. 420–434. DOI: [10.1109/TPDS.2017.2752706](https://doi.org/10.1109/TPDS.2017.2752706).
- [91] "The Time-Triggered Protocols". In: *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Boston, MA: Springer US, 1997, pp. 171–191. ISBN: 978-0-306-47055-4. DOI: [10.1007/0-306-47055-1_8](https://doi.org/10.1007/0-306-47055-1_8). URL: https://doi.org/10.1007/0-306-47055-1_8.
- [92] H. Kopetz et al. "The time-triggered Ethernet (TTE) design". In: *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*. 2005, pp. 22–33. DOI: [10.1109/ISORC.2005.56](https://doi.org/10.1109/ISORC.2005.56).
- [93] Roman Obermaisser. *Event-Triggered and Time-Triggered Control Paradigms*. Vol. 22. Jan. 2005. ISBN: 978-0-387-23043-6. DOI: [10.1007/978-0-387-23044-3](https://doi.org/10.1007/978-0-387-23044-3).
- [94] Roman Obermaisser et al. "Adaptive Time-Triggered Multi-Core Architecture". In: *Designs* 3.1 (2019). ISSN: 2411-9660. DOI: [10.3390/designs3010007](https://doi.org/10.3390/designs3010007). URL: <https://www.mdpi.com/2411-9660/3/1/7>.

- [95] K. Goossens, J. Dielissen, and A. Radulescu. "AETHEReal network on chip: concepts, architectures, and implementations". In: *IEEE Design Test of Computers* 22.5 (2005), pp. 414–421. DOI: [10.1109/MDT.2005.99](https://doi.org/10.1109/MDT.2005.99).
- [96] Andreas Hansson et al. "CoMPSoC: A template for composable and predictable multi-processor system on chips". In: *ACM Trans. Des. Autom. Electron. Syst.* 14.1 (Jan. 2009). ISSN: 1084-4309. DOI: [10.1145/1455229.1455231](https://doi.org/10.1145/1455229.1455231). URL: <https://doi.org/10.1145/1455229.1455231>.
- [97] Benny Akesson and Kees Goossens. "Architectures and modeling of predictable memory controllers for improved system integration". In: *2011 Design, Automation Test in Europe*. 2011, pp. 1–6. DOI: [10.1109/DATE.2011.5763145](https://doi.org/10.1109/DATE.2011.5763145).
- [98] Paul Rosenfeld and et al. "DRAMSim2: A Cycle Accurate Memory System Simulator". In: *IEEE Computer Architecture Letters* 10.1 (2011), pp. 16–19. DOI: [10.1109/L-CA.2011.4](https://doi.org/10.1109/L-CA.2011.4).
- [99] Shang Li et al. "DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator". In: *IEEE Computer Architecture Letters* 19.02 (July 2020), pp. 106–109. ISSN: 1556-6064. DOI: [10.1109/LCA.2020.2973991](https://doi.org/10.1109/LCA.2020.2973991). URL: <https://doi.org/10.1109/LCA.2020.2973991>.
- [100] Lukas Steiner et al. "DRAMSys4.0: A Fast and Cycle-Accurate SystemC/TLM-Based DRAM Simulator". In: *Embedded Computer Systems: Architectures, Modeling, and Simulation: 20th International Conference, SAMOS 2020, Samos, Greece, July 5–9, 2020, Proceedings*. Samos, Greece: Springer-Verlag, 2020, pp. 110–126. ISBN: 978-3-030-60938-2. DOI: [10.1007/978-3-030-60939-9_8](https://doi.org/10.1007/978-3-030-60939-9_8). URL: https://doi.org/10.1007/978-3-030-60939-9_8.
- [101] Sudipto Chakraborty. "Vivado Design Tools". In: *Designing with Xilinx® FPGAs: Using Vivado*. Ed. by Sanjay Churiwala. Cham: Springer International Publishing, 2017, pp. 17–21. ISBN: 978-3-319-42438-5. DOI: [10.1007/978-3-319-42438-5_2](https://doi.org/10.1007/978-3-319-42438-5_2). URL: https://doi.org/10.1007/978-3-319-42438-5_2.
- [102] Yosab Bebawy et al. "Time-Triggered Inference on FPGAs". In: *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2024, pp. 1–7. DOI: [10.1109/ETFA61755.2024.10710675](https://doi.org/10.1109/ETFA61755.2024.10710675).
- [103] Yosab Bebawy, Aniebiet Micheal Ezekiel, and Roman Obermaisser. "Superscalar Time-Triggered Versatile-Tensor Accelerator". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 44.7 (2025), pp. 2503–2515. DOI: [10.1109/TCAD.2025.3528355](https://doi.org/10.1109/TCAD.2025.3528355).
- [104] Yosab Bebawy, Abdulmajeed Alhumaidi, and Roman Obermaisser. "Deterministic Caching and Pre-fetching for High-Performance Superscalar Time-Triggered Tensor Accelerators". In: *ACM Transactions on Embedded Computing Systems (Under Review)* (2026).
- [105] Saikat Bandopadhyay. "Designing with Xilinx® FPGAs: Using Vivado". In: ed. by Sanjay Churiwala. Cham: Springer International Publishing, 2017, pp. 127–140. ISBN: 978-3-319-42438-5. DOI: [10.1007/978-3-319-42438-5_11](https://doi.org/10.1007/978-3-319-42438-5_11). URL: https://doi.org/10.1007/978-3-319-42438-5_11.
- [106] Colin J. Theaker and Graham R. Brookes. "Memory Management — Paging Systems". In: *A Practical Course on Operating Systems*. London: Macmillan Education UK, 1983, pp. 92–108. ISBN: 978-1-349-17138-5. DOI: [10.1007/978-1-349-17138-5_10](https://doi.org/10.1007/978-1-349-17138-5_10). URL: https://doi.org/10.1007/978-1-349-17138-5_10.

- [107] Jian Fang and et al. "In-Memory Database Acceleration on FPGAs: a survey". In: *The VLDB Journal* 29.1 (Oct. 2019), pp. 33–59. ISSN: 1066-8888. DOI: [10.1007/s00778-019-00581-w](https://doi.org/10.1007/s00778-019-00581-w). URL: <https://doi.org/10.1007/s00778-019-00581-w>.
- [108] Qing Yun. "Vision Transformers (ViTs) for Feature Extraction and Classification of AI-Generated Visual Designs". In: *IEEE Access* 13 (2025), pp. 69459–69477. DOI: [10.1109/ACCESS.2025.3562130](https://doi.org/10.1109/ACCESS.2025.3562130).
- [109] Zina Chkirbene et al. "Large Language Models (LLM) in Industry: A Survey of Applications, Challenges, and Trends". In: *2024 IEEE 21st International Conference on Smart Communities: Improving Quality of Life using AI, Robotics and IoT (HONET)*. 2024, pp. 229–234. DOI: [10.1109/HONET63146.2024.10822885](https://doi.org/10.1109/HONET63146.2024.10822885).
- [110] Zhiyuan Ma et al. "Efficient Diffusion Models: A Comprehensive Survey From Principles to Practices". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 47.9 (2025), pp. 7506–7525. DOI: [10.1109/TPAMI.2025.3569700](https://doi.org/10.1109/TPAMI.2025.3569700).
- [111] Zonghan Wu et al. "A Comprehensive Survey on Graph Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1 (2021), pp. 4–24. DOI: [10.1109/TNNLS.2020.2978386](https://doi.org/10.1109/TNNLS.2020.2978386).
- [112] Han Zhang et al. "Multi-Modal Data-Enhanced Foundation Models for Prediction and Control in Wireless Networks: A Survey". In: *IEEE Communications Surveys Tutorials* 28 (2026), pp. 4359–4393. DOI: [10.1109/COMST.2025.3648785](https://doi.org/10.1109/COMST.2025.3648785).
- [113] Tobias Meuser et al. "Revisiting Edge AI: Opportunities and Challenges". In: *IEEE Internet Computing* 28.4 (2024), pp. 49–59. DOI: [10.1109/MIC.2024.3383758](https://doi.org/10.1109/MIC.2024.3383758).
- [114] Yunfei Shen. "Time-Sensitive Networking: Standards and Application Prospects". In: *2023 International Conference on Information Processing and Network Provisioning (ICIPNP)*. 2023, pp. 479–484. DOI: [10.1109/ICIPNP62754.2023.00105](https://doi.org/10.1109/ICIPNP62754.2023.00105).