2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 14-15, 2025

Fast, Flexible, Timing-accurate and Open-Source Performance Modeling Method for Compute Accelerators

Vishal Chovatiya, Infineon Technologies, Bangalore, India (*vishal.chovatiya@infineon.com*)

Andrew Stevens, Infineon Technologies, Munich, Germany (*andrew.stevens@infineon.com*)

Snehith Shenoy, Infineon Technologies, Bangalore, India (*snehith.shenoy@infineon.com*)

***Abstract*— The proliferation of Artificial Intelligence and Machine Learning (AI/ML) applications has fueled demand for specialised compute accelerators, creating complex design challenges across hardware and software domains. Traditional Register-Transfer Level (RTL) simulation approaches face significant limitations due to their inherent complexity, slow simulation speeds, and extensive development resource requirements. This paper presents a fast, flexible, and timing-accurate performance modeling method for compute accelerators using CorePerfDSL, a domain-specific language developed initially for CPU pipeline modeling. We demonstrate the methodology's effectiveness by implementing and validating a comprehensive mini-NPU (Neural Processing Unit) performance model. The approach separates functional and timing concerns, enabling rapid architectural exploration while maintaining predictive accuracy. Experimental validation using MLPerf Tiny inference benchmarks shows the performance model can predict mini-NPU accelerator performance with mean absolute error less than 10% compared to RTL simulation for 84% of evaluated layers. The method achieves significantly higher simulation speeds than RTL while providing timing-accurate results suitable for design space exploration and early software validation of compute accelerators.**

Keywords*— *Performance Modeling; Compute Accelerators; Virtual Prototyping; Design Space Exploration; CorePerfDSL; Functional Model; Timing Model; Neural Processing Unit.

I. Introduction

ML accelerators and NPUs significantly outperform traditional CPUs for AI/ML workloads. However, their design presents significant design-space exploration challenges and requires the timely delivery of complex supporting software stacks. Traditional RTL simulation of HDL Designs is unable to meet these developmental challenges. Modification and ab-initio implementation are time-consuming, and the simulation speed is too low with costly / low-availability RTL emulation systems.

Virtual prototyping addresses these challenges. Functional models allow early software validation without sacrificing simulation speed, while performance models provide critical insights for architectural exploration, helping developers make informed design decisions.

This work demonstrates the applicability and extension of CorePerfDSL for compute accelerator performance modeling. The principal contributions include: (1) manual development of a timing-accurate mini-NPU performance model incorporating bus contention effects, (2) establishment of a virtual instruction method for dataflow-to-pipeline mapping, (3) development of a comprehensible code generation strategy for performance models, (4) creation of a custom communication interface for non-standard trace points, and (5) experimental validation against RTL using MLPerf Tiny inference benchmarks.

II. Related Work

A. Cycle-Accurate and Functional Modeling

Early CPU-focused frameworks like SMARTS[6] and ESECS[7] employed statistical sampling[8] to accelerate simulations, though these required pre-existing cycle-accurate models, limiting novel design exploration. The gem5 simulator[5] advanced this paradigm by integrating detailed functional models of processor microarchitectures, but this level of detail comes at additional development cost and compromises design space exploration flexibility.

B. *Instruction Set Simulation(ISS) Extensions*

ISS-based approaches[9][10][11] decouple timing models from functional behaviour at the ISA level, enabling faster software development. However, these frameworks rely on ad-hoc programming of timing details, making design-space exploration difficult.

C. *High-Level Abstraction for Accelerators*

Aladdin[12] pioneered pre-RTL power-performance simulation, enabling rapid design space exploration through parametric modeling. While effective for static configurations, it struggles to capture dynamic behaviours in modern AI-accelerators. SystemC[14] / TLM expanded transaction-level modeling, but achieving abstraction without losing timing accuracy remains challenging. Proprietary HLS-driven solutions (Siemens-EDA[15]) improve productivity but suffer from architectural rigidity for design-space exploration.

D. *Domain-Specific Language(DSL) Advancements*

CorePerfDSL[1] marked a paradigm shift by introducing a declarative DSL for CPU pipeline timing. Its syntax models data dependencies, resource contention, and latency variations, generating timing-accurate models without cycle-level simulation. Extensions[2] have demonstrated scalability to advanced features like out-of-order execution, sub-pipelining, etc.

E. *Comparative Analysis and Research Gaps*

While discrete-event frameworks like Sparta[13] provide robust microarchitectural simulation utilities, they require significant boilerplate code and lack clear functional/timing separation.

Current AI accelerator virtual prototyping thus suffers from three critical gaps:
1. Abstraction Limitations: Timing-accurate predictions without cycle-level simulations.
2. Portability: Tight coupling between timing models and specific ISS environments hinders heterogeneous system integration.
3. Design Space Exploration: Existing frameworks often require extensive parameter tuning and/or detailed cycle-accurate modeling for new designs, limiting rapid exploration of architectural alternatives.

CorePerfDSL[1] uniquely addresses these through declarative modeling and microarchitectural separation, providing a foundation that can extend to accelerator-specific scenarios. By adapting Kunzelmanns's parallel functional unit and sub-pipelining mechanisms[2], we establish a framework for dataflow accelerator Modeling that preserves simulation speed while capturing dynamic scheduling effects.

III. METHODOLOGY

A. *mini-NPU Architecture*

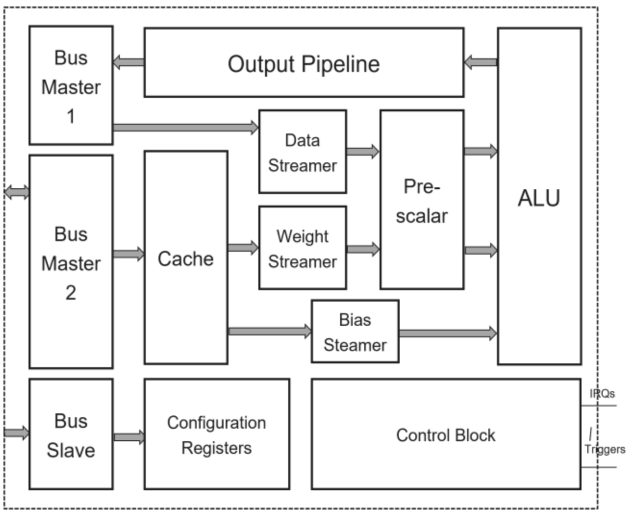

Figure 1: mini-NPU Architecture

The mini-NPU is a specialised Neural Processing Unit designed for inference workloads on resource-constrained devices with low power requirements. The accelerator is optimised to execute common neural network operations efficiently while maintaining a small footprint and power envelope suitable for edge deployments.

The mini-NPU architecture consists of several key components organised in a dataflow-oriented design. The Arithmetic Logic Unit (ALU) is at the core, which performs primary computational operations. The architecture features a Bus Master and a Bus Slave for communication with the host system and employs a series of streamers to move data through the processing pipeline efficiently.

The mini-NPU implements a data processing pipeline optimised for neural network inference with several stages: Input stage features three primary data streamers responsible for fetching input, weight and bias (or scale) data from memory; Pre-scalar stage prepares data for processing in the ALU; ALU stage processed data then passes through an output pipeline consisting of stages: pre-activation scaling, activation function unit, post-activation scaling, offset addition, clipping, output packing, and memory write-back.

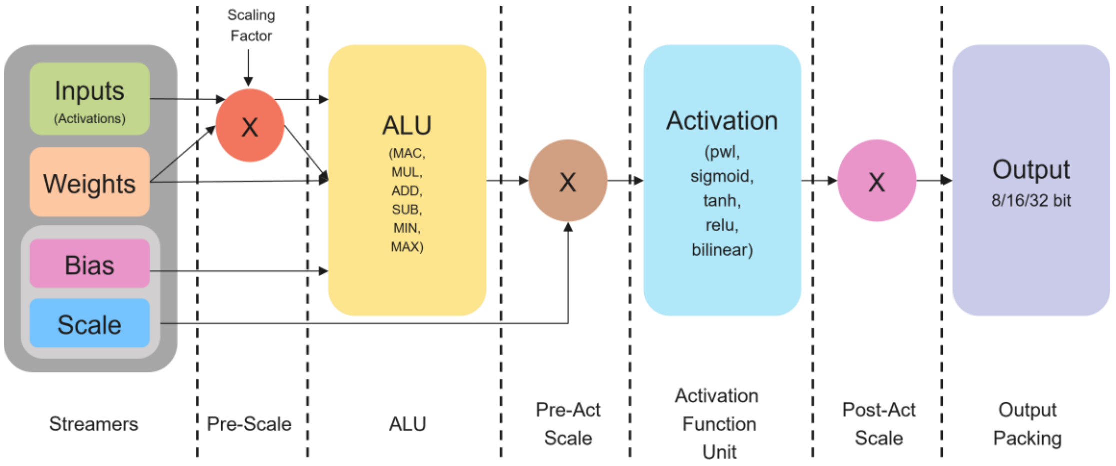

Figure 2: mini-NPU Data Processing Pipeline

A notable feature is the mini-NPU's sparsity-aware hardware design, allowing efficient processing of sparse neural networks by skipping unnecessary computations when encountering zero values in weights or activations. The Control Block manages execution flow, handling interrupts and triggers from the host system, while Configuration Registers allow the host to set up the accelerator for specific neural network topologies.

B. *Performance Modeling Method*

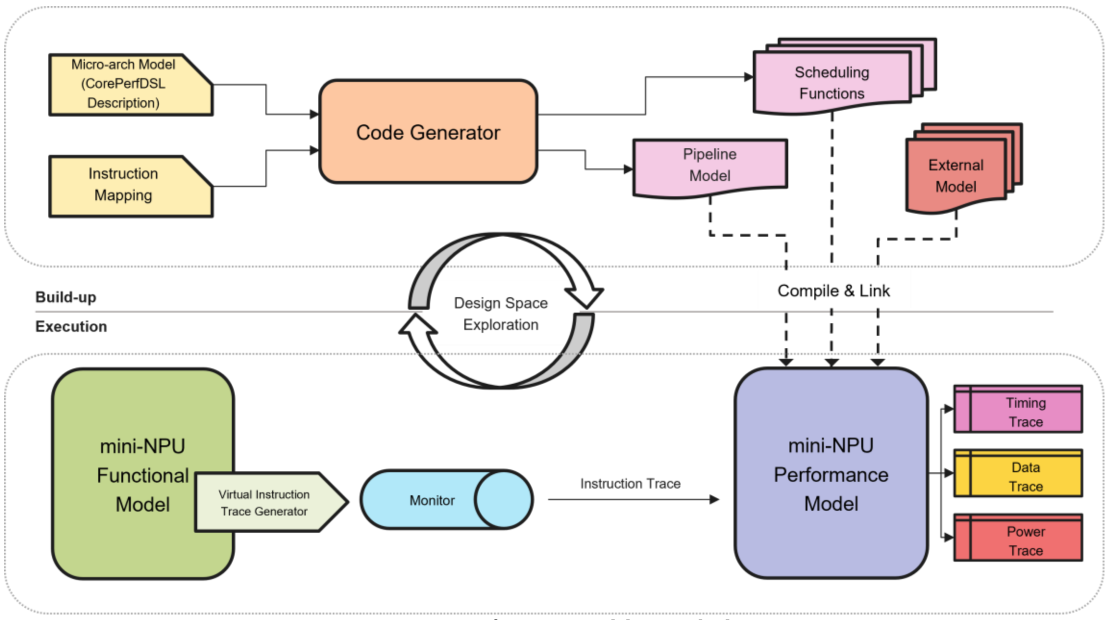

Figure 3: Performance Modeling Method

The proposed performance modeling method provides a comprehensive framework for accurately predicting and analysing mini-NPU accelerator behaviour. This method follows a two-phase approach: Build-up and Execution, with design space exploration as an iterative feedback mechanism.

The Build-up phase focuses on creating the necessary components for performance simulation. This includes: (1) Microarchitectural model using CorePerfDSL to formally describe the mini-NPU architecture, capturing pipeline, stage arrangements, hardware components, interconnections, and operational characteristics, (2) Instruction Mapping establishing detailed mapping of neural network operations to the mini-NPU virtual instruction set, (3) Code Generator processing both micro-architectural description and instruction mapping to produce scheduling functions and pipeline models, and (4) External Models capturing dynamic microarchitectural aspects not directly represented in CorePerfDSL, e.g. timing models for external memories and buses.

The Execution phase implements actual performance simulation and analysis. The mini-NPU Functional Model includes a Virtual Instruction Trace Generator, creating instruction sequences that would be executed by actual hardware when running target neural network workloads. A Monitor hooks to the functional model, capturing and supplying instruction traces to the performance model. The mini-NPU Performance Model accepts instruction traces as input and generates comprehensive timing traces.

A key strength is the method's support for iterative design space exploration. Performance results from the Execution phase inform potential architectural modifications, parameter adjustments, or instruction set refinements. These refinements are incorporated back into Build-up phase components, allowing designers to rapidly evaluate trade-offs and optimise the mini-NPU architecture for specific performance targets, power constraints, or application requirements.

CorePerfDSL is an open-source project that provides a domain-specific language for CPU core performance modeling. A complete workspace setup, including CorePerfDSL descriptions for different CPU architectures, a code generator, and CPU benchmarks to run on the generated performance models, along with the ETISS[4] simulator, is available on GitHub[19].

C. *mini-NPU Performance Model Implementation*

The mini-NPU performance model implementation breaks down into two main parts: virtual instruction creation in the functional model and manual CorePerfDSL performance model implementation. Developing an effective performance model requires creating a well-defined virtual instruction set that accurately represents the accelerator's operations while maintaining appropriate abstraction levels.

Virtual instruction development involved breaking down neural network operations into fundamental tasks, aligning with the mini-NPU's data processing pipeline. Based on functional decomposition and the pipeline structure consisting of Fetch, ALU, and Output stages, virtual instructions were categorized into logical groups: Configuration (setup operations), Data Transfer (load/store operations), Computation (core arithmetic operations), Activation Functions (non-linear operations), and Scaling & Clipping (data conditioning operations).

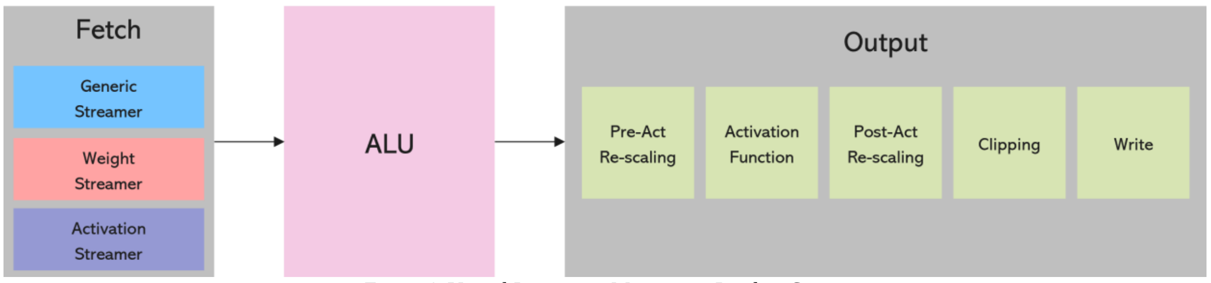

Figure 4: Virtual Instruction Mapping to Pipeline Stages

The virtual instructions were mapped to the mini-NPU's three primary pipeline stages. The Fetch Stage involves three parallel streamer units preparing data for computation, with instructions including configuration setups and data loading operations. The ALU Stage performs core computational operations (e.g. MAC, element-wise addition, subtraction, etc.). The Output Stage encompasses post-processing computed data before memory write-back, including sub-stages for pre-activation rescaling, activation functions, post-activation rescaling, clipping, and write operations.

The key challenge in designing this virtual instruction set was finding optimal abstraction levels. The instruction set avoids creating separate instructions for each state in the finite state machine, which would result in unmanageable complexity. Conversely, overly abstract instructions would fail to capture critical micro-architectural details necessary for accurate performance modeling. The virtual instruction set represents the outer loop of the modeled FSM in the functional model, providing sufficient detail to capture performance-critical behaviors while maintaining manageability.

The CorePerfDSL implementation incorporates advanced constructs, including sub-pipelining, sub-stages, and parallel functional units. The micro-architectural description follows a hierarchical structure: Pipeline consisting of three stages (Fetch, ALU, Output), Stages subpipelines or microactions arranged linearly or in parallel. Microactions are fundamental building blocks representing individual operations associated with resource models defining behaviour and timing characteristics.

Mini-NPU employs a shared bus for multiple streamers that facilitates data transfers between memory and the (pre)fetch buffer. Since CorePerfDSL does not natively support bus modeling, the external resource models must address this. The external bus model is responsible for arbitrating access to the shared bus, ensuring that only one streamer can utilise the bus at any given time, and managing contention to optimise bus utilisation. The implemented bus model only models bus contention and not bus arbitration due to a single instruction scheduling window. Although this approach may introduce minor inaccuracies in the relative timing of individual stages, it effectively captures the overall impact of bus contention in the performance model. Additionally, because this mechanism is implemented in the external model, it allows for the inclusion of bus arbitration penalties as needed.

To model bus contention, two key modifications were made to the external resource model API of CorePerfDSL, (1) The start or request time is now passed to the external resource model API. This enables the bus contention model to determine when bus access is initiated, assessing potential contention. (2) The external resource model API was updated from *Time getDelay()* to *Time getCompletionTime(const Time& start_time)* to reflect the need for latency calculations based on the specific request time.

D. *Performance Model Scheduler - Design and Derivation*

Currently, the instruction scheduling element of the mini-NPU performance model is implemented manually to demonstrate the applicability of CorePerfDSL for accelerator modelling. However, future work envisions leveraging an automated code generator to produce executable performance models directly from CorePerfDSL descriptions.

The code generated by existing tools is not easily comprehensible and straightforward to debug. The generated code lacks a clear structure, established C++ best practices, and is insufficiently documented in relation to the original CorePerfDSL description. This lack of clarity complicates the process of understanding and correlating the generated code with its high-level specification. The general strategy for implementing a microaction execution is as follows:

1. The microaction *start time* is determined as the maximum of (a) the exit cycle of the current or previous pipeline stage, (b) the availability time of required data or resources, as indicated by the connector reads.
2. The microaction *completion time* is computed by adding the resource delay to the start time.
3. After completion, connectors are updated to reflect resource release or new data availability.
4. The stage *exit time* is determined as the maximum of (a) the completion time of all the microactions, (b) any back pressure from the subsequent pipeline stage.
5. The stage exit time is updated, and relevant timing information is logged for analysis.

In cases where a stage is bypassed (i.e., no microaction is executed for a particular instruction), the start time for the next active stage is set to the last exit time of that stage.

E. *Communication Interface between Functional Model and Performance Model*

CPU performance models use standardised ISA registers for communication with functional models, but mini-NPU lacks such standards, requiring custom communication interfaces. Two approaches facilitate communication: (1) *Virtual Instructions* that communicate functional behaviour of operations, triggering corresponding microactions in the performance model, (2) *State Variables* that pass state information from the functional model needed for dynamic timing calculations in external models, data fetch address exemplifies this need, as latency depends on memory hierarchy and data value count.

The naive and fast way to pass state variable data between the functional and performance models is to use read-only pointers. The limitation of this approach is that (1) the performance model cannot run independently of the functional model, limiting testing capabilities, and (2) adding new functional model features requires boilerplate code for pointer passing.

The better approach is to use state variables as attributes to virtual instruction with benefits (1) Direct performance model access to state variables from virtual instructions, (2) Independent performance model operation from functional model, (3) Loose coupling allowing separate thread/process execution (4) Support for manual/artificial virtual instruction traces. This approach provides better modularity and testing flexibility while maintaining necessary communication between models.

IV. Results

The performance model validation was conducted using four MLPerf Tiny benchmarks[16]: (1) Anomaly Detection, (2) Image Classification, (3) Keyword Spotting, and (4) Visual Wake Words, which represent standardised inference workloads for edge AI applications. The experimental methodology involved several key steps to compare the CorePerfDSL-based performance model and RTL simulation results accurately.

All MLPerf Tiny benchmark models required conversion to Infineon's packed tensor format, optimising memory efficiency through custom weight tensor representations. Comprehensive configuration files were generated for each neural network layer within the benchmark models, containing all necessary parameters, including layer dimensions, memory addresses, and operational parameters. RTL simulation results serve as ground truth for performance comparison, providing cycle-accurate execution timing for each layer under identical configuration conditions.

A. *Aggregate Performance Analysis*

The comprehensive evaluation across 82 layers from four diverse neural network workloads reveals strong overall accuracy for the CorePerfDSL-based mini-NPU performance model. Analysis shows that 84% of all evaluated layers achieve prediction errors within ±10% compared to RTL simulation, with more than half of the layers (58%) achieving errors below ±5%. This level of accuracy represents substantial improvement over analytical models while maintaining simulation speeds that are orders of magnitude faster than cycle-accurate RTL verification.

Table 1: Error Rates

Absolute Error	Layers	% of Total Layers
≤ 2%	10	-12%
≤ 5%	48	-58%
≤ 10%	69	-84%
> 10%	12	-14%

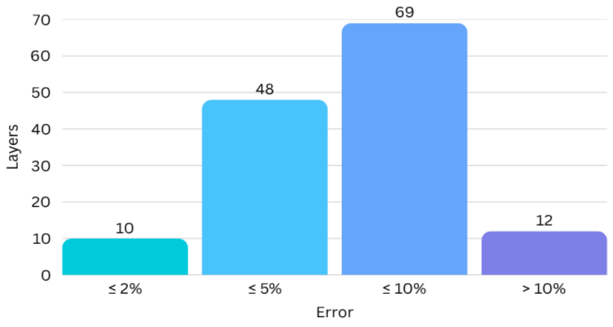

Figure 5: Absolute Errors vs Layers

B. *Performance Model Accuracy by Operation Type*

Operation type analysis reveals distinct performance characteristics aligning with underlying hardware implementation complexity. Fully connected operations achieve the highest accuracy with a mean absolute error of only 2.44%, validating the CorePerfDSL model's effectiveness for dense matrix multiplication operations. Depthwise convolution operations demonstrate exceptional accuracy with mean error of 3.47% and remarkably low variability, indicating robust modeling of this critical mobile neural network primitive.

Table 2: Error Analysis by Operation Type

Operation Type	Layer Count	Mean Absolute Error (%)	Standard Deviation (%)	Max Error (%)
FullyConnected	13	2.44	2.25	8.74
Conv2D	28	6.67	4.93	20.68
DWConv2D	17	3.47	1.63	7.14
Add	3	14.23	0.03	14.26
AvgPool2D	3	5.52	5.42	13.17
Softmax Components	18	13.38	22.22	79.55

Standard 2D convolutions show moderate accuracy with higher variability (mean = 6.67%), reflecting diverse geometric configurations and memory access patterns across different kernel sizes and channel depths. Element-wise addition operations consistently overestimate execution time with approximately 14% error across all instances, indicating systematic modeling error for data-movement-intensive operations.

Average pooling operations show moderate accuracy (mean = 5.52%) with notable variability, reflecting outlier behaviour (with error of 13.17%) that indicates that while the model captures general timing trends, specific pooling configurations may require more detailed modeling or calibrating to achieve higher accuracy.

Softmax operations exhibit the highest errors and variability (mean = 13.38%), particularly for exponential components, achieving maximum errors up to 79.55%. This reflects the complexity of modeling specialised hardware units implementing non-linear mathematical functions, where precise timing depends on algorithmic implementation details not fully captured in the current CorePerfDSL description.

V. Analysis and Limitations

Although the proposed performance modelling methodology produced good overall results for the mini-NPU case study, simple instruction-by-instruction scheduling of pipeline state microactions in the generated scheduler limits the precision with which arbitration of resource conflicts between microactions is performed.

A. Bus Arbitration Modeling

The scheduler derived from the CorePerfDSL microarchitecture description schedules microactions of each virtual instruction based only on the scheduling of its immediate predecessor. This means that external resource timing models (e.g. for memory accesses) can only account for resource contention, not arbitration. In effect, the performance model hardwires in an assumption that arbitration of resource contention prioritises microactions from earlier instructions (later pipeline stages) over those from later instructions.

Our performance model nonetheless produced good results for the mini-NPU use-case because, for most operations, the available memory bandwidth is saturated without significant backpressure in the pipeline. The overall delay introduced by memory contention is thus largely insensitive to the prioritisation applied: prioritising a later instruction rather than an earlier one simply results in a still-later instruction being delayed due to contention with the earlier instruction. In scenarios with lower memory bandwidth utilisation (e.g. Softmax), the inaccurate resolution of contention was a significant contributor to the observed timing inaccuracies.

B. Prefetcth and Prefetch Cancellation Modeling

The simple single instruction scheduling implementation also prevents accurate modelling of speculative prefetching with cancellation and similar mechanisms. Prefetching and cancellation are readily modeled as a microaction triggering fetches/prefetch sequences. Given a suitable external timing model (e.g. a model of buffer occupancy over time, with models of arbitration and memory access timing), accurate timing for memory traffic and the delivery of (pre)fetched data can be computed. However, the effects of prefetch cancellation are extremely sensitive to the precise scheduling of the cancelling microaction. Unlike memory contention, inaccuracies from inaccurate modeling of arbitration change the overall volume of (modeled) memory traffic. The resulting cumulative timing error is noticeable in overall timing, even with high memory bandwidth demand. This effect appears to be the cause of high-error "outliers" in the timing predictions for convolution operations.

VI. Conclusion and Future Work

This research successfully demonstrates the applicability and effectiveness of CorePerfDSL for performance modeling of compute accelerators, specifically through developing and validating a comprehensive mini-NPU performance model. The experimental evaluation across four diverse MLPerf Tiny benchmarks reveals that the CorePerfDSL-based approach achieves a mean absolute error below 10% for 84% of evaluated layers compared to RTL simulation, while providing substantially faster simulation speeds.

The method effectively separates functional and timing concerns, enabling rapid architectural exploration without sacrificing predictive accuracy for critical design decisions. The systematic evaluation across different neural network architectures validates the generalizability and robustness of the approach for realistic edge AI workloads.

Engineers can quickly evaluate the effects of different memory hierarchies, compute unit configurations, and pipeline organisations, enabling informed design decisions early in the development process. The model allows accurate timing analysis in design space exploration, helping answer critical "what-if" questions, including identifying optimal parameters for specific workloads, pinpointing performance bottlenecks, and determining best trade-offs between memory and performance.

Future research directions should focus on addressing identified limitations while expanding the methodology's scope and automation capabilities.

A. Multi-Instruction Scheduler

We aim to address the limitations by replacing our current simple single instruction scheduling model with a model maintaining a rolling "window" of partially scheduled instructions whose timing depends on unresolved resource contention. By maintaining and updating earliest-possible scheduling times for microactions in this window, contention can be accurately identified and resolved:

1. Each instruction in the operation sequence is scheduled and appended to the window based on the scheduling of its predecessor.
2. Where a final scheduling of a microaction cannot be computed due to unresolved resource contention or dependencies on other microactions, the earliest possible scheduling is computed instead.

3. Let *t* be the earliest possible start time of the earliest microaction(s) for which resource contention has yet to be resolved.
4. If the earliest microaction in the latest instruction is scheduled to start after or at time *t*:
    a) Resource contention at time *t* is resolved based on all potentially contending operations with earliest-possible start times *t*.
    b) After resource contention at time *t* is resolved, the schedules of the operations for which contention has been resolved are updated and made final, and later microactions are rescheduled to reflect these new final schedules.
    c) Instructions, all of whose micro-actions are fully scheduled, are identified, and all but the latest are removed from the scheduling window.

The multi-instruction scheduling model will eliminate the underlying causes of worst-case inaccuracies observed for the mini-NPU use case and provide a robust basis for use on a wider class of accelerator designs.

B. *Performance Model Code-generation*

Enhancing automated code generation capabilities would significantly improve the methodology's accessibility and reduce design space exploration effort. Applying the CorePerfDSL approach to other accelerator domains, such as cryptographic processors, would demonstrate the framework's generalizability beyond neural network accelerators. Advanced features like dynamic power estimation integration and support for multi-accelerator system modeling, would enhance the framework's utility for complex system-level design optimisation.

ACKNOWLEDGMENT

This work has been supported by the project MANNHEIM-FlexKI funded by the German Federal Ministry of Education and Research (BMBF) under contract no.01IS22086L.

REFFERENCES

[1] C. Foik et al., "CorePerfDSL: A flexible processor description language for software performance simulation," FDL, 2022.

[2] R. Kunzelmann, "Modeling Advanced CPU Pipeline Features for Fast and Accurate Performance Simulation of a RISC-V Microprocessor," Master Thesis, TUM, 2022.

[3] C. Foik et al., "Flexible Generation of Fast and Accurate Software Performance Simulators From Compact Processor Descriptions," IEEE TCAD, 2024.

[4] D. Mueller-Gritschneder et al., "The extendable translating instruction set simulator (etiss) interlinked with an mda framework for fast risc prototyping," RSP, 2017.

[5] N. Binkert et al., "The gem5 Simulator," SIGARCH Comput. Archit., 2011.

[6] R. E. Wunderlich et al., "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," ISCA, 2003.

[7] E. K. Ardestani and J. Renau, "ESESC: A fast Multicore Simulator Using Time-Based Sampling," HPCA, 2013.

[8] D. C. Powell and B. Franke, "Using Continuous Statistical Machine Learning to Enable High-Speed Performance Prediction in Hybrid Instruction-/Cycle-Accurate Instruction Set Simulators," CODES+ISSS, 2009.

[9] I. Böhm et al., "Cycle-Accurate Performance Modeling in an Ultra-Fast Just-In-Time Dynamic Binary Translation Instruction Set Simulator," SAMOS, 2010.

[10] M.-C. Chiang et al., "A QEMU and SystemC-Based Cycle-Accurate ISS for Performance Estimation on SoC Development," IEEE TCAD, 2011.

[11] V. Herdt et al., "Fast and Accurate Performance Evaluation for RISC-V using Virtual Prototypes," DATE, 2020.

[12] Y. S. Shao et al., "Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customised architectures," ISCA, 2014.

[13] SPARTA: A Modeling framework for performance analysis of hardware architectural platforms, https://github.com/sparcians/map.

[14] SystemC: IEEE Std. 1666-2023, https://www.accellera.org/downloads/standards/systemc.

[15] W. Zheng, "Early Design and Validation of an AI Accelerator's System Level Performance Using an HLS Design Methodology," DVCon Workshop, 2021

[16] Vijay Janapa Reddi, Christine Cheng, "MLPerf Inference Benchmark", 2020 47th Annual International Symposium on Computer Architecture(ISCA).

[17] "Unsupervised Detection of Anomalous Sounds for Machine Condition Monitoring", DCASE 2020 Challenge Task 2, http://dcase.community/challenge2020/task-unsupervised-detectionof-anomalous-sounds.

[18] CIFAR10 dataset, https://www.cs.toronto.edu/~kriz/cifar.html.

[19] Example workspace for software performance simulation with ETISS-based performance simulator, https://github.com/tum-ei-eda/PerformanceSimulation_workspace.