MASTER'S THESIS 2026

Machine-Learning-Based Correction of Analytical NPU Performance Models

Joaquim Gouveia, Oliver Gouveia

ISSN 1650-2884
LU-EIT-EX: 2026-1143

DEPARTMENT OF ELECTRICAL
AND INFORMATION TECHNOLOGY
LTH | LUND UNIVERSITY

EXAMENSARBETE

EIT

LU-EIT-EX: 2026-1143

Machine-Learning-Based Correction of Analytical NPU Performance Models

Maskininlärningsbaserad korrigering
av analytiska prestandamodeller för NPU:er

Joaquim Gouveia, Oliver Gouveia

Machine-Learning-Based Correction of Analytical NPU Performance Models

Joaquim Gouveia
jo0714ho-s@student.lu.se

Oliver Gouveia
ol3505jo-s@student.lu.se

June 8, 2026

Master's thesis work carried out at Arm Sweden AB.

Supervisors: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se
Fabio Pancot, fabio.pancot@arm.com
William Isaksson, william.isaksson@arm.com

Examiner: Pietro Andreani, pietro.andreani@eit.lth.se

"Essentially, all models are wrong, but some are useful."

– George E. P. Box

Abstract

Modern Neural Processing Units (NPUs) must be evaluated long before silicon exists, yet early tools force a difficult compromise: fast analytical performance models enable exploration, but may diverge from slower RTL simulation, which provides a more accurate timing reference. This thesis investigates whether that gap can be learned and corrected. Using early-available block-command descriptors, sequence summaries, and analytical cycle estimates from Arm's NPU modelling flow, supervised XGBoost regressors are trained as a correction layer rather than a replacement for the existing model. The method is evaluated on anonymised single-block and short-sequence workloads across three NPU units, comparing corrected predictions against RTL cycle measurements with a 5% acceptance criterion. Results show large improvements in difficult cases, raising several low baseline pass rates above 90% while substantially reducing median and tail errors. SHAP-based analysis further identifies workload regimes linked to systematic mismatch, making the correction model not only more accurate but also a practical diagnostic tool for model improvement.

Keywords: Neural Processing Unit, Performance modelling, RTL simulation, Machine learning correction, XGBoost, Model interpretability

Acknowledgements

First and foremost, we would like to thank our supervisors at Arm, Fabio Pancot and William Isaksson. Thank you for all the guidance, support, and feedback throughout this project, but also for the many conversations, laughs, and good moments along the way. We have learned a lot from both of you, and we are truly grateful for the time and energy you have put into helping us.

We would also like to thank our academic supervisor, Jonas Skeppstedt, for his support, guidance, and constructive feedback throughout the thesis process. We are especially grateful for his inspiring courses and for the passion with which he teaches, which has had a lasting influence on us and helped spark our interest in the topics explored in this thesis.

Special thanks go to Andreas Nevailander and the entire modelling team at Arm for their help, discussions, and for creating such a welcoming environment.

Finally, we would like to thank our parents and all of our friends for making these five years truly unforgettable. Your support, encouragement, and friendship have meant a great deal to us and have helped bring us to where we are today.

Confidentiality Statement

This thesis was carried out in an industrial hardware-development context at Arm, using internal verification and performance-modelling data. For confidentiality reasons, hardware-specific unit names, operation identifiers, command fields, descriptor names, and selected implementation details have been anonymised.

The anonymised labels are used consistently throughout the thesis and do not affect the prediction task, evaluation methodology, or interpretation of the results. Any descriptor-level explanations in this thesis are intended to describe anonymised workload regimes, not to disclose internal command fields, operation names, or hardware mechanisms.

Contents

Acknowledgements **i**

Confidentiality Statement **ii**

Abbreviations **vi**

1 Introduction **1**
1.1 Motivation . . . 2
1.2 Problem Statement and Scope . . . 3
1.3 Research Questions . . . 4

2 Background and Related Work **6**
2.1 NPU Execution Model . . . 6
2.1.1 MATMUL to Block Commands . . . 7
2.2 Performance Modelling Levels . . . 8
2.3 Gradient Boosting . . . 11
2.3.1 XGBoost . . . 12
2.4 SHAP for Interpretability . . . 14
2.5 Related Work Summary . . . 16
2.5.1 Simulation Fidelity and RTL Bottlenecks . . . 16
2.5.2 Learning from Early-Available Inputs . . . 17
2.5.3 Measurement-Driven Simulator Correction . . . 17
2.5.4 Prediction from Design Representations . . . 18

3 Data and Methodology **21**
3.1 Prediction Problem Overview . . . 21
3.2 Single-Block Prediction . . . 22
3.2.1 Sample Definition . . . 22
3.2.2 Input Features . . . 23
3.2.3 Prediction Target . . . 24
3.3 Sequence-Level Prediction . . . 24

3.3.1 Sequence Sample Definition . . . 25
3.3.2 Sequence Features . . . 25
3.4 Target Formulation . . . 26
3.5 Learning Models . . . 28
3.6 Evaluation . . . 30
3.7 Interpretability and Error Analysis . . . 31

4 Results **35**
4.1 Experiment Overview . . . 35
4.2 Exploratory Baseline Analysis . . . 35
4.2.1 Baseline Acceptance Rates . . . 35
4.2.2 Baseline Error Structure . . . 36
4.2.3 Simple Baseline Correction Models . . . 38
4.3 Single-Block Correction Results . . . 38
4.3.1 Unit X . . . 39
4.3.2 Unit Y . . . 42
4.3.3 Unit Z . . . 44
4.4 Sequence-Level Results . . . 46
4.4.1 Unit X . . . 47
4.4.2 Unit Y . . . 48
4.5 MATMUL Case Study . . . 48
4.6 Explainability and Error Analysis . . . 49

5 Discussion **55**
5.1 Interpretation of the Correction Results . . . 55
5.2 Target and Metric Trade-offs . . . 57
5.3 Sequence Modelling and Features . . . 59
5.4 Explainability and Diagnostic Value . . . 60
5.5 Limitations and Threats to Validity . . . 62
5.5.1 Feature- and Cycle-Space Coverage . . . 62
5.5.2 Limited Sequence Length . . . 63
5.5.3 Idealised Timing Configuration . . . 64
5.5.4 Cross-Unit and Scheduling Effects . . . 64

6 Conclusions **66**
6.1 Research Questions Revisited . . . 66
6.2 Main Contributions . . . 67
6.3 Future Work . . . 68

A Additional Results **74**
A.1 Simple Correction Baselines . . . 74

A.2 Target-Formulation Sweeps 76
A.2.1 Unit Y 77
A.2.2 Unit Z 78

Abbreviations

NPU	Neural Processing Unit
RTL	Register-Transfer Level
ML	Machine Learning
HW	Hardware
MAE	Mean Absolute Error
RMSE	Root Mean Squared Error
SHAP	Shapley Additive Explanations
blk_cmd	Block Command
CPU	Central Processing Unit
GPU	Graphics Processing Unit
PPA	Power, Performance, and Area
TOSA	Tensor Operator Set Architecture
FPGA	Field-Programmable Gate Array
kNN	k-Nearest Neighbours

Chapter 1

Introduction

Neural-network workloads are increasingly executed on specialised accelerators rather than only on general-purpose CPUs or GPUs. Neural Processing Units (NPUs) are designed to exploit the structure of tensor computations, such as matrix multiplications, convolutions, reductions over tensor dimensions, and data-layout transformations, in order to improve performance and energy efficiency. However, this specialisation also makes early performance estimation challenging: the runtime of an operation is not determined only by its high-level mathematical form, but also by how that operation is represented and issued to the hardware, including tensor shapes, precision, layout, tiling, data movement, and unit-specific execution modes.

This creates a practical challenge during hardware development. Many design and optimisation decisions must be made before final silicon is available, and once hardware exists, using it as the main vehicle for design iteration is far too late and costly. RTL simulation provides a detailed pre-silicon timing reference, but it is computationally expensive and too slow to support broad, repeated exploration across many workloads and design alternatives. Development workflows therefore rely on faster performance models at higher levels of abstraction to support agile hardware-software co-design and design-space exploration [13]. These models make early performance estimation practical, but their speed comes from simplifications that can cause their cycle estimates to deviate from RTL-observed behaviour.

This thesis investigates whether machine learning can reduce this gap by learning a correction layer on top of an existing fast analytical hardware model for Arm's NPU. The aim is not to replace the analytical model or RTL simulation, but to improve the alignment between early model estimates and RTL-measured execution cycles using information available before RTL results are known. In this setting,

machine learning appears in two distinct roles: neural-network workloads motivate the specialised accelerator being modelled, while supervised learning is used as a tool for improving the accelerator's performance model. Beyond prediction accuracy, the thesis also examines whether the learned correction can provide diagnostic value by identifying workload regimes and command-level features associated with systematic model–RTL mismatch.

1.1 Motivation

The usefulness of an early performance model depends on a trade-off between speed, availability, and fidelity. A model that is too slow cannot be used routinely across many generated workloads or design alternatives, while a model that is too abstract may fail to reflect important RTL timing behaviour. In this context, early performance estimates are therefore valuable because they support architectural exploration, software tuning, and performance analysis before the final implementation is available [13]. In the NPU development setting considered in this thesis, the analytical hardware model occupies the fast and practical end of this trade-off: it can provide estimates early enough to support iterative analysis, especially when RTL is incomplete or too slow for broad exploration [1], but it does not reproduce all implementation-level timing effects.

This practicality comes from abstraction. To remain fast and usable, the analytical model may omit or simplify parts of the low-level behaviour that ultimately determines runtime, including microarchitectural effects, timing interactions, and implementation-specific behaviour [1]. As a result, its estimates may deviate from those observed in more detailed RTL simulation. Such discrepancies matter because decisions based on inaccurate model estimates may not fully reflect the behaviour of the design that is eventually implemented.

Improving alignment with RTL is therefore highly valuable, but doing so directly is often costly. Increasing model fidelity typically requires substantial engineering effort and may reduce the speed that makes the model useful in the first place [1, 14]. This motivates the present work, which explores a data-driven alternative to manually increasing model fidelity: using machine learning to learn and reduce the remaining discrepancy relative to RTL without requiring the fast baseline model itself to be rebuilt at a substantially higher level of detail.

A further motivation is that model discrepancies are useful to understand, since they can carry information about where the analytical model fails to capture RTL behaviour. If a learned correction consistently depends on particular command

fields, workload shapes, layouts, or operation modes, this can help indicate where the analytical model and RTL differ most strongly. Interpretability methods such as SHAP [6] can therefore support model-debugging and correlation analysis by turning a trained correction model into evidence about which feature groups drive the predicted mismatch. Such analysis should not be interpreted as a direct causal explanation of the underlying hardware-model discrepancy, but it can provide strong indications of which workload regimes, command fields, and model paths should be inspected further.

1.2 Problem Statement and Scope

The problem considered in this thesis is to improve the agreement between a fast analytical hardware-performance model and RTL-observed execution-cycle behaviour. For each workload instance considered, the available inputs consist of early-stage execution descriptors, such as block-command or instruction-level parameters, together with the cycle estimate produced by the analytical model. The reference output is the corresponding execution-cycle count measured in RTL simulation.

The thesis formulates this as a supervised learning problem. Rather than predicting performance entirely from scratch, the learned model is used as a correction layer on top of the analytical estimate. In this setting, the model may either predict RTL-aligned execution cycles directly or learn the residual discrepancy between the analytical estimate and the RTL measurement, with the corrected prediction reconstructed from the baseline estimate and the learned correction.

The scope is limited to improving prediction for the measured block-level or sequence-level cases available in the collected datasets. The work does not attempt to replace RTL simulation, model full-graph execution time, capture compiler scheduling effects, or rebuild the analytical model at a higher level of implementation detail. The objective is instead to evaluate how far alignment with RTL can be increased using information that is available early enough to preserve the practical advantages of the existing analytical hardware-performance model.

1.3 Research Questions

RQ1 Can machine learning be used to predict the performance discrepancy between NPU hardware models and RTL implementations?

RQ2 Can machine learning-based performance models provide insights into the factors contributing to performance discrepancies between hardware model predictions and RTL simulation results?

Chapter 2

Background and Related Work

2.1 NPU Execution Model

The Neural Processing Unit considered in this thesis is a specialised accelerator for neural-network workloads. Compared with a general-purpose CPU, which executes a program as a stream of relatively small instructions over registers and memory operands, an NPU is organised around structured tensor computations. At a high level, it consists of specialised hardware units for different classes of work. The level of execution considered in this thesis is the work issued to such individual NPU units, rather than a full-network execution across the complete accelerator.

Before work can be issued to these units, a neural-network model defined in a framework such as PyTorch is first represented as a graph of tensor operations. These operations can be described using an intermediate representation such as TOSA, the Tensor Operator Set Architecture [9]. TOSA defines hardware-independent tensor operations, including elementwise operations, convolutions, and matrix multiplication. In this thesis, TOSA is used as a public way to describe the logical operation associated with lower-level NPU execution, without exposing hardware-specific operation encodings or the full set of operations supported by the accelerator.

Before execution, a logical tensor operation is lowered into hardware-facing work. This lowering process determines how the operation is issued to the NPU, for example through choices of tiling, layout, precision, traversal order, and unit-specific execution modes. These choices affect performance because the same high-level operation can correspond to different lower-level execution forms. For instance, matrix multiplications with different shapes, layouts, or reduction structures can

place different demands on hardware resources and data movement, and therefore require different numbers of execution cycles [11].

At the abstraction level used in this thesis, the lowered work is represented by block commands, abbreviated `blk_cmd`. A block command is a unit-level description of a tensor operation, tile, or sub-operation, together with the parameters needed to execute it on a particular NPU unit. In this limited sense, block commands are analogous to CPU instructions: both describe work issued to hardware.

Block commands therefore form the common execution-level reference point used throughout this thesis: they connect the logical tensor-operation view to the unit-level behaviour estimated by the analytical hardware model, measured in RTL simulation, and used to construct the machine-learning datasets. The following MATMUL example illustrates this relationship between a familiar tensor operation, a TOSA-level representation, and the block-command-level execution view used throughout the thesis.

2.1.1 MATMUL to Block Commands

To make the origin of a block command more concrete, consider a matrix multiplication written at the PyTorch framework level as:

```
output = torch.matmul(input_A, input_B)
```

At a tensor-operator level, the same operation can be represented as a TOSA `MATMUL`. One example used during this work has the logical form:

```
input_A = shape(1, 256, 7).type(BF16)
input_B = shape(1, 7, 32).type(BF16)

MATMUL in(input_A, input_B).out(output)
```

This representation describes the mathematical operation: a BF16 matrix multiplication with batch size 1, output rows $M = 256$, reduction dimension $K = 7$, and output columns $N = 32$. Before execution on the NPU, this logical operation is lowered into hardware-facing block commands. In the simplest case, the operation may correspond to a single MATMUL block command. In other cases, the same logical operation may be represented by several block commands. For illustration,

a reduction dimension with $K = 7$ can be decomposed into partial reductions in several mathematically equivalent ways. One possible decomposition is

$$K = 1 + 1 + 4 + 1 = 7,$$

which produces four block commands that together represent the same source-level MATMUL operation. Thus, the single-block and multi-block representations are equivalent at the source-workload level, but not at the execution-schedule level: they describe the same logical computation, but differ in how that computation is issued to the NPU after lowering, for example due to compiler choices, scheduling decisions, or interactions with other unit-level work.

This example illustrates the level of abstraction used in the thesis. The PyTorch and TOSA forms describe what computation is being performed, while the block-command representation describes how the work is issued to the NPU. The exact hardware command fields are not needed here; what matters is that the performance model, RTL simulation, and the learning datasets operate on this block-command-level representation.

2.2 Performance Modelling Levels

Hardware development typically relies on multiple modelling levels because no single model can simultaneously provide high execution speed, low implementation cost, and full fidelity to the final design. This thesis considers three such levels: the functional/analytical model, the cycle-approximate model, and the RTL model. These levels occupy different points in the trade-off between speed and timing fidelity. More abstract models support fast exploration and evaluation at a larger scale, while more detailed models provide closer correspondence to the implemented design at substantially higher computational and engineering cost [1, 14]. The resulting modelling stack is therefore best understood as a hierarchy of abstractions with distinct roles in the development workflow. In practice, these abstractions may be used both complementarily and, in some settings, as alternative paths to improved timing estimation.

At the highest level relevant to this thesis, the functional or analytical model provides a fast representation of hardware behaviour and baseline performance estimates. Its purpose is to support early exploration, software analysis, and practical runtime estimation without reproducing the implementation in full detail. In this setting, execution cost is estimated through analytical formulas, heurist-

ics, counters, and other simplified rules derived from hardware knowledge. This makes the model attractive in practice because it is fast, scalable, and relatively inexpensive to develop and use. However, because its timing behaviour is generated through abstraction rather than through a detailed reconstruction of the implementation, its estimates may deviate from those observed in more faithful models [13].

A more detailed abstraction is provided by the cycle-approximate model. In contrast to the functional or analytical model, the cycle-approximate model aims to follow the implementation much more closely by recreating timing-relevant behaviour in software. In this sense, it represents an attempt to mirror RTL behaviour as closely as possible without directly using RTL simulation. Such a model can provide substantially improved timing fidelity, but it also comes with significant engineering cost, since implementing and maintaining a faithful cycle-approximate representation requires considerable development effort [1, 14]. In practice, cycle-approximate models are often used alongside fast functional or analytical models, providing a more detailed timing-oriented estimate when higher fidelity is needed without making such detailed simulation the default for all analyses.

At the most detailed pre-silicon level, RTL models describe the hardware at the register-transfer level and therefore provide the most implementation-faithful executable representation available during development. For this reason, RTL simulation is treated in this thesis as the primary reference against which higher-level model estimates are evaluated and correlated. Its main drawback is computational cost: RTL simulation is substantially slower and less scalable than higher-level models, which makes it impractical as the sole basis for iterative performance estimation or broad design-space exploration [1, 14]. Although RTL is not immune to bugs or regressions, it remains, within the workflow considered here, the most reliable executable timing reference available.

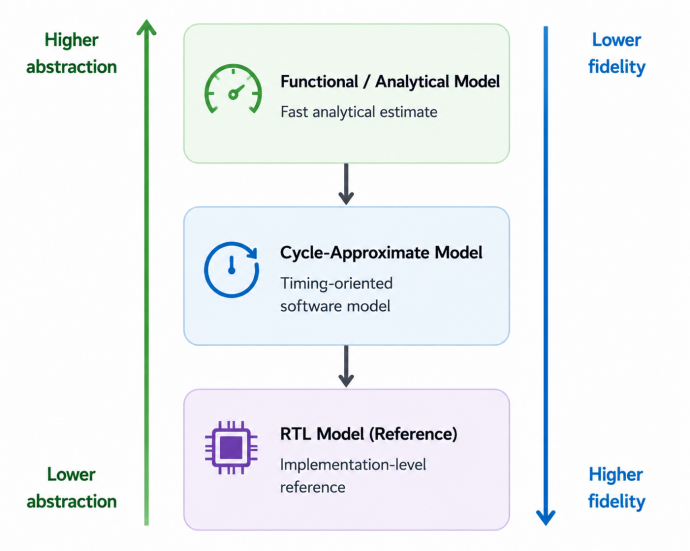

Figure 2.1: Hierarchy of performance-modelling abstractions used in the development workflow. Higher-level analytical models provide fast and scalable estimates, while RTL simulation provides the most reliable pre-silicon reference. Cycle-approximate models occupy an intermediate point in the trade-off between speed and timing fidelity.

This hierarchy frames the prediction problem studied in this thesis. The functional/analytical model provides the fast baseline estimate, while RTL simulation provides the reference measurement used to evaluate alignment. The cycle-approximate model represents the conventional route towards higher timing fidelity, but at greater engineering and runtime cost. The proposed approach instead asks whether a learned correction layer can reduce part of the gap between the fast analytical model and RTL without rebuilding the baseline model at a substantially higher level of detail.

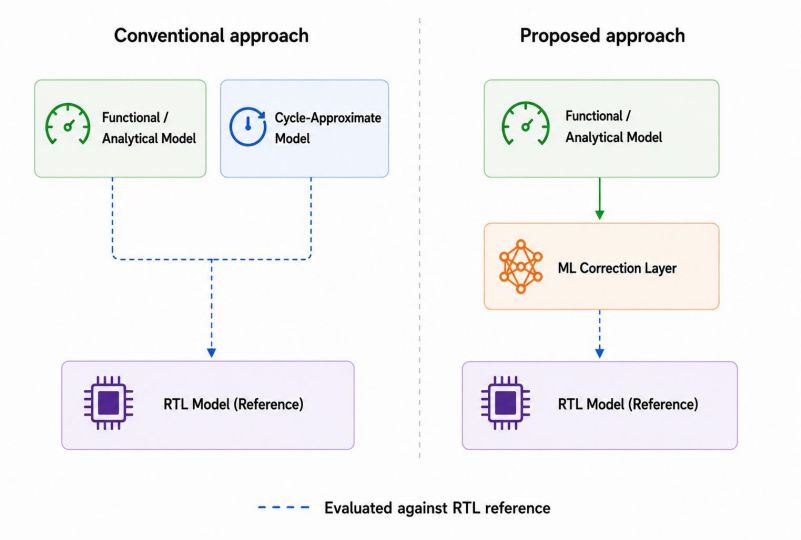

Figure 2.2: Conventional and proposed modelling paths. The conventional route towards higher timing fidelity uses a more detailed cycle-approximate model, while the proposed approach applies an ML correction layer on top of the fast functional/analytical model and evaluates the corrected estimate against RTL.

2.3 Gradient Boosting

Gradient boosting is a supervised machine learning ensemble method in which a predictor is built as a sum of simpler models added sequentially. In the case of regression trees, the final predictor can be written as

$$F_M(\mathbf{x}) = \sum_{m=1}^{M} f_m(\mathbf{x})$$

where each f_m is a weak learner, typically a regression tree. The core idea is that the model is not learned in one step; instead, new learners are added iteratively so that each stage improves the current approximation. For mean square error regression, this can be understood intuitively as fitting the new learner to the residuals of the current model. More generally, gradient boosting can be interpreted as optimisation in function space, where at each iteration a new function is chosen to move the model in a direction that reduces the loss [2].

Let $\hat{y}_i^{(t-1)}$ denote the prediction for sample i at iteration $t-1$. The model is then updated as

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i),$$

where f_t is the learner added at iteration t. Thus, each stage of boosting refines the current predictor by modelling structure that remains unexplained by the previous ensemble. Consequently, the framework is applicable to a broad range of supervised learning problems beyond simple least-squares regression.

For tabular datasets, gradient boosting is especially effective as the weak learners can model nonlinearities, threshold effects, and conditional feature interactions without requiring a fixed parametric form [2].

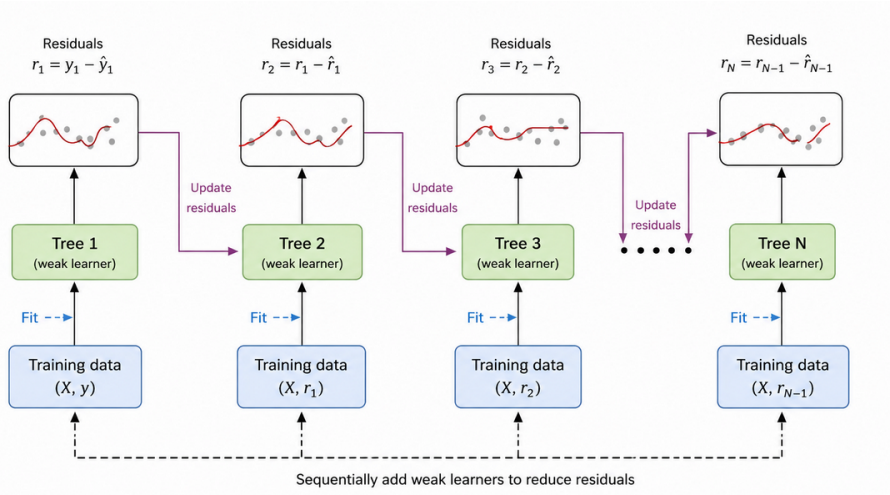

Figure 2.3: Simplified illustration of the sequential residual correction process in gradient boosting.

2.3.1 XGBoost

XGBoost is a highly optimised implementation of gradient boosted decision trees. Chen and Guestrin formulated the method as an additive tree model in which the prediction for sample x_i is

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), \qquad f_k \in \mathcal{F},$$

where $\mathcal{F}$ denotes the space of regression trees. Each tree maps an input sample to a leaf weight, and the final prediction is the sum of the leaf contributions across all trees [2].

A key feature of XGBoost is that learning is formulated through a regularised

objective:

$$\mathcal{L} = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k),$$

where $l(y_i, \hat{y}_i)$ is the training loss and $\Omega(f_k)$ penalises the complexity of tree f_k. In the formulation given by Chen and Guestrin, the regularisation term is

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2,$$

where T is the number of leaves in the tree and w_j is the score associated with leaf j. This means that XGBoost not only seeks low training error but also discourages overly complex trees and large leaf weights, which helps improve generalisation.

At boosting iteration t, the model adds a new tree f_t to the current predictor:

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i).$$

Rather than optimising the full objective from scratch after each addition, XGBoost uses a second-order Taylor approximation of the loss around the current prediction. This gives the approximate objective

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2 \right] + \Omega(f_t),$$

where

$$g_i = \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)}), \qquad h_i = \partial^2_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)}).$$

Thus, the new tree is fitted using both first-order and second-order information from the loss. This is one of the main technical characteristics that distinguishes XGBoost from simpler boosting formulations.

For a fixed tree structure, the training instances are partitioned into leaf sets I_j. The objective can then be rewritten in terms of leaf-wise gradient and Hessian sums,

$$G_j = \sum_{i \in I_j} g_i, \qquad H_j = \sum_{i \in I_j} h_i,$$

which leads to the optimal leaf weight

$$w_j^* = -\frac{G_j}{H_j + \lambda}.$$

Substituting this back into the objective yields a scoring expression for the quality of a tree structure, from which the gain of a split can be derived. In practice, this gives an efficient criterion for deciding whether a split improves the model enough to justify the increase in complexity [2].

From a modelling perspective, XGBoost is well suited to structured regression problems with heterogeneous engineered features and nonlinear interactions. Tree ensembles can naturally represent conditional effects, threshold behaviour, and feature interactions that would be difficult to encode explicitly in linear models. A further practical advantage is that XGBoost generally does not require numerical features to be normalised or standardised. Since tree splits are based on feature thresholds, features with different physical units can be used directly after parsing and encoding. XGBoost also includes support for sparse inputs and missing values during split finding, which is helpful for heterogeneous tabular feature sets [2]. This profile matches the regression problems considered in this thesis, where the inputs consist of analytical baseline estimates together with structured hardware-command and sequence-related features.

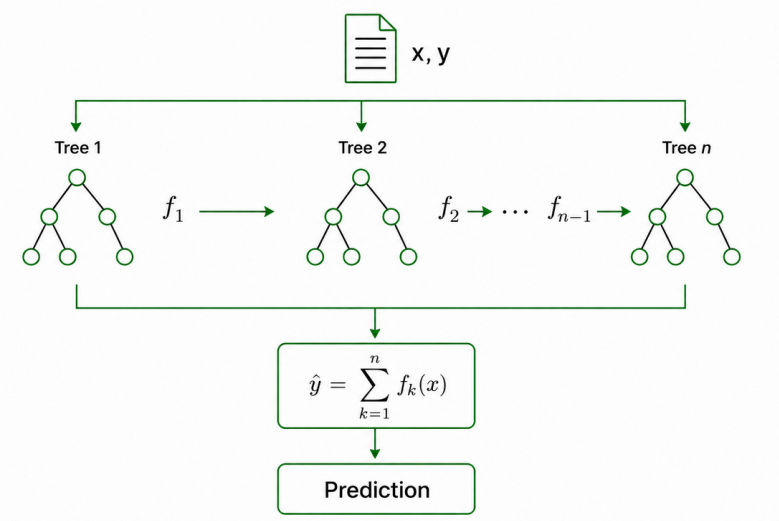

Figure 2.4: High-level illustration of XGBoost as an ensemble of decision trees.

2.4 SHAP for Interpretability

As previously mentioned, tree-based ensemble methods such as gradient boosting can capture nonlinear relationships, threshold effects, and feature interactions,

which makes them useful for structured regression problems. However, this flexibility also makes their predictions less directly interpretable than those of simpler models. In the context of this thesis, interpretability is relevant because the learned models are used not only to improve prediction accuracy, but also to analyse which early-available features are associated with discrepancies between analytical hardware-model estimates and RTL-observed behaviour.

SHAP values provide a feature-attribution method for explaining individual model predictions [6]. The method is based on Shapley values from cooperative game theory and assigns each input feature a contribution to a specific prediction. For a model f and an input sample x, the prediction can be written as

$$f(x) = E[f(X)] + \sum_{i=1}^{M} \phi_i,$$

where $E[f(X)]$ is the average output of the explained model over a background dataset, M is the number of input features, and ϕ_i is the SHAP value assigned to feature i. In this thesis, the explained model is the trained XGBoost model, not the analytical HW model. For one specific sample, each SHAP value describes how one feature moves the XGBoost output away from this average output. A positive SHAP value pushes the XGBoost output upward, while a negative SHAP value pushes it downward.

The Shapley value for a feature is defined by considering its marginal contribution across possible subsets of the remaining features:

$$\phi_i = \sum_{S \subseteq F \setminus \{i\}} \frac{|S|!(M - |S| - 1)!}{M!} \left[f_{S \cup \{i\}}(x_{S \cup \{i\}}) - f_S(x_S) \right],$$

where F is the full set of input features, $M = |F|$, and S is a subset that does not contain feature i. Intuitively, this expression measures how much feature i changes the prediction on average when it is added to different combinations of other features. Directly evaluating all such subsets is expensive, but efficient algorithms exist for tree-based models, making SHAP practical for gradient-boosted decision trees such as XGBoost [7].

SHAP values can be interpreted both locally and globally. A local explanation describes how the features of one input sample contribute to that sample's XGBoost output. A global view can be obtained by aggregating SHAP values over many samples, for example by averaging their absolute values, which indicates which

features have the largest average contribution to the model's outputs. In the context of residual correction, this distinction is useful because local explanations can show why a particular correction was predicted, while global summaries can indicate which types of execution descriptors tend to influence the learned correction model.

SHAP values should be interpreted as explanations of the trained model rather than as direct proof of causality in the underlying hardware. A feature with high SHAP importance is influential for the model's predictions, but this does not necessarily mean that the feature alone causes the corresponding RTL discrepancy. For this reason, SHAP analysis is best viewed as an exploratory tool for interpreting learned model behaviour and guiding later error analysis, rather than as a substitute for detailed hardware-level investigation.

2.5 Related Work Summary

Related work relevant to this thesis spans four overlapping areas: methods for balancing simulation fidelity and throughput [1, 14], machine-learning approaches for performance prediction from early-available execution information [8, 4, 15], learning-based calibration of approximate simulators against more faithful measurements [12], and ML methods that map earlier hardware-design representations to later implementation metrics [3]. The present work is most closely related to learning-based performance modelling and measurement-driven correction, since it learns a corrective layer on top of an existing hardware model using early-available execution descriptors and RTL-measured execution cycles as the target.

2.5.1 Simulation Fidelity and RTL Bottlenecks

A first line of related work addresses the longstanding trade-off between simulation speed and fidelity [1, 14]. RTL simulation remains a critical tool for hardware design, but its runtime often bottlenecks iterative development and performance analysis [1]. To reduce this cost, prior work has explored both direct acceleration of RTL simulation and the use of more abstract simulators [1, 5, 14]. For example, RTLFlow accelerates batched RTL simulation by translating RTL simulation code into CUDA kernels, illustrating one route towards improving throughput without changing the reference level of abstraction [5]. FireAxe illustrates another route towards improving RTL simulation throughput, using FPGA acceleration for large-scale RTL designs while still operating close to the RTL reference [14]. These works

motivate the central problem addressed in this thesis from the opposite direction: rather than making RTL simulation itself faster, the thesis investigates whether a fast analytical model can be made more RTL-aligned through a learned correction layer.

2.5.2 Learning from Early-Available Inputs

A second line of work uses machine learning to predict performance from early-available execution information, such as instruction content, execution context, or workload descriptors [8, 4, 15]. Ithemal predicts basic-block throughput directly from instruction opcodes and operands, demonstrating that learned models can outperform hand-written analytical throughput estimators [8]. SimNet applies machine learning to accelerate architecture simulation by predicting instruction latencies from static instruction properties and dynamic processor state [4]. Similarly, recent work on RISC-V performance estimation combines a fast functional simulator, cycle-accurate RTL implementations, and machine learning to obtain fast and accurate performance prediction across microarchitectures [15].

These works are closely related because they show that timing behaviour can often be inferred from representations that are cheaper or earlier than full detailed simulation [8, 4, 15]. However, unlike the present work, they do not primarily focus on learning a correction from hardware-model predictions to RTL-observed performance. Concorde is also relevant because it combines analytical modelling with machine learning rather than treating ML as a complete replacement for modelling. It uses compact analytical representations of microarchitectural behaviour together with learned components to enable fast CPU performance prediction for design-space exploration [10]. However, its goal is to construct a surrogate CPU performance model across microarchitectural configurations, whereas the present thesis starts from an existing NPU analytical model and learns a correction to improve agreement with RTL execution-cycle measurements.

2.5.3 Measurement-Driven Simulator Correction

The line of work closest in spirit to this thesis treats model error itself as something that can be learned and reduced from measurements.

DiffTune is a particularly relevant example: rather than replacing a simulator with a black-box predictor, it learns parameters of an existing basic-block CPU

simulator from coarse-grained end-to-end measurements and then plugs the learned parameters back into the original simulator to reduce prediction error [12]. This is closely related to the present thesis, but the learned component is inserted differently: DiffTune calibrates parameters inside the simulator, whereas this thesis learns an external correction layer applied to the analytical model output.

More broadly, this line of work treats the gap between a fast approximate model and a more faithful reference as something that can be learned and reduced from data. The present thesis follows this discrepancy-learning view: the analytical model provides a fast but imperfect estimate, while the learned model predicts the remaining difference to the RTL reference. This distinguishes the approach from direct cycle prediction, since the baseline hardware-model estimate remains an explicit part of the prediction pipeline.

2.5.4 Prediction from Design Representations

Adjacent work has explored predicting downstream power, performance, and area (PPA) metrics directly from RTL representations. For example, MasterRTL proposes a pre-synthesis PPA estimation framework that transforms HDL into a bit-level simple operator graph and predicts timing, power, and area before synthesis [3]. This line of work is relevant because it similarly seeks early prediction from pre-implementation representations, but it differs from the present thesis in both inputs and targets: it operates on RTL design structure for PPA estimation, whereas this thesis learns to correct an existing hardware performance model using execution-level features and RTL-measured execution cycles.

This distinction is important for the scope of the present work. RTL-structural features could potentially improve prediction accuracy, but they would also make the method dependent on a later-stage representation. The thesis therefore focuses on block-command-level and model-derived features in order to preserve compatibility with the existing early-stage performance-modelling workflow and to evaluate how far RTL alignment can be improved without using RTL structure as an input.

Overall, prior work suggests three main routes to accurate early performance estimation: increasing simulator fidelity or throughput, learning performance directly from early-available execution features, or calibrating approximate models against more faithful references. The present thesis is positioned closest to the latter two directions. It differs from direct learned performance prediction by keeping the analytical hardware model as an explicit baseline, and it differs from RTL-structural prediction by using block-command-level and model-derived features

rather than RTL design representations. The resulting gap addressed by this thesis is therefore narrower but practically important: learning an RTL-aligned correction layer for an existing fast NPU performance model.

Chapter 3

Data and Methodology

3.1 Prediction Problem Overview

The prediction task in this thesis is defined as cycle-count correction for workloads executed on selected units within the NPU. The target quantity throughout the work is the execution-cycle count observed in RTL simulation, which is treated as the timing reference for evaluation. Depending on the unit and dataset, prediction is performed either at the single-block level or at the sequence level.

For single-block prediction, one sample corresponds to one executed block command (`blk_cmd`) together with its analytical-model estimate and associated descriptors. The target variable is the RTL-measured execution-cycle count of that block.

For sequence-level prediction, one sample corresponds to an executed sequence of block commands, and the target variable is the RTL-measured execution-cycle count of the full sequence execution. The sequence datasets considered in this work include two cases: short sequences of block commands, and sequences where multiple block commands jointly represent one logical operation. The MATMUL example in Section 2.1.1 illustrates the latter case, where one logical operation is split across several block commands.

The experiments focus on selected compute-oriented NPU units. For confidentiality, the results chapter reports these units using anonymised labels, while single-block and sequence-level settings are considered where relevant.

The learning problem is formulated as model correction rather than standalone cycle prediction. The analytical performance model is treated as the baseline pre-

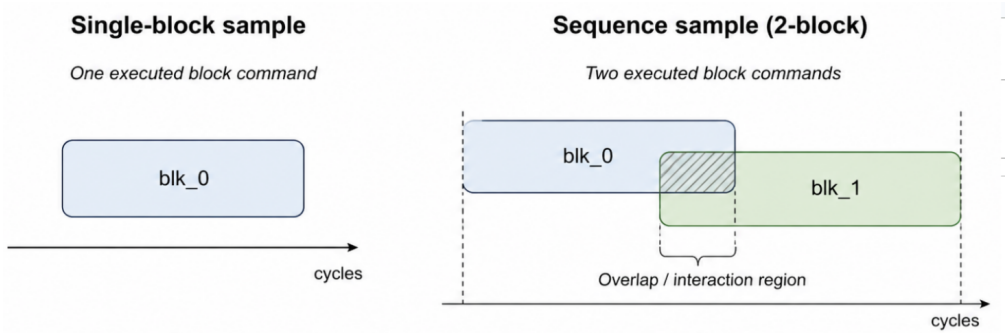

Figure 3.1: Illustration of single-block prediction and a conceptual sequence-level prediction setting, where multiple block commands may overlap or interact during execution.

dictor, and the ML model uses block- or sequence-level descriptors together with that baseline estimate to produce an RTL-aligned execution-cycle prediction. The scope is limited to the measured unit-level block and sequence cases; it does not include full-graph execution time, compiler scheduling effects, cross-unit scheduling, or system-level runtime effects.

3.2 Single-Block Prediction

This section defines the single-block prediction setting used in the main correction experiments. It first describes how one executed block command is represented as one supervised-learning sample, then summarises the input features used by the models, and finally defines the RTL-measured cycle count used as the prediction target.

3.2.1 Sample Definition

In the single-block setting, one sample corresponds to one executed block command. Each sample contains an execution-level workload description, the analytical HW model estimate for that workload, and the corresponding RTL-measured execution-cycle count. The workload description and analytical estimate form the model input, while the RTL measurement is kept separate and used only as the supervised-learning target.

Formally, a single-block sample can be written as

$$(\mathbf{x}_i, y_i^{\text{model}}, y_i^{\text{RTL}}),$$

where $\mathbf{x}_i$ denotes the block-command descriptors, y_i^{model} is the analytical HW model execution-cycle estimate, and y_i^{RTL} is the RTL-measured execution-cycle count. The prediction task is to use $\mathbf{x}_i$ and y_i^{model} to produce a corrected prediction $\hat{y}_i^{\text{corr}}$ that is closer to y_i^{RTL} than the original analytical estimate.

The underlying workloads are generated through a combination of directed test intent and constrained randomisation. Therefore, the sample is defined by the resolved workload that was actually executed, rather than only by the high-level intended test configuration. This is important because the model input must describe the concrete block command seen by both the analytical HW model and RTL.

The workloads are not hand-written one by one. Instead, they are generated from test specifications that describe the type of case to create, while constrained randomisation selects the concrete parameter values. For example, a specification may request a certain operation or class of tensor shape, but the final dimensions, formats, strides, and other command parameters are resolved during generation, satisfying the legal input space. Therefore, each sample is defined by the concrete block command that was actually executed, not only by the high-level generation intent. This is the command evaluated by both the analytical HW model and the RTL simulation, and it is therefore the representation used as input to the ML model.

Only information available before the RTL measurement is known is included in the input representation. This prevents target leakage and keeps the prediction setting aligned with the intended use case: improving early analytical HW model estimates without relying on RTL results at prediction time.

3.2.2 Input Features

The single-block feature vector combines the analytical HW model estimate with descriptors of the executed block command. Depending on the anonymised unit and operation family, these descriptors may represent operation type, execution mode, data format, precision, shape-related quantities, layout-related fields, and other computation parameters. Hardware-specific field names and selected implementation details are anonymised where required.

Some descriptors are used directly, while others are transformed into derived quantities that summarise the scale or structure of the workload. For example, shape-related fields may be combined into volume-like or ratio-based descriptors. Discrete descriptors are encoded before training. The exact feature set differs across unit-operation groups, but the same principle is used throughout: features are included only when they describe workload information available before RTL execution-cycle measurements are known.

3.2.3 Prediction Target

The target for single-block prediction is the RTL-measured execution-cycle count of the executed block command. The model is not given this value as input. Instead, it uses the block-command descriptors and the analytical HW model estimate to produce an RTL-aligned prediction. The target can be learned directly as RTL cycles or indirectly through one of the correction formulations described in Section 3.4.

3.3 Sequence-Level Prediction

The sequence-level setting extends the single-block formulation from one block command to a short structured execution context. In this setting, one sample corresponds to the execution of a complete sequence, and the target is the RTL-measured execution-cycle count of that sequence. The sequence-level task was conducted as a further experiment since the runtime of a sequence is not necessarily equal to the sum of the independently measured runtimes of its constituent commands. Ordering effects, overlap, shared state, or other unit-level interactions may affect the total execution time.

Furthermore, although a sequence can be viewed intuitively as the runtime of its independently run constituent block commands minus possible overlap, the effective contribution of each block command may change when it is executed in sequence. A block command that has one standalone runtime may take a different effective amount of time when preceded by another command. The sequence model therefore aims to learn joint execution behaviour, rather than simply summing standalone block-command predictions.

3.3.1 Sequence Sample Definition

A sequence sample can be written as

$$(\mathbf{s}_i, y_{\text{seq},i}^{\text{model}}, y_{\text{seq},i}^{\text{RTL}}),$$

where $\mathbf{s}_i$ denotes the sequence-level workload description, $y_{\text{seq},i}^{\text{model}}$ is the analytical HW model estimate for the full sequence, and $y_{\text{seq},i}^{\text{RTL}}$ is the RTL-measured execution-cycle count of the full sequence. The prediction task is to use the sequence descriptors and the analytical sequence estimate to produce a corrected prediction $\hat{y}_{\text{seq},i}^{\text{corr}}$.

The sequence datasets considered in this thesis contain short sequences only. Depending on the dataset, the sequence may either consist of a short two-command sequence, or of a multi-block reduction sequence where multiple block commands jointly represent one logical operation. The MATMUL example in Section 2.1.1 corresponds to a reduction, where one source-level operation may be represented by multiple block commands after lowering.

3.3.2 Sequence Features

The sequence feature vector follows the same principle as the single-block representation, but extends it with compact information about the constituent commands and their execution context. Each sequence sample includes the analytical HW model estimate for the full sequence, compact descriptors of the sequence elements, and summary features describing relationships between the constituent elements where applicable.

The descriptors may include operation identifiers, the number of block commands represented by the sequence, selected shape or workload summaries, and simple relationships between sequence elements. The exact representation depends on the sequence type, but the common goal is to describe the structured workload context without using the RTL sequence measurement as an input feature.

For the deployable sequence models, learned single-block predictions are used as context features where applicable. These provide estimated standalone costs for the constituent block commands or subsequences without using standalone RTL measurements as inputs. This keeps the deployable sequence model aligned with the intended prediction setting, where RTL measurements are not available at prediction time.

For selected experiments, diagnostic oracle-context variants were also evaluated by replacing the learned standalone-cost estimates with standalone RTL measurements. These variants are not deployable, since the required RTL measurements would not be available at prediction time. However, they provide a useful comparison for separating sequence-level modelling error from errors introduced by imperfect single-block predictions.

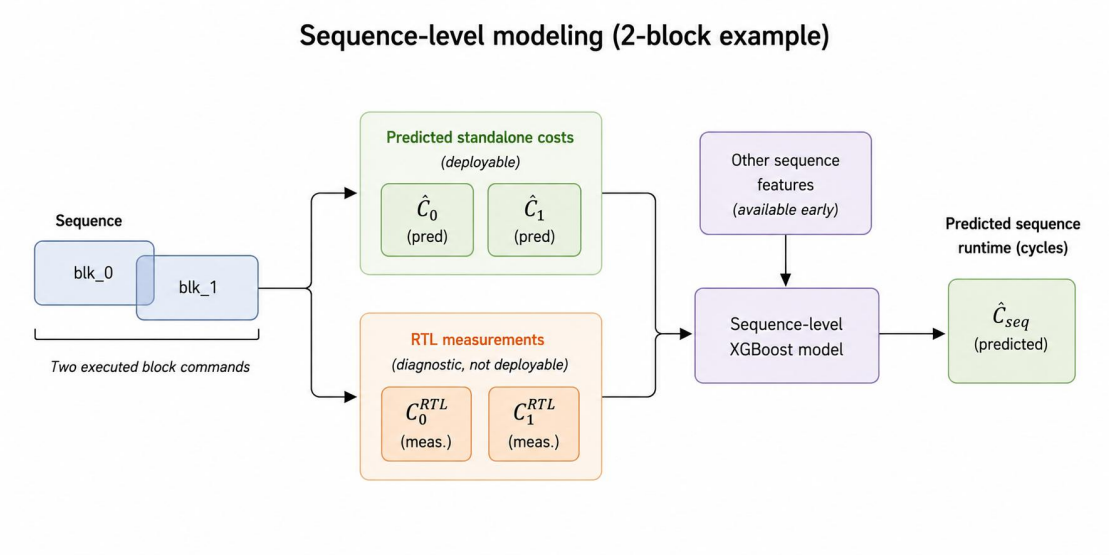

Figure 3.2: Conceptual sequence-level feature construction. The sequence model uses context features derived either from predicted standalone costs, in the deployable setting, or from standalone RTL measurements, in the diagnostic RTL-context setting.

The final result is a tabular feature matrix consisting of workload descriptors, derived numerical quantities, encoded categorical variables, analytical HW model estimates, and sequence-context features where applicable. The RTL execution-cycle measurements are kept separate and are used only to define the learning target, as described in the next section.

3.4 Target Formulation

For each obtained sample, two cycle-count quantities are available: the execution-cycle count measured in RTL simulation and the corresponding estimate produced by the analytical hardware model. Let y_{RTL} denote the RTL-measured execution-

cycle count and let y_{model} denote the analytical-model estimate for the same workload instance. The target formulation determines whether the learning model predicts RTL execution cycles directly or predicts a correction relative to the analytical-model estimate.

Four target formulations were considered in this work. The first formulation uses the RTL-measured execution-cycle count directly:

$$t_{\text{rtl}} = y_{\text{RTL}}.$$

In this case, the prediction produced by the learned model is already an estimate of RTL-aligned execution cycles:

$$\hat{y}_{\text{corr}} = \hat{t}_{\text{rtl}}.$$

The second formulation uses an additive residual target, defined as the difference between the RTL measurement and the analytical-model estimate:

$$t_{\text{res}} = y_{\text{RTL}} - y_{\text{model}}.$$

Here, the learned model predicts the correction that should be added to the analytical estimate. The corrected execution-cycle prediction is reconstructed as

$$\hat{y}_{\text{corr}} = y_{\text{model}} + \hat{t}_{\text{res}}.$$

The third formulation uses a direct ratio between the RTL measurement and the analytical-model estimate:

$$t_{\text{ratio}} = \frac{y_{\text{RTL}}}{y_{\text{model}}}.$$

This target represents the discrepancy as a multiplicative correction factor. The corrected execution-cycle prediction is reconstructed by multiplying the analytical estimate by the predicted ratio:

$$\hat{y}_{\text{corr}} = y_{\text{model}} \cdot \hat{t}_{\text{ratio}}.$$

The fourth formulation uses a logarithmic ratio between the RTL measurement and the analytical-model estimate:

$$t_{\log} = \log\left(\frac{y_{\mathrm{RTL}}}{y_{\mathrm{model}}}\right).$$

This target also represents the discrepancy as a multiplicative correction, but in log space. The corrected execution-cycle prediction is reconstructed by exponentiating the predicted target and multiplying by the analytical estimate:

$$\hat{y}_{\mathrm{corr}} = y_{\mathrm{model}} \cdot \exp(\hat{t}_{\log}).$$

The ratio and log-ratio formulations require positive RTL and analytical-model cycle counts and were therefore applied only to valid samples satisfying this condition.

Regardless of the target formulation used during training, all final predictions are converted back to execution-cycle space before evaluation. This allows the different target choices to be compared using the same cycle-level metrics and against the same RTL reference.

3.5 Learning Models

The datasets considered in this thesis are heterogeneous across hardware units, operation families, and prediction settings. Different groups expose different command descriptors and may exhibit different forms of discrepancy relative to RTL. For this reason, the learning problem was decomposed into separate regression tasks rather than treated as a single global prediction problem.

The evaluated datasets span three anonymised NPU units, denoted Unit X, Unit Y, and Unit Z. In the single-block experiments, separate models were trained for each anonymised unit-operation group reported in Chapter 4. This allows each model to specialise to the feature space and runtime behaviour of the corresponding workload group, while avoiding the need to describe unit-specific operation families in detail.

Sequence-level prediction was also decomposed into separate learning problems. For Unit X, separate sequence models were trained for two distinct classes of short block-command sequences. This decomposition reflects the fact that the sequence

classes expose different execution structures: one class emphasises interactions between separate commands, while the other represents a single logical operation distributed across multiple block commands. For Unit Y, one sequence model was trained for two-command sequences. No sequence-level dataset was available for Unit Z in this work, so Unit Z was evaluated only in the single-block setting.

All primary experiments use XGBoost regression as the learning method. XGBoost was selected because the dataset is structured and tabular, containing a mixture of numerical command descriptors, encoded categorical fields, tensor-shape information, and analytical-model features. Gradient-boosted decision trees are well suited to this type of input because they can represent nonlinear relationships, threshold effects, and feature interactions without requiring all interactions to be manually specified.

The same overall modelling procedure was used across all units and operation families. Each model receives the extracted feature representation together with one of the target formulations defined above. Depending on the experiment, the model therefore predicts either RTL execution cycles directly, an additive residual correction, or a multiplicative correction represented in logarithmic form.

In addition to the primary XGBoost experiments, simpler correction baselines were evaluated as reference points. These included Linear/Ridge regression, K-means-based correction, and kNN-based correction. The purpose was to test how much of the mismatch between the HW model and the RTL could be captured by simpler correction mechanisms before relying on a more flexible tree-ensemble model. The Linear/Ridge models test whether the mismatch can be explained by a simple global correction, while the K-means and kNN baselines test whether grouping or local similarity between samples is sufficient to improve the analytical estimate.

For each simple baseline family, a small predefined hyperparameter sweep was evaluated and the best variant was selected according to pass@5%, matching the primary evaluation metric used for the main experiments. The detailed sweep settings and results are reported in Appendix A.1 and discussed in Chapter 5. The main results in Section 4.3 focus on XGBoost, while the simpler baselines are used as comparative reference points.

The XGBoost hyperparameters were intentionally kept fixed across the main experiments unless otherwise stated. This design choice was made to improve comparability across operation families and units. The goal of the thesis is primarily to evaluate the learning formulation and correction capability rather than to maximise performance through extensive per-family hyperparameter optimisation. Keeping the hyperparameters fixed therefore reduces the risk that observed performance

differences are caused mainly by different tuning effort rather than differences in workload behaviour or dataset characteristics.

The hyperparameters used throughout the main experiments were:

Table 3.1: XGBoost hyperparameters used in the main experiments.

Parameter	Value
`n_estimators`	1500
`max_depth`	6
`learning_rate`	0.03
`subsample`	0.9
`colsample_bytree`	0.9
`objective`	`reg:squarederror`
`random_state`	0

Here, `n_estimators` defines the maximum number of boosting rounds, `max_depth` limits the depth of each tree, and `learning_rate` controls the contribution of each boosting stage. The `subsample` and `colsample_bytree` parameters introduce row and feature subsampling during training, which acts as regularisation and helps reduce overfitting. The squared-error objective was used because the prediction task is formulated as a regression problem.

For each unit-operation group, the dataset was first split into disjoint training and test subsets using an 80/20 split. The training subset was then used to fit the correction models, with a validation subset used during training for early stopping. Early stopping was based on validation-set performance and was used to reduce overfitting and determine the effective number of boosting rounds. The test subset was held out until final evaluation. Splitting was performed at the resolved sample level after feature extraction. Duplicate samples with identical input descriptors and target cycle counts were removed before splitting. Consequently, no RTL target values or test-set samples were used during training.

3.6 Evaluation

Each trained model was evaluated using the same general procedure. For each test sample, the original analytical estimate y_{model} and the corrected prediction $\hat{y}_{\text{corr}}$ are both compared against the RTL reference y_{RTL}. Since the goal is correction rather than standalone prediction, results are reported as a before and after comparison between the analytical model and the learned correction.

Although the models may be trained using different target formulations, all final predictions are reconstructed into execution-cycle space before evaluation. This allows direct-cycle, additive-residual, ratio, and log-ratio models to be compared using the same physical quantity and the same RTL reference.

The evaluation focuses on pass@5%, MAE, median absolute relative error, 95th-percentile absolute relative error, and regression rate. Additional exploratory metrics such as RMSE, R^2, and correlation were also considered during analysis. The primary acceptance metric is pass@5%, which follows the acceptance-style guideline used in the modelling and verification context. It is defined as the fraction of samples whose relative prediction error is at most five percent:

$$\text{pass@5\%} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}\left(\frac{|y_i - \hat{y}_i|}{y_i} \leq 0.05\right).$$

To characterise the distribution of errors, median and 95th-percentile absolute relative errors are also reported. Finally, because the method acts as a correction layer, the evaluation records a regression rate defined with respect to the $\pm 5\%$ acceptance criterion. In this work, a regression is counted only when a sample that was within the acceptance interval before correction is moved outside the interval after correction. Let

$$e_i^{model} = \frac{|y_i^{RTL} - y_i^{model}|}{y_i^{RTL}}, \qquad e_i^{corr} = \frac{|y_i^{RTL} - \hat{y}_i^{corr}|}{y_i^{RTL}}.$$

The reported regression rate is then

$$\text{Reg.} = \frac{|\{i : e_i^{model} \leq 0.05 \wedge e_i^{corr} > 0.05\}|}{|\{i : e_i^{model} \leq 0.05\}|} \cdot 100.$$

Thus, the metric measures the fraction of originally accepted samples that are moved outside the acceptance threshold by the correction.

3.7 Interpretability and Error Analysis

In addition to evaluating prediction accuracy, an interpretability and error analysis was conducted to better understand which features influenced the learned corrections and to identify possible systematic sources of discrepancy between the

analytical model and RTL. This analysis is directly related to the second research question, which asks whether ML-based correction models can help identify factors associated with performance-model mismatch.

The main interpretability method used in this thesis was SHAP analysis, as introduced in Section 2.4. Since several of the trained models predict a correction relative to the analytical baseline, the SHAP values are interpreted as contributions to the predicted correction rather than as direct explanations of RTL execution cycles. A positive SHAP value indicates that a feature increases the predicted correction, while a negative SHAP value indicates that it decreases the predicted correction.

The analysis was performed at both global and local levels. First, global SHAP summaries were used to identify which feature groups contributed most strongly to the learned correction across a full test set. Features were also grouped into broader descriptor categories, such as baseline-cycle descriptors, workload-scale descriptors, source-shape descriptors, and layout-related descriptors. Here, workload-scale descriptors denote derived quantities that summarise the size or issued form of the workload, for example how the tensor work is represented after lowering into block-command-level execution. This grouping was used to make the attribution results interpretable without exposing confidential command-field names.

Second, feature-level SHAP summary plots were inspected to identify whether individual descriptors showed structured attribution patterns. In particular, descriptors with large SHAP spread or clear separation of feature values across the SHAP axis were treated as candidates for further analysis. Such patterns indicate that the model may be using the descriptor to separate workload regimes that require different correction behaviour.

Third, local SHAP explanations were computed for samples with the largest baseline errors. This step was used to compare the features that dominate the correction globally with the features that dominate the worst baseline-miss cases. The purpose was to identify whether the largest model–RTL discrepancies were associated with the same descriptors that were important on average, or with more specific workload regimes.

Finally, SHAP findings were followed by cohort-based error analysis. For selected SHAP-highlighted descriptors, samples were grouped by descriptor value or descriptor regime, and the baseline and corrected errors were compared across those groups. This step was important because SHAP values explain the trained model, but do not by themselves prove that a feature causes the underlying hardware-model discrepancy. Cohort analysis was therefore used to test whether

the descriptors highlighted by SHAP also corresponded to observable error structure in the data.

The interpretability workflow can be summarised as follows: global SHAP was used to identify influential feature groups, feature-level SHAP was used to detect structured descriptor behaviour, local SHAP was used to analyse the worst baseline misses, and cohort analysis was used to verify whether the highlighted descriptors corresponded to systematic error regimes.

In this thesis, the full interpretability workflow was applied as a focused diagnostic case study to operation `Z.A`, rather than exhaustively to every trained model. Operation `Z.A` was selected because it showed both a substantial baseline mismatch and strong correction performance, making it suitable for analysing what information the correction model had learned. The case study is presented in Section 4.6 and discussed further in Section 5.4.

Chapter 4

Results

4.1 Experiment Overview

This chapter evaluates whether XGBoost correction models improve agreement between the NPU hardware model and RTL execution-cycle measurements. Results are reported for single-block prediction and, where available, sequence-level prediction. The primary engineering metric is pass@5%, defined as the fraction of predictions within 5% relative error of RTL. MAE, relative-error percentiles, and regression rate are used as supporting metrics.

4.2 Exploratory Baseline Analysis

4.2.1 Baseline Acceptance Rates

Before applying any learned correction, the analytical HW model baseline was evaluated against the RTL execution-cycle reference for each unit-operation group. Figure 4.1 reports the corresponding pass@5% rate, showing that starting baseline accuracy differs substantially across operations and units: some operation groups are already close to the RTL reference, while others have much lower baseline acceptance rates.

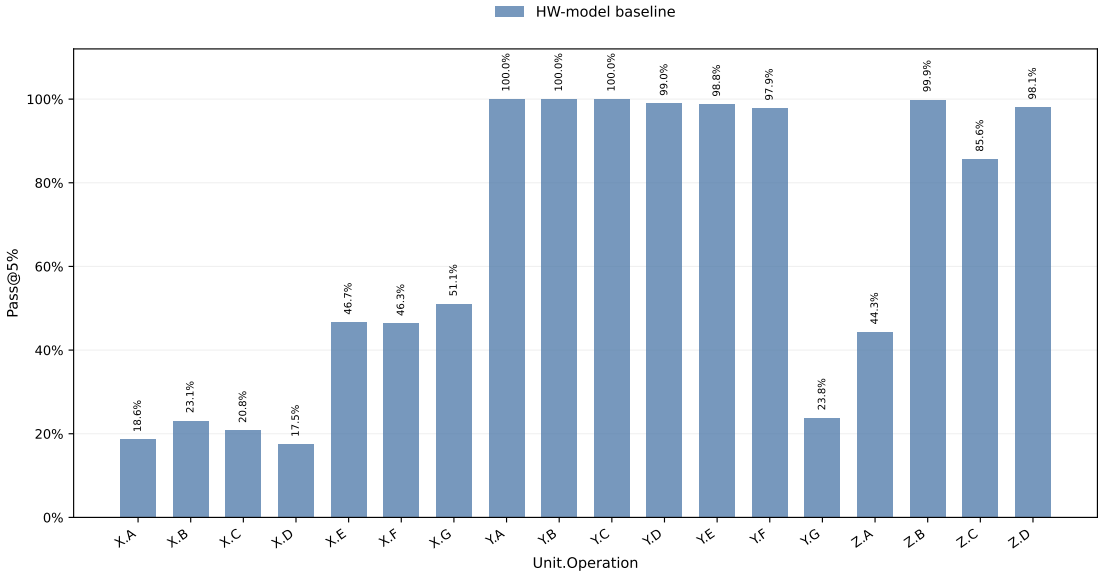

Figure 4.1: Baseline pass@5% rates for the analytical HW model predictions before correction, grouped by unit and operation.

4.2.2 Baseline Error Structure

To better understand the baseline mismatch beyond aggregate pass rates, Figure 4.2 shows representative scatter plots comparing the analytical HW model cycle estimate with the corresponding RTL-observed execution cycles. Each point corresponds to one sample, and the dashed diagonal indicates perfect agreement between the HW model baseline and RTL. The examples were selected to illustrate that the baseline error is not uniform across operation families, but instead appears with different structures depending on the unit and operation.

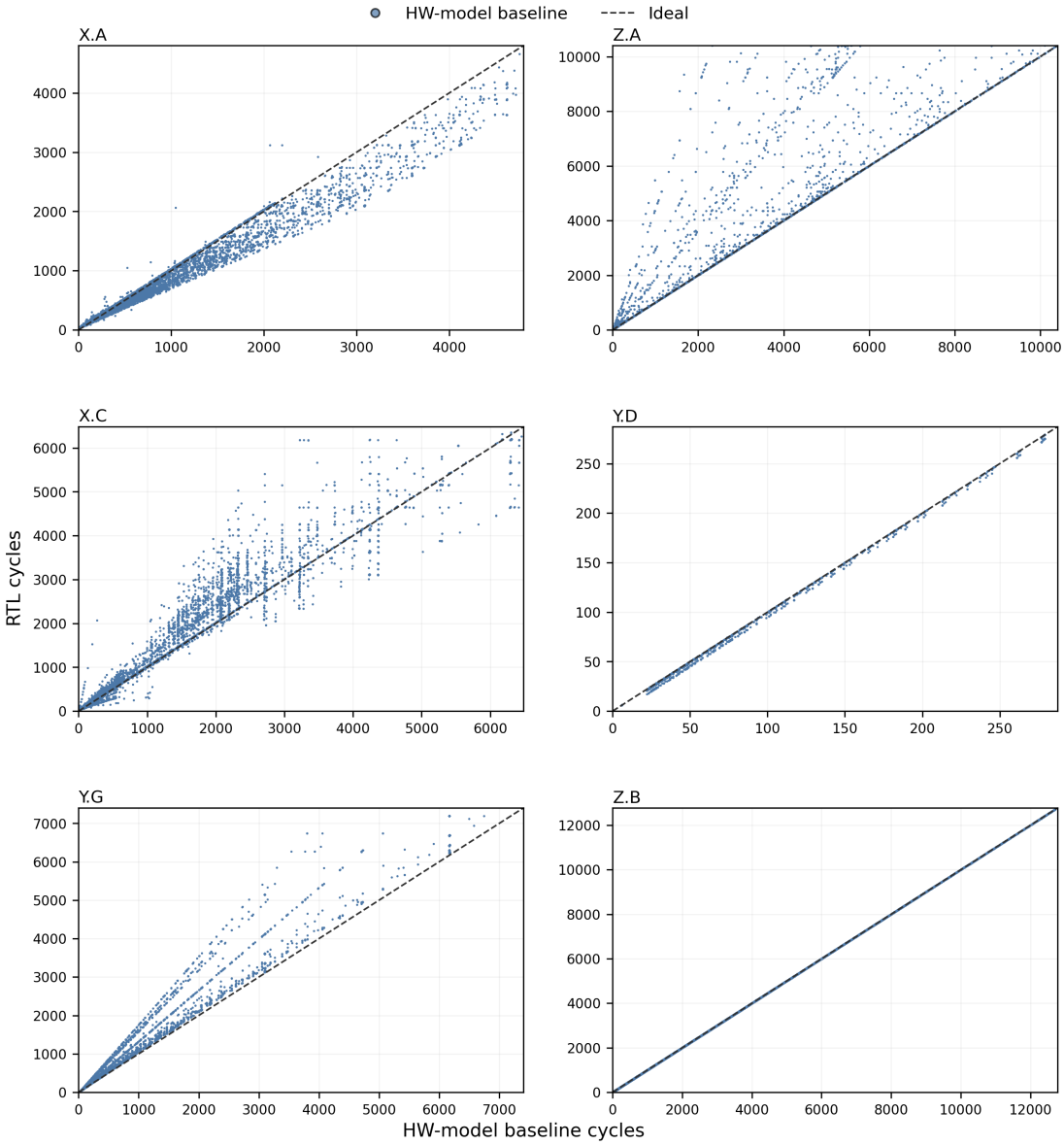

Figure 4.2: Baseline mismatch examples, comparing the analytical hardware-model predictions against RTL-observed execution cycles before correction.

The examples show several distinct baseline error patterns. In X.A, most points lie below the ideal line, which corresponds to systematic overprediction by the HW model. Z.A and Y.G show the opposite pattern, with points lying largely above the ideal line and absolute error increasing with cycle count. This pattern is consistent with a scale-dependent, multiplicative-looking mismatch. X.C also

shows scale-dependent error, but with a larger vertical spread for similar HW-model estimates, meaning that similar baseline predictions can map to a wide range of RTL-observed cycle counts. Y.D is much closer to the ideal line, but still shows a small offset-like deviation, corresponding to an additive-looking mismatch. Finally, Z.B is almost perfectly aligned with the ideal line, showing that the baseline model is already highly accurate for some operation families. Overall, the figure illustrates that baseline mismatch can range from near-perfect agreement to systematic overprediction, systematic underprediction, and high-spread error regimes.

4.2.3 Simple Baseline Correction Models

As an intermediate reference point before the main XGBoost experiments, three simpler correction approaches were evaluated for each unit-operation group: Linear/Ridge correction models, K-means-based correction models, and kNN-based correction models. For each operation and model class, the best configuration was selected according to pass@5%, matching the primary evaluation metric used throughout the chapter. The full results are reported in Appendix A.1.

Overall, the simple correction models provide mixed results. Among the three simple baseline families, kNN gives the strongest results for several of the more difficult cases, especially in Unit X and for Y.G. For Unit X, kNN improves over the HW model baseline for most operations and outperforms both Linear/Ridge and K-means in several cases, but the resulting pass@5% values remain well below the XGBoost-corrected results reported in Section 4.3. kNN also gives a large improvement for Y.G and performs strongly for Z.C. In contrast, Linear/Ridge gives the strongest simple-baseline result for Z.A, where it reaches a pass@5% close to the best corrected result. Overall, the simple baselines recover part of the HW-model mismatch in selected cases, but remain substantially weaker than XGBoost for the difficult Unit X operations and Y.G.

4.3 Single-Block Correction Results

This section reports the main single-block correction results for the evaluated NPU units. For each unit-operation group, the HW model baseline is compared with the best XGBoost correction model selected for that group. The selected target formulation is shown in the Target column of the result tables, and the full target-formulation sweeps are reported in Appendix A.2.

All predictions are evaluated in execution-cycle space against the same RTL reference, regardless of the target formulation used during training. The primary metric is pass@5%. Median error and p95 error denote the median and 95th-percentile absolute relative errors, respectively, and are reported in percent. MAE is reported in execution cycles. The P→F column reports the pass-to-fail regression rate in percent: among samples that pass the 5% threshold before correction, it gives the fraction that fail the threshold after correction. For compactness, the tables use HW for the HW model baseline, XGB for XGBoost, and Dir, Res, Rat, and Log for the direct, residual, ratio, and log-ratio target formulations, respectively.

The following subsections present the results separately for Unit X, Unit Y, and Unit Z. Operation labels are anonymised where required. For each unit, the table reports the numerical metrics, the first figure compares pass@5% before and after correction, and the second figure summarises the signed relative-error distribution for each operation using 5th–95th percentile intervals, 25th–75th percentile intervals, and median markers.

4.3.1 Unit X

The Unit X single-block experiments cover seven anonymised operations, denoted X.A–X.G. Table 4.1 summarises the HW model baseline and the selected XGBoost correction result for each operation. For all seven Unit X operations, the selected target formulation is log-ratio.

XGBoost improves pass@5% for all Unit X operations. The HW model baseline ranges from 17.5% to 51.1% pass@5%, while the corrected models reach between 90.7% and 98.2%. The corrected median relative error is below 1% for every operation, showing that the typical prediction error is strongly reduced.

Table 4.1: Unit X single-block correction results for anonymised operations X.A–X.G.

OP	Samples	Model	Target	pass@5%	Med err	p95 err	MAE	P→F
X.A	51,362	HW	–	18.6	17.86	60.34	59.95	–
		XGB	Log	98.2	0.46	3.46	3.17	0.90
X.B	49,818	HW	–	23.1	15.20	47.92	80.66	–
		XGB	Log	96.7	0.52	3.78	16.43	3.38
X.C	49,940	HW	–	20.8	18.38	51.06	76.49	–
		XGB	Log	93.4	0.55	6.16	23.55	7.14
X.D	48,367	HW	–	17.5	14.60	48.98	269.93	–
		XGB	Log	98.2	0.29	2.63	20.54	3.04
X.E	49,908	HW	–	46.7	6.09	45.83	89.98	–
		XGB	Log	92.1	0.58	6.61	43.62	2.54
X.F	49,913	HW	–	46.3	6.13	45.83	93.06	–
		XGB	Log	92.3	0.58	6.49	45.27	2.52
X.G	49,901	HW	–	51.1	4.70	45.57	110.77	–
		XGB	Log	90.7	0.60	7.79	79.98	3.77

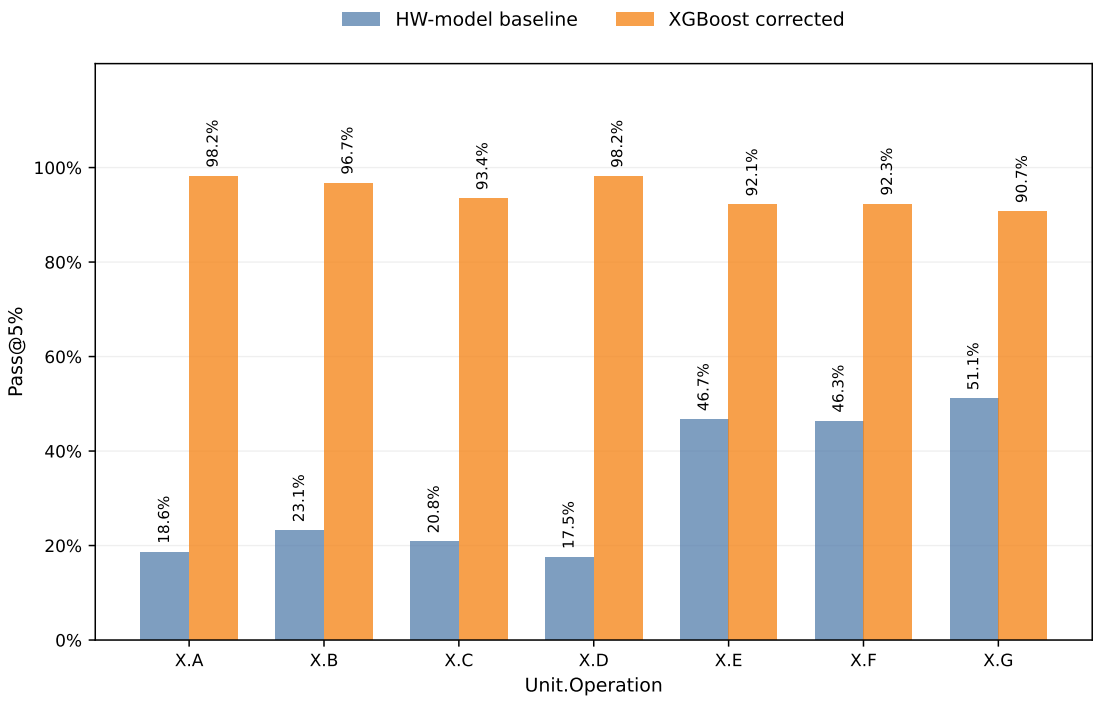

Figure 4.3: Pass@5% comparison between the HW model baseline and the XGBoost-corrected predictions for Unit X single-block operations.

The remaining errors are more visible in the tail metrics. X.A and X.D achieve the lowest corrected p95 errors, while X.E–X.G retain the largest corrected p95 errors. Under the pass-to-fail regression definition, the largest regression rate is observed for X.C, where 7.14% of samples that originally passed the 5% threshold fail it after correction. For the other Unit X operations, the pass-to-fail regression rate remains between 0.90% and 3.77%. Figure 4.4 shows the same pattern visually: after correction, the signed-error intervals contract strongly towards zero and the corrected medians lie close to the ideal error level.

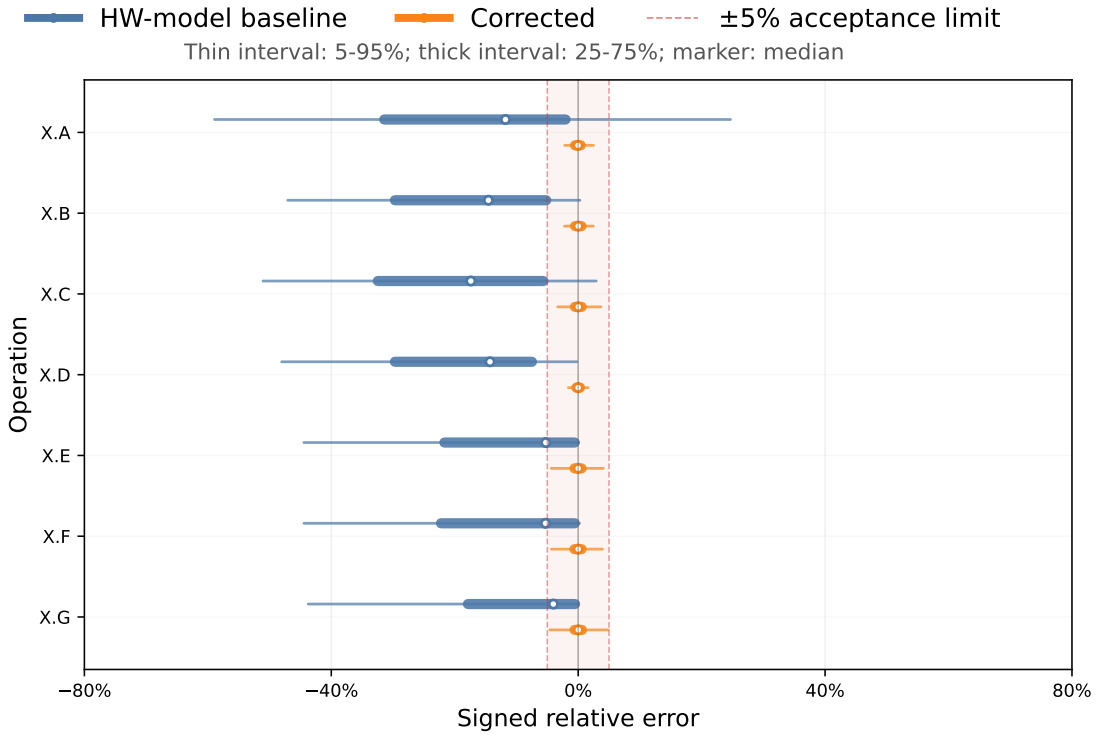

Figure 4.4: Signed relative-error summary for Unit X single-block operations before and after correction. The figure highlights the contraction of the corrected error distributions towards zero across all Unit X operations. Thin intervals show the 5th–95th percentile range, thick intervals show the 25th–75th percentile range, and markers show the median. The shaded region indicates the ±5% acceptance interval.

4.3.2 Unit Y

The Unit Y single-block experiments cover seven anonymised operations, denoted Y.A–Y.G. Table 4.2 reports the HW model baseline and selected XGBoost correction results for each operation.

Table 4.2: Unit Y single-block correction results for anonymised operations Y.A–Y.G.

OP	Samples	Model	Target	pass@5%	Med err	p95 err	MAE	P→F
Y.A	29,893	HW	–	100.0	0.00	0.05	0.12	–
		XGB	Res	100.0	0.00	0.00	0.00	0.01
Y.B	29,655	HW	–	100.0	0.00	0.00	0.00	–
		XGB	Log	100.0	0.00	0.00	0.00	0.00
Y.C	29,744	HW	–	100.0	0.00	0.67	26.56	–
		XGB	Log	100.0	0.00	0.03	0.26	0.00
Y.D	29,669	HW	–	99.0	1.10	2.56	2.17	–
		XGB	Res	100.0	0.00	0.00	0.00	0.00
Y.E	29,699	HW	–	98.8	0.29	2.05	2.74	–
		XGB	Res	100.0	0.00	0.00	0.00	0.02
Y.F	29,808	HW	–	97.9	0.09	1.18	11.24	–
		XGB	Log	98.5	0.06	1.11	2.35	0.73
Y.G	29,449	HW	–	23.8	16.29	66.20	529.76	–
		XGB	Log	98.9	0.04	1.36	3.97	0.30

Unit Y shows a different pattern from Unit X because some operations, denoted Y.A–Y.C, are already at 100.0% pass@5%, and Y.D–Y.F are also close to the acceptance threshold before the correction. For these operations, XGBoost mainly reduces the remaining error magnitude rather than producing a large pass@5% gain.

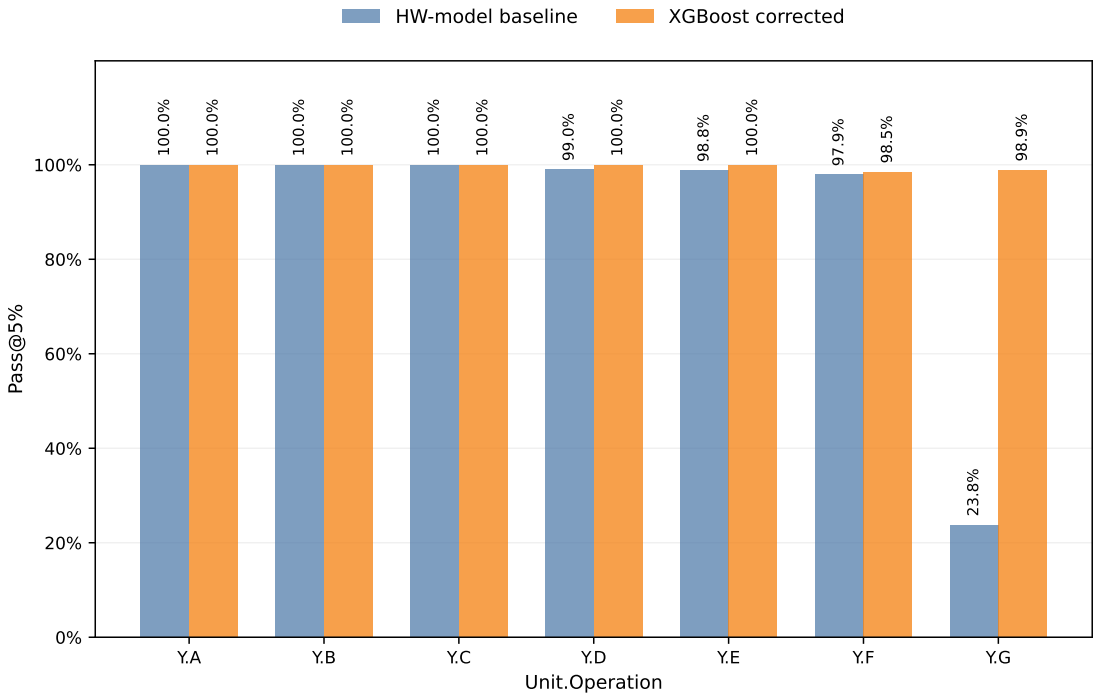

Figure 4.5: Pass@5% comparison between the HW model baseline and the XGBoost-corrected predictions for Unit Y single-block operations.

The main exception is Y.G, where the HW model baseline achieves only 23.8% pass@5%. The selected XGBoost correction increases this to 98.9%, while reducing the MAE from 529.76 cycles to 3.97 cycles, the median error from 16.29% to 0.04% and the p95 error from 66.20% to 1.36%. Figure 4.6 shows that the large negative baseline errors for Y.G are largely removed after correction.

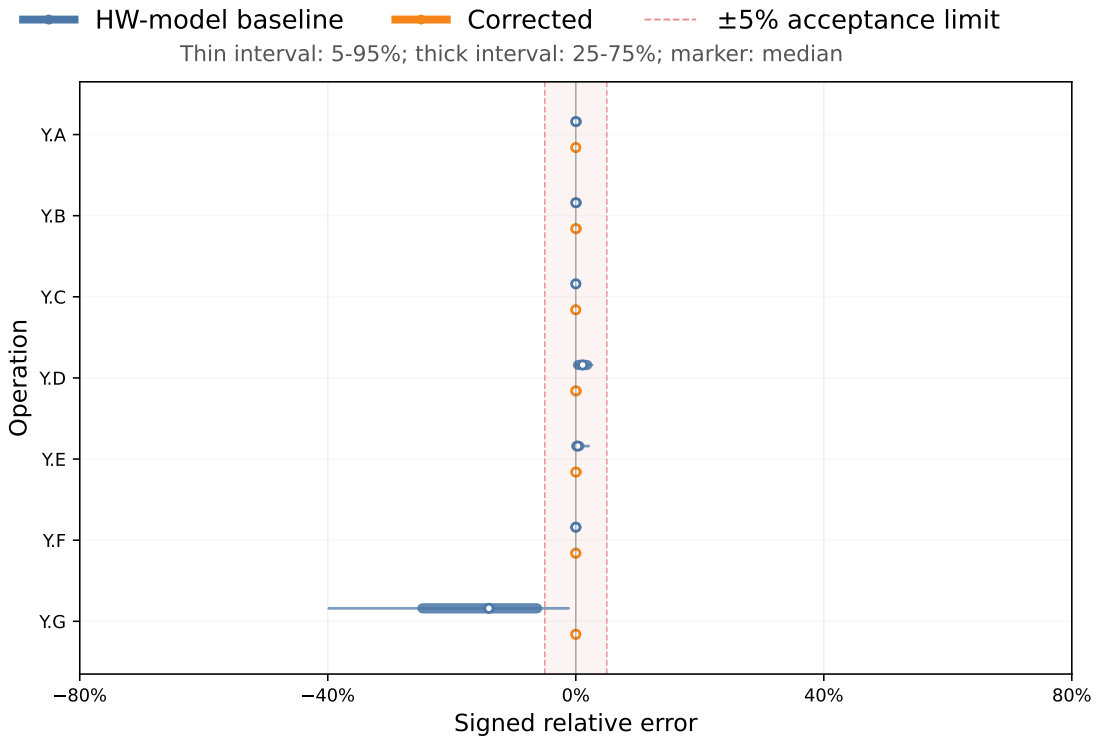

Figure 4.6: Signed relative-error summary for Unit Y single-block operations before and after correction. The figure highlights that most Unit Y operations are already close to zero before correction, while Y.G shows the clearest visible contraction after correction. Thin intervals show the 5th–95th percentile range, thick intervals show the 25th–75th percentile range, and markers show the median. The shaded region indicates the ±5% acceptance interval.

4.3.3 Unit Z

The Unit Z single-block experiments cover four anonymised operations, denoted Z.A–Z.D. Table 4.3 reports the HW model baseline and selected XGBoost correction for each operation.

Table 4.3: Unit Z single-block correction results for anonymised operations Z.A–Z.D.

OP	Samples	Model	Target	pass@5%	Med err	p95 err	MAE	P→F
Z.A	35,039	HW	–	44.3	11.96	70.85	1411.72	–
		XGB	Dir	99.0	0.50	2.57	44.54	1.30
Z.B	38,401	HW	–	99.9	0.00	0.15	0.33	–
		XGB	Res	100.0	0.00	0.00	0.00	0.00
Z.C	39,951	HW	–	85.6	0.00	14.00	18.13	–
		XGB	Log	99.9	0.00	0.12	11.36	0.01
Z.D	59,135	HW	–	98.1	0.08	1.77	2.17	–
		XGB	Res	100.0	0.00	0.01	0.01	0.00

XGBoost also improves pass@5% for all Unit Z operations. The largest gain is observed for Z.A, where pass@5% increases from 44.3% to 99.0%. Z.C also improves from 85.6% to 99.9%, while Z.B and Z.D are already close to the RTL reference before correction and reach 100.0% pass@5% after correction.

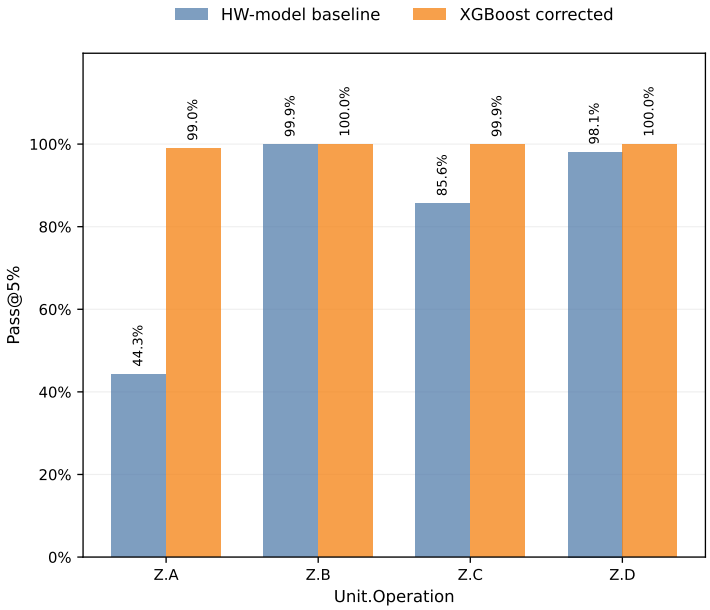

Figure 4.7: Pass@5% comparison between the HW model baseline and the XGBoost-corrected predictions for Unit Z single-block operations.

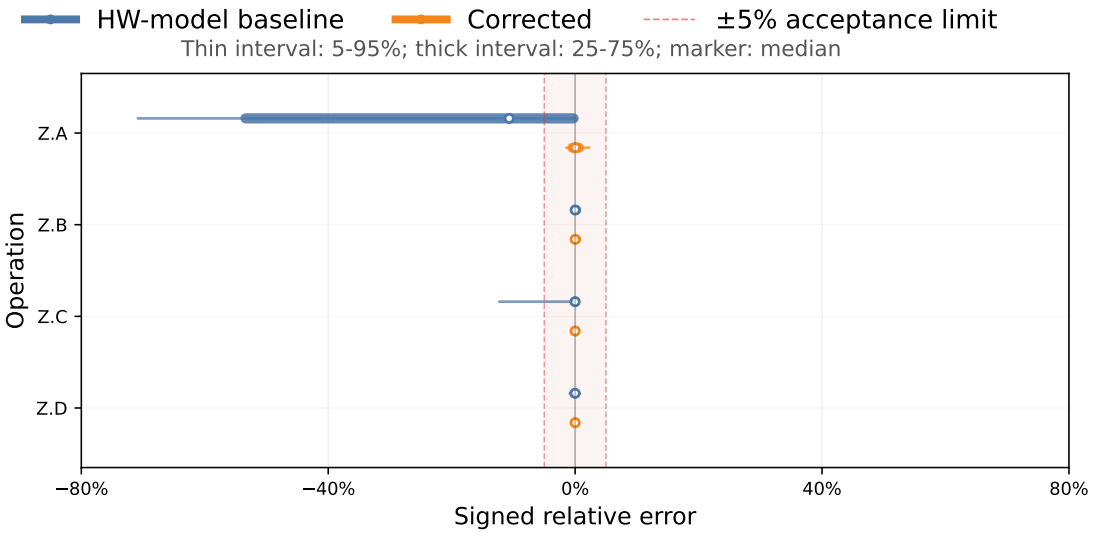

Figure 4.8: Signed relative-error summary for Unit Z single-block operations before and after correction. The figure highlights the large baseline spread for Z.A and the much tighter corrected error intervals for all Unit Z operations. Thin intervals show the 5th–95th percentile range, thick intervals show the 25th–75th percentile range, and markers show the median. The shaded region indicates the ±5% acceptance interval.

The tail-error metrics show that the corrected predictions are tightly concentrated for all Unit Z operations. Z.A retains the largest corrected p95 error among the Unit Z operations, but this is still reduced substantially from 70.85% to 2.57%. Figure 4.8 confirms that the corrected signed-error distributions are concentrated near zero for all four operations.

4.4 Sequence-Level Results

This section reports the sequence-level correction experiments. Unlike the single-block setting, each sample corresponds to the measured execution-cycle count of a full sequence execution. The same evaluation metrics as in Section 4.3 are used: pass@5%, median absolute relative error, p95 absolute relative error, MAE, and P→F regression rate.

As described in Section 3.3, the deployable sequence models use predicted standalone costs from the corresponding single-block correction models as sequence-

context features. Where available, a diagnostic RTL-context variant is also reported. This variant replaces the predicted standalone costs with standalone RTL measurements and is therefore not deployable, but provides an oracle-context comparison. The distinction between the deployable and diagnostic settings is illustrated in Figure 3.2.

4.4.1 Unit X

For Unit X, sequence-level correction was evaluated in two settings: sequences of two independent block commands and sequences where multiple block commands jointly implement one logical operation. Table 4.4 compares the HW model baseline, the deployable predicted-context XGBoost model, and the diagnostic RTL-context XGBoost model for each setting.

Table 4.4: Unit X sequence-level correction results. "2-blk" denotes two independent block commands, and "Joint" denotes sequences where multiple block commands jointly implement one logical operation.

Seq.	Model	Samples	pass@5%	Med err	p95 err	MAE	P→F
2-blk	HW	89,587	42.2	7.03	42.37	135.65	–
	XGB (pred)	89,587	86.4	1.29	8.90	74.93	3.91
	XGB (RTL)	89,587	93.8	0.91	5.42	42.72	0.92
Joint	HW	49,865	39.3	11.52	96.15	88.24	–
	XGB (pred)	49,865	92.3	0.31	6.50	50.81	4.27
	XGB (RTL)	49,865	93.7	0.29	5.70	44.75	3.53

The Unit X sequence-level results show that the correction approach also extends beyond isolated block commands. For two independent block-command sequences, the deployable predicted-context model increases pass@5% from 42.2% to 86.4%, while reducing the median relative error from 7.03% to 1.29% and the p95 error from 42.37% to 8.90%. The diagnostic RTL-context variant improves further to 93.8% pass@5%, indicating that more accurate standalone block-cost context can still provide additional sequence-level benefit.

For the joint-operation sequence class, the improvement is even stronger. The HW model baseline reaches only 39.3% pass@5%, while the deployable XGBoost model increases this to 92.3% and reduces the median relative error from 11.52% to 0.31%. The RTL-context variant gives a smaller additional gain, reaching 93.7% pass@5%

with a p95 error of 5.70%. This suggests that the sequence model captures most of the correction for this sequence class already in the deployable setting, although oracle standalone costs still reduce the tail error and MAE.

4.4.2 Unit Y

For Unit Y, sequence-level correction was evaluated for sequences of two independent block commands. Table 4.5 compares the HW model baseline with the deployable predicted-context XGBoost sequence model.

Table 4.5: Unit Y sequence-level correction results. "2-blk" denotes two independent block commands.

Seq.	Model	Samples	pass@5%	Med err	p95 err	MAE	P→F
2-blk	HW	26,965	71.62	0.30	19.87	151.08	–
	XGB (pred)	26,965	98.33	0.09	2.16	8.95	0.18

For Unit Y, the HW model baseline in the two-block sequence setting reaches 71.62% pass@5%. The predicted-context XGBoost model improves this to 98.33%, while reducing the p95 relative error from 19.87% to 2.16% and the MAE from 151.08 to 8.95 cycles. The pass-to-fail regression rate is only 0.18%, meaning that very few samples that originally passed the 5% threshold are pushed outside the tolerance band after correction. Thus, in this evaluated sequence setting, the correction provides both a large aggregate improvement and a low acceptance-level regression risk.

4.5 MATMUL Case Study

This section returns to the MATMUL example introduced in Section 2.1.1. The goal is to compare the HW model and XGBoost-corrected predictions for the two concrete, source-equivalent but execution-distinct representations described there: a single-block execution and a reduction-sequence execution. Both originate from the same BF16 MATMUL operation, but differ in how the reduction dimension is issued to the NPU.

For confidentiality reasons, the absolute RTL cycle counts and model predictions are not reported. Instead, Table 4.6 reports the signed relative error of the HW

model estimate and the XGBoost-corrected prediction with respect to the RTL reference. In the single-block case, the HW model underpredicts RTL by 8.25%, while the XGBoost correction reduces the error to 0.04%. In the reduction-sequence case, the HW model underpredicts RTL by 2.24%, while the XGBoost correction reduces the error to 0.57%.

The case study therefore shows that the corrected predictions remain close to RTL for both execution forms, even though the same source-level workload is represented by different block-command schedules.

Table 4.6: MATMUL case study comparing source-equivalent single-block and reduction-sequence executions. Absolute cycle counts are omitted for confidentiality; values report signed relative error with respect to RTL execution cycles.

Execution form	K split	HW model error	XGBoost error
Single block	7	-8.25%	$+0.04\%$
Reduction sequence	$1+1+4+1$	-2.24%	-0.57%

4.6 Explainability and Error Analysis

To examine whether the learned correction models also provide insight into the sources of discrepancy, a detailed SHAP analysis was conducted for operation Z.A. Although the main result table reports the direct-cycle model as the selected Z.A model under the pass@5% criterion, the SHAP analysis in this section uses the log-ratio Z.A model. This choice was made because the aim of the case study is to analyse the learned discrepancy relative to the analytical baseline. A direct-cycle model explains variation in predicted RTL cycle counts, whereas the log-ratio model explains variation in the multiplicative correction applied to the HW model estimate. This makes the analysis more directly aligned with the diagnostic question of where the analytical model and RTL differ.

Figure 4.9 shows the grouped global SHAP analysis for operation `Z.A`. The largest contribution comes from baseline-cycle descriptors, which is expected because the correction model is trained on top of the analytical baseline and uses baseline-derived cycle information as part of its input representation. However, workload-related descriptors such as workload-scale, source-shape, and layout-related descriptors also contribute. This means that the learned log-ratio correction depends not only on baseline cycle magnitude, but also on descriptors of the executed workload structure.

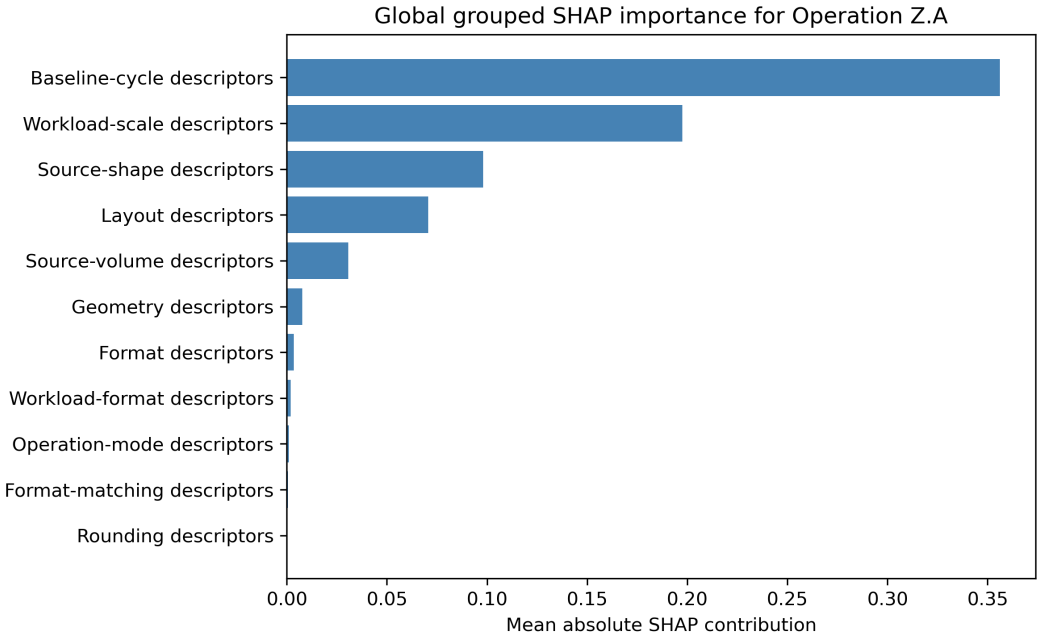

Figure 4.9: Global grouped SHAP attribution summary for operation `Z.A`. The figure shows the relative contribution of anonymised descriptor groups to the learned log-ratio correction.

The feature-level SHAP summary in Figure 4.10 provides a more detailed view of the same attribution pattern. Several of the highest ranked descriptors have contributions on both sides of zero, showing that their effect is not a single uniform adjustment across all samples.

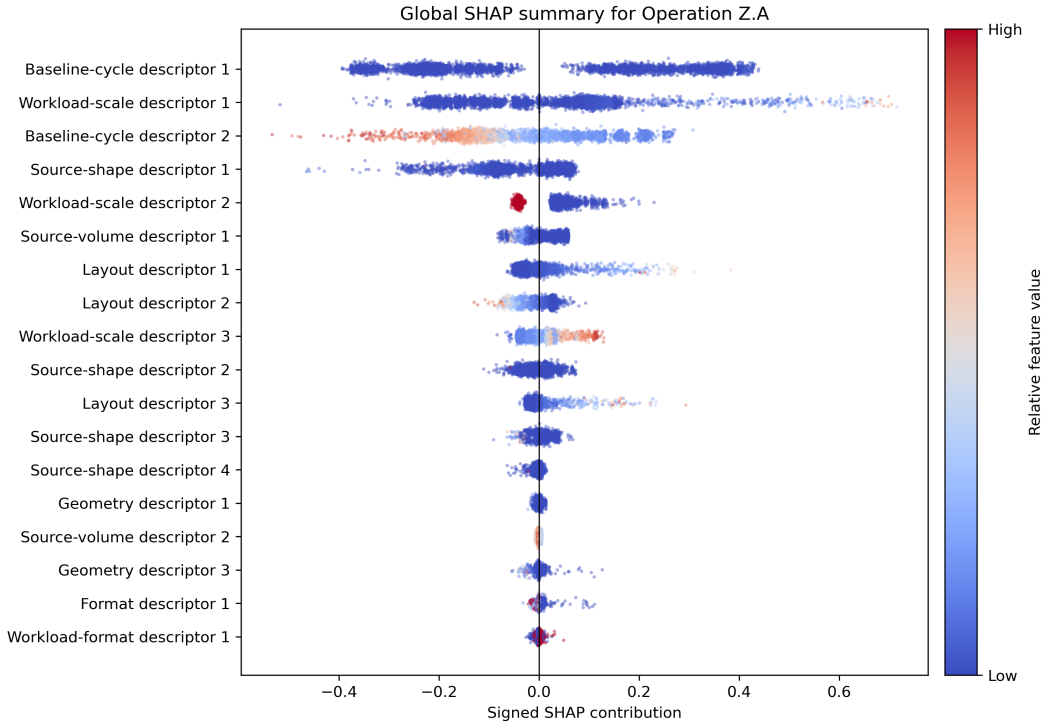

Figure 4.10: SHAP beeswarm plot for operation `Z.A`. The figure shows how the most influential features affect the predicted log-ratio correction across individual samples.

A local SHAP analysis was also conducted for the 20 samples with the largest baseline errors. In contrast to the global attribution summary, where baseline-cycle descriptors had the largest overall contribution, the local explanations for these worst baseline misses were dominated by workload-scale and layout-related descriptors.

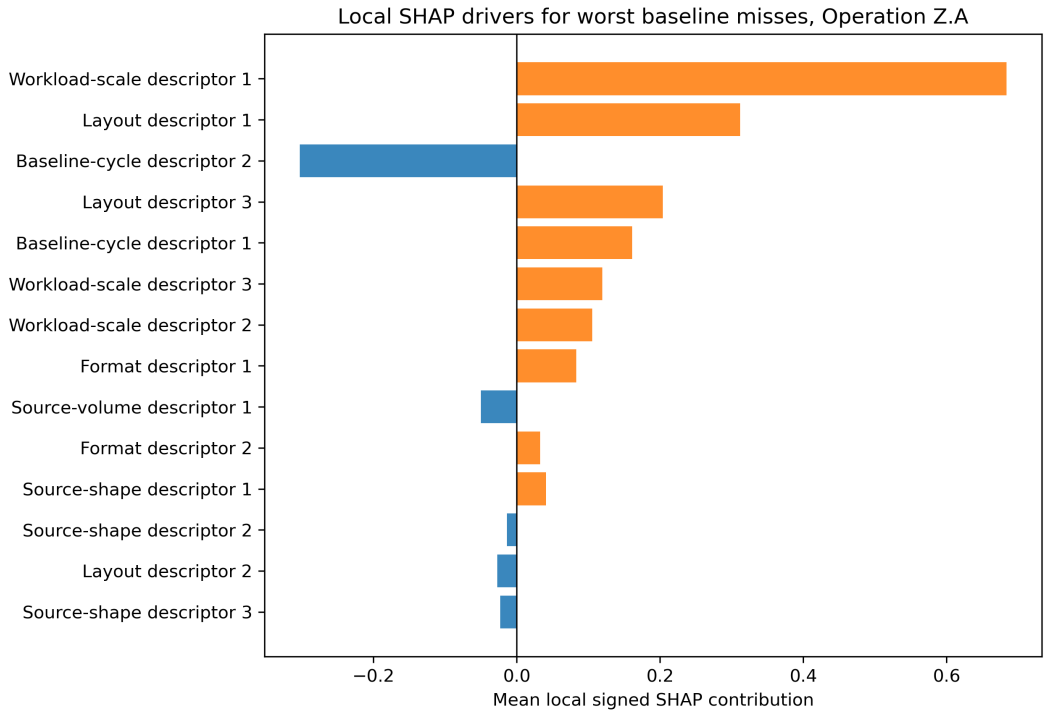

Figure 4.11: Local SHAP attribution summary for the 20 samples with the largest baseline errors for operation `Z.A`. The figure highlights which grouped features contribute most strongly, and in which direction, for the largest analytical-model misses.

To follow up on the SHAP results, cohort splits were inspected for the most influential attributes. Among these, two workload-scale descriptors showed the clearest connection between attribution behaviour and baseline-error structure. Workload-scale descriptor 1 had the widest SHAP spread in the beeswarm plot and was also the dominant positive contributor in the local analysis of the 20 largest baseline misses. Workload-scale descriptor 2 showed a clear separation in the beeswarm plot, with different descriptor values appearing on opposite sides of the SHAP axis.

These descriptors were therefore important not only because they received high SHAP importance, but also because they separated samples into regimes with different baseline-error behaviour. This suggests that the correction model had learned to distinguish between cases where the analytical model already matched RTL well and cases where the analytical estimate was systematically too low. The following figures test this interpretation by grouping samples according to the two workload-scale descriptors and comparing the baseline and corrected relative

errors. The diagnostic interpretation of these regimes is discussed in Section 5.4.

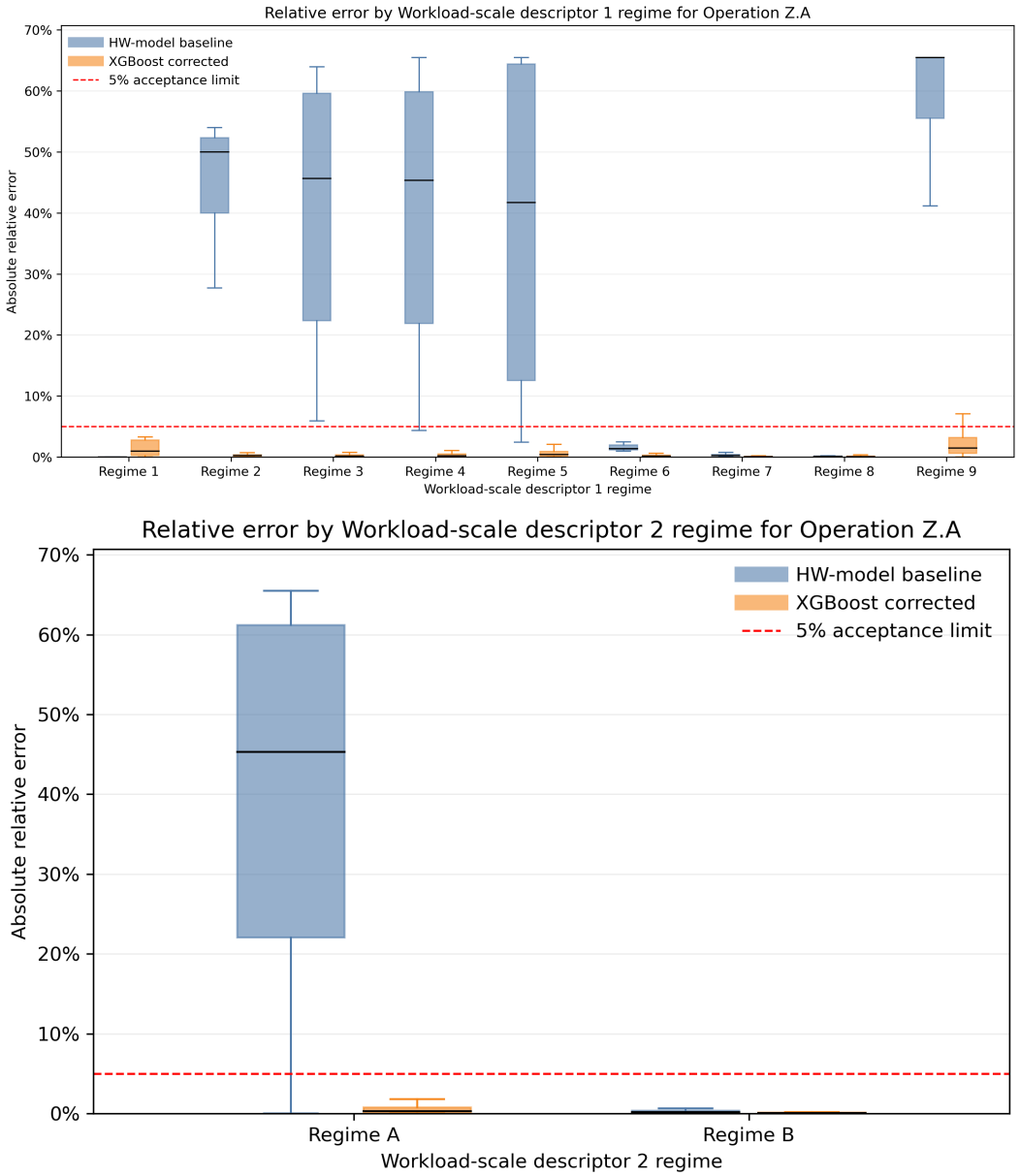

Figure 4.12: Absolute relative error for operation `Z.A` grouped by two workload-scale descriptors. The plots show that the analytical HW model baseline error is concentrated in specific descriptor regimes, while the corrected predictions remain close to RTL across regimes.

Chapter 5

Discussion

5.1 Interpretation of the Correction Results

The single-block results show that the learned correction models can substantially improve the agreement between the analytical HW model baseline and RTL-observed execution cycles. This supports the main premise of the thesis: the discrepancy between the fast analytical model and RTL is not merely random noise, but contains structure that can be learned from early-available command-level features and baseline cycle estimates. The correction models are therefore best interpreted not as replacements for the existing HW model, but as data-driven calibration layers that preserve the analytical estimate as the starting point and learn the remaining mismatch. For the evaluated single-block setting, the combination of the analytical HW model and the learned correction layer can therefore also be viewed as a lightweight alternative path towards RTL-aligned timing estimates, complementing the more engineering-intensive route of building a more detailed cycle-approximate model.

The strongest evidence for this appears in Unit X, where all seven operation families start from relatively low baseline pass@5% values. The HW model baseline ranges from 17.5% to 51.1% pass@5%, while the corrected XGBoost models reach between 90.7% and 98.2%. The available single-block features therefore capture much of the mismatch for these operations. The fact that the corrected median error is below 1% for every Unit X operation further shows that the typical prediction error is strongly reduced, even though some tail errors remain.

The Unit X results are also where the use of a flexible nonlinear model is most clearly justified. Among the simple correction baselines, kNN is the strongest

alternative for Unit X. It outperforms Linear/Ridge and K-means for all Unit X operations and improves over the original HW model baseline for six of the seven operations. This suggests that local similarity in feature space captures part of the model–RTL mismatch. However, kNN still remains far below the XGBoost results: the best Unit X kNN result is 49.6% pass@5%, while the selected XGBoost models reach between 90.7% and 98.2%. For Unit X, local neighbour-based interpolation is therefore insufficient; the remaining improvement appears to require conditional feature interactions and threshold-like regimes.

Unit Y and Unit Z show a more mixed pattern because several operations already have high baseline accuracy before correction. For operations such as Y.A–Y.C, Z.B, and Z.D, the HW model baseline is already close to RTL, so there is limited room for improvement under the pass@5% metric. In these cases, the corrected model mainly reduces residual error magnitude rather than producing a large acceptance-rate gain. However, the difficult cases in these units show that the correction approach is not limited to Unit X. In particular, Y.G improves from 23.8% to 98.9% pass@5%, and Z.A improves from 44.3% to 99.0%. The approach is most useful when the analytical baseline contains a systematic mismatch, while already accurate operations naturally show smaller gains. Y.G also illustrates the intermediate value of kNN. The kNN baseline raises pass@5% from 23.8% to 77.6%, showing that local neighbourhood information captures a substantial part of the discrepancy. Nevertheless, XGBoost reaches 98.9%, indicating that local interpolation alone is insufficient for the full correction.

At the same time, Z.A shows that the most flexible model is not always the most appropriate correction mechanism. The best Linear/Ridge correction for Z.A reaches 99.4% pass@5%, slightly above the selected XGBoost model at 99.0%, while kNN reaches 76.6%. For Z.A, the dominant mismatch appears closer to a smooth global bias than to a highly local nonlinear effect. This is an important distinction for practical use: XGBoost is valuable when the residual depends on nonlinear feature interactions, but a simpler correction may be preferable when the mismatch is dominated by a smooth global bias. Model choice should therefore be guided by the observed error structure, not only by the overall strength of the learning algorithm.

The choice of target formulation and evaluation metric also affects how these improvements should be interpreted. This trade-off is discussed separately in Section 5.2.

Finally, the results show that correction is not uniformly beneficial for every individual sample. This is visible mainly in the remaining p95 errors and, to a lesser extent, in the pass-to-fail regression-rate metric. Some operations, especially X.E–

X.G, retain higher corrected tail errors than the other Unit X operations, while X.C shows the largest pass-to-fail regression rate among the Unit X single-block results. However, the updated regression metric shows that acceptance-level regressions are generally limited: most samples that pass the 5% threshold before correction remain within tolerance after correction. This distinction is important because a stricter metric based on any increase in absolute error would count many small numerical perturbations as regressions, even when both the original and corrected predictions remain well inside the acceptance band. In practice, learned correction is most compelling for operation families with clear baseline mismatch. For operations that are already near-perfect, it should be applied more cautiously, especially if per-sample regressions are unacceptable.

5.2 Target and Metric Trade-offs

The target-formulation sweep shows that the way the correction task is defined has a substantial effect on the learned model. Although all predictions are ultimately reconstructed into execution-cycle space, the model is trained in different target spaces depending on whether it predicts RTL cycles directly, an additive residual, a ratio, or a logarithmic ratio. These formulations are not equivalent. A direct-cycle target emphasises absolute cycle error, an additive residual target emphasises missing or excess cycles relative to the analytical estimate, while ratio and log-ratio targets emphasise multiplicative discrepancy between the HW model and RTL.

This distinction matters because the baseline mismatch is not the same for all operation families. Several difficult cases show scale-dependent behaviour, where the absolute error grows with the cycle count. In such cases, the mismatch is better interpreted as a relative or multiplicative error than as a fixed additive offset. This explains why the log-ratio formulation is particularly effective for Unit X: all Unit X single-block operations selected log-ratio as the best XGBoost target, and the ratio-based targets consistently outperform the direct and additive residual targets under the primary pass@5% metric and generally improve the relative-error distribution as observed in the Appendix sweeps. The log-ratio target is well aligned with this type of error because it asks the model to learn a multiplicative correction factor in log space, rather than forcing it to explain the same relative discrepancy as a larger and larger absolute residual at higher cycle counts.

The same pattern appears in some of the difficult operations outside Unit X. Operation `Y.G`, for example, has a strongly inaccurate baseline and selects the log-ratio target. The corrected model improves pass@5% from 23.8% to 98.9%,

while reducing the median relative error from 16.29% to 0.04% and the p95 error from 66.20% to 1.36%. This is a case where the multiplicative target formulation is well matched to the dominant evaluation objective, because the main problem is not a small fixed cycle offset but a large relative mismatch between the analytical estimate and RTL.

However, the results also show that no single target formulation is universally best. Several operations that are already close to RTL, or whose remaining discrepancy is closer to an offset-like error, select residual targets instead. Examples include `Y.A`, `Y.D`, `Y.E`, `Z.B`, and `Z.D`. In these cases, the analytical model is already highly correlated with RTL, and the useful correction is often small. A residual target can therefore be sufficient, and sometimes preferable, because the model only needs to learn a small additive adjustment rather than a scale factor.

This also means that target selection should be interpreted cautiously for already well-correlated operations. When these models already reach 100% pass@5%, the primary metric becomes saturated and no longer separates the target formulations clearly. However, improvements may still be visible in the secondary metrics. For example, `Y.C` already has 100% baseline pass@5%, but XGBoost reduces the p95 relative error from 0.67% to 0.03% and the MAE from 26.56 to 0.26 cycles. In such cases, p95 error, MAE, and pass-to-fail regression provide a more informative view of the remaining differences between target formulations than pass@5% alone.

Operation Z.A also shows why target selection cannot be reduced to a single universal rule. The baseline error for Z.A is severe, but the selected XGBoost target is direct-cycle prediction rather than log-ratio. At the same time, the target sweep shows that the log-ratio formulation gives lower median and p95 relative error than the direct target, while the direct target gives the highest pass@5% and lower MAE than the ratio-based alternatives. The residual target also gives a slightly lower MAE than the direct target, but its much lower pass@5% shows that minimising average absolute cycle deviation alone is not sufficient for the acceptance objective. This illustrates that a target that gives the best acceptance rate is not necessarily the same target that gives the lowest median error, the lowest tail error, or the lowest absolute cycle error.

The target should therefore be chosen according to the intended use of the corrected model: pass@5% if acceptance within a relative tolerance band is most important, p95 error if tail behaviour is more important, or MAE if absolute cycle deviation is the main concern.

This analysis suggests that learned correction should not necessarily be applied uniformly to all operation families. For operations with clear systematic mismatch,

such as `Y.G`, `Z.A`, and the Unit X operations, the correction provides large and practically meaningful improvements. For operations where the analytical baseline is already nearly perfect, the main risk is not aggregate accuracy loss or pass-to-fail regression, but unnecessary per-sample perturbation that may be undesirable even when predictions remain within tolerance. A practical deployment could therefore use a conservative policy: apply learned correction where the baseline is known to have systematic error, but bypass or gate the correction for operation regimes where the analytical model is already within tolerance.

Overall, the sweep results lead to two main conclusions. First, multiplicative targets, especially log-ratio, are well suited when the baseline error behaves like a relative scale error, which is common in the more difficult operation families. Second, the target formulation should be selected together with the evaluation objective, whether that objective is maximising pass@5%, minimising p95 relative error, minimising MAE, or reducing the risk of per-sample regressions.

5.3 Sequence Modelling and Features

The sequence-level results extend the correction approach beyond isolated block-command prediction. This is important because the runtime of a sequence is not necessarily equivalent to the sum of independently measured block runtimes. As discussed in the methodology, sequence execution may include overlap, ordering effects, shared state, and other unit-level interactions. The fact that the learned models still improve sequence-level agreement with RTL therefore shows that at least part of this additional sequence behaviour is predictable from compact execution-level features.

A notable aspect of the sequence experiments is that the feature representations are deliberately simple. For the short two-command sequence, the deployable model uses only a small amount of the sequence context: the operation associated with each block command and the predicted standalone cost of each constituent block. It does not require a full per-block descriptor profile. Despite this compact representation, the Unit X model improves pass@5% from 42.2% to 86.4%, and the Unit Y model improves from 71.62% to 98.33%. For these two-block sequences, much of the discrepancy can be explained by the operation pair and the estimated standalone costs.

The comparison between predicted-context and RTL-context variants gives additional insight into where the remaining error comes from. In the Unit X independent-block setting, replacing predicted standalone costs with standalone RTL measure-

ments increases pass@5% from 86.4% to 93.8%. Part of the remaining sequence error is therefore inherited from imperfect standalone block-cost estimates. In other words, the sequence model can benefit from more accurate constituent-block context. However, the deployable predicted-context model already provides a substantial improvement over the HW model baseline, which means that the learned single-block predictions are sufficiently informative to support useful sequence-level correction without requiring standalone RTL measurements at prediction time.

The reduction-sequence setting is different because the sequence represents a single logical operation distributed across multiple block commands, rather than simply two independent commands placed next to each other. For this case, the feature representation is somewhat richer, using aggregate statistics over standalone cost estimates and selected shape-related descriptors, together with the number of blocks in the reduction. However, it is still a compact representation: it summarises the sequence instead of exposing the full per-block command profile. The strong reduction result therefore suggests that the important correction signal is captured by coarse sequence-level summaries such as the scale of the constituent blocks, the number of blocks, and aggregate shape information. The deployable reduction model reaches 92.3% pass@5%, while the diagnostic RTL-context variant only increases this to 93.7%. The smaller gap to the RTL-context variant means that the reduction representation already captures most of the useful sequence-level information in this experiment.

One practical implication is that sequence-level correction does not necessarily require a highly detailed representation of every constituent block command. For the evaluated settings, compact summaries were sufficient to obtain large improvements over the analytical HW model baseline. This is encouraging for deployment because it keeps the sequence model close to the early-stage information available in the performance-modelling flow and avoids making the correction layer depend on detailed RTL-only measurements. This claim is limited to the short, controlled sequences evaluated here. Longer workloads may introduce ordering, queueing, reuse, and cross-command effects that are not visible in these datasets. The sequence experiments are therefore best viewed as an intermediate step between single-block correction and full workload-level prediction.

5.4 Explainability and Diagnostic Value

The explainability analysis in Section 4.6 shows that the learned correction models can provide more than aggregate accuracy improvements. The main value of

the analysis is not simply that certain features have high SHAP importance, but that the attribution and cohort results expose where the baseline mismatch is concentrated.

The global SHAP summary first showed that the correction is still strongly anchored in the analytical baseline, since baseline-cycle descriptors have the largest overall contribution. This is expected because the model is intended to act as a correction layer on top of the existing HW performance model. More importantly, the model is not merely learning a global offset or scale adjustment. The correction also depends on block-command descriptors, including source-shape fields, layout-related fields, and workload-scale fields derived from the `blk_cmd` representation. In other words, the model uses structural information about the executed workload to decide how the prediction should be changed.

Further observations can be drawn from the beeswarm plot in Figure 4.10. Workload-scale descriptor 1 has a wide attribution spread, meaning that it can have a large positive or negative effect on the predicted correction depending on the value of this feature for a given sample. In the local analysis of the largest baseline misses, shown in Figure 4.11, the same descriptor becomes the dominant positive contributor. This is consistent with the baseline mismatch in Figure 4.2, where `Z.A` is shown to be mainly underpredicted by the analytical model. For the samples where the analytical model underestimates RTL most severely, the correction model therefore relies heavily on workload-scale descriptor 1 to increase the predicted cycle count. Workload-scale descriptor 2 provides a different signal, as its values are more clearly separated on opposite sides of the SHAP axis, suggesting that it divides the data into regimes with different correction behaviours.

The cohort analysis makes this interpretation more concrete. When the data are split by workload-scale descriptor 1, the baseline error is highly uneven across regimes. Some regimes have large errors far above the 5% acceptance threshold, while others are already well correlated with RTL. This shows that the analytical model is not uniformly inaccurate for operation Z.A. The mismatch is concentrated in specific workload-scale regimes. The split by workload-scale descriptor 2 is even more direct, since the error is mainly located in one regime. Thus, the two descriptors together identify a small set of workload regimes where the analytical model and RTL disagree most strongly.

The engineering interpretation is that the workload-scale descriptors captured more than the amount of issued work. They also separated cases where the tensor shape and layout aligned cleanly with the regular execution pattern assumed by the analytical model. In the well-correlated regimes, the tensor workload fits this regular pattern, so the analytical model tracks RTL closely. In the poorly cor-

related regimes, the workload reaches a boundary or alignment case: part of the tensor does not line up cleanly with the regular pattern assumed by the HW model, while execution still has to handle that boundary. In those cases, additional work may be needed to handle edges, unused lanes, or less efficient data movement, which can make RTL take more cycles than the analytical model predicts.

This made the SHAP result directly useful for debugging: the descriptors pointed to the alignment-related regime where the analytical model was missing cost, rather than merely to larger workloads.

This explains why the highlighted descriptors were diagnostically useful. They should not be interpreted as direct causal explanations of the RTL behaviour by themselves. Instead, they acted as indicators of the regime in which the analytical model systematically underestimated RTL cycles. The value of the SHAP analysis was therefore not only that it ranked important features, but also that it helped translate a broad operation-level mismatch into a specific and testable modelling hypothesis: the analytical model captured the regular regime well, but missed part of the cost associated with an irregular alignment or boundary regime.

The practical role of explainability is illustrated in the proposed workflow. SHAP first identified the descriptors used by the correction model, local explanations showed that these descriptors dominated the worst baseline misses, and cohort analysis confirmed that the corresponding descriptor regimes had systematically different baseline errors. The resulting evidence gave the modelling team a focused region of the workload space to inspect in the performance-modelling pipeline, helped make the underlying correlation issue apparent from the data, and contributed to the modelling issue being resolved.

5.5 Limitations and Threats to Validity

This section discusses the main limitations and threats to validity of the present work, focusing on aspects of the experimental scope that affect how broadly the results can be interpreted.

5.5.1 Feature- and Cycle-Space Coverage

One main limitation found throughout the thesis concerns the coverage of the generated feature and cycle spaces. The datasets are derived from the verification team's existing testlists, guided randomisation, and targeted additions introduced

during this work. While this gives broad practical coverage of known and relevant workload regimes, it cannot guarantee that all corner cases or rare feature combinations are represented. As the design and verification environment evolve, new edge cases may continue to appear. Consequently, the results apply to the evaluated workload distribution and should not be treated as exhaustive evidence over the full legal workload space. In other words, some rare feature combinations, corner cases, or extreme cycle-count regimes may be absent or only weakly represented in the evaluated data.

This does not imply that the learned correction models cannot generalise to such cases. However, if a workload regime is not present, or only sparsely present, in the training and test data, this thesis cannot make a strong empirical claim about correction quality in that regime. The reported results therefore demonstrate performance on the generated and evaluated workload distribution, while leaving open how well the same models would behave on untested corner cases.

5.5.2 Limited Sequence Length

A second limitation is that the sequence-level experiments are restricted to controlled short-sequence settings. One setting consists of two block commands executed as a short unit-level sequence, while the reduction setting represents one logical operation split across multiple block commands. These settings provide a manageable first step beyond single-block correction, but they do not capture arbitrary-length command streams.

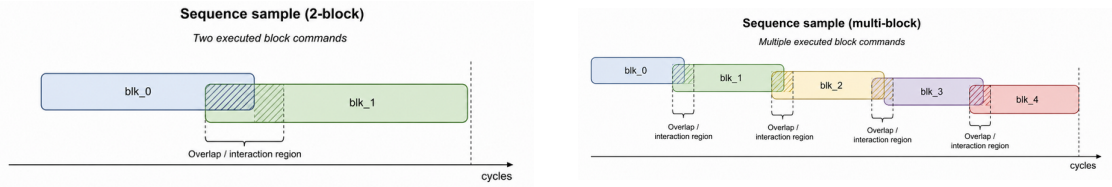

Figure 5.1: Comparison of the evaluated two-block sequence setting with a longer command stream.

However, longer command streams may exhibit additional behaviour that is not visible in two-block cases. Examples include accumulated ordering effects, queueing effects, resource reuse across more than two commands, state carried over across several commands, and more complex interactions between commands competing for the same unit-level resources. This limits the sequence-level claim to an intermediate step between isolated block-command correction and full workload-level

prediction, rather than as evidence that the approach fully captures arbitrary-length command streams.

5.5.3 Idealised Timing Configuration

A third limitation is that the experiments were performed under an idealised unit-level timing configuration through the test setup used for data generation. This setting does not modify the RTL implementation of the unit itself, but simplifies parts of the surrounding verification environment. Compared with the non-idealised test setup, it removes randomised command latency and testbench-modelled interface backpressure. As a result, commands are issued under cleaner timing conditions and the relevant interfaces do not introduce artificial stalls.

5.5.4 Cross-Unit and Scheduling Effects

A final limitation is that the models do not explicitly capture a full execution context in the NPU. The experiments focus on unit-level block commands and selected short sequences, while complete workloads may involve dependencies between units, global scheduling decisions, shared-resource contention and overlap between work issued to different parts of the NPU. The experiments therefore model overlap and interaction only within the evaluated unit-level settings. No explicit modelling of overlap between different units was conducted. Consequently, the reported results should be interpreted as correction results for work performed within individual units, rather than as complete NPU workload-runtime predictions.

Chapter 6

Conclusions

This thesis investigated whether machine learning can be used as a correction layer between a fast analytical NPU performance model and RTL-observed execution-cycle measurements. The goal was not to replace the analytical model or RTL simulation, but to test whether the remaining gap between them can be reduced using block-command and sequence-level information available before RTL results are known. Overall, the results show that this is possible for the evaluated unit-level workloads. The learned models improve RTL alignment in several difficult cases, and the explainability analysis shows that the same models can also help identify workload regimes where the analytical model and RTL disagree.

6.1 Research Questions Revisited

RQ1: Can machine learning be used to predict the performance discrepancy between NPU hardware models and RTL implementations?

Yes. For the evaluated workloads, the learned correction models predict a large part of the discrepancy between the analytical HW model and RTL. The clearest evidence is found in the operation families where the baseline model is inaccurate. For Unit X, all seven operations improve from low baseline pass@5% values to above 90% after correction. Similar improvements are seen for difficult operations in the other units, such as Y.G and Z.A. The sequence-level results also show that the approach extends beyond isolated block commands in the evaluated two-block and reduction settings.

Some operations are already close to RTL before applying ML, so the gain is

smaller and conservative use of correction becomes more relevant, especially if unnecessary perturbations or pass-to-fail regressions are unacceptable. The main conclusion is therefore that ML correction is most useful when the analytical model has a systematic mismatch that is visible in the available features. The simple correction baselines further show that the learnable discrepancy is not exclusive to XGBoost. kNN and Linear/Ridge also recover part of the mismatch in selected cases, which supports the conclusion that the discrepancy contains learnable structure. However, XGBoost is the only evaluated model that consistently provides large improvements across the difficult nonlinear cases.

RQ2: Can machine learning-based performance models provide insights into the factors contributing to performance discrepancies between hardware model predictions and RTL simulation results?

Yes, but mainly as a diagnostic aid. The SHAP and cohort analysis for Z.A showed that the learned correction was driven not only by the baseline cycle estimate, but also by workload-scale, shape, and layout-related descriptors. These descriptors separated cases where the analytical model already matched RTL well from cases where it systematically underestimated RTL. This helped narrow a broad operation-level model–RTL mismatch to specific workload regimes and supported an alignment-related modelling hypothesis.

These explanations should not be treated as proof of the hardware root cause. They explain what the trained model uses, not why the hardware behaves that way. However, they can point the modelling team towards the parts of the workload space where further investigation is most useful.

6.2 Main Contributions

The main contribution of this thesis is to show that RTL correlation for an analytical NPU performance model can be approached as a supervised learning problem. Instead of treating the analytical model and RTL measurements only as outputs to compare manually, the work uses their discrepancy as a learnable signal. In this sense, the thesis introduces a data-driven correction strategy for this hardware-modelling setting: the analytical model remains the starting point, but gradient-boosted tree models are used to learn the remaining systematic mismatch from early-available execution descriptors. The results show that this is not only a plausible formulation, but also a highly effective one for several of the evaluated workload groups.

A second contribution is the diagnostic view of learned correction. The models are not used only to improve prediction accuracy, but they also provide a way to study where and how the analytical model disagrees with RTL. By combining feature attribution with cohort-based error analysis, the correction model can help narrow broad correlation problems to specific workload regimes and descriptor patterns. This is useful from an industrial perspective because it turns part of the model-correlation workflow into a quantitative process: simulation and model data can be used not only to measure that a mismatch exists, but also to guide where modelling effort should be focused next.

6.3 Future Work

A natural continuation of this work is to explore how learned correction models can be used as a bridge between prediction and analytical-model improvement. In this thesis, the correction layer is applied externally: the HW model produces an estimate, and the learned model adjusts it towards RTL. A useful next step would be to translate recurring correction patterns, feature attributions, and cohort-level error regimes back into modelling rules, missing terms, or targeted updates to the analytical model. This would require applying the diagnostic workflow more systematically across operations, sequence types, and model revisions, and combining the ML-based feature analysis with modelling-team expertise so that influential features and feature interactions can be related back to concrete modelling assumptions or simplifications. The goal would be to move beyond using the ML model only to identify mismatch regimes, and towards a workflow where those regimes are traced back to analytical-model assumptions and validated through targeted model updates.

In the longer term, this diagnostic workflow could also be combined with agentic AI systems. Code-oriented agents could use SHAP findings, cohort-level error regimes, and modelling-team constraints to help inspect analytical-model code, suggest candidate fixes, and generate targeted validation tests. The ML model would therefore identify where the model–RTL mismatch occurs, while the agentic system could assist in turning those findings into concrete, reviewable modelling updates. Such a workflow would still require expert supervision and RTL-based validation, but could make the path from learned diagnostics to analytical-model improvement more automated and systematic.

Dataset generation could also be made more systematic. The datasets used in this thesis were built from available verification workloads and targeted additions,

which gives practical coverage of relevant workload regimes but does not guarantee exhaustive coverage of the legal feature and cycle space. The correction model could also guide workload selection for future RTL simulation. For example, samples from regions that previously showed high correction error, samples with high predicted uncertainty, large disagreement between the analytical and learned models, or unusual feature combinations could be prioritised for RTL simulation. This would turn the workflow into an active learning loop, where the model is not only trained on existing simulation data, but also helps guide the creation of new data in the parts of the workload space where correlation is weakest.

A further research direction is to extend the approach beyond the short sequence settings studied here. The sequence-level results suggest that compact execution representations can capture a substantial part of the sequence-level timing behaviour, but the evaluated sequences are still controlled and limited in scope. Future work could study longer command streams, interactions between different units, compiler scheduling effects, and more realistic workload fragments. This would also raise a representation question: as command streams grow, it may become difficult to describe an indefinite-length sequence using a fixed set of tabular features for XGBoost. Sequence-oriented models, hierarchical summaries, or learned embeddings of command streams could therefore be investigated as alternatives for larger-scale runtime prediction.

Finally, the modelling approach itself could also be extended with broader deployment-oriented model evaluation and uncertainty-aware prediction. Although XGBoost was a strong choice for the tabular correction tasks studied in this thesis, practical deployment also depends on training time, inference latency, memory footprint, and ease of integration into the existing performance-modelling pipeline. Future work could therefore compare XGBoost with alternative gradient-boosted tree frameworks, such as LightGBM or CatBoost, evaluating not only correction accuracy and pass@5%, but also retraining cost as new RTL data becomes available and inference overhead during large-scale workload exploration. In addition, future models could predict not only a corrected cycle estimate, but also a confidence interval, calibrated uncertainty estimate, or risk score indicating whether correction is likely to help. Such information could support a conservative deployment policy, where learned correction is applied mainly in regimes with systematic baseline error and acceptable runtime overhead, and bypassed when the analytical model is already within tolerance or when correction uncertainty is high.

Bibliography

[1] Scott Beamer. A case for accelerating software RTL simulation. *IEEE Micro*, 2020.

[2] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, 2016. doi: 10.1145/2939672.2939785. URL `https://arxiv.org/abs/1603.02754`.

[3] Wenji Fang, Yao Lu, Shang Liu, Qijun Zhang, Ceyu Xu, Lisa Wu Wills, Hongce Zhang, and Zhiyao Xie. MasterRTL: A pre-synthesis PPA estimation framework for any RTL design. In *Proceedings of the 42nd IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2023. doi: 10.48550/arXiv.2311.08441. Also available as arXiv:2311.08441.

[4] Lingda Li, Santosh Pandey, Thomas Flynn, Hang Liu, Noel Wheeler, and Adolfy Hoisie. SimNet: Accurate and high-performance computer architecture simulation using deep learning. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(2), 2022. doi: 10.1145/3530891.

[5] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. From RTL to CUDA: A GPU acceleration flow for RTL simulation with batch stimulus. In *Proceedings of the 51st International Conference on Parallel Processing*, ICPP '22, New York, NY, USA, 2022. Association for Computing Machinery. doi: 10.1145/3545008.3545091.

[6] Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems 30 (NeurIPS 2017)*, 2017. doi: 10.48550/arXiv.1705.07874. URL `https://proceedings.neurips.cc/paper/2017/hash/8a20a8621978632d76c43dfd28b67767-Abstract.html`. Also available as arXiv:1705.07874.

[7] Scott M. Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M. Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. From local explanations to global understanding with explainable AI for trees. *Nature Machine Intelligence*, 2:56–67, 2020. doi: 10.1038/s42256-019-0138-9. URL `https://www.nature.com/articles/s42256-019-0138-9`. Preprint available as arXiv:1905.04610.

[8] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 4505–4515. PMLR, 2019. URL `https://proceedings.mlr.press/v97/mendis19a.html`.

[9] ML Platform. Tensor operator set architecture (TOSA) specification v1.0.1. `https://www.mlplatform.org/tosa/tosa_spec_1_0_1.html`, 2025. Accessed 2026-04-27.

[10] Arash Nasr-Esfahany, Mohammad Alizadeh, Victor Lee, Hanna Alam, Brett W. Coon, David Culler, Vidushi Dadu, Martin Dixon, Henry M. Levy, Santosh Pandey, Parthasarathy Ranganathan, and Amir Yazdanbakhsh. Concorde: Fast and accurate CPU performance modeling with compositional analytical-ML fusion. *arXiv preprint arXiv:2503.23076*, 2025. doi: 10.48550/arXiv.2503.23076.

[11] NVIDIA. Matrix multiplication background user's guide. `https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html`, 2023. Accessed 2026-05-06.

[12] Alex Renda, Yishen Chen, Charith Mendis, and Michael Carbin. Difftune: Optimizing CPU simulator parameters with learned differentiable surrogates. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 442–455, 2020. doi: 10.1109/MICRO50266.2020.00045.

[13] Tyler Sorensen, Aninda Manocha, Esin Tureci, Marcelo Orenes-Vera, Juan L. Aragón, and Margaret Martonosi. A simulator and compiler framework for agile hardware-software co-design evaluation and exploration: Invited talk. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '20)*, 2020. doi: 10.1145/3400302.3415751.

[14] Joonho Whangbo, Edwin Lim, Chengyi Lux Zhang, Kevin Anderson, Abraham Gonzalez, Raghav Gupta, Nivedha Krishnakumar, Sagar Karandikar, Borivoje Nikolic, Yakun Sophia Shao, and Krste Asanovic. Fireaxe: Partitioned FPGA-accelerated simulation of large-scale RTL designs. In *Proceedings of the 51st Annual International Symposium on Computer Architecture*, pages 501–515, 2024. doi: 10.1109/ISCA59077.2024.00044.

[15] Weiyan Zhang, Mehran Goli, Muhammad Hassan, and Rolf Drechsler. Efficient ML-based performance estimation approach across different microarchitectures for RISC-V processors. In *2023 26th Euromicro Conference on Digital System Design (DSD)*, 2023. URL `https://agra.informatik.uni-bremen.de/doc/konf/DSD2023_WZ.pdf`.

Appendix A

Additional Results

This appendix reports supplementary results that support the main Results chapter. Section A.1 summarises the simple Linear/Ridge, K-means and kNN correction baselines used in Section 4.2.3. Section A.2 reports the full target-formulation sweeps for the single-block XGBoost experiments. These results are included for completeness, while the main text focuses on the best corrected models and representative comparisons.

A.1 Simple Correction Baselines

For each operation, Linear/Ridge, K-means, and kNN correction baselines were evaluated. For each baseline family, the reported value is the best pass@5% obtained over the predefined sweep shown in Table A.1. The best variant was selected separately for each operation and baseline family using pass@5%, matching the primary evaluation metric used in the main results. The HW model pass@5% is included as the uncorrected reference. XGBoost results are not repeated here; they are reported in Section 4.3.

Table A.1: Hyperparameter grids used for the simple correction baselines.

Baseline family	Evaluated variants
Linear/Ridge	Linear regression; Ridge with $\alpha \in \{1, 10, 100, 1000, 10000, 100000, 1000000, 10000000\}$
K-means	$k \in \{32, 64, 128\}$
kNN	$k \in \{5, 10, 25, 50\}$

Table A.2: Simple correction baseline results. For each baseline family, the table reports the best pass@5% obtained over the predefined sweep.

OP	HW	Linear/Ridge	K-means	kNN
X.A	18.6	20.4	17.9	37.7
X.B	23.1	30.3	26.2	39.9
X.C	20.8	30.0	24.5	41.0
X.D	17.5	31.0	26.9	46.9
X.E	46.7	27.1	30.9	47.5
X.F	46.3	26.8	30.4	47.4
X.G	51.1	29.5	30.2	49.6
Y.A	100.0	100.0	100.0	100.0
Y.B	100.0	100.0	100.0	100.0
Y.C	100.0	100.0	100.0	100.0
Y.D	99.0	99.5	99.2	99.7
Y.E	98.8	99.7	99.1	99.5
Y.F	97.9	93.7	97.1	97.1
Y.G	23.8	47.8	29.4	77.6
Z.A	44.3	99.4	43.7	76.6
Z.B	99.9	100.0	99.9	100.0
Z.C	85.6	92.2	92.1	99.2
Z.D	98.1	98.4	98.4	99.3

A.2 Target-Formulation Sweeps

Table A.3: Target-formulation sweep for Unit X single-block XGBoost correction models. Results use the HW model cycle estimate as an input feature.

OP	Samples	Target	pass@5%	Med. err.	p95 err.	MAE
X.A	51,362	Direct	94.9	0.92	5.05	7.41
		Residual	96.6	0.56	4.20	3.43
		Ratio	97.8	0.51	3.61	3.47
		Log-ratio	98.2	0.46	3.46	3.17
X.B	49,818	Direct	75.6	2.59	12.12	68.62
		Residual	87.8	1.34	8.76	19.21
		Ratio	96.2	0.55	4.19	18.06
		Log-ratio	96.7	0.52	3.78	16.43
X.C	49,940	Direct	79.7	2.16	11.26	29.59
		Residual	84.3	1.37	10.84	25.73
		Ratio	92.8	0.58	6.60	24.31
		Log-ratio	93.4	0.55	6.16	23.55
X.D	48,367	Direct	81.6	1.79	10.67	43.34
		Residual	88.7	0.99	8.60	19.65
		Ratio	97.9	0.31	2.82	21.68
		Log-ratio	98.2	0.29	2.63	20.54
X.E	49,908	Direct	70.4	2.56	16.84	67.64
		Residual	79.1	1.11	15.08	44.27
		Ratio	91.3	0.65	7.01	47.40
		Log-ratio	92.1	0.58	6.61	43.62
X.F	49,913	Direct	72.5	2.35	15.85	65.75
		Residual	79.3	1.09	14.92	43.77
		Ratio	91.7	0.65	6.83	49.17
		Log-ratio	92.3	0.58	6.49	45.27
X.G	49,901	Direct	58.5	3.84	24.91	138.91
		Residual	79.3	0.82	18.08	79.50
		Ratio	90.0	0.64	8.21	84.44
		Log-ratio	90.7	0.60	7.79	79.98

A.2.1 Unit Y

Table A.4: Target-formulation sweep for Unit Y single-block XGBoost correction models. Results use the HW model cycle estimate as an input feature.

OP	Samples	Target	pass@5%	Med. err.	p95 err.	MAE
Y.A	29,893	Direct	97.7	0.00	2.50	1.06
		Residual	100.0	0.00	0.00	0.00
		Ratio	100.0	0.00	0.00	0.02
		Log-ratio	100.0	0.00	0.00	0.02
Y.B	29,655	Direct	100.0	0.00	0.00	0.00
		Residual	100.0	0.00	0.00	0.00
		Ratio	100.0	0.00	0.00	0.00
		Log-ratio	100.0	0.00	0.00	0.00
Y.C	29,744	Direct	94.7	0.05	5.34	13.78
		Residual	99.8	0.00	0.02	0.06
		Ratio	100.0	0.00	0.03	0.27
		Log-ratio	100.0	0.00	0.03	0.26
Y.D	29,669	Direct	100.0	0.00	0.04	0.00
		Residual	100.0	0.00	0.00	0.00
		Ratio	100.0	0.00	0.00	0.00
		Log-ratio	100.0	0.00	0.00	0.00
Y.E	29,699	Direct	97.5	0.01	2.84	1.21
		Residual	100.0	0.00	0.00	0.00
		Ratio	100.0	0.00	0.03	0.04
		Log-ratio	100.0	0.00	0.03	0.04
Y.F	29,808	Direct	94.7	0.07	5.36	2.97
		Residual	96.0	0.04	3.52	1.72
		Ratio	98.4	0.06	1.26	2.50
		Log-ratio	98.5	0.06	1.11	2.35
Y.G	29,449	Direct	94.3	0.05	5.96	5.84
		Residual	95.3	0.04	4.53	3.88
		Ratio	98.7	0.05	1.49	4.47
		Log-ratio	98.9	0.04	1.36	3.97

A.2.2 Unit Z

Table A.5: Target-formulation sweep for Unit Z single-block XGBoost correction models. Results use the HW model cycle estimate as an input feature.

OP	Samples	Target	pass@5%	Med. err.	p95 err.	MAE
Z.A	35,039	Direct	99.0	0.50	2.57	44.54
		Residual	87.9	0.82	7.65	44.01
		Ratio	97.8	0.18	2.75	53.08
		Log-ratio	98.3	0.13	2.24	56.40
Z.B	38,401	Direct	98.7	0.37	2.47	13.36
		Residual	100.0	0.00	0.00	0.00
		Ratio	100.0	0.00	0.00	0.01
		Log-ratio	100.0	0.00	0.00	0.01
Z.C	39,951	Direct	83.6	1.42	15.39	138.12
		Residual	95.8	0.27	3.72	16.23
		Ratio	99.9	0.00	0.12	11.66
		Log-ratio	99.9	0.00	0.12	11.36
Z.D	59,135	Direct	86.2	1.30	13.50	239.10
		Residual	100.0	0.00	0.01	0.01
		Ratio	100.0	0.01	0.11	0.53
		Log-ratio	100.0	0.01	0.12	0.59

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY | LUNDS TEKNISKA HÖGSKOLA | PRESENTED 2026-06-08

MASTER'S THESIS Machine-Learning-Based Correction of Analytical NPU Performance Models

STUDENTS Joaquim Gouveia, Oliver Gouveia

SUPERVISOR Jonas Skeppstedt (LTH), Fabio Pancot (Arm), William Isaksson (Arm)

EXAMINER Pietro Andreani (LTH)

AI for AI Hardware: Predicting Chip Performance Before Silicon

POPULAR SCIENCE SUMMARY **Joaquim Gouveia, Oliver Gouveia**

AI chips are designed years before they become physical products. To support early design decisions, engineers in the semiconductor industry use fast software models to forecast how the hardware will perform, but these forecasts can make repeatable mistakes. This thesis shows how machine learning can learn to spot and correct these mistakes.

Modern artificial intelligence depends on specialised chips that can process huge amounts of data quickly and efficiently. One example is a Neural Processing Unit, or NPU: a type of processor built especially for neural-network workloads, including the calculations behind image recognition, speech processing, and generative AI. These processors accelerate tools we rely on for many everyday tasks, from using ChatGPT to unlocking a phone with Face ID.

But designing such hardware creates a difficult question: how do engineers know how fast a chip will be before it exists?

One answer is simulation. Very detailed simulations can describe hardware close to how it will actually be built, but they are often too slow to use for every design idea. Faster analytical models are therefore used earlier in development. These models are useful because they can quickly estimate how many clock cycles a piece of work will take. However, they simplify the complex reality of computer chips. Like a route planner estimating travel time before you start driving, they give a useful early prediction, but traffic, roadworks, and other real-world details can make the final result different.

This thesis investigates whether machine learning can improve these fast estimates. The model first makes its usual prediction of how long a task will take. A machine-learning model then looks at that prediction, together with basic information about the task, and learns how the estimate should be adjusted based on more detailed simulations. In this sense the route planner learns from previous journeys. If it often underestimates travel time on certain roads or in certain traffic conditions, it can adjust future estimates before the next trip. In the same way, the learning model in this thesis recognises situations where the fast chip-performance estimate is usually too optimistic or too pessimistic.

The results show that many differences between the fast model and the detailed simulation are not random. They contain patterns that machine learning can learn. In several difficult cases, the original fast estimate was within five percent of the detailed simulation result for only a minority of test cases. After correction, more than 90 percent passed this five-percent mark, and some cases came close to 99 percent. The method also worked for short sequences of tasks, not only for isolated pieces of work.

The approach was also useful for understanding the errors. By analysing which inputs affected the correction, the thesis identified types of cases where the analytical model tended to underestimate the runtime. In other words, the method not only improved the prediction, it also helped point towards where the mismatch appeared.

There is also a fitting twist: machine learning is used to improve predictions for chips that are themselves designed to accelerate machine-learning workloads. Better early estimates can help engineers compare design choices, find performance issues sooner, and use expensive detailed simulations where they are most useful.

The work does not remove the need for detailed simulation, but it shows that simulation data can be reused to make fast models more accurate.