LICENTIATE THESIS

Machine Learning for Predictive Modeling and Abstraction in Industrial-Scale Systems

Edin Jelačić

Department of Computer Science and Computer Engineering

Mälardalen University

edin.jelacic@mdu.se

Main supervisor: Prof. Tiberiu Seceleanu

Co-supervisors: Prof. Cristina Seceleanu, Prof. Ning Xiong, Assoc. Sen. Lecturer Peter Backeman

Contents

Abstract ix

Sammanfattning xi

Acknowledgments xiii

List of Publications xv

I Thesis 1

1 Introduction 3

1.1 Research context and narrative . . . 4
1.2 Research goal and questions . . . 6
1.3 Summary of contributions . . . 6
1.4 Outline of the thesis . . . 7

2 Background 9

2.1 Neural networks . . . 9
2.1.1 Feedforward neural networks . . . 10
2.1.2 Long short-term memory networks . . . 12
2.2 Neural network verification and abstraction . . . 15
2.2.1 Formal verification of neural networks . . . 15
2.2.2 Abstraction-refinement for neural networks . . . 17
2.3 Conformal prediction and uncertainty quantification . . . 19
2.3.1 Split conformal prediction for regression . . . 20
2.3.2 Shapley values for feature attribution . . . 22
2.4 Cache memory and hardware performance simulation . . . 24

2.4.1 Memory hierarchy and cache operation 24
2.4.2 The DynamoRIO Cachesim simulator 25
2.5 SBT for automated driving systems 27
2.5.1 Scenario-based testing 27
2.5.2 OpenSCENARIO and OpenDRIVE 28

3 Research overview 29
3.1 Motivation and research gaps 29
3.2 Research goal 31
3.3 Research questions 31
3.4 Research methodology 33
3.4.1 Formal input reduction 33
3.4.2 Guaranteed load forecasting 35
3.4.3 Cache data-driven modeling 35
3.4.4 Scenario compilation (HASCO) 35
3.4.5 Cross-cutting methodological pattern - Reduce, Represent, Validate . . 36

4 Contributions 39
4.1 Thesis contributions 39
4.1.1 Abstraction-based reduction of neural networks 39
4.1.2 Forecasting with uncertainty and explanations 40
4.1.3 Data-driven cache miss prediction 41
4.1.4 Hybrid AI Simulation Compiler (HASCO) 46
4.2 Individual contribution of the thesis author 50

5 Related work 51
5.1 Abstraction and feature selection 51
5.2 Uncertainty quantification in practical domains 52
5.3 ML for hardware performance modeling 53
5.4 LLMs for scenario generation 54

6 Discussion and limitations 57
6.1 Discussion 57
6.1.1 Addressing the research gaps 57

6.1.2 The Role of intermediate representations 58
6.1.3 Design for domain expert utilization 58
6.2 Limitations and threats to validity 59
6.2.1 Paper A 59
6.2.2 Paper B 59
6.2.3 Paper C 60
6.2.4 Paper D 60

7 Conclusion and future work 61
7.1 Conclusion 61
7.2 Future Work 62

Bibliography 65

II Included Papers 77

8 Paper A 79
8.1 Introduction 81
8.2 Background 82
8.2.1 Neural Networks 82
8.2.2 Verification of NNs 84
8.2.3 Abstracting NNs 84
8.3 Removing Inputs by Over/Under-estimation 88
8.4 Identifying Insignificant Inputs 91
8.4.1 Estimating Impact of Inputs 94
8.5 Experimental Evaluation 95
8.5.1 Polynomial Coefficient Estimation 95
8.5.2 Identifying Demanding Functions 95
8.5.3 Repeated Experiments 97
8.6 Related Work 97
8.7 Conclusions and Future Work 98
8.7.1 Future Work 98

Bibliography 99

9 Paper B **101**
9.1 Introduction 103
9.1.1 Objectives and Contributions 103
9.2 Our Methodology 104
9.2.1 Data Extraction and Preprocessing 104
9.2.2 Conformal Prediction Framework 106
9.2.3 Shapley Value Integration 110
9.2.4 System Integration & GUI Application 112
9.3 Experimental Setup 117
9.3.1 Dataset Description 117
9.3.2 Experimental Design 117
9.4 Results 118
9.4.1 Coverage, Interval Length, and Prediction Error 119
9.4.2 Forecasting Performance 119
9.4.3 Additivity Check 121
9.4.4 Sensitivity Analysis over Ablations 122
9.5 Discussion 125
9.6 Related Work 126
9.7 Conclusion and Future Work 126

Bibliography **129**

10 Paper C **133**
10.1 Introduction 135
10.2 Background 138
10.2.1 Memory hierarchy 138
10.2.2 Cache memory 139
10.2.3 Neural Networks 139
10.2.4 DynamoRIO Cachesim 141
10.3 The Approach 141
10.4 Implementation 142
10.4.1 DynamoRIO Instruction Features 143
10.4.2 Tokenization 143

10.4.3 Model and Training Procedure 145
10.5 Results 147
10.5.1 Performance Metrics 148
10.5.2 Execution 150
10.6 Discussion 156
10.7 Related Work 158
10.8 Conclusion and Future work 159

Bibliography **161**

11 Paper D **181**
11.1 Introduction 182
11.1.1 The specification gap 183
11.1.2 The need for code co-generation 184
11.2 Background 186
11.2.1 The OpenSCENARIO standard, the `scenariogeneration` library and esmini 186
11.2.2 Scenario-based testing and reconstruction 187
11.3 Related Work 187
11.4 Methodology: the HASCO compiler 188
11.4.1 Static world generation (geospatial pipeline) 189
11.4.2 Vehicle extraction module 189
11.4.3 Dynamic scenario synthesis (semantic pipeline) 190
11.4.4 Multi-strategy synthesis 190
11.4.5 The heuristic translation layer 191
11.4.6 The forensic judge (validation loop) 194
11.5 Evaluation methodology 195
11.5.1 Quantitative analysis: executability 195
11.5.2 Qualitative analysis: semantic fidelity 196
11.5.3 Cost-benefit analysis 198
11.6 Threats to validity 200
11.7 Discussion 202
11.8 Conclusion and future work 202

Bibliography **205**

11.9 Prompts and LLM Context . . . 208

11.10 Worked example: cyclist–truck collision (German report) . . . 213

Abstract

Machine learning is increasingly called upon to guide decisions in critical industrial applications. Its predictive power promises gains in efficiency, yet its black-box nature and lack of guarantees pose risks in contexts where behavior must remain analyzable and safe. This thesis asks how machine learning can be made trustworthy, explainable, and efficient enough for engineers to deploy in practice. Three gaps hamper broader adoption. Few works provide formal or statistical guarantees on ML outputs paired with explanations that engineers can act on (Gap A). Data-driven models that generalize across hardware configurations without retraining remain rare, and existing simulators are prohibitively slow (Gap B). Many contributions address individual components of industrially motivated problems without combining them into validated end-to-end pipelines (Gap C). To address Gap A, we apply abstraction to neural networks, showing that inputs with negligible effect on the output can be formally identified and removed, producing simpler yet bounded models open to verification. We then introduce a conformal prediction framework for CPU load forecasting that provides statistically guaranteed coverage intervals, combined with Shapley value analysis to trace individual task contributions to the predicted load. To address Gap B, we develop a data-driven cache memory surrogate using long short-term memory networks, reproducing cache miss distributions across unseen hardware configurations at a fraction of the simulator's computational cost. To address Gap C, we present HASCO, a Hybrid AI Simulation Compiler that translates natural language accident reports into executable vehicular simulation scenarios through a structured compilation approach with deterministic validation. Together, these contributions establish a path toward machine learning that is not merely powerful but trustworthy, explainable, and practically deployable in the industrial workflow.

Sammanfattning

Maskininlärning används i allt högre grad för att vägleda beslut i kritiska industriella tillämpningar. Dess prediktiva kraft utlovar effektivitetsvinster, men dess svarta låda-natur och avsaknad av garantier medför risker i sammanhang där systemets beteende måste förbli analyserbart och säkert. Denna avhandling undersöker hur maskininlärning kan göras tillförlitlig, förklarbar och tillräckligt effektiv för att ingenjörer ska kunna använda den i praktiken. Tre kunskapsluckor försvårar bredare tillämpning. Få studier erbjuder formella eller statistiska garantier på ML-resultat i kombination med förklaringar som ingenjörer kan agera på (Lucka A). Datadrivna modeller som generaliserar över hårdvarukonfigurationer utan omskolning är sällsynta, och befintliga simulatorer är orimligt långsamma (Lucka B). Många forskningsbidrag behandlar enskilda komponenter av industriellt motiverade problem utan att sammanföra dem i validerade helhetspipelines (Lucka C). För att adressera Lucka A tillämpar vi abstraktion på neurala nätverk och visar att insignifikanta indata formellt kan identifieras och avlägsnas, vilket ger enklare men begränsade modeller öppna för verifiering. Vi introducerar sedan ett ramverk för konform prediktion av processorbelastning som ger statistiskt garanterade konfidensintervall, kombinerat med Shapley-värdesanalys för att spåra enskilda uppgifters bidrag till den predikterade belastningen. För att adressera Lucka B utvecklar vi en datadriven cacheminnesurrogat baserad på LSTM-nätverk, som reproducerar cachemiss-fördelningar över tidigare osedda hårdvarukonfigurationer till en bråkdel av simulatorns beräkningskostnad. För att adressera Lucka C presenterar vi HASCO, en hybrid AI-simuleringskompilator som översätter naturspråkliga olycksrapporter till exekverbara fordonssimuleringsscenarier genom en strukturerad kompileringsansats med deterministisk validering. Sammantaget stakar dessa bidrag ut en väg mot maskininlärning som inte bara är kraftfull, utan även tillförlitlig, förklarbar och praktiskt användbar i det industriella arbetsflödet.

“Možete me otpustiti ali ovaj mi čekić iz ruku ne možete oteti!”

“You can fire me, but you can not take this hammer from my hands!” — Muhamed Buza, my late grandfather

In memory of those who did not survive to see their dreams fulfilled.

To the fallen of Bosnia and Herzegovina, 1992–1995.

Acknowledgments

My father has a habit of saying to me that I always somehow choose the most complicated and challenging ways to achieve things, but still achieve them, be it fixing a loose power outlet or working my way towards a profession. This has shown itself thoroughly to be the case with my education, where I was driven by curiosity towards exploring increasingly complex and abstract phenomena and trying to squeeze every last bit of juice from the Universe as we are able to observe it. This I did to the best of my ability by way of reasoning and reading as much as I could of said Universe and all the ebbs and flows therein, from societal dynamics, to electrical engineering, machine learning and all else contained within the magical science of computing. Though I believe my father was quite right about this aspect of my personality, nowhere was this as true as in my doctoral studies. This journey I have undertaken, to wring out as much of science in the short time that has been allotted to us has proven to be, for better or for worse, infinite in both its depth and breadth. It is hardly possible to set foot on this path with loads lighter than those of others. Every single passing minute and every idea bounced back and forth in stuffy, dimly lit meeting rooms, makes this journey increasingly formidable for those hoping to undertake it in the future. We are all aware of the ongoing proverbial flood of AI research work. We bear witness to online chatrooms set abuzz with news of newer, faster, flashier models. We see images proliferating online from, and take part in, grandiose research conferences where seemingly the fate of the humanity's computing appears to be decided over coffee-stained dinner tables and tired, suited keynote speakers. We spare a thought also for all those grinding in silence, those comp-sci students holed up in cosy rooms, looking at rain-covered windows over overheated noisy laptops displaying LaTeX highlighted text. If research is easy, you are likely not doing it right.

That said, my process of getting to this particular position in my life of defending a Licentiate Thesis has been a challenging patchwork composed of decisions whose outcomes I was aware of and of many more whose I could not foresee. For the former, I weighed options

and counter-options painfully many times, and for the latter, I have been blessed by positive guidance on all sides so as not to err. This came from various sources. Firstly from my wife Azra, who has been with me through a Russian novel's worth of challenges and stood as my support and told me honestly when I was wrong. It came from my parents whose words of encouragement have given me hope even when times were incredibly trying. It is hard to forget many a night past the library's closing time, studying late on a streetlamp-lit park bench on the wooded outskirts of Ljubljana during my master's studies, with handwritten algorithm notes on one side and a videocall to mom, ever smiling, and dad curiously advising on the other. This work belongs to you all far more than it does to me.

Regarding the doctoral research itself, I owe an enormous debt to my supervisors Tiberiu Seceleanu, Cristina Seceleanu, Ning Xiong and Peter Backeman, whose invaluable ideas, numerous brainstorms and constructive feedback have made this body of work possible in the first place.

I would also like to thank my colleagues and fellow PhD students at Mälardalen University and beyond: Aldin Beriša, Abdulkarim Habbab, Sarmad Bashir, Zenepe Satka, Madiha Umar, Leo Hatvani, Mikael Salari, Branko Miloradović, Sahar Mobaiyen, Anna Friebe, Daniel Bujosa and many others. From all of you I learned so much, and I hope I managed to give back at least a little.

Finally, I would like to thank my numerous esteemed industrial partner colleagues in industries across Sweden. Your support was instrumental in manifesting my ideas into reality.

The work presented in this thesis has been supported by the Swedish Knowledge Foundation through the PerFlex (*Performant and Flexible digital Systems through Verifiable Artificial Intelligence*) project, grant no. 20220033; the IGP-IDS (*International Guest Professor of Intelligent Distributed Systems*) project, grant no. 20230147; and the ACICS (*Assured Cloud Platforms for Industrial Cyber-Physical Systems*) project, grant no. 20190038; as well as by Mälardalen University through the TSS-INSID (*In-Silico Driving Assurance Using Machine-Learning-Powered Verification and Synthesis for Safe Autonomous Vehicles*) project of the Trusted Smart Systems initiative.

Edin Jelačić, Västerås, May 2026

List of Publications

Papers included in this thesis[1]

Paper A *AISoLA 2023*

Peter Backeman, Edin Jelačić, Cristina Seceleanu, Ning Xiong, Tiberiu Seceleanu. "Abstraction-based Reduction of Input Size for Neural Networks." *First International Conference on Bridging the Gap between AI and Reality (AISoLA 2023)*, October 2023.

Paper B *COMPSAC 2025*

Edin Jelačić, Cristina Seceleanu, Peter Backeman, Ning Xiong, Tiberiu Seceleanu, Axel Jantsch. "A Conformal Prediction-Based Framework for CPU Load Forecasting: A Black-Box Approach." *49th IEEE International Conference on Computers, Software, and Applications (COMPSAC 2025)*, July 2025.

Paper C *STTT 2025*

Edin Jelačić, Cristina Seceleanu, Ning Xiong, Peter Backeman, Sharifeh Yaghoobi, Tiberiu Seceleanu. "Machine Learning-Based Cache Miss Prediction." *International Journal on Software Tools for Technology Transfer (STTT)*, April 2025.

Paper D *AEiC 2026*

Edin Jelačić, Rong Gu, Cristina Seceleanu, Ning Xiong, Peter Backeman, Tiberiu Seceleanu, Zhennan Fei, Ali Nouri. "HASCO: A Hybrid AI Simulation Compiler for Semantic Accident Reconstruction." *30th Ada-Europe International Conference on Reliable Software Technologies (AEiC 2026)*, June 2026.

[1]The included papers have been reformatted to comply with the thesis layout.

Part I

Thesis

Chapter 1

Introduction

Modern industrial environments generate vast amounts of structured, semi-structured, and unstructured data, both from system operations and from the engineers operating them [1, 2, 3]. At the same time, industrial environments often suffer from a lack of optimization with regards to problem-solving, either due to the cyber-physical limitations of the systems being worked on, or due to the sheer magnitude of complexity of the processes required to solve these problems [4, 5]. These two simultaneous intricacies, alongside the pervasiveness of machine learning (ML) in modern contexts, open up approaches that may have previously been unthinkable for developers actively tackling these problems.

The ability of universal function approximators to provide powerful insights into vast amounts of data, particularly industry-specific measurements and aggregated expert knowledge, is now a well-established phenomenon [6, 7]. We thus treat ML as a pragmatic tool for reasoning from large quantities of industrial data, and show that this data is crucial both for understanding how these systems operate and for solving the problems arising in industrial environments in novel ways.

The key component towards increased adoption of ML in industrial contexts is attaining stakeholder trust with respect to their particular use-case and environment. There exist tradeoffs with utilizing an ML system that must be clearly communicated between ML practitioners and stakeholders, such as the complexity of evaluating ML-produced results. Cost is a further hurdle: the introduction of ML pipelines must justify the investment to stakeholders. It is for these reasons that a key aspect of any ML introduction must be a solution that ensures *trustworthiness*, *explainability*, and *efficiency*.

These three notions, we define as follows: by *trustworthiness* we refer to the level of trust

that stakeholders may reasonably have in the results of ML-based analysis, stemming from inherent properties of the ML system and all of its core components. By *explainability* we refer to the clarity of the reasoning behind ML-based analysis and the ability of the ML system to manifest its logic in an understandable way. By *efficiency* we mean the ability of the ML system to demonstrably be a net positive for the stakeholders in terms of cost-benefit analysis.

We strive to extend these three notions, individually or altogether, as much as it was possible to the industrial domain. That said, there exist several industrial-scale challenges that our research has focused on tackling. First, an industrial partner in the energy sector operates a dual-core real-time protection and control device deployed in electrical substation environments. The device runs a configurable set of protection, monitoring, and control functions that customers enable or disable according to the requirements of their installation. Because the device operates under hard real-time constraints, a missed protection deadline can leave transmission infrastructure unprotected, and engineers must verify before deployment that a given function configuration will not overload the processor [8, 9]. Second, evaluating how software performs under different hardware cache configurations traditionally requires running an instruction-level simulator for each configuration, a process that incurs overhead orders of magnitude slower than native execution [10]. For an industrial partner operating server-class computing infrastructure, this cost becomes prohibitive when exploring a design space spanning multiple cache sizes and core counts. Third, the validation of automated driving systems requires large libraries of concrete test scenarios, yet translating the thousands of accident reports that exist as natural language text into executable simulation artifacts remains a manual, expert-driven process.

These challenges motivate an overarching research goal that is introduced formally in Section 1.2.

1.1 Research context and narrative

The work presented in this thesis does not follow a single sequential research thread. Rather, it addresses several industrially motivated problems that share common concerns: the trustworthiness of ML outputs, the efficiency of data-driven alternatives to traditional tools, and the explainability of results to practicing engineers. Each contribution tackles one or more of these concerns in a different domain and at a different level of abstraction.

Our first concern has been that neural networks (NNs), being fundamental components upon which many ML systems are built [11], often become unnecessarily large and carry the severe drawback of their black-box nature for practitioners modeling safety-critical systems [12]. Significant work has been done in the field of formal NN verification [13], with the Marabou verification tool [14] being of particular interest. Elboher et al. [15] utilized Marabou for formal verification of an abstracted NN reduced in size by counter-example guided abstraction refinement [16]. In our first contribution [17], we extend this idea. A related but distinct challenge arises in a different industrial context: rather than simplifying a model's inputs, the question becomes how to quantify the uncertainty of a model's outputs and explain what drives them. For the protection device described above, engineers need to know not just the forecast of CPU load for a given task configuration, but how uncertain that forecast is and which individual tasks contribute most to the load. We addressed this by designing a conformal prediction framework that wraps a base neural network with statistically guaranteed prediction intervals, combined with Shapley value attribution to identify each task's contribution. The framework was validated on real measurements from the device, and the industrial partner has confirmed its usefulness and intends to deploy a specialized version.

For the cache simulation problem, we developed an LSTM-based model that learns cache miss distributions directly from x86 program execution traces, accepts cache configuration parameters as inputs, and generalizes to hardware configurations not seen during training, effectively reducing the need for repeated simulator runs to a single forward pass. The model was evaluated on four benchmark programs across various configuration combinations, including cache sizes outside the training range.

Finally, in collaboration with Volvo Cars, we designed and implemented HASCO, the Hybrid AI Simulation Compiler, which translates natural language accident reports into executable OpenSCENARIO/OpenDRIVE simulation artifacts. The pipeline combines LLM-based semantic extraction with deterministic validation and repair loops across eight languages.

These contributions illustrate how the same core concerns, guarantees, transparency and usability, manifest across different industrial ML contexts. The specific research gaps motivating each contribution are detailed in Section 3.1.

Beyond these three qualities, a shared design pattern unifies all four contributions at the architectural level: each one *reduces* the problem to its essential structure, *represents* that structure in an intermediate form amenable to automated analysis, and *validates* the result against a

criterion independent of the method that produced it. This *reduce, represent, validate* pattern is formalized in Subsection 3.4.5 and examined in detail in Chapter 6.

1.2 Research goal and questions

Our overarching research goal (RG) is:

> **RG:** Design and evaluate a trustworthy ML method for industrial-scale automation, with explainability and workflow integration.

To realize this aim, we pose four research questions. First, how can the influence of individual inputs on a neural network's output be identified with formal or principled guarantees? Second, how can forecasting methods provide statistically guaranteed prediction intervals and per-feature attribution in industrial systems? Third, how can data-driven models replace expensive simulation processes while maintaining fidelity? Finally, how can natural language narratives be compiled into executable, validated simulation scenarios with minimized expert repair effort? These questions are addressed in detail in Section 3.3.

Throughout this thesis, fidelity is used in two distinct senses. In the context of performance modeling, it refers to accuracy at both the distributional level: how closely predicted cache miss patterns track the simulator at each subsequence and the aggregate level: how well total miss counts align across a full program execution. In the context of scenario generation, fidelity refers to syntactic validity: the generated artifact executes without error and semantic fidelity: the simulation reflects the causal narrative of the source report. The tradeoff between generation speed and fidelity in the latter sense is examined quantitatively in Paper D.

Both uses share a common concern: that a data-driven or generative system should reproduce not just the surface statistics of its target, but the underlying behavior that engineers actually care about. In Paper C this means tracking both the distribution of misses over time and their aggregate total. In Paper D it means producing code that both executes and tells the right story.

1.3 Summary of contributions

This thesis presents four contributions that together address the posed research questions:

1. **Abstraction-based input reduction for neural networks:** A formal method for identifying and eliminating neural network inputs of negligible influence.

2. **Forecasting with uncertainty and explanations:** A model-agnostic framework providing statistically guaranteed prediction intervals on CPU load.

3. **Data-driven cache miss prediction:** An LSTM network trained on x86 assembly execution traces to predict cache miss distributions.

4. **HASCO: Hybrid AI Simulation Compiler:** A compilation pipeline translating natural language road accident reports into executable OpenSCENARIO/OpenDRIVE simulation artifacts.

They are described in detail in Chapter 4.

1.4 Outline of the thesis

The thesis consists of two parts. **Part I** provides the research context, methodology, and synthesis of results:

- Chapter 2 presents the technical background on neural networks, formal verification, conformal prediction, cache memory simulation, and scenario-based testing.
- Chapter 3 details the research gaps, research goal, research questions, and the methodology used across the four papers.
- Chapter 4 describes the contributions of each included paper in detail, with figures, formal statements, and a mapping of papers to research questions. It also summarizes the key empirical results.
- Chapter 5 presents a survey of related work positioning each contribution against the state of the art.
- Chapter 6 discusses the findings, limitations, and threats to validity.
- Chapter 7 concludes and outlines future work.

Part II includes the four published papers, reformatted to comply with the thesis layout.

Chapter 2

Background

This chapter presents the technical foundations necessary to understand the contributions of this thesis.

2.1 Neural networks

Neural networks (NNs) are a widely used class of machine learning models [18]. They are computational structures composed of calculation units connected between each other by edges, in a way that loosely resembles the connections of neurons by way of synapses in a biological neural network, from which they draw their name. A distinction is commonly made in literature to term the computational model an artificial neural network (ANN) to avoid confusion. An ANN receives one or more numerical inputs, transforms them by way of passing said inputs through its internal grid of artificial neurons and outputs it at the end. Each artificial neuron, packed with a set of individual parameters, represents a mathematical transformation $f_{NN}(\cdot)$ of the input value or signal provided to it (termed x_{IN}), with connections to subsequent neurons representing a data flow path for subsequent processing, often with an application of a transformation function of an identical format with different parameters within the same NN. This is visualized in Fig 2.1.

Usually these neurons are arranged in one or more layers, starting with the input layer that accepts the inputs vector $\overrightarrow{x_{IN}} = [x_{IN\ 1}, x_{IN\ 2}, \ldots, x_{IN\ K}]$ of size K, one or more intermediate layers (typically termed *hidden* layers) and an output layer with no subsequent connections that outputs an output vector $\overrightarrow{y_{OUT\ 1}} = [y_{OUT\ 1}, y_{OUT\ 2}, \ldots, y_{OUT\ L}]$ of size L, where K is not necessarily equal to L, as visualized in Fig 2.2. Note that in this figure, each neuron is

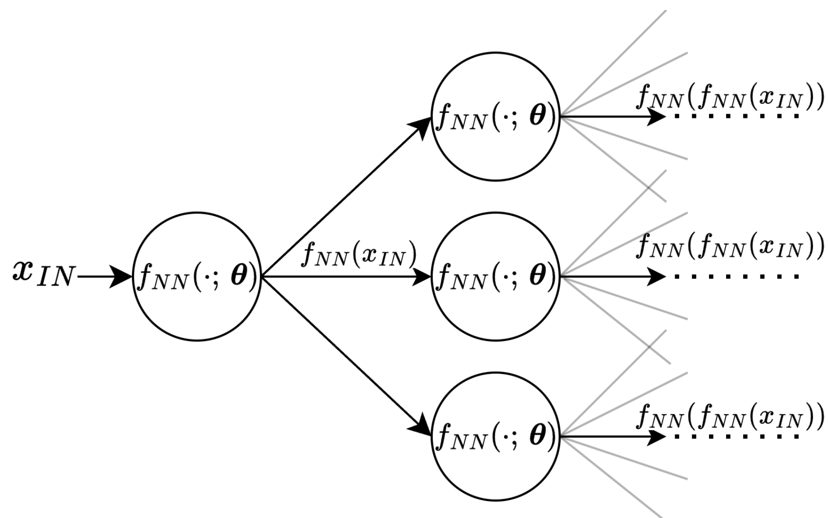


Figure 2.1: The path of data from input to a neuron in an ANN towards other neurons in the network, each neuron equipped with a set of function parameters θ.

connected to each of the neurons in the subsequent layer, although this is not necessarily the case with ANNs. We call such neural networks fully connected.

Artificial NNs are utilized for modeling relationships in data consisting of inputs and outputs by way of approximating these relationships as a form of statistical regression. The fundamental aspect of ANNs which enables this regression to manifest from the process of modifying its internal neuron parameters is that the function they utilize is nonlinear. The universal approximation theorems are a group of existence theorems that state that arbitrarily complex combinations of non-polynomial functions can theoretically approximate any continuous function at a desired degree of accuracy, with more complex combinations being more capable in this respect than simpler ones [19]. In context of the ANN, regression of complex relationships found in non-closed-form real-world data is made possible by increasing depth of the neural network (i.e. the number of layers) or width of the network (the number of neurons in the layers).

2.1.1 Feedforward neural networks

A feedforward neural network (FNN) is the most foundational ANN architecture, named for the unidirectional flow of data from input to output with no feedback loops or cycles. More formally, an FNN is organized into an ordered sequence of layers: an input layer receiving the input vector $\overrightarrow{x_{IN}}$, one or more hidden layers performing intermediate transformations, and an output layer producing the prediction $\overrightarrow{y_{OUT}}$, as visualized in Fig. 2.2. Each neuron in a given layer receives the outputs of all neurons in the preceding layer, transforms them via an activation

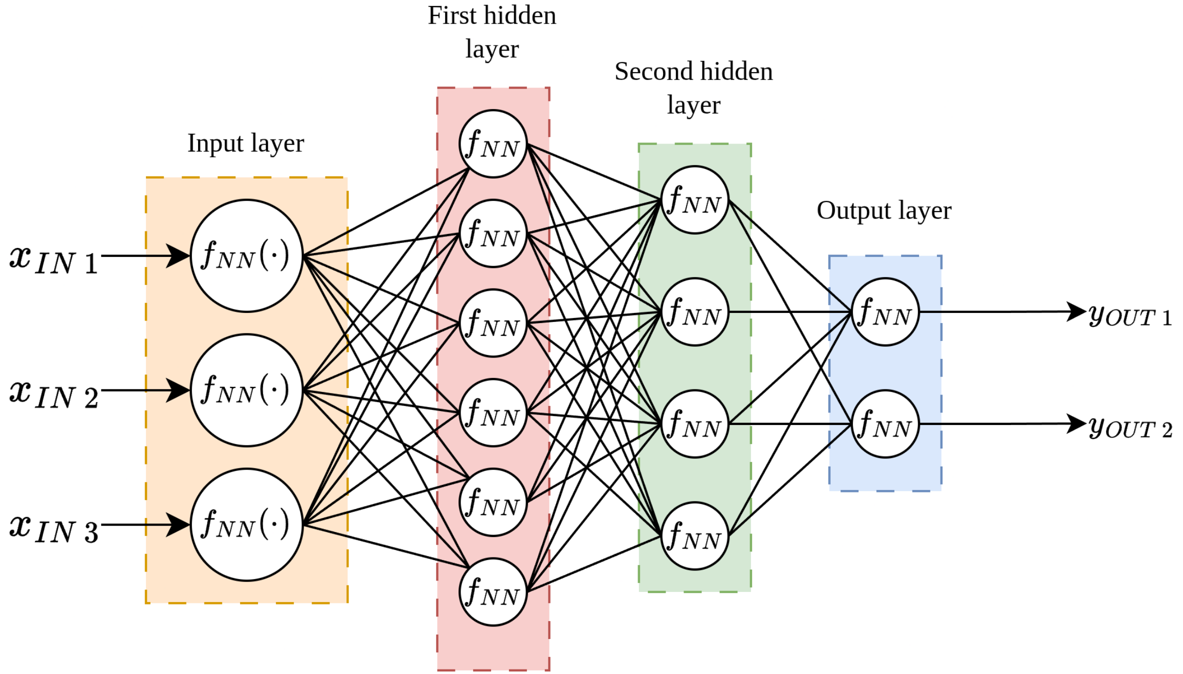


Figure 2.2: Example of a fully connected artificial neural network, with three inputs and two outputs. Each neuron computes $f_{NN}(\cdot; \boldsymbol{\theta}_i)$, where $\boldsymbol{\theta}_i$ denotes the parameters (weights and bias) of neuron i.

function $f_{NN}(\cdot; \boldsymbol{\theta}_i)$, and passes the result forward.

The computation performed by a single neuron H_i^ℓ in hidden layer ℓ can be written explicitly as:

$$H_i^\ell = \sigma\left(\sum_j W_{[i,j]}^\ell \cdot H_j^{\ell-1} + B_{[i]}^\ell\right),$$

where $W_{[i,j]}^\ell$ is the weight of the edge connecting neuron j in layer $\ell-1$ to neuron i in layer ℓ, $B_{[i]}^\ell$ is a scalar bias term associated with neuron i, and $\sigma(\cdot)$ is the activation function. The weights and biases collectively constitute the parameter set $\boldsymbol{\Theta}$ of the network, and it is these parameters that are adjusted during training to fit the model to observed data. The output layer applies an analogous linear combination, commonly without an activation function, to produce the final prediction.

The choice of activation function $\sigma(\cdot)$ is central to the expressive power of the network. The most widely adopted activation function in modern FNNs is the Rectified Linear Unit (ReLU), defined as $\mathrm{ReLU}(z) = \max(0, z)$. Its piecewise-linear nature provides nonlinearity sufficient to enable universal approximation while remaining computationally tractable and amenable to formal analysis, a property that is directly relevant to the verification approach taken in Paper A. The input and output layers conventionally apply no activation function, i.e., the identity, so that

the network's input and output values are not distorted at the boundary.

Training an FNN proceeds by presenting the network with labeled input-output pairs and adjusting $\mathbf{\Theta}$ to minimize a loss function $\mathcal{L}(\mathbf{\Theta})$ measuring the discrepancy between predicted and true outputs. This is achieved via backpropagation, which computes the gradient of $\mathcal{L}$ with respect to each parameter by the chain rule, and a gradient-descent optimizer then updates the parameters in the direction that reduces the loss. Because the network computes predictions strictly from the current input $\overrightarrow{x_{IN}}$ without reference to any prior inputs, FNNs are well-suited to problems in which inputs are independently and identically distributed and no temporal dependency exists between successive observations. Note that these inputs can be in the form of tabular ordered data or otherwise arbitrarily formatted inputs, such as text, images or (multidimensional) matrices, so long as the network is adapted to handle these input shapes. Additionally, we talk often discuss the regressive capabilities of NNs, however adding additional layers, such as the SoftMax function on top of NN outputs enables various modes of operation, such as classification of inputs.

2.1.2 Long short-term memory networks

As noted above, feedforward neural networks process each input independently and carry no memory of prior inputs. This makes them unsuitable for tasks in which the output depends not only on the current input but on a sequence of preceding inputs, where the temporal ordering of observations is itself informative. Such problems call for a recurrent architecture, in which the network maintains an internal state that is updated at each time step and carries information forward through the sequence. The most straightforward recurrent architecture, the base recurrent neural network (RNN), achieves this by feeding a hidden state h_{t-1} (passed through an activation function, most commonly $\tanh$) from the previous time step alongside the current input x_t into each computation step, producing an updated hidden state h_t and an output, as can be seen in Fig. 2.3.

Formally, the prediction at each step depends on the entire history of inputs up to that point, i.e., $p(\mathbf{y} \mid (\mathbf{x}_t, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_{t-k}); \mathbf{\Theta})$, where k denotes the effective depth of the temporal dependency. In practice, however, RNNs struggle to capture long-range dependencies. The repeated application of the same weight matrices during backpropagation through time causes gradients to either shrink toward zero (the vanishing gradient problem of repeatedly exponentiating values lesser than 1.0) or grow without bound (the exploding gradient problem of repeatedly exponen-

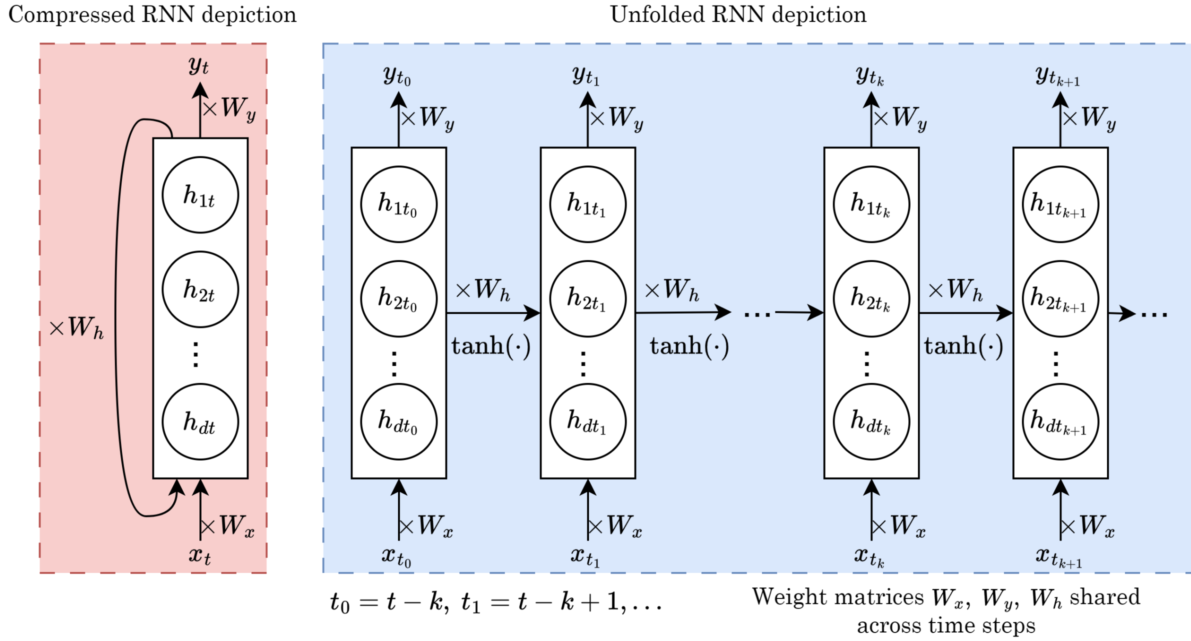


Figure 2.3: Compressed and unfolded depictions of a vanilla RNN processing a sequence of inputs $x_{t_0}, \ldots, x_{t_{k+1}}$, with shared weights W_x, W_h, and W_y across time steps.

tiating values greater than 1.0), both of which prevent the network from learning dependencies that span many time steps [20].

The long short-term memory (LSTM) network, introduced by Hochreiter and Schmidhuber [20], addresses this limitation through a fundamentally different internal cell structure. Rather than relying on a single hidden state to carry all temporal information, an LSTM cell maintains two separate quantities: a hidden state h_t, which serves as the short-term output of the cell, and a cell state c_t, which acts as a long-term memory that can persist across many time steps with only minor modification at each step. The flow of information into and out of the cell state is regulated by three learned, nonlinear gating mechanisms, as illustrated in Fig. 2.4.

The *forget gate* f_t decides what fraction of the previous cell state c_{t-1} to retain:

$$f_t = \sigma(W_f \cdot [h_{t-1},\, x_t] + b_f)\,,$$

where $\sigma(\cdot)$ denotes the sigmoid function, W_f and b_f are learned parameters, and $[h_{t-1}, x_t]$ denotes the concatenation of the previous hidden state and the current input. The *input gate* i_t and a candidate cell vector $\tilde{c}_t$ together determine what new information is written to the cell state:

$$i_t = \sigma(W_i \cdot [h_{t-1},\, x_t] + b_i)\,, \qquad \tilde{c}_t = \tanh(W_c \cdot [h_{t-1},\, x_t] + b_c)\,.$$

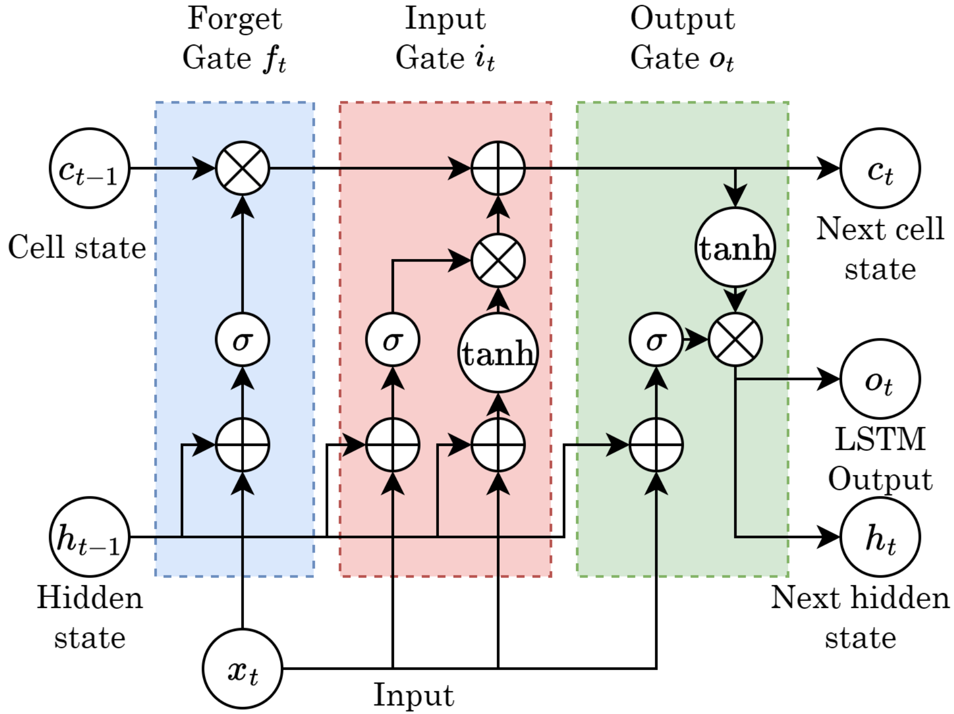


Figure 2.4: LSTM with labeled cell architecture

The cell state is then updated as an elementwise combination of the retained old state and the new candidate:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t,$$

where $\odot$ denotes elementwise multiplication. Finally, the *output gate* o_t determines which portion of the cell state is exposed as the hidden state:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o), \qquad h_t = o_t \odot \tanh(c_t).$$

Because the cell state c_t is updated additively and its gradient path bypasses the repeated weight multiplications that afflict vanilla RNNs, the vanishing gradient problem is substantially mitigated, enabling the network to learn dependencies across sequences of considerable length [21].

This architecture makes LSTMs well-suited to the problem addressed in Paper C, in which the target quantity, the distribution of cache misses across a program's execution, depends on the sequence of previously executed instructions. The memory access behaviour of a program at any given point is not independent of what has executed before it; rather, it reflects patterns such as repeated loop bodies, data locality structures, and instruction fetch sequences that unfold over many consecutive steps. An FNN, operating on individual instruction snapshots, would be unable to capture these dependencies. By contrast, an LSTM operating on subsequences of the

execution trace retains a learned summary of prior instructions in its cell state, allowing it to condition each prediction on the relevant execution history.

2.2 Neural network verification and abstraction

At present, we are observing a staggering increase in the prevalence of deep-learning based mechanisms, such as NNs, being utilized for increasingly important tasks. These tasks range from healthcare, malware detection, self-driving vehicles to credit approvals and more. As mentioned, NNs, through a process of statistical regression learned from data, can come to represent highly complex nonlinear functions often impossible to formalize in closed form. Though the individual computations that occur within NN nodes are relatively simple, the overall functioning of the mechanism remains extremely difficult to do and is in fact an NP-hard problem [22]. This begs the question: how can we know that NNs will not generate incorrect outputs? Since NNs have effectively infinite possible inputs, irrelevant of how many test inputs have been verified to correctly evaluate, it is always possible to generate adversarial inputs, i.e. inputs that are extremely difficult to evaluate manually as incorrect [23], such as adding noise to an image otherwise easily classified by the network, or imperceptibly changing a single numerical value in an input vector that changes the output dramatically in an undesired direction [24]. Since, therefore, we cannot settle for testing NNs, we desire instead formal guarantees of NN behavior, and more specifically, we desire a process for verification of behavioral properties that an NN is expected to satisfy.

2.2.1 Formal verification of neural networks

The three most significant types of properties that are most sought after in NNs are *robustness*, *safety* and *consistency* [25].

Robustness of an NN model is the ability of the NN model to preserve its output modeling performance under finite bounded perturbations of its inputs. Robustness is typically the most sought after property that NNs are functionally required to satisfy, due mainly to two reasons: it is a highly generic property of engineered systems (i.e. there should be resilience in a system's functioning to small changes in the inputs to the system) and secondly, because NNs are notoriously prone to robustness violations [26]. This property, or lack thereof, can have more or less far-reaching effects, depending on the use-case within which the NN is placed. Images of

muffins having small patches of chocolate should not make the classifier believe that it is seeing an image of a yellow puppy, and small inaudible noises added to an audio recording should not make the audio-classifier unable to detect speakers. More critically, it has been shown that visual deep-learning based classifiers in autonomous vehicles suffered from being unable to detect traffic signs on the open road when the signs were slightly vandalized (e.g. sprayed by graffiti or have a sticker on them) [27], something that would be nearly imperceptible to the human operator.

Safety may in NNs be defined as the inability of NNs to lead an operator (in this case, the operator that is using NNs as decision-making systems) into a bad state. A well known example in literature is the implementation of an NN-based aircraft collision avoidance system (A-CAS) [28], and in scenarios like these, if an aircraft is approaching from the left, the NN model must decide to turn the aircraft right in all possible ways of it approaching.

Consistency, is the property of NNs that enable them to, colloquially speaking, produce similar outputs for similar inputs, without dramatic unexpected output shifts that appear seemingly out of nowhere. For a concrete example, an NN model predicting CPU load might assign a higher predicted load to a task configuration that is a strict subset of a heavier one, which violates the monotonicity that domain knowledge and training set measurements dictate should hold.

Having established the properties we wish NNs to satisfy, the question becomes how to express them precisely enough for formal analysis. Verification of an n-layer network f with input $\mathbf{x} \in \mathcal{D}_{\mathbf{x}} \subseteq \mathbb{R}^{k_0}$ and output $\mathbf{y} \in \mathcal{D}_{\mathbf{y}} \subseteq \mathbb{R}^{k_n}$ requires a *specification* ϕ that formalizes the desired input-output behavior. Such a specification takes the general form: for all $\mathbf{x} \in \mathcal{X}$, the output $f(\mathbf{x})$ lies in $\mathcal{Y}$, where $\mathcal{X} \subseteq \mathcal{D}_{\mathbf{x}}$ constrains the inputs of interest and $\mathcal{Y} \subseteq \mathcal{D}_{\mathbf{y}}$ constrains the acceptable outputs. Verification then amounts to proving that $\mathbf{x} \in \mathcal{X} \Rightarrow f(\mathbf{x}) \in \mathcal{Y}$ [25]. The three properties described above map directly onto this framework: a robustness specification constrains $\mathcal{X}$ to a bounded neighborhood around a nominal input and $\mathcal{Y}$ to the set of outputs preserving the correct classification; a safety specification constrains $\mathcal{X}$ to a set of operationally relevant scenarios and $\mathcal{Y}$ to the set of outputs that do not lead to a hazardous state; and a consistency specification encodes a relational constraint, such as monotonicity, that must hold across pairs of inputs in $\mathcal{X}$. Significant work has been done in the field of formal NN verification [13], with many tools arising [29]. Of particular interest is the Marabou verification tool [14], which can prove a linear bound on the output of an NN given constraints on its

inputs, provided the NN uses piecewise-linear activation functions. Concretely, a Marabou query consists of a set of input constraints $\mathcal{X}$ and a desired output bound, and Marabou either returns a counterexample (an assignment to the inputs that violates the bound) or confirms that no such assignment exists, thereby proving the bound holds over the entire input region.

Example 1. *Consider a network with two inputs $x_1, x_2 \in [0, 10]$, one hidden layer with two ReLU neurons H_1, H_2, and a single output node O, with weights $e(x_1, H_1) = 2$, $e(x_2, H_1) = 0$, $e(x_1, H_2) = 0$, $e(x_2, H_2) = 1$, $e(H_1, O) = 1$, $e(H_2, O) = 1$, and all biases zero. The output is then $\mathrm{NN}(\mathbf{x}) = \mathrm{ReLU}(2x_1) + \mathrm{ReLU}(x_2) = 2x_1 + x_2$, which over $[0, 10]^2$ attains a maximum of 30. To verify that the output never exceeds 31, one poses the query $\mathrm{NN}(\mathbf{x}) > 30$ subject to $x_1, x_2 \in [0, 10]$; in practice encoded as $-\mathrm{NN}(\mathbf{x}) \leq -30$ since Marabou requires constraints in less-than-or-equal form. If Marabou returns unsatisfiable, the bound is proven.*

This query mechanism is directly exploited in Paper A, where Marabou is applied to a *difference network* $\mathrm{NN}_i^{\mathrm{diff}}$ to establish an upper bound on the maximum change in output induced by varying a single input x_i, forming the basis for the input significance analysis described in Section 8.

2.2.2 Abstraction-refinement for neural networks

Verification of NN properties by way of constraint queries over individual inputs, although a very powerful technique, runs into the scaling problem relatively quickly.

Example 2. *Consider a single-neuron network $y = \mathrm{ReLU}(5x_1 + 3x_2 - 1)$ with $x_1, x_2 \in [1/5, 1/3]$. We wish to verify that $y \leq 4$ over this input region. The network encoding ψ and specification ϕ are:*

$$\psi \triangleq (5x_1 + 3x_2 - 1 > 0 \Rightarrow y = 5x_1 + 3x_2 - 1) \ \wedge \ (5x_1 + 3x_2 - 1 \leq 0 \Rightarrow y = 0)$$
$$\phi \triangleq \forall x_1, x_2 \in [1/5,\ 1/3] : \psi \Rightarrow y \leq 4$$

Each implication in ψ is logically equivalent to a disjunction, giving the equivalent form:

$$\psi \ \equiv \ (5x_1 + 3x_2 - 1 \leq 0 \ \vee \ y = 5x_1 + 3x_2 - 1) \ \wedge \ (5x_1 + 3x_2 - 1 > 0 \ \vee \ y = 0)$$

Although each disjunct is a linear constraint, their union is non-convex. To see why, consider the two feasible sets induced by the ReLU: $A = \{(t, y) : t \leq 0\}$, a half-space, and $B = \{(t, y) : y = t\}$, a hyperplane, where $t = 5x_1 + 3x_2 - 1$. Both A and B are individually convex, but

their union is not. Taking $(-1, 5) \in A$ *and* $(2, 2) \in B$*, their midpoint* $(0.5, 3.5)$ *satisfies neither* $t \leq 0$ *nor* $y = t$*, and therefore lies in neither set.*

Consequently, constraint-based solvers such as Marabou must handle ψ by exhaustive case splitting over ReLU activation patterns: for each neuron, the solver separately considers the inactive branch ($t \leq 0,\ y = 0$) and the active branch ($t > 0,\ y = t$). For a network with n ReLU neurons this yields up to 2^n cases, which is what drives the NP-hardness of NN verification [22].

The exponential case-splitting cost of direct verification motivates the use of *abstraction*, which trades exactness for scalability while preserving soundness. The key idea is to replace the exact reachable set $f(\mathcal{X})$ with an over-approximation $\hat{f}(\mathcal{X})$ such that $f(\mathcal{X}) \subseteq \hat{f}(\mathcal{X})$, and reason about sets of behaviors rather than individual executions. Formally, soundness is guaranteed by construction: any property proven on $\hat{f}(\mathcal{X})$ holds for the real network f. Instead of enumerating ReLU activation patterns exhaustively, abstraction collapses the exponential case splits into convex relaxations. This enables polynomial-time propagation of bounds.

The cost of this approach is incompleteness: since $f(\mathcal{X}) \subseteq \hat{f}(\mathcal{X})$ but $\hat{f}(\mathcal{X}) \not\subseteq f(\mathcal{X})$, the abstract network may exhibit spurious behaviors that the original network cannot, and may therefore fail to prove properties that do actually hold. However, this is acceptable in the verification context: false positives are permitted, whereas false negatives are not. The result is a tractable conservative relaxation that preserves correctness guarantees and provides the foundation for the abstraction-refinement methodology of Elboher et al. [15]. The authors introduced an abstraction-refinement methodology that constructs a smaller network NN' such that $NN(x) \leq NN'(x)$ for all inputs x, enabling faster verification. The core idea is to classify hidden nodes by their contribution direction ("green" for output-increasing, "red" for output-decreasing) and merge nodes of the same color. Coloring proceeds backwards from the output node: a node H_i^ℓ is *green* if all its outgoing edges lead to green successors with non-negative weights or red successors with non-positive weights, and *red* in the symmetric case; nodes that satisfy neither condition are *colorless*. Colorless nodes are resolved by *splitting*: the node is replaced by two copies with identical incoming weights, one inheriting the edges that increase the output (green) and the other those that decrease it (red), leaving the network's computed function unchanged. Once all nodes are colored, nodes of the same color can be merged, yielding a strictly smaller over-approximating network NN' whose verified upper bounds are guaranteed to hold for the original NN. Paper A reuses this coloring and splitting procedure but applies it to the *input layer* rather than hidden nodes, which is what enables the input significance analysis

described in Paper A.

2.3 Conformal prediction and uncertainty quantification

A point forecast from a machine learning model carries no information about how much that prediction should be trusted. In industrial deployment, this is rarely sufficient: an engineer deciding whether a task configuration is safe to deploy on a real-time processor needs to know not just the expected CPU load, but how uncertain that estimate is. *Uncertainty quantification* (UQ) is the field concerned with producing such estimates alongside model predictions, typically in the form of prediction intervals $[\hat{y}_{\mathrm{low}}, \hat{y}_{\mathrm{high}}]$ that are intended to contain the true value with some specified probability.

Several families of methods exist for UQ in regression. Bayesian approaches, such as Bayesian neural networks and Gaussian processes, model uncertainty by placing a prior distribution over model parameters and computing a posterior after observing data; the predictive uncertainty then follows from integrating over this posterior [30]. Ensemble methods, such as deep ensembles [31], train multiple models with different initializations and treat the spread of their predictions as a proxy for uncertainty. Monte Carlo Dropout applies dropout at inference time to simulate sampling from an approximate posterior [32]. While these approaches are widely used, they share a common limitation: their uncertainty estimates are only as reliable as the modeling assumptions they rest on, and none provides a finite-sample, distribution-free guarantee that the produced intervals will contain the true value with at least the desired probability.

Conformal prediction (CP) occupies a distinct position among UQ methods precisely because it provides such a guarantee without requiring any assumptions about the data distribution or the model architecture, making it applicable as a post-hoc wrapper around any pre-trained predictor. This model-agnostic property was a decisive factor in the design of Paper B, where there existed an extreme sparsity of the observed task configurations, more specifically 7,129 samples out of $2^{27} \approx 134\,\mathrm{M}$ possibilities. This magnitude of sparsity makes the prior-dependent estimates of Bayesian methods untenable.

Conformal prediction (CP) is a distribution-free framework for uncertainty quantification in machine learning [33]. Unlike Bayesian methods, CP requires no prior distribution over the output space and makes no parametric assumptions about the data-generating process. Its sole

requirement is *exchangeability*: the calibration and test points must be drawn from the same distribution in a way that is invariant to permutation, a condition satisfied by any IID sampling procedure, such as the one in Paper B.

The core mechanism of CP is the *nonconformity score* $s(x, y)$, a scalar that measures how poorly a candidate label y fits a given input x according to a pre-trained model. Given a set of calibration points $\{(x_i, y_i)\}_{i=1}^n$ with computed scores $s_i = s(x_i, y_i)$, CP constructs a prediction set $\mathcal{C}(x)$ for a new input x such that the true label y is included with probability at least $1 - \alpha$:

$$\mathbb{P}\big(y \in \mathcal{C}(x)\big) \;\geq\; 1 - \alpha,$$

where $\alpha \in (0, 1)$ is a user-specified miscoverage level, typically taken to be $\alpha = 0.1$. This guarantee is *marginal* rather than *conditional*: it holds in expectation over the joint distribution of calibration and test points. This means that, if one were to repeat the procedure many times with freshly drawn data, the empirical coverage would converge to $1-\alpha$ (typically taken to be 0.9) on average. The term *marginal* reflects that the probability is taken by marginalizing over the input distribution $P(X)$, so the guarantee says nothing about coverage for any particular input value or subgroup. A *conditional* guarantee, by contrast, would require $\mathbb{P}(y \in \mathcal{C}(x) \mid X = x) \geq 1-\alpha$ to hold for each specific x, conditioning on the input rather than averaging over it. Vovk [34] showed that exact conditional coverage is in general unachievable with finite samples without additional assumptions, which is why marginal coverage is the standard guarantee offered by CP. In practice, this means that for a subpopulation of inputs that is systematically harder to predict (for example, rare task configurations in the Paper B setting) the actual coverage may fall below $1 - \alpha$ even when the marginal guarantee holds globally.

Crucially, CP is *finite-sample* and *distribution-free*, meaning it does not improve asymptotically nor require large samples to become valid, making it an excellent choice for industrially-motivated use-cases.

2.3.1 Split conformal prediction for regression

Split conformal prediction [33] partitions the available data into three disjoint sets: a *training* set used to fit the base model, a *calibration* set used to compute nonconformity scores, and a *test* set used for evaluation. In Paper B, this partition follows a 37.5/37.5/25% split across $N = 7{,}129$ observations, where each observation is a binary task configuration vector $\vec{f} \in \mathbb{B}^{27}$ paired with a mean CPU load measurement $y \in \mathbb{R}^3$ (core 0, core 1, and dual-core load).

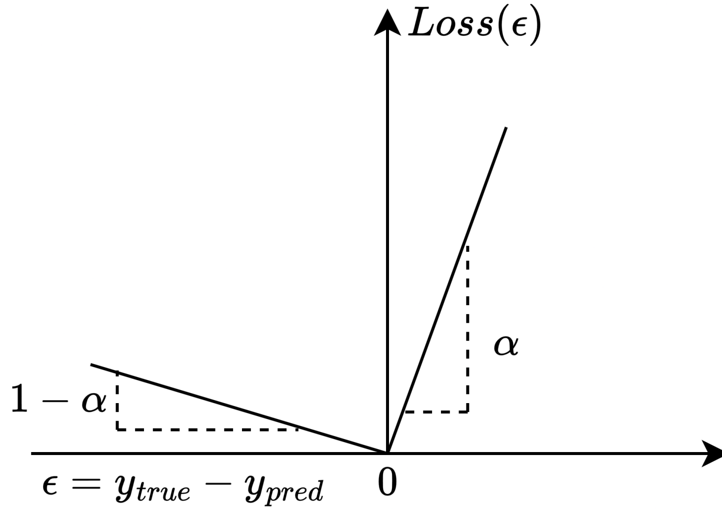


Figure 2.5: Pinball loss (Quantile loss) function

The procedure proceeds in four steps. First, a base neural network $\mathbf{M}$ is trained on $(X_{\text{train}}, y_{\text{train}})$ to produce point forecasts $\hat{y} = \mathbf{M}(x)$. Second, two quantile regressors are fitted on the training residuals: a lower regressor QR_{lower} targeting the $\alpha/2$ quantile and an upper regressor QR_{upper} targeting the $1-\alpha/2$ quantile, both trained on the asymmetric pinball (quantile) loss, seen in Fig. 2.5.

These regressors provide initial interval estimates but without coverage guarantees; the conformalization step corrects for this. Third, for each calibration point $(x_i, y_i) \in (X_{\text{cal}}, y_{\text{cal}})$, the nonconformity score is computed as:

$$s(x_i, y_i) = \max\bigl(\text{QR}_{\text{lower}}(x_i) - y_i,\ y_i - \text{QR}_{\text{upper}}(x_i)\bigr),$$

which measures how far the true label falls outside the initial quantile interval. Fourth, the corrected quantile

$$\hat{q} = \text{Quantile}\left(s_1, \dots, s_n;\ \frac{\lceil (n+1)(1-\alpha) \rceil}{n}\right)$$

is computed from the n calibration scores, where the $\lceil (n+1)(1-\alpha) \rceil / n$ level incorporates a finite-sample correction [33]. For a new input x, the final prediction interval is:

$$\text{Lower bound} = \text{QR}_{\text{lower}}(x) - \hat{q}, \tag{2.1}$$

$$\text{Upper bound} = \text{QR}_{\text{upper}}(x) + \hat{q}. \tag{2.2}$$

By construction, this interval satisfies the marginal coverage guarantee: for a fresh test point (x, y) drawn exchangeably with the calibration data,

$$\mathbb{P}\bigl(y \in [\text{Lower bound},\ \text{Upper bound}]\bigr) \ \geq\ 1-\alpha.$$

In Paper B, $\alpha = 0.1$, targeting 90% coverage. This guarantee holds regardless of the underlying distribution of CPU load values, the architecture of the base model M, or the density of the observed task configurations.

2.3.2 Shapley values for feature attribution

Shapley values originate in cooperative game theory [35] where the problem is to fairly distribute the total payoff of a coalition of players among its individual members. The central idea is that each player's fair share should equal their average marginal contribution across all possible orderings in which the coalition could have been assembled. Translated to machine learning, the "players" are input features, the "coalition" is the set of features provided to a model, and the "payoff" is the model's prediction. The Shapley value ϕ_i for feature i is then defined as:

$$\phi_i = \sum_{S \subseteq \Phi \setminus \{i\}} \frac{|S|!\,(|\Phi| - |S| - 1)!}{|\Phi|!} \big[\gamma(S \cup \{i\}) - \gamma(S)\big],$$

where Φ is the full set of features, S ranges over all subsets that do not include feature i, and $\gamma(S)$ is the model's expected output when only the features in S are active, with the remaining features marginalized out [36]. The combinatorial prefactor is the probability that coalition S would arise if features were added in a uniformly random order, ensuring that each ordering is weighted equally. The resulting values satisfy three axioms that together characterize a unique fair attribution: *efficiency* (the values sum to the total prediction minus a baseline), *symmetry* (equally contributing features receive equal values), and *null player* (a feature that changes no prediction receives zero).

Although Shapley values appear to be a silver bullet for feature attribution, calculating them is a very difficult problem. The sum over S in the Shapley formula ranges over all $2^{|\Phi|-1}$ subsets of $\Phi \setminus \{i\}$. For a model with $|\Phi| = 27$ binary features, as in Paper B, this is $2^{26} \approx 67\,\mathrm{M}$ subsets per feature, and $27 \times 67\,\mathrm{M}$ evaluations of $\gamma(\cdot)$ in total, each of which requires a forward pass through the model. Deng and Papadimitriou [37] showed that computing exact Shapley values is #P-hard in general, belonging to the complexity class of counting solutions to decision problems in the NP set. This places it in a complexity class widely believed to be harder than NP [38]. Practically, this means that exact computation is feasible only for shallow tree models with few features; for neural networks and other non-linear models, approximation is necessary.

The SHAP library [36] provides two principal approximations. *KernelSHAP* is model-

agnostic: it approximates Shapley values by fitting a weighted linear model over a sample of coalitions, where the weights are chosen to satisfy the Shapley axioms. The key practical caveat is that KernelSHAP treats absent features by marginalizing over the training data distribution. Here, $\gamma(S)$ is estimated by replacing the absent features with samples drawn from their marginal distribution, rather than conditioning on the present features. This can introduce error when features are correlated, since the marginalization implicitly assumes independence among features. *TreeSHAP* [39] exploits the recursive structure of decision trees and tree ensembles to compute exact Shapley values in polynomial time, avoiding the sampling approximation entirely. However, TreeSHAP is only applicable to tree-based models. Paper B uses KernelSHAP, as the base model $\mathbb{M}$ is a neural network; the correlation caveat is partially mitigated by the ablation study in Paper B, which cross-validates attributions by measuring the effect of withholding individual tasks on the CPU load directly.

However, a crucial aspect to keep in mind is that, by itself, attribution does not imply causation. A Shapley value quantifies how much a feature *contributes* to a model's prediction, not how much it *causally influences* the underlying system being modeled. This distinction matters in practice. If two features are correlated the model's predictions will depend on both, but the credit for the shared variance will be split between them in a way that reflects their statistical co-occurrence in the training data rather than any causal mechanism. A feature may receive a high Shapley value simply because it is a proxy for a third, unobserved variable that is the true driver of the outcome. Conversely, a feature that is genuinely causal may receive a low Shapley value if it is collinear with another feature that the model uses instead. In the Paper B context this is illustrated by the correlation between FUNC 12 and FUNC 13 documented in the ablation study: ablating FUNC 12 shifts the attributed contribution of FUNC 13, indicating that the two share explanatory credit rather than acting independently. Shapley values therefore provide *associative* explanations that are useful for understanding model behavior and for communicating which task configurations drive load estimates, but they should not be interpreted as identifying the tasks that, if individually removed, would cause the load to decrease by the attributed amount.

2.4 Cache memory and hardware performance simulation

The performance of software running on modern processors is fundamentally shaped by how well the memory system keeps up with the processor's demand for data. Understanding this relationship requires a brief account of how memory is organized, why caches exist, and what happens when they fail to serve a request in time.

2.4.1 Memory hierarchy and cache operation

Modern processors operate on data held in *registers*: compact, fixed-width storage locations (typically 32 or 64 bits) that the processor can access in a single clock cycle. Registers are the fastest storage available, but there are typically very few of them. A typical x86-64 (the 64-bit extension of the x86 instruction set architecture, the dominant instruction-set architecture in desktop and server processors [40]) processor exposes sixteen general-purpose 64-bit registers to the programmer [41]. Programs must therefore load data from *main memory* (DRAM) when it is not already in a register. DRAM offers gigabytes of capacity, but it lies physically and architecturally outside the processor die, and accessing it introduces latencies on the order of tens to hundreds of nanoseconds, often exceeding two hundred clock cycles on a modern processor running at several gigahertz [40]. This disparity between processor speed and memory latency, which has widened steadily over decades as CPU clock rates have grown faster than DRAM access times, is commonly referred to as the *memory wall* [42].

Cache memory is the principal hardware solution to this latency gap. It is a set of fast, on-die SRAM memories arranged in a hierarchy of levels, each larger and slower than the one before it. The L1 cache is split into a data cache (L1D) and an instruction cache (L1I), each typically a few kilobytes in size, and can be accessed in two to five clock cycles. The L2 cache is unified and larger, typically tens to hundreds of kilobytes, with access latencies of around ten cycles. The last-level cache (LLC or L3) is shared across cores in multicore processors, may range from a few megabytes to tens and even hundreds of megabytes in contemporary server processors, and serves as the final buffer before a request must be escalated to DRAM [40]. When a processor needs data, it checks L1D (or L1I for instructions), then L2, then LLC, and only issues a DRAM request if all levels fail. This sequence is known as a *cache miss* at the LLC level. The processor pipeline stalls until the requested data is returned, which is what makes high miss rates directly observable as degraded application throughput.

Cache misses are conventionally classified into three categories. *Compulsory misses* (cold misses) occur the first time a memory address is accessed, since the data has never been loaded into any cache level. *Capacity misses* arise when the working set of a program is too large to fit in the cache, so previously loaded data is evicted before it can be reused. *Conflict misses* are an artifact of cache associativity: a set-associative cache maps each memory address to a limited number of *ways* within a *cache set*, and if more addresses than ways map to the same set, eviction is forced even when the overall cache is not full. The relative contribution of each miss type depends on the program's memory access pattern and on the cache configuration, which is precisely why varying cache size and core count produces meaningfully different miss distributions, and why those distributions are informative targets for a predictive model.

Cache performance is of particular significance in industrial software running on server-class and embedded processors. Workloads that process large data structures such as databases, signal processing pipelines and image analysis algorithms are frequently limited not by compute throughput but by memory bandwidth and cache efficiency [42, 43]. A program whose working set exceeds the LLC may stall on cache misses for a majority of its execution time; the same program running with a larger or better-configured cache may run several times faster. For hardware architects and performance engineers, predicting how miss rates change across cache configurations is therefore a core design-space exploration task, one that traditionally requires running a simulator for each configuration under study.

2.4.2 The DynamoRIO Cachesim simulator

DynamoRIO [10] is a dynamic binary instrumentation framework that operates by injecting analysis code into a running application at the machine-code level, without requiring source code or recompilation. Its `drcachesim` tool, referred to throughout this thesis as *Cachesim*, intercepts every memory access the application performs at runtime, records the accessed address and access type (instruction fetch, memory load, or memory store), and replays these accesses against a configurable cache hierarchy to compute hit and miss statistics at each level.

Instrumentation works in two stages. First, a tracer client is injected into the target process via DynamoRIO's code manipulation layer, which intercepts every memory reference and writes it to a trace buffer. Second, the accumulated trace is fed to the cache simulator, either online via a named pipe or offline from stored trace files. The simulator then processes each memory reference sequentially, updating the simulated cache state and tallying hits and

misses per cache level and per core. This approach is entirely microarchitecture-agnostic: because Cachesim operates on the memory reference stream rather than on a model of a specific processor pipeline, it does not require any description of the target CPU's internal structure. This property was central to choosing Cachesim over a more comprehensive simulator such as gem5 [44]. gem5 supports detailed, cycle-accurate simulation of full system models, but doing so requires specifying a concrete microarchitectural configuration, including pipeline width, branch predictor type, memory controller, and more. Since the goal of Paper C is to build a model that reasons about cache behavior across a range of configurations rather than replicating one specific processor, tying the data collection to a single gem5 system model would have constrained the generality of the approach. Cachesim avoids this by abstracting away all pipeline details and exposing only the cache hierarchy as a configurable parameter space.

The parameters that Cachesim accepts for the cache hierarchy are: L1D cache size and associativity, L1I cache size and associativity, last-level cache (LL) size and associativity, cache line size, core count, and replacement policy. The default values, as specified in the official DynamoRIO documentation, are: L1D size 32 KB, L1I size 32 KB, both with associativity 8; LL size 8 MB with associativity 16; cache line size 64 bytes; and LRU replacement policy. In Paper C, the cache sizes and core count were varied across a controlled grid as the experimental variables, while associativity, line size, and replacement policy were left at their defaults. This decision reflects the study's focus on the effect of capacity and parallelism on miss distributions, rather than on the subtler effects of replacement policy or set structure, which are left as directions for future work.

The principal limitation of dynamic instrumentation is overhead. Because DynamoRIO intercepts every memory access at runtime, instrumented execution is substantially slower than native execution. This is not specific to DynamoRIO: Cachegrind, one of the tools available in the Linux program profiler Valgrind [45], which operates on the same principle, runs at approximately $50\times$ slower than native execution by its own documentation, and DynamoRIO's drcachesim in online mode is known to be slower still. For a single program run on a single cache configuration this is acceptable. Evaluating a program across C configurations, however, requires C separate instrumented runs, each carrying the same overhead. In Paper C the training set spans 324 configuration-program combinations (3 L1D sizes $\times$ 3 L1I sizes $\times$ 3 LL sizes $\times$ 3 core counts $\times$ 4 benchmarks), each executed up to $K = 25 \times 10^6$ instructions. The data collection cost of this training corpus directly motivates the ML substitute: once trained, the

LSTM model accepts cache parameters as input features alongside the instruction trace and produces miss distribution predictions across all configurations in a single forward pass, without any further instrumented execution. DynamoRIO Cachesim thus plays a dual role in Paper C, as both the bottleneck being replaced and the ground truth against which the replacement is evaluated.

2.5 Scenario-based testing for automated driving systems

In this section we introduce the motivation behind the challenges of automated driving system validation, the shift from pure data-based to scenario-based testing approaches, the ASAM driving standard and the need for a more efficient scenario reconstruction mechanism.

2.5.1 Scenario-based testing

The validation of Automated Driving Systems (ADSs) has undergone a fundamental methodological shift. The classical approach of driving billions of miles on public roads and measuring the absence of failures is now widely understood to be statistically infeasible for establishing the safety of systems that must handle rare, safety-critical events. Wachenfeld et al. [46] demonstrated that the frequency of serious accidents in naturalistic driving data is so low that no practical test fleet could accumulate sufficient mileage to provide statistically meaningful coverage of the critical scenarios an ADS must handle.

This realization has driven the industry toward Scenario-Based Testing (SBT), in which the ADS is validated not against random mileage but against a curated library of concrete, safety-critical traffic situations [47, 48]. The methodology was formalized by the PEGASUS project [49], as part of which Menzel et al. [50] established the widely adopted three-level scenario taxonomy: *functional* scenarios describe situations in natural language; *logical* scenarios parameterize these descriptions with ranges and distributions; and *concrete* scenarios specify exact values for every parameter, producing a machine-executable test case. The VVM project [51] subsequently extended these frameworks to complex urban environments, cementing simulation as the cornerstone of industrial safety argumentation.

Populating scenario libraries at the required scale currently depends on two methods. Manual engineering requires domain experts to hand-script scenarios in tools such as IPG CarMaker or VIRES Virtual Test Drive, which produces high-fidelity results but is unscalable since ev-

ery parameter must be explicitly specified and maintained [52]. Sensor-based reconstruction converts LiDAR and camera logs from test fleets into simulation files, but is strictly limited to situations the fleet has actually encountered [53]. Neither approach can efficiently exploit the large corpus of accident reports such as police records, news articles or insurance filings that document thousands of safety-critical events daily in textual form. This data scarcity gap is the motivating context for Paper D.

2.5.2 OpenSCENARIO and OpenDRIVE

The executable format underpinning SBT is ASAM OpenSCENARIO [54], which defines dynamic scenario content via a hierarchical XML storyboard. A scenario is not a sequence of animation frames but a state machine: `Acts`, `ManeuverGroups`, `Maneuvers`, and `Events` are nested in a strict hierarchy, with each `Event` activated by a `Trigger` that fires when a runtime condition is met. The static road network is defined separately in a corresponding OpenDRIVE file specifying lane geometry and road topology. These two files must be mutually consistent: every entity's position, trajectory, and trigger condition must be expressed in a coordinate system derived from the road geometry, and every trajectory must be geometrically realizable within the lane structure the map defines.

This consistency requirement is the central challenge for programmatic generation. A single semantic action such as an overtaking maneuver requires coordinated updates across the road entity's initial position, a trajectory in the road's local coordinate frame, speed profile waypoints that respect the vehicle's physical limits, a trigger condition referencing the relative position of another entity, and a road network in which the required lane structure exists. Any inconsistency among these elements, a position outside valid lane bounds, a trigger referencing an entity absent from the catalogue, or a speed specified in the wrong units, causes the simulator to reject the file. For a human expert writing a scenario iteratively this is manageable; for a probabilistic generator such as an LLM, which is prone to losing consistency over long token sequences, it constitutes a systematic failure mode. Any locally plausible but globally inconsistent hallucination propagates into a file that is syntactically well-formed but physically or semantically invalid, and that a downstream simulator rejects without a clear error message. This is the precise failure mode the Judge loop in Paper D is designed to detect and repair. An emerging body of work applies LLMs to scenario synthesis from natural language; this literature is reviewed in Section 5.4.

Chapter 3

Research overview

This chapter provides a description of the research gaps, research goal, research questions, and the methodology used to conduct the research activities underlying the included papers.

3.1 Motivation and research gaps

Based on our analysis of the state of the art in applied ML, we have found a persistent discord between novel modeling techniques and their industrial application. Many methods remain at the "proof-of-concept" stage without integration into real-world systems. Sinha et al. [55] highlight the obstacles in deploying AI in industry; Schmitt [56] describes how deep models often face resistance in business settings; and Shahedi et al. [57] document how technically excellent tools fail to achieve adoption when trust or usability is lacking. Although a core component of our thesis work has initially been the exploration of ML techniques for solving various aspects of industrially inspired problems, we have also taken serious steps toward alleviating at least some of the concerns that precede satisfying and efficient ML adoption.

This exploration has yielded the following intertwined research gaps:

Gap A: Trustworthiness and explainability are under-addressed in ML for industrial-scale systems. A significant number of studies in ML applied to industrial systems emphasize point accuracy and heuristics, with comparatively fewer works providing statistical or formal guarantees suitable for operational decision-making, and fewer still pairing those guarantees with explainability such that engineers can act on them [58]. In a relevant work [33], Angelopoulos and Bates show distribution-free guarantees for ML outputs but without integration into

system-level workflows; Zhou et al. [59] note scalability challenges and the rarity of pairing output guarantees with practical industrial ML. This gap directly concerns the *trustworthiness* of ML outputs: without guarantees, engineers cannot responsibly act on predictions. It also concerns *explainability*, though in a specific and limited sense: not the interpretability of a network's internal reasoning, but the ability to state formally which inputs influence the output (Paper A) and to attribute individual predictions to input features (Paper B via Shapley values). We do not claim to open the black box of neural network decision-making; the explainability contribution is in the verification of input influence and in local, associative attribution, not in the network's reasoning process.

Gap B: Generalizable, fast, low-level performance modeling is sparse. Traditional simulators (e.g. PIN [60], Valgrind [61], DynamoRIO [10], Q-EMU [62] and gem5 [44]) are accurate but slow and expertise-heavy; ML substitutes that are both fast and generalize across workloads/hardware, though actively in development by researchers in the field, remain rare and often architecture-tied [63, 64, 65, 66]. In their work, [67] Renda et al. particularly pointed out how CPU simulators are often not unified generalizable mechanisms but rather abstraction of at most several composed or individual processor design concepts. This gap is primarily one of *efficiency*: the engineering cost of repeated simulation runs is the bottleneck, and a data-driven surrogate that generalizes across configurations addresses it directly. *Trustworthiness* enters through the requirement that the surrogate be evaluated rigorously against the simulator it replaces, including on configurations outside the training range.

Gap C: End-to-end, usable integration is scarcely treated. Many works address individual components: uncertainty quantification, explainability, performance modeling, or robustness, yet few combine all those pieces in operational pipelines evaluated under the full suite of constraints (latency, resource limits, human interpretability, deployment constraints, drift, etc.) [68, 69, 70]. Even where individual components exist (uncertainty quantification, attribution, or ML-based performance models), integrated pipelines that

- start from the kinds of inputs engineers actually have,
- end in runnable artifacts and human-usable outputs and
- are evaluated as such

are limited [71]. Liang et al. [72] show in a novel approach how LLMs may be utilized for robotics control purposes, however with domain limitations. Huang et al. [73] show that the translation from natural language to actions but without full integration into human workflows. Nouri et al. [74] showcase the potential of simulation-guided LLM code generation for use-case-specific software with expert validation, while Xiao et al. [75] showcase a multi-level generative framework, illustrating modularity but lacking full end-to-end usability. This gap spans *efficiency*, in reducing manual engineering effort through automation, and *trustworthiness*, in ensuring that generated artifacts are validated against deterministic criteria rather than accepted on faith. Explainability is not a primary concern in this gap.

Table 3.1 summarizes which contributions address which gaps.

Table 3.1: Mapping of research gaps to contributions.

Gap	**Addressed by**
Gap A: Trustworthiness and explainability	Formal input reduction, Guaranteed load forecasting
Gap B: Fast, generalizable performance modeling	Cache surrogate modeling
Gap C: End-to-end integration	Scenario compilation (HASCO)

3.2 Research goal

Taken together, the gaps highlight that current research provides valuable isolated contributions yet falls short of offering a coherent methodology. Addressing this requires a unifying research direction that explicitly targets trustworthiness, automation, and usability at scale. This leads to the following research goal:

> **RG:** Design and evaluate a trustworthy ML method for industrial-scale automation, with explainability and workflow integration.

3.3 Research questions

To realize the research goal, we pose four research questions, each motivated by the industrial and research challenges identified in Gaps A–C, and addressed by the included papers.

RQ 1: How can the influence of individual inputs on a neural network's output be identified with formal or principled guarantees? Engineers demand clarity on which inputs drive a model's predictions. Most attribution methods are heuristic: they produce plausible explanations without stating how reliable those explanations are. For safety- or cost-sensitive applications this is insufficient. RQ 1 targets *trustworthiness* through guarantees on influence bounds and a limited form of *explainability* through identifying *which* inputs matter, though not *how* the network uses them internally. The boundaries of what 'explainability' means here are stated in Gap A above. The contribution is in bounding and attributing input influence and not in exposing the network's internal reasoning. RQ 1 is addressed primarily by Paper A, which provides formal bounds via Marabou-verified difference networks, and secondarily by Paper B, which provides principled attribution via Shapley values grounded in game-theoretic axioms.

RQ 2: How can forecasting methods provide statistically guaranteed prediction intervals and per-feature attribution for deployment in industrial embedded systems? Most current forecasting tools provide only point predictions or heuristic uncertainty estimates. Classical approaches that offer guarantees are often too computationally heavy or opaque for real-time use. RQ 2 is the question that most directly addresses all three thesis concerns: *trustworthiness* via finite-sample coverage guarantees, *explainability* via task-level Shapley attribution that communicates prediction drivers to engineers, and *efficiency* via a framework lightweight enough for deployment on embedded hardware. RQ 2 is addressed by Paper B.

RQ 3: How can data-driven models replace expensive simulation processes while maintaining evaluated fidelity at both distributional and aggregate levels? Traditional simulation tools, whether instruction-level cache simulators or manual expert scenario scripting, are accurate but prohibitively slow or labour-intensive. ML-based surrogates promise speed, but a fast model that sacrifices the fidelity engineers need to make sound decisions is not useful. This question concerns both *efficiency*, replacing repeated costly processes with learned alternatives, and *trustworthiness*, requiring that fidelity be evaluated rigorously rather than assumed. The tradeoff between speed and fidelity is operationalized differently in each paper: in Paper C through the dual metrics of subsequence accuracy (MSE, RMSE, R^2) and aggregate reproduction error (E_{REP}), and in Paper D through the cost-benefit analysis that plots the quality score S_{avg} against token consumption across synthesis strategies. RQ 3 is addressed primarily by Paper C and secondarily by Paper D.

RQ 4: How can unstructured domain knowledge be transformed into executable, validated engineering artifacts with minimized expert intervention? Industrial workflows frequently depend on knowledge locked in unstructured formats: natural language reports, legacy documentation, expert notes. Translating this knowledge into artifacts that automated tools can consume is currently a manual, expert-driven bottleneck. This question targets *efficiency*, automating a process that currently requires skilled labour, and *trustworthiness*, requiring that generated artifacts be validated through deterministic checks rather than accepted on faith. RQ 4 is addressed by scenario compilation (HASCO), which compiles natural language accident reports into executable OpenSCENARIO simulation artifacts with deterministic validation.

3.4 Research methodology

The research presented in this thesis follows a design science methodology [76, 77] complemented by mixed-methods evaluation [78]. The central orientation is toward the creation and validation of artifacts that address industrially and research-motivated problems, following the cycle of *build*, *evaluate*, and *reflect*. This integration of design, evaluation, and iterative refinement aligns with what Holz et al. [79] describe as characteristic of computing research: the blending of design, experimentation, and reflection. The research method is visualized in Figure 3.1.

3.4.1 Formal input reduction

The first line of research followed what Wieringa [76] terms *technical action research*: an artifact (the difference network construction) is built, then assessed on benchmark models to test its ability to identify inputs of negligible influence. Evaluation combined the following formal proof obligation: ensuring $NN_i^-(\mathbf{x}^{-i}) \leq NN(\mathbf{x}) \leq NN_i^+(\mathbf{x}^{-i})$ for all $\mathbf{x}$, with empirical tests of accuracy preservation after reduction. Paper A is primarily a formal methods contribution: the algorithms and their correctness proofs (Theorems 1 and 2) are the principal output. No large-scale empirical evaluation on industrial-size networks has been conducted; the current evaluation uses the running example presented in the paper. Benchmarking on larger networks is explicitly identified as future work and is acknowledged as a limitation in Chapter 6.

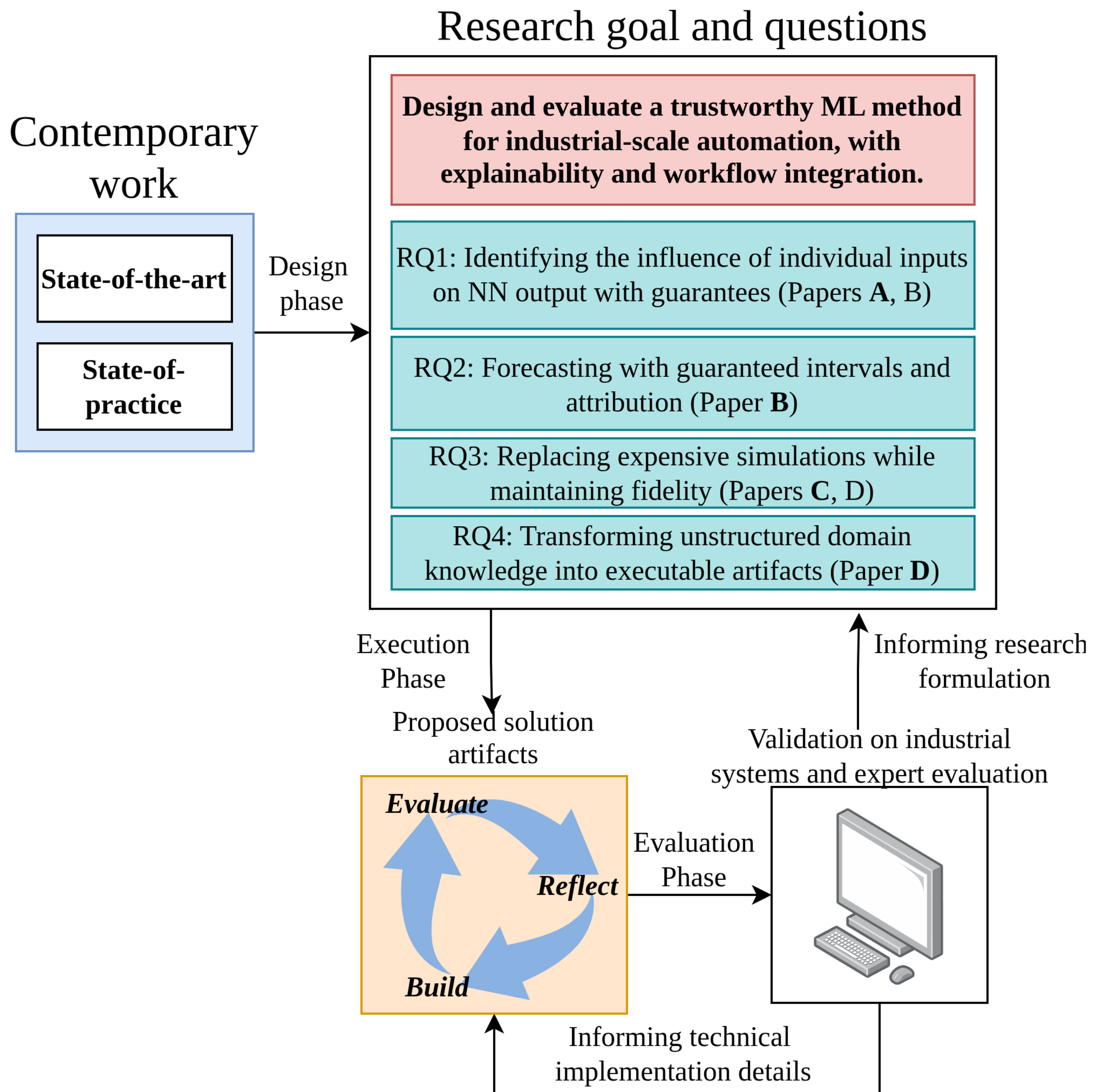


Figure 3.1: Thesis research process.

3.4.2 Guaranteed load forecasting

A forecasting pipeline was designed coupling conformal prediction [80, 33] with Shapley-based attribution [36]. Experiments were run on task-trace datasets collected from industrial hardware, with evaluation focusing on empirical coverage, interval width, and threshold exceedance detection. The Shapley component was validated by measuring consistency of attributions across ablations. A prototype GUI was built and tested with industrial partners. A formal user study with controlled participants was not conducted. However, industrial partners reviewed the prototype system and confirmed its usefulness for their configuration workflow; they have indicated intent to deploy a specialized version.

This constitutes an informal usability validation and is acknowledged as a limitation.

3.4.3 Cache data-driven modeling

The methodology reflects what Brown [81] calls *design as a search process*: exploring alternative model architectures, embedding strategies, and sequence lengths, then evaluating against DynamoRIO Cachesim ground truth. Models were trained on instruction traces, validated on Sysbench and image-processing workloads, and assessed on both subsequence fidelity (MSE, RMSE, R^2) and aggregate accuracy (E_{REP}). Generalization was explicitly tested on cache configurations unseen during training, including sizes outside the training range. Overfitting was mitigated through several design choices: a 90/10 train/validation split during model training, dropout between LSTM layers as a regularization mechanism, and evaluation on cache configurations entirely absent from the training set. The overestimation observed on the GFTT benchmark at extreme cache sizes (Paper C, Fig. 18) is consistent with extrapolation beyond the training distribution rather than overfitting to it, as an overfit model would perform well on interpolated configurations but poorly on all unseen ones, which is not the observed pattern.

3.4.4 Scenario compilation (HASCO)

The approach aligns with *design science for socio-technical systems* [82], as the pipeline not only generates artifacts (scenarios) but is designed to interface with human experts in a review workflow. Evaluation combined objective measures (executability rate determined by the esmini simulator) with a qualitative fidelity rubric applied by the research team, who assessed semantic alignment between generated scenarios and source accident reports. This constitutes

a *mixed-methods design* [78]. The reliance on researcher assessment rather than independent domain expert evaluation is acknowledged as a threat to validity in Chapter 6.

3.4.5 Cross-cutting methodological pattern - Reduce, Represent, Validate

Beyond the per-contribution methodologies described above, a shared architectural pattern emerged across all four lines of work. Each contribution follows a three-phase structure:

1. ***Reduce***: constrain the problem's dimensionality or scope to its essential structure. This must be done in accordance with the most significant aspects of the problem and the technical limitations of solving the problem.

2. ***Represent***: encode that structure in an intermediate form that is amenable to automated analysis. This must be done in a way that preserves what was identified as essential in Reduce, while ensuring that the next phase's assessment can be performed automatically.

3. ***Validate***: assess the result against a criterion that is independent of the method that produced it.

Figure 3.2 visualizes the pattern. The reduction process may be an investigative method of questioning the problem, or an automated mechanism by which the complexity of the problem naturally reduces. The intermediate representation is to be understood broadly: it may be code, a pipeline, a dataset, a database schema, a learned model, an ontology, or some combination of these, depending on what is necessary to solve the reduced core problem. The validation may take the form of analyzing the solution artifact as a whole, its individual aspects or merely the outputs. The three phases are not strictly sequential. Even within Represent, candidate representations are tried, prototyped, and informally assessed against how well they would support the eventual analysis, with poor candidates discarded before one is finalized. Take the following example:

Example 3. *A researcher is tasked by an industrial partner to establish an ML-based pipeline for predictive maintenance on a factory floor. The machinery on the floor is very complex, and its interfacing with the rest of the factory can impact its operating characteristics. The researcher must first attempt to* ***reduce*** *the problem to its solvable, approachable core. The researcher must ask: what are the technologies available for measurement, what is the budget the researcher is allowed for the pipeline deployment, what are specific aspects of the forecasted*

maintenance that cannot be reduced away, and so on. The researcher then analyzes both the state-of-the-art and state-of-practice with regards to the specific aspects of the core problem. Having done so, the researcher can start with prototyping core components of the solution, and later implementing a potential solution, which then manifests a technical and theoretical ***representation*** *of the core problem. Perhaps the researcher decides to utilize a dataset comprised of 3D laser scanning of factory products as an input to ML-enabled error detection. This representation is formal, structured and analyzable from a technical standpoint. The researcher then* ***validates*** *the representation, getting positive and negative results (do anomalous 3D scans actually correlate with future factory machinery breakdown and to what extent, could visual camera imaging do a better job?) and using the attained information as a structural critique to iterate upon.*

All three phases are present in every contribution, but the relative emphasis differs: some contributions derive their primary novelty from the reduction step, others from the representation or the validation mechanism. Table 3.2 maps each contribution to the research questions it addresses and indicates where the primary methodological weight falls within the three-phase pattern.

Table 3.2: Mapping of contributions to research questions and methodological phases. "•" = primary, "○" = secondary.

	RQ 1	**RQ 2**	**RQ 3**	**RQ 4**	**Reduce**	**Represent**	**Validate**
Formal input reduction	•				•	○	○
Guaranteed load forecasting	○	•			○	•	•
Cache data-driven modeling			•		•	•	○
Scenario compilation (HASCO)			○	•	○	•	•

This pattern ensures separation of concerns: the data-driven component handles learning from data, extracting patterns, and managing ambiguity, while the validation phase enforces constraints, provides guarantees, and catches inconsistencies. The pattern is examined in further detail in Chapter 6.

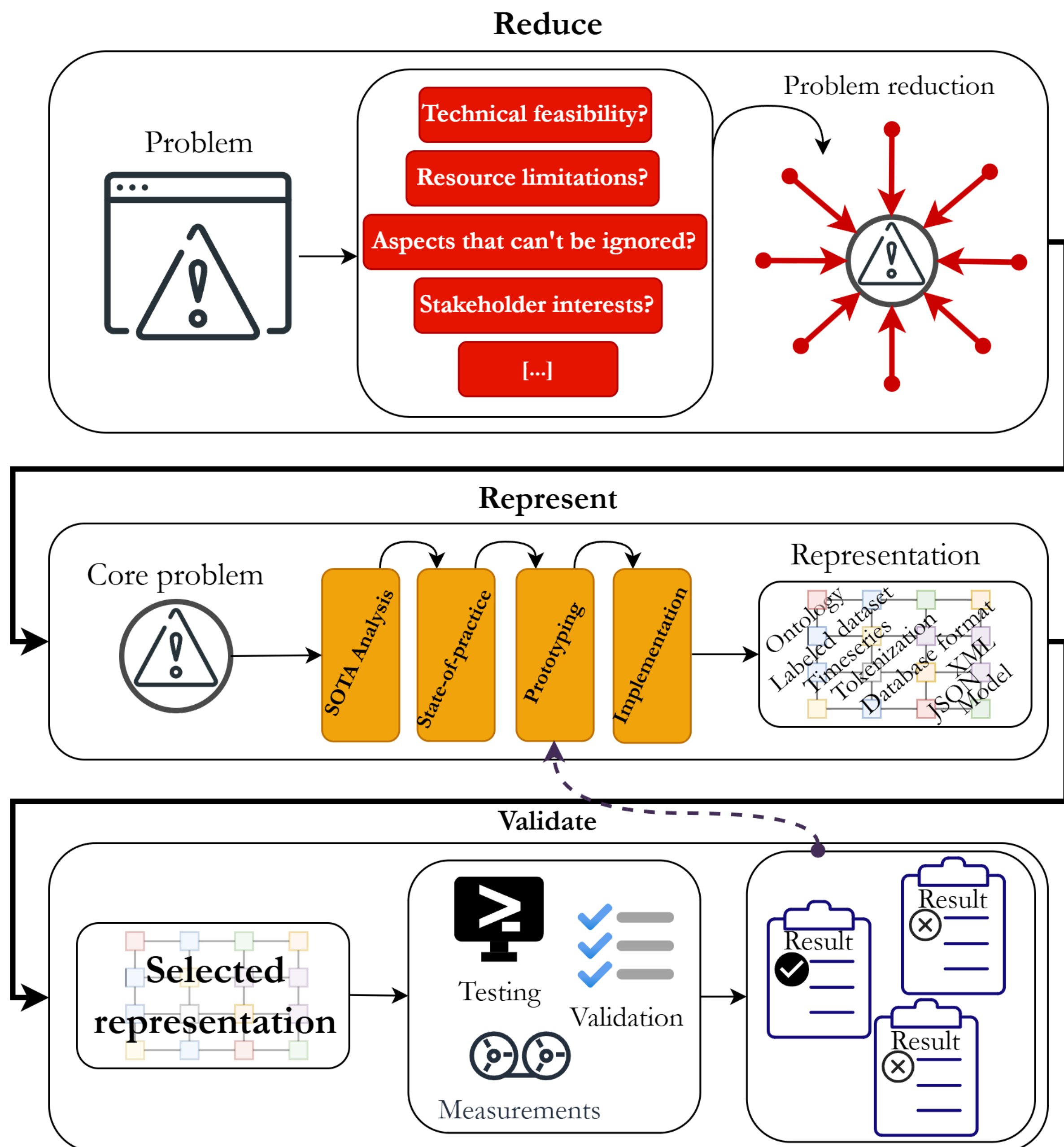


Figure 3.2: The *reduce, represent, validate* pattern. Each contribution constrains the problem (Reduce), encodes it in an intermediate form (Represent), and assesses the result against an independent criterion (Validate).

Chapter 4

Contributions

This chapter presents a summary of the contributions of the included papers, along with a mapping of each contribution to the research questions. The full papers are presented in Part II.

4.1 Thesis contributions

The contributions of this thesis can be understood as four independent but complementary artifacts, each addressing a different research challenge toward trustworthy, automation-oriented machine learning in industrial systems.

4.1.1 Abstraction-based reduction of neural networks

As a first contribution, we develop a principled mechanism for simplifying neural networks while retaining provable guarantees. We introduce an abstraction method [17] that constructs paired over- and under-approximating variants with the i-th input removed, NN_i^+ and NN_i^- respectively, of a trained neural network NN with piecewise-linear activation functions, such that for any input vector $\mathbf{x}$:

$$NN_i^-(\mathbf{x}^{-i}) \;\leq\; NN(\mathbf{x}) \;\leq\; NN_i^+(\mathbf{x}^{-i}),$$

where $\mathbf{x}^{-i}$ denotes $\mathbf{x}$ with its i-th component removed. We then design an algorithm to generate a *difference network* NN_i^{diff} such that:

$$NN_i^{diff}(\mathbf{x}) = NN_i^+(\mathbf{x}) - NN_i^-(\mathbf{x}).$$

The difference network bounds how much the i-th input can affect the total output. We then query this network using Marabou [14] to formally establish whether the output is bounded

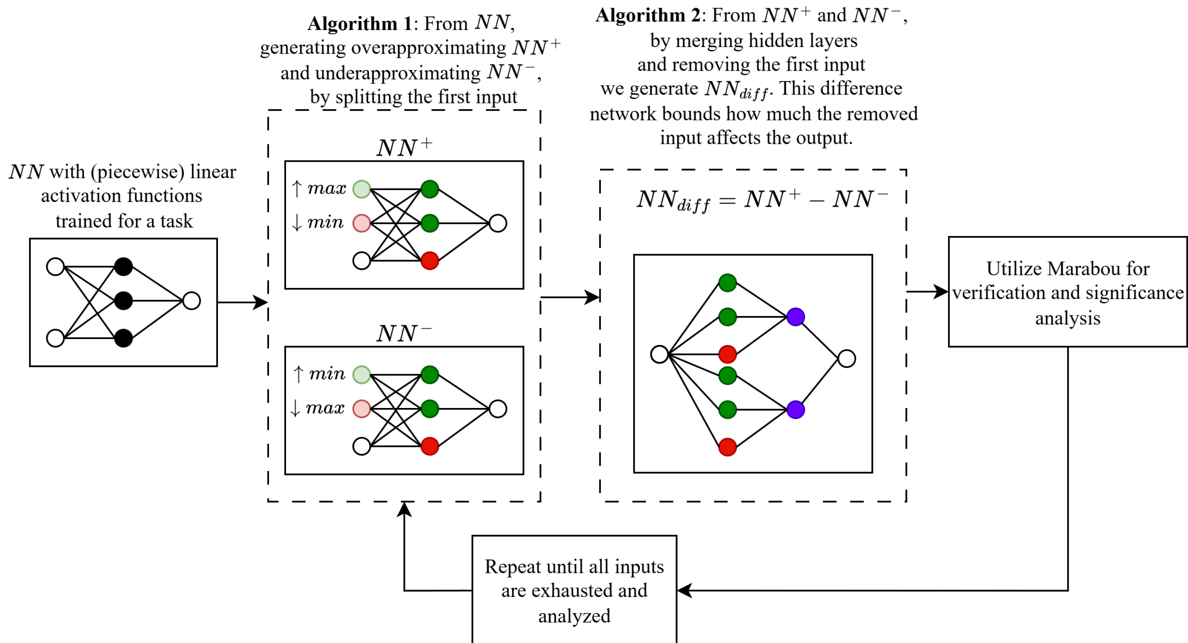


Figure 4.1: Formal construction of difference networks for abstraction (Paper A).

below a given limit L. If so, input i is deemed *insignificant* and may be safely removed. This contribution *extends* the hidden-layer abstraction framework of Elboher et al. [15] to the input layer, which was not previously addressed.

This makes it possible to safely discard features with negligible influence while preserving system behavior within known bounds. Unlike heuristic feature selection, the approach provides formal assurance on how outputs are affected when inputs are discarded.

Results. The method is demonstrated on a network with two inputs, three hidden ReLU neurons, and one output, trained on $f(x_1, x_2) = 100x_1 + x_2$ with $x_1, x_2 \in [0, 1]$. Applying Algorithm 1 to remove x_1 produces an over-approximating network in which the worst-case contribution of x_1 is absorbed into the hidden layer bias. The corresponding difference network NN_1^{diff} is queried via Marabou, which correctly identifies x_1 as significant. As a primarily formal contribution, the algorithms and their correctness proofs are the principal output. Evaluation on industrial-scale networks is identified as future work.

4.1.2 Forecasting with uncertainty and explanations

We develop an uncertainty-aware CPU load forecasting framework designed to support practical safety assessment in industrial computing platforms. The framework wraps a black-box regres-

sion model with conformal prediction to provide guaranteed prediction intervals. This contribution is *new*: while conformal prediction and Shapley attribution each exist independently, their combination into a single deployable framework for embedded CPU load forecasting has not been proposed before.

Given a nonconformity score function $s(x, y)$ and a calibration set of size m, prediction intervals are defined as:

$$\Gamma^{\alpha}(x) = \{y : s(x, y) \leq q_{1-\alpha}\},$$

where $q_{1-\alpha}$ is the $(1 - \alpha)$ quantile of the calibration scores. This guarantees that the interval covers the true value with probability at least $1 - \alpha$, regardless of the data distribution. In our implementation, nonconformity scores are:

$$s(x, y) = \max\bigl(QR_{lower}(x) - y,\ y - QR_{upper}(x)\bigr),$$

with $\alpha = 0.1$ (targeting 90% coverage). To make forecasts actionable, Shapley values are integrated: each task i receives a score ϕ_i representing its marginal contribution to the predicted load, enabling engineers to see not only the predicted load level but also which tasks are most responsible.

Results. Across all 27 task ablations and the full baseline, the framework achieves coverage rates of 0.88–0.90 on Core 0, Core 1, and Dual Core measurements, consistently near the nominal $1 - \alpha = 0.90$ target. Mean interval lengths for Core 0 fall within the device measurement system's $\pm 2\%$ margin of error. The additivity check confirms internal consistency of the Shapley decomposition across all 27 ablations; the sensitivity analysis confirms industrial expert domain knowledge, with FUNC_12 and FUNC_13 identified as the most load-impactful tasks. Industrial partners reviewed the framework and confirmed its practical utility, expressing intent to deploy a specialized instance.

A prototype GUI was implemented to make the framework accessible to device engineers. Screenshots of the GUI are shown in Figures 4.4 and 4.5.

4.1.3 Data-driven cache miss prediction

We develop data-driven models for cache miss prediction at the microarchitectural level. This contribution is *new*: existing ML approaches to cache modeling either assume fixed microarchitectures or target narrow use cases. Parameterizing cache configuration as explicit model inputs

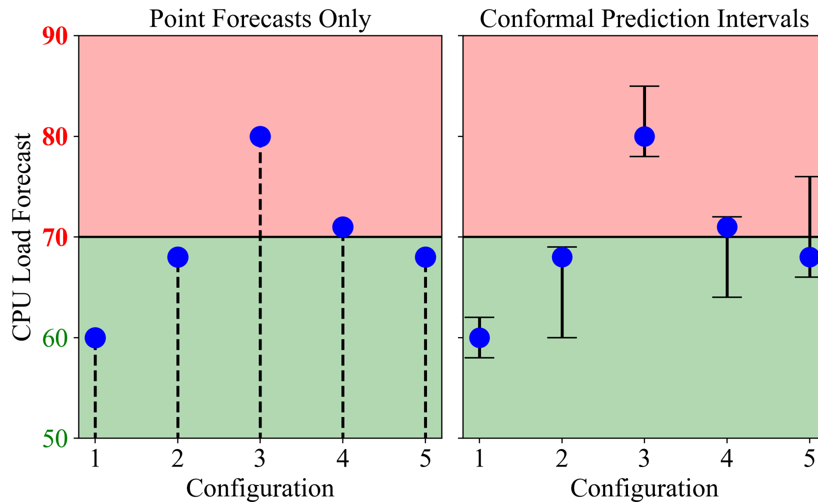


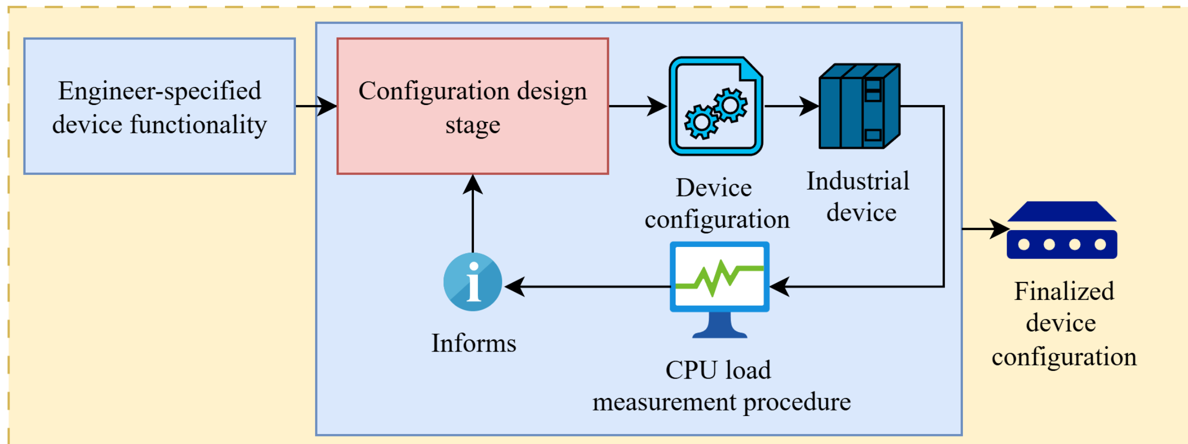
Figure 4.2: Point forecasting vs. conformal prediction intervals for CPU load (Paper B). The CP intervals clarify the safety status of ambiguous configurations.

for cross-configuration generalization is a novel design choice. X86 program execution traces are tokenized, embedded, and processed with stacked LSTM layers. Cache size and core count are passed as additional features, making the model hardware-agnostic. The network outputs a predicted cache miss distribution for the three cache levels (L1D, L1I, LL) across instruction subsequences.

Evaluation is designed around two complementary criteria. *Distributional fidelity* is assessed via MSE, RMSE, and R^2, capturing how closely the predicted miss distribution tracks the simulator over time. *Aggregate accuracy* is measured by E_{REP}, the relative discrepancy between predicted and simulated total miss counts which captures whether global system-level behavior is preserved. This dual evaluation directly addresses RQ 3.

Results. The model achieves E_{RMSE} values of 2–6 across most configurations, with positive R^2 for CPU and FileIO benchmarks. Most configurations achieve E_{REP} below 30%. Notable exceptions occur for the GFTT benchmark at large L1I cache sizes (64 kB), where aggregate miss counts are overestimated by up to $10\times$; these outliers occur at cache sizes well outside the training distribution and are classified as extrapolation failures. Each execution of 25M instructions runs in approximately 45 seconds under DynamoRIO Cachesim and 17.35 seconds per parallel block under the ML model, yielding a 21% average per-block reduction.

High level overview example of contemporary industrial device deployment



Proposed augmentation via conformally calibrated regression and Shapley attribution

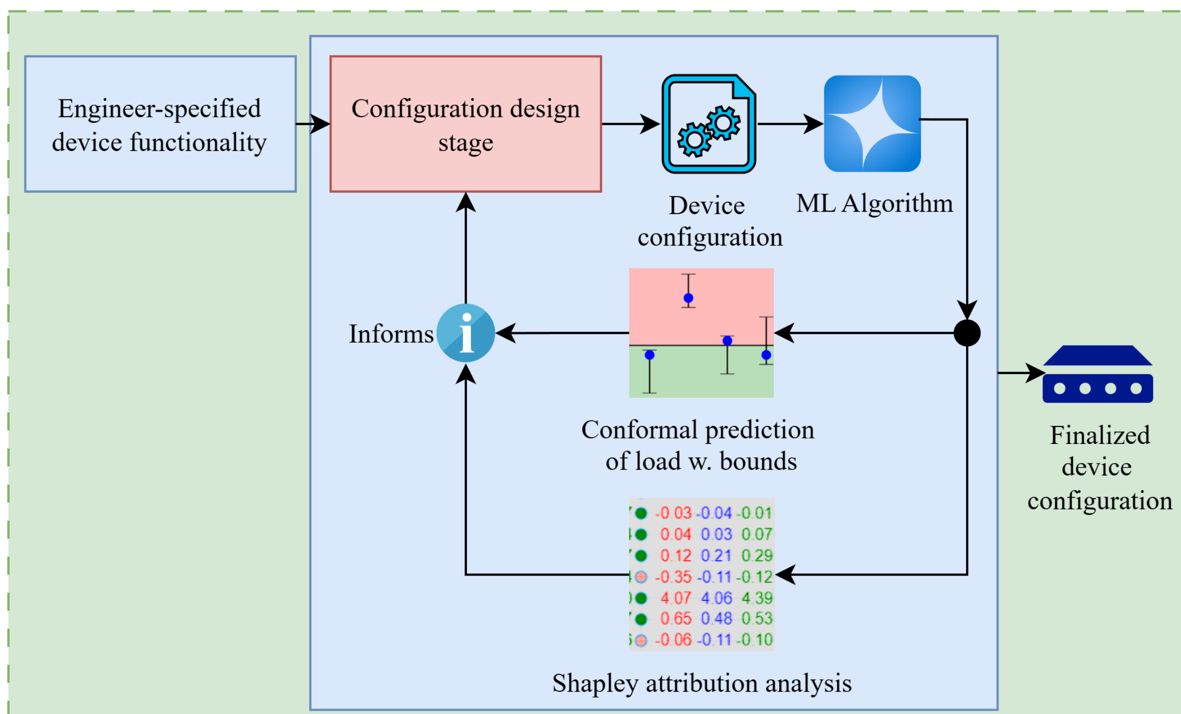


Figure 4.3: High-level overview contrasting current state-of-practice (top) with the proposed framework (bottom), providing prediction intervals and task-level attributions (Paper B).

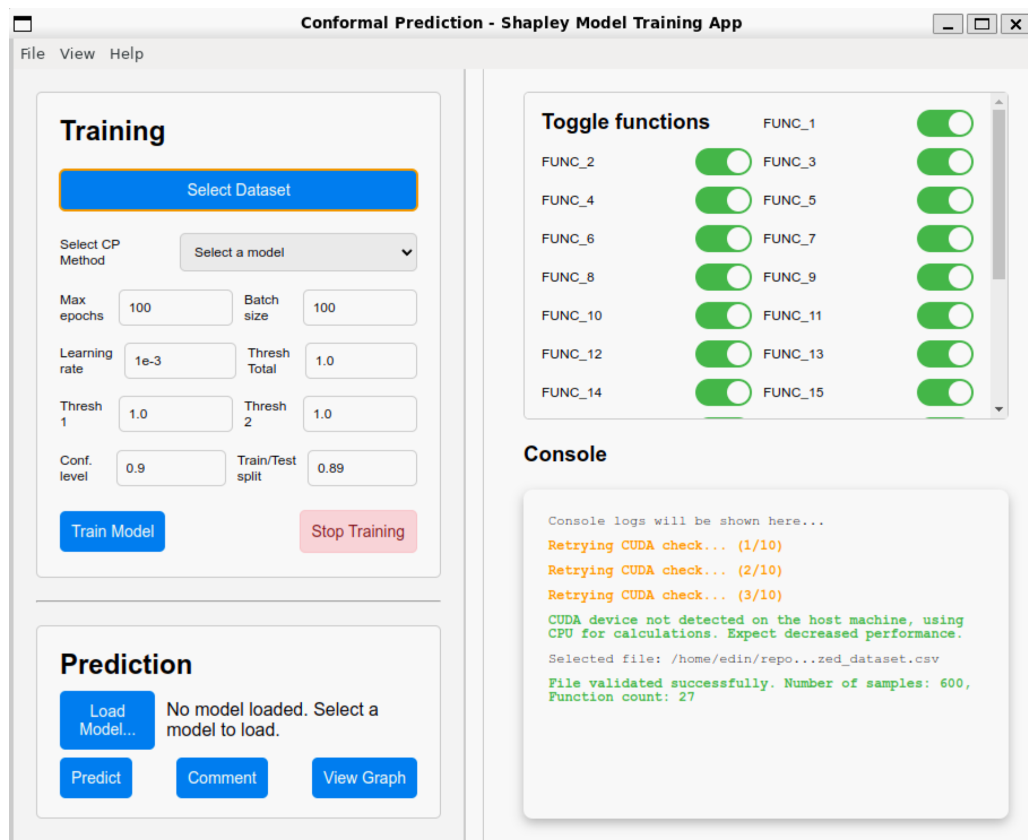


Figure 4.4: Initial GUI screen showing Training, Prediction, task Toggle, and Console sections (Paper B).

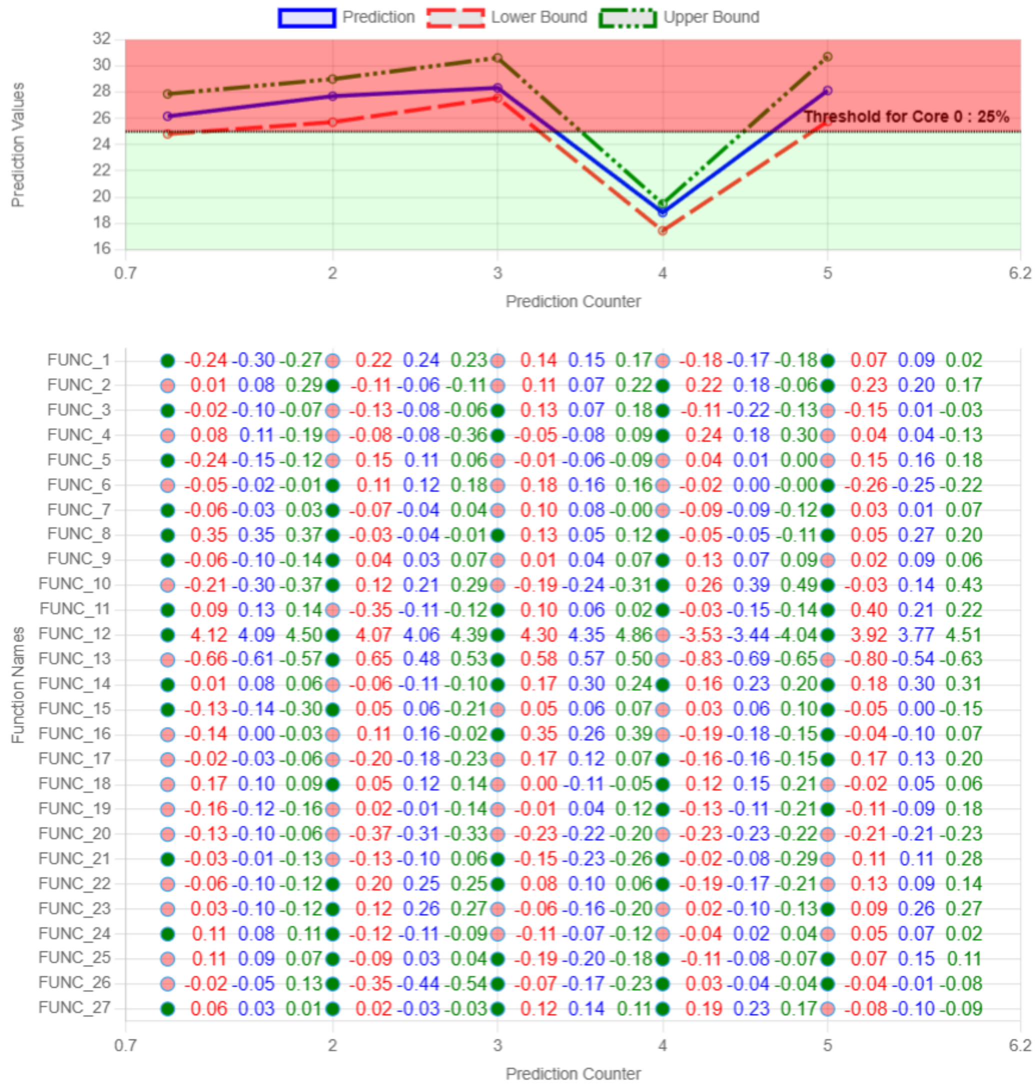


Figure 4.5: CPU load forecaster output for Core 0 with 5 predictions, 25% threshold, and task-level Shapley contributions (Paper B).

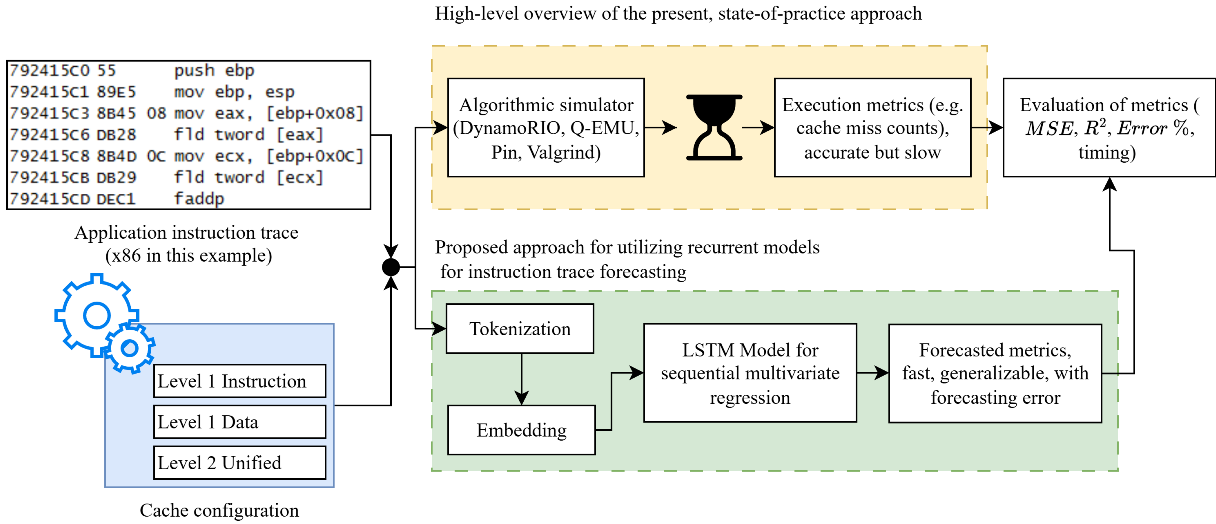


Figure 4.6: Proposed industrial simulation paradigm contrasted with the traditional approach (Paper C).

4.1.4 Hybrid AI Simulation Compiler (HASCO)

The final contribution is HASCO, a Hybrid AI Simulation Compiler that transforms natural-language accident reports into runnable OpenSCENARIO/OpenDRIVE simulation artifacts, as shown in Figure 4.8. This contribution is *new* in its compilation-based framing and *adapts* the code co-generation philosophy of Nouri et al. [83] to the specific challenges of OpenSCENARIO XML synthesis, adding a forensic Judge loop and SHACL-based ontology validation not present in prior work. The central design insight is that scenario creation should be treated as a *compilation task* rather than a generative one: the LLM handles semantic extraction from the source text, while deterministic layers handle the syntactic engineering of valid XML.

Three compilation strategies are implemented, each introducing a progressively more structured intermediate representation between the LLM and the final XML output. Mode 1 (Direct XOSC) asks the LLM to produce OpenSCENARIO XML directly. Mode 2 (Python) asks the LLM to generate a Python script that programmatically constructs the scenario via the `scenariogeneration` library. Mode 3 (Ontology) asks the LLM to populate a JSON-LD instance of a scenario ontology whose core classes (actors, events, start poses, trigger conditions) are defined by a SHACL (Shapes Constraint Language) schema. SHACL constraints enforce structural invariants on the ontology instance before compilation begins: for example, that every actor has exactly one start pose, that every event references an actor present in the catalogue, and that speed values fall within physically plausible ranges. Instances that violate

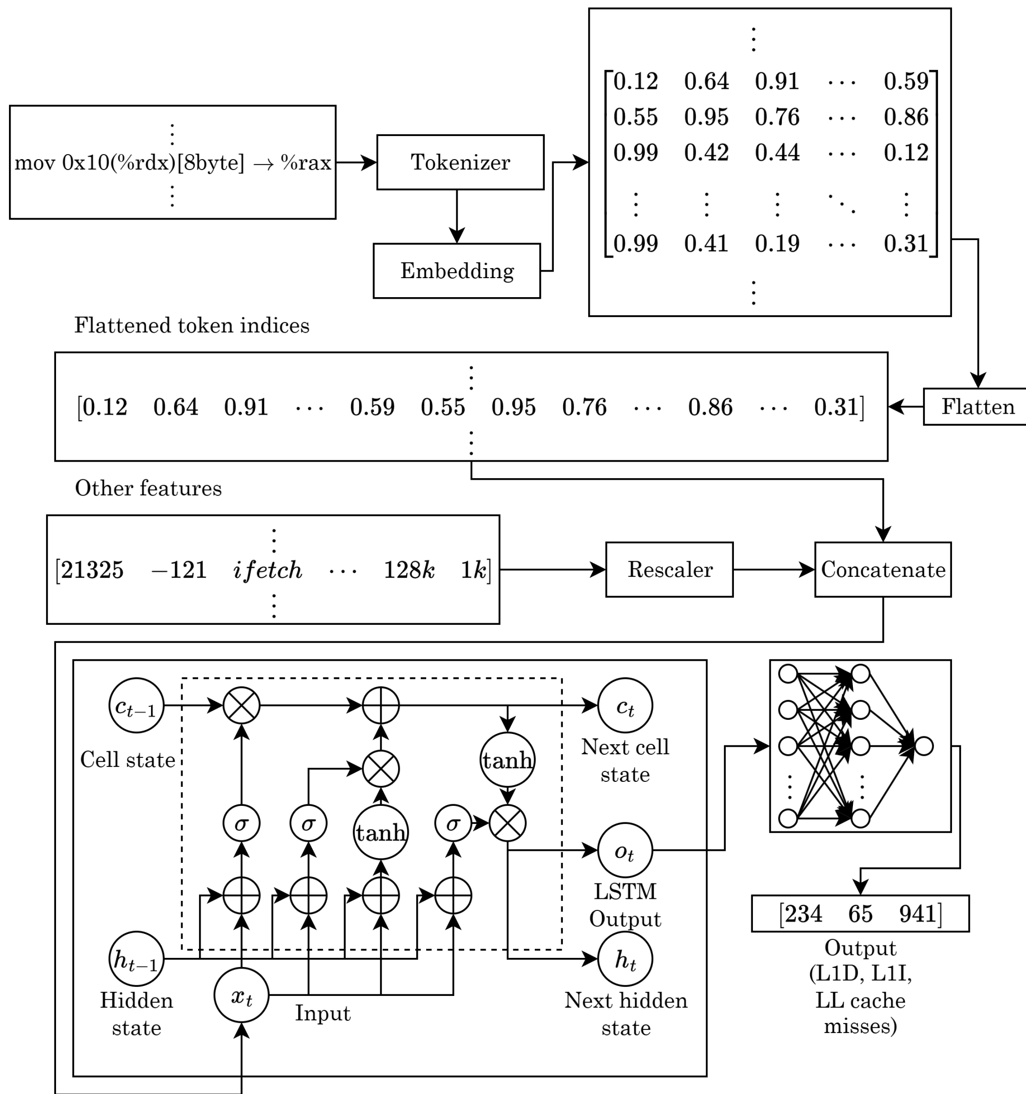


Figure 4.7: LSTM-based cache miss prediction pipeline (Paper C).

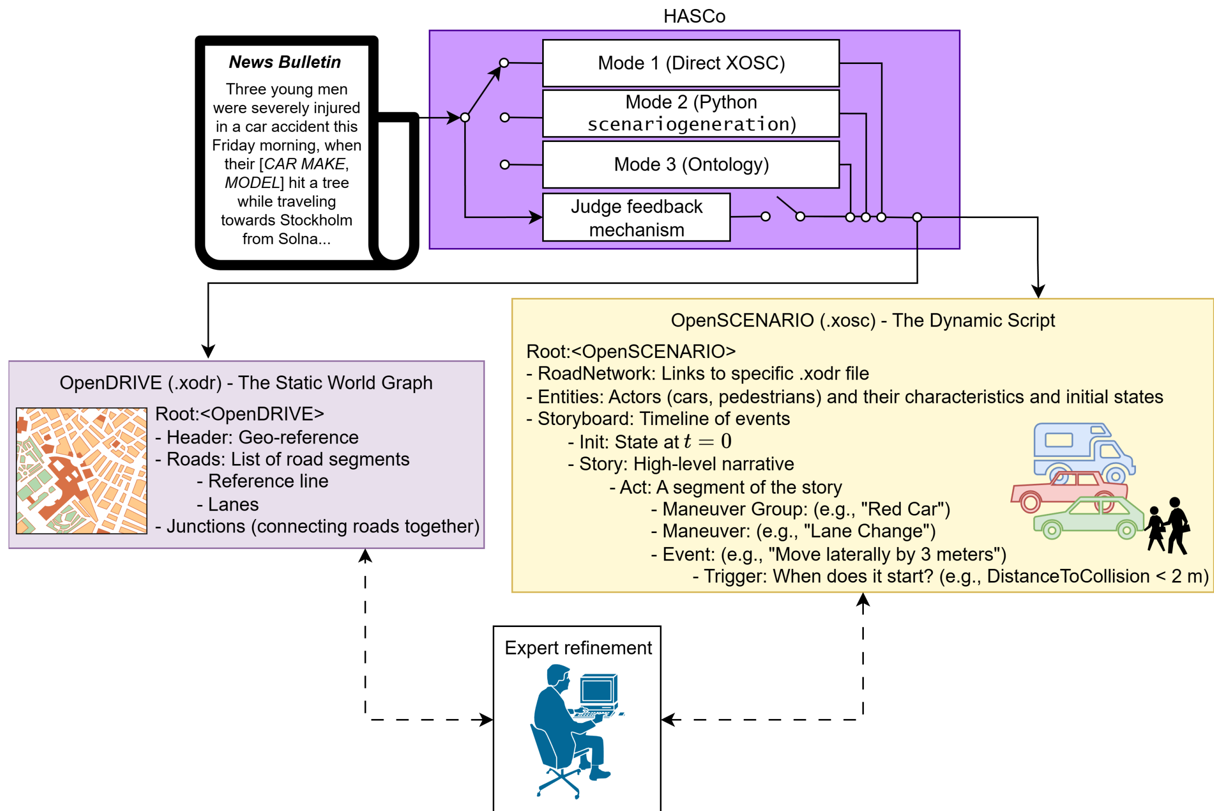


Figure 4.8: HASCO high-level overview (Paper D).

these constraints are rejected before any XML is generated, catching a class of semantic errors that would otherwise propagate silently into the final artifact. A deterministic compiler then translates validated ontology instances into OpenSCENARIO XML. The progressive structuring is deliberate: each successive mode shifts more responsibility from the LLM to deterministic components. The three modes are visible in more detail in Fig. 4.9.

A forensic Judge loop provides self-healing validation. Before a scenario is accepted, the pipeline attempts to execute it in the esmini simulator [84]. Syntax failures trigger automatic repair via error feedback to the LLM. Scenarios that execute but produce no meaningful action (actors spawning but never moving) are detected by a semantic alignment check that compares the simulation's event log against the original report. The pipeline additionally includes a geospatial module that retrieves real-world road networks via OpenStreetMap for each report's location, and a vehicle extraction module that queries realistic actor dimensions.

Results. Across six operating modes ($N = 40$ accident reports), the ontology-guided compiler with Judge achieves 95% executability; the Python-based compiler with Judge achieves 90%; direct XML synthesis achieves only 5%-7.5% regardless of the Judge. With the Judge en-

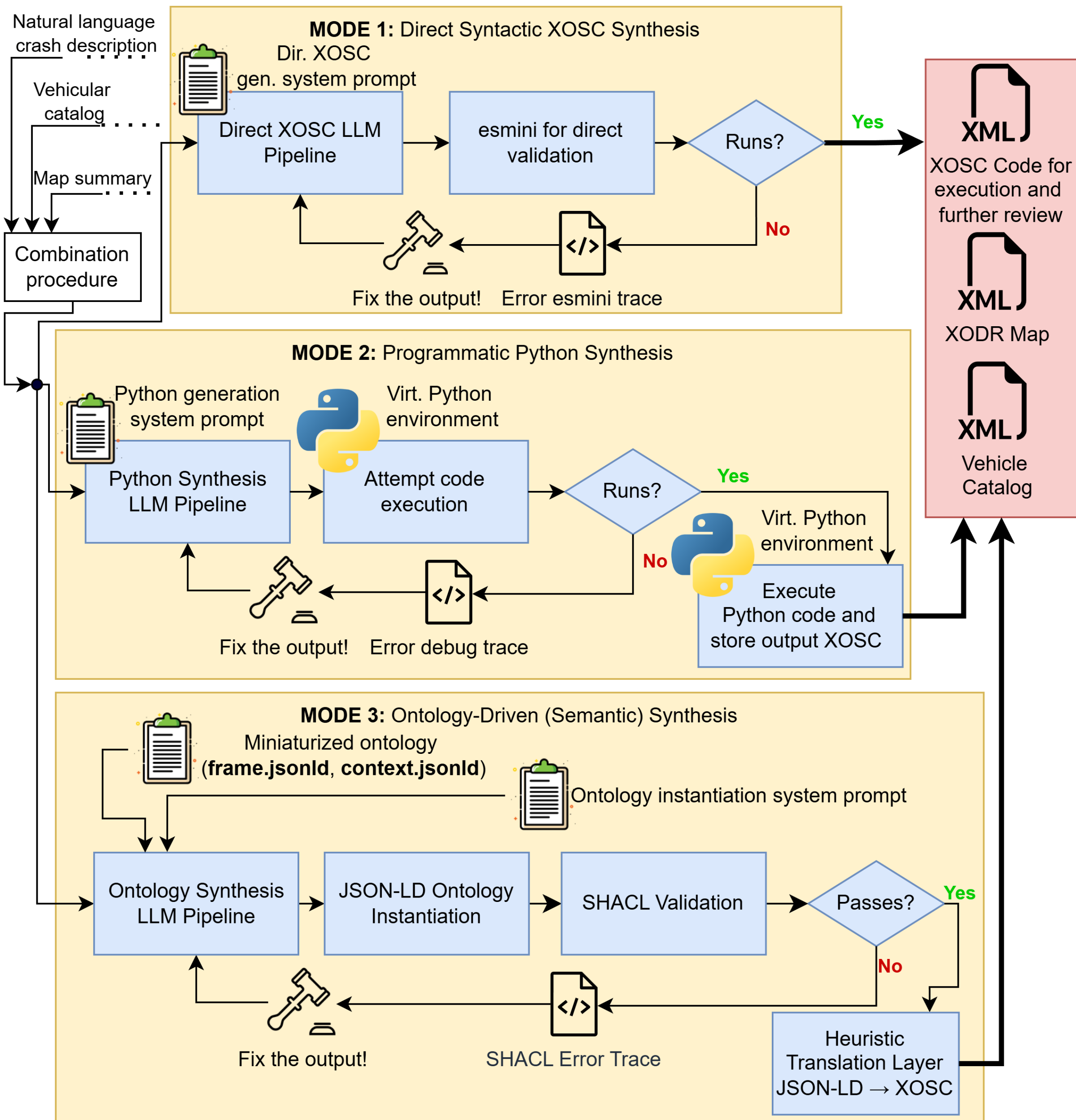


Figure 4.9: HASCO's multi-strategy synthesis architecture (Paper D).

abled, Python mode achieves 62.5% acceptable quality (30% Production, 32.5% Draft); in Ontology mode, the Judge reduces semantic failures by 36.3%. Ontology mode without Judge is the most cost-effective configuration, achieving the highest quality-to-token-cost ratio at 92.5% executability. The corpus spans eight languages; two reports failed across all modes due to infrastructure limitations (geocoding, complex extraction) rather than generation logic failures.

4.2 Individual contribution of the thesis author

I am the primary author and driver of Papers B, C, and D, and a co-leading contributor to Paper A. In all papers, senior co-authors contributed ideas, discussions, experimental feedback, and extensive reviews.

Paper A. I contributed to the conceptual development of the input layer abstraction extension, the implementation and testing of the difference network construction, and the writing of the paper. The theoretical framework builds on prior work by Elboher et al. [15] to which I did not contribute.

Paper B. I was the lead contributor, responsible for the design of the full CP and Shapley integration framework, data extraction and preprocessing, all implementation and experimentation, GUI development, and primary writing.

Paper C. I was the lead contributor, responsible for the LSTM architecture design, tokenization and embedding pipeline, the DynamoRIO wrapper tool, hyperparameter optimization, all experimentation, and primary writing.

Paper D. I was the lead contributor, responsible for the overall HASCO system architecture, all three synthesis modes, the ontology and SHACL schema, the heuristic translation layer, the Judge loop, the evaluation methodology, and primary writing. Zhennan Fei and Ali Nouri contributed through Volvo Cars domain expertise and review.

Chapter 5

Related work

This chapter positions each contribution against the most closely related prior work. The technical foundations underlying each area are presented in Chapter 2; here the focus is on what existing approaches achieve, where they fall short, and how the contributions of this thesis address those shortcomings.

5.1 Abstraction and feature selection

Work on verification-driven abstraction has developed methods to simplify neural networks while preserving formal guarantees. Elboher et al. [15] introduced an abstraction-refinement methodology that merges hidden-layer nodes by color (output-increasing vs. output-decreasing), producing a smaller over-approximating network whose verified bounds hold for the original. Subsequent extensions by Elboher et al. [85] retain information across verification queries on similar networks, and Gokulanathan et al. [86] identify hidden nodes whose output is provably near zero, enabling their removal. Both lines of work operate exclusively on hidden layers; the input layer is left untouched.

These abstraction methods rely on underlying verification tools to discharge proof obligations on the simplified networks. The tool used in Paper A is Marabou [14], an SMT-based solver for piecewise-linear networks. Other verification approaches have since scaled further: α, β-CROWN [87, 88], the repeated winner of VNN-COMP, uses GPU-accelerated bound propagation and branch-and-bound to verify networks with millions of parameters. The abstraction-based approach of Paper A is in principle complementary to such scalable verifiers, as the difference network it constructs could be analyzed by any tool that supports piecewise-linear

networks.

A separate stream of work addresses network compression without certification requirements. Pruning and quantization [89] reduce network size for deployment efficiency but provide no formal guarantees on how the output changes after compression. Statistical dimensionality reduction via PCA [90] produces a transformed feature space that is more compact but no longer consists of original input features, making it difficult to interpret which physical inputs matter. Classical feature selection methods [91] rank inputs by statistical criteria (mutual information, correlation, wrapper-based search) but without bounding how the network's output deviates when a feature is removed.

Remaining gap. All existing approaches either target internal layer reduction rather than input reduction, produce transformed feature spaces rather than interpretable subsets of original inputs, or lack formal bounds on the output deviation induced by feature removal. Paper A addresses this gap by extending the property-preserving abstraction principle of Elboher et al. specifically to the input layer. The resulting difference network provides a formally verifiable bound on how much each input can influence the output across the entire input domain.

It is important to clarify what this method does and does not provide in terms of interpretability. The abstraction does not explain the neural network's internal decision-making process; the network remains a black box in that sense. What the method provides is a formal answer to a different question: given a tolerance threshold L, can input x_i change the output by more than L under any assignment of the remaining inputs? This is interpretability of input *influence*, verified across the entire domain, not interpretability of the network's *reasoning*. The two are complementary: understanding which inputs matter is valuable even when the mechanism by which they matter remains opaque.

5.2 Uncertainty quantification in practical domains

Forecasting in safety-critical domains requires calibrated uncertainty estimates. Conformal prediction was originally introduced by Shafer and Vovk [80] and surveyed comprehensively by Angelopoulos and Bates [33]. Recent extensions include adaptive variants such as jackknife+ with rescaled scores [92] and reweighted nonconformity scores [93], both targeting tighter intervals under distribution shift. In industrial settings, electricity price forecasting has adopted CP [94, 95], and Wess et al. [96] applied CP to DNN accelerator latency estimation on embed-

ded hardware. Wess et al. is the closest prior work to Paper B, but targets inference latency of a neural network accelerator rather than task-configuration CPU load, and does not address feature attribution.

Bayesian approaches offer an alternative path to uncertainty quantification. Bayesian neural networks place a prior over model weights and compute a posterior after observing data [30]; deep ensembles [31] train multiple models and treat prediction spread as a proxy for uncertainty; Monte Carlo Dropout [32] applies dropout at inference time to approximate posterior sampling. These methods are widely used, but their uncertainty estimates are only as reliable as the modeling assumptions they rest on, and none provides a finite-sample, distribution-free guarantee that the produced intervals will contain the true value with at least the desired probability.

In the specific context of Paper B, the choice of CP over Bayesian alternatives was driven by a concrete data constraint. The input space consists of $2^{27} \approx 134\,\mathrm{M}$ possible binary task configurations, of which only 7,129 were observed. A Bayesian network over this space would require estimating conditional probability tables over a joint space vastly larger than the available sample, making the prior dominant and the posterior unreliable. CP sidesteps this entirely: it requires only exchangeability of the calibration and test data, which the experimental protocol directly satisfies (tasks toggled sequentially via script, producing IID samples by design). This is why we chose a model-agnostic solution that avoids the need to specify any prior distribution or graph structure over the task space.

Remaining gap. Existing work does not combine CP coverage guarantees with Shapley-based task-level attribution in the embedded CPU load context. No prior work provides an engineer with both a statistically guaranteed prediction interval and a per-feature decomposition of what drives that prediction, in a framework lightweight enough for deployment on the target device. Paper B addresses this gap.

5.3 Machine learning for hardware performance modeling

The emergence of ML-based alternatives to traditional simulators has been driven by the need for faster exploration of hardware design spaces. Ithemal [65] applies hierarchical LSTMs to predict basic-block throughput from x86 assembly, but operates at the basic-block level where program-level cache behavior is not modeled. SimNet [64] uses CNNs for instruction latency prediction and can include cache and branch predictor behavior through its history context, but

requires gem5 cycle-accurate traces for training, tying it to a specific simulation infrastructure. TAO [66] extends DL-based simulation to predict multiple performance metrics including cache misses, and introduces microarchitecture-agnostic embeddings for transfer learning across configurations; however, it still requires detailed cycle-accurate training data from gem5 and operates at instruction-level granularity rather than predicting cache miss distributions across program execution. Jha et al. [97] demonstrated NN-based cache miss prediction from instruction traces, and Zeng et al. [98] integrated LSTM predictors into hardware prefetchers; both target narrow use cases rather than cross-configuration generalization.

It is worth clarifying that Paper C does not replace simulation with pure ML. DynamoRIO Cachesim is the ground-truth simulator used both for generating training data and for evaluating the model's predictions. The ML model is designed as a faster surrogate that, once trained, can predict cache miss distributions for new configurations without requiring an additional instrumented run for each one. The relationship is complementary: simulation provides the trusted baseline, and ML provides the speed.

Remaining gap. Existing ML approaches either model instruction throughput under a negligible-cache-miss assumption, operate on a fixed microarchitecture, or target a specific narrow use case. None models cache miss *distributions* across multiple cache levels and core counts simultaneously, with hardware parameters as explicit model inputs enabling comparison across hypothetical configurations not seen during training. Paper C addresses this gap.

5.4 LLMs for scenario generation and code co-generation

Several recent works have applied large language models to generating scenario descriptions or simulation artifacts from natural language inputs.

TARGET [99] generates test scenarios from traffic rules via LLM-guided knowledge extraction, but focuses on regulatory compliance rather than accident reconstruction and does not address the challenge of generating valid OpenSCENARIO XML from unstructured text. Chat2Scenario [100] extracts scenario parameters from naturalistic driving datasets using LLMs, but is an extraction framework that requires a pre-existing dataset of matching scenarios and does not synthesize new XML structures. ScenicNL [101] converts police reports into the Scenic probabilistic programming language, which is more concise than OpenSCENARIO but is not the automotive industry standard and has limited toolchain support in production valida-

tion workflows.

On the code generation side, Nouri et al. [83] argue for a code co-generation paradigm in which LLMs generate candidate code that is immediately subjected to automated sanity checks, with failures fed back into the generation loop. This generate-fast, eliminate-fast philosophy directly motivates the Judge mechanism in Paper D, where a deterministic validation loop running esmini [84] provides explicit failure signals that the LLM can act on.

Remaining gap. Existing LLM-based scenario generation frameworks demonstrate the promise of text-to-simulation pipelines but rarely account for the repair cost and trustworthiness of generated outputs. None integrates a deterministic validation loop specifically targeting the syntactic and semantic hallucinations inherent in verbose OpenSCENARIO XML generation, and none evaluates semantic fidelity against the original forensic narrative as the primary quality metric. Paper D addresses this gap through the HASCO compilation architecture and its qualitative fidelity rubric.

Chapter 6

Discussion and limitations

6.1 Discussion

6.1.1 Addressing the research gaps

Gap A (guarantees and explainability). Paper A provides formal bounds on the approximation error introduced by input reduction, established constructively via Theorems 1 and 2 and mechanically verifiable through Marabou queries. Paper B provides statistically guaranteed coverage intervals via conformal prediction, with finite-sample validity under the sole assumption of data exchangeability, augmented by Shapley attribution to trace the source of predictions. Together, these contributions demonstrate two complementary modes of guarantee: formal (Paper A) and statistical (Paper B).

Gap B (generalizable performance modeling). Paper C demonstrates generalization across cache configurations and program families by encoding hardware parameters as model inputs and evaluating on configurations unseen during training, including sizes outside the training range. The failure cases at extreme extrapolation are characterized explicitly, providing an honest boundary condition for the method's scope.

Gap C (end-to-end integration). Paper D integrates the principles established across the thesis: structured intermediate representation (echoing Paper A's abstraction), statistically grounded evaluation (echoing Paper B's coverage thinking), and data-driven components operating alongside deterministic validators (echoing Paper C's hybrid approach). The result is an end-to-end pipeline that a practitioner can use directly.

6.1.2 The Role of intermediate representations

A recurring theme across all four contributions is the value of intermediate representations (IRs) as a mechanism for managing complexity. The *reduce, represent, validate* pattern from Subsection 3.4.5 manifests in each of the four contributions as follows. In the abstraction-based reduction of neural networks, the reduction step eliminates inputs whose influence is bounded below a threshold; the representation is the difference network NN_i^{diff}; and the validation is the Marabou query that either confirms insignificance or returns a counterexample. In forecasting with uncertainty and explanations, the reduction is implicit in the choice to model CPU load as a function of 27 binary task flags rather than the full system state; the representation is the conformalized prediction interval augmented with Shapley attributions; and the validation is the coverage guarantee, verified empirically on a held-out test set against the nominal 90% target. In data-driven cache miss prediction, the reduction maps raw x86 execution traces to tokenized subsequences with cache parameters appended; the representation is the LSTM's learned encoding of instruction sequences; and the validation is the dual-metric evaluation against DynamoRIO Cachesim ground truth at both distributional and aggregate levels. Finally, in HASCO, the reduction extracts semantic content from natural language into a structured form (Python script or JSON-LD ontology); the representation is the intermediate artifact that the deterministic compiler consumes; and the validation is the esmini execution check followed by the semantic alignment assessment against the original report.

6.1.3 Design for domain expert utilization

Both contributions B and D produce their outputs autonomously, without requiring human intervention during execution. What distinguishes them from purely automated systems is that their outputs are designed to be consumed, interpreted, and acted upon by domain experts rather than by downstream software. Paper B does not make the deployment decision for the engineer; it provides a prediction interval and a per-task attribution breakdown that equip the engineer to make that decision with quantified confidence. Paper D does not claim to produce finished simulation scenarios; it produces candidate artifacts that a validation engineer can inspect, refine, and approve. In both cases, the ML system's role terminates at producing structured, interpretable output. The judgment remains with the domain expert, and this reflects a deliberate design choice. Namely, in safety-critical industrial contexts, the value of ML lies not in replacing expert judgment but in presenting experts with richer, better-quantified evidence than they

would otherwise have.

6.2 Limitations and threats to validity

6.2.1 Paper A

The method is currently limited to networks with a single hidden layer. Extension to multiple layers requires resolving color conflicts in intermediate nodes, identified as future work. No large-scale empirical evaluation of efficiency on industrial-scale networks has been conducted; the current evaluation consists of the running example in the paper. This is acknowledged as the primary limitation of Paper A: the contribution is theoretical, with the algorithms and their correctness proofs (Theorems 1 and 2) as the principal output. Empirical evaluation on industrial-scale networks with high-dimensional inputs is explicitly identified as future work and does not diminish the formal contribution, but it does mean that the practical scalability of the method remains undemonstrated.

6.2.2 Paper B

The dataset covers only 27 of the hundreds of tasks running on the device. The IID assumption, while directly satisfied by the experimental design (tasks toggled sequentially via script), may not hold in a continuous production deployment where the task workload shifts over time. The conformal prediction framework provides marginal coverage; conditional coverage guarantees (per-subgroup) are not addressed and are identified as future work. Overfitting was not identified as a concern in Paper B for several reasons. The base model is trained on only 37.5% of the data using MSE loss, with the remaining data reserved for calibration and testing. The coverage evaluation is performed on a fully held-out test set that the model never sees during training or calibration. The empirical coverage rates of 0.88-0.90 across all 27 task ablations, consistently near the nominal 90% target, demonstrate that the framework is not overfitting to the calibration set: an overfit model would show inflated coverage on calibration data but degraded coverage on the test set, which is not the observed pattern.

6.2.3 Paper C

The model consistently overestimates aggregate cache miss counts in the GFTT benchmark at large cache sizes, suggesting that the MSE training objective does not adequately penalize aggregate bias. The model has been evaluated on four benchmark programs; generalization to a broader range of industrial applications remains to be demonstrated. Complex workloads such as FileIO, which involve synchronous disk operations and irregular access patterns, expose limitations of the current feature set. Overfitting is mitigated by several design choices. Dropout of 0.05 is applied between LSTM layers as regularization. More importantly, the test set comprises entirely different cache configurations from the training set, making the evaluation an out-of-distribution generalization test rather than a conventional train/test split. The overestimation observed on GFTT at extreme cache sizes is characteristic of extrapolation beyond the training distribution rather than overfitting: an overfit model would perform well on interpolated configurations within the training range but poorly on all unseen ones, whereas the observed pattern is strong performance on interpolated configurations with degradation only at the extremes.

6.2.4 Paper D

The semantic fidelity evaluation relies partly on expert human judgment and partly on the Judge LLM, both of which introduce subjectivity. The Judge LLM may itself hallucinate "alignment." The model temperature was not configurable, limiting reproducibility analysis across multiple runs. The system's fidelity is bounded by external API reliability (geocoding, OSM data quality). Additionally, the qualitative fidelity rubric (bins 1-3) was assessed by the research team rather than independent domain experts, introducing potential bias in the semantic evaluation.

Chapter 7

Conclusion and future work

7.1 Conclusion

This thesis set out to advance three qualities that are necessary for responsible adoption of machine learning in industrial-scale systems: *trustworthiness*, *explainability*, and *efficiency*. Four contributions, each addressing a different industrial problem, have been presented. We refer to them here by their core function: *formal input reduction* (Paper A), *guaranteed load forecasting* (Paper B), *cache surrogate modeling* (Paper C), and *scenario compilation* (Paper D, HASCO).

Formal input reduction demonstrated that neural network inputs can be identified as significant or insignificant with provable bounds on the induced output deviation, via Marabou-verified difference networks. This contribution advances *trustworthiness* by replacing heuristic feature selection with formal guarantees, and provides a limited but concrete form of *explainability*: engineers learn which inputs matter, with a certificate that the answer holds across the entire input domain.

Guaranteed load forecasting demonstrated that statistically valid prediction intervals can be generated in a model-agnostic manner for CPU load on an industrial protection device, with per-task Shapley attribution identifying each task's contribution to the predicted load. This contribution is the one that most directly advances all three qualities: *trustworthiness* through finite-sample coverage guarantees, *explainability* through associative attribution that communicates prediction drivers to engineers, and *efficiency* through a framework lightweight enough for deployment on the target embedded hardware.

Cache surrogate modeling demonstrated that an LSTM trained on program execution traces

can replicate cache miss distributions across hardware configurations not seen during training, including cache sizes outside the training range. This contribution advances *efficiency* by replacing repeated simulator runs with a single forward pass, and *trustworthiness* by evaluating fidelity against DynamoRIO Cachesim ground truth at both distributional and aggregate levels, with failure cases at extreme extrapolation characterized explicitly.

Scenario compilation demonstrated that natural language accident reports can be compiled into executable OpenSCENARIO simulation artifacts through a multi-strategy pipeline with deterministic validation, achieving 95% executability on 40 real-world reports across eight languages. This contribution advances *efficiency* by automating a process that currently requires expert manual effort, and *trustworthiness* by validating generated artifacts through deterministic checks rather than accepting LLM output at face value.

What unifies these four contributions, beyond the shared concern for trustworthiness, explainability, and efficiency, is the common architectural pattern of *reduce, represent, validate*.

Formal input reduction reduces to a difference network and validates via Marabou. Guaranteed load forecasting reduces to a conformalized interval and validates via held-out coverage. Cache surrogate modeling reduces to tokenized instruction subsequences representing the simulated source code execution and validates via dual-metric comparison against the simulator. Scenario compilation reduces to a structured intermediate representation (Python script or JSON-LD ontology) and validates via esmini execution and semantic alignment checks.

The overarching scientific insight is that *separation of concerns* between what ML does well (learning from data, extracting patterns, handling ambiguity) and what deterministic methods do well (enforcing constraints, providing guarantees, catching inconsistencies) is what makes ML deployable in industrial contexts. The reduce, represent, validate pattern makes this separation manifest: the ML component is never the sole arbiter of correctness. This principle constitutes the thesis's broader methodological contribution, a reusable design pattern for deploying ML in contexts where outputs must be not only accurate but also demonstrably warranted.

7.2 Future Work

Scaling formal input reduction. The current method handles single-hidden-layer networks. Extension to deeper networks requires resolving color conflicts in intermediate layers, potentially via the refinement loop described by Elboher et al. but not implemented here. Large-scale

empirical evaluation on industrial-scale networks would establish practical viability.

Conditional coverage and incremental learning. The conformal prediction framework currently provides marginal coverage. Conditional coverage methods [102] would provide per-subgroup guarantees. Incremental learning strategies would enable continuous model refinement as new device data becomes available, which would improve overall *efficiency* in the framework's long-term deployment.

Richer features and extended benchmarks. Extracting additional features from DynamoRIO (register usage, branch predictor state, individual instruction latencies) and evaluating on a broader suite of industrial software applications, particularly complex I/O-heavy workloads, would improve coverage of the generalization space. Batch normalization, alternative recurrent architectures (GRU, temporal convolutional networks), and positional encoding for token embeddings are also worth exploring.

Deterministic safety metrics and kinematic realism. Future work will focus on replacing the qualitative fidelity rubric with deterministic safety metrics (responsibility-sensitive safety checks) that mathematically validate the physical causality of synthesized collisions. This would shift the validation step of the *reduce, represent, validate* pattern from partly subjective assessment to fully deterministic verification, strengthening *trustworthiness*. Integrating constraint-based motion planning into the heuristic translation layer would improve kinematic realism.

Toward a unified pipeline. A longer-term research direction is to integrate the four contributions into a single framework: formally reduced models, with guaranteed uncertainty bounds and traceable explanations, supported by fast data-driven simulation, embedded in an end-to-end compilation pipeline with deterministic validation. Such a framework would instantiate the *reduce, represent, validate* pattern at the system level, offering industrial practitioners a unified toolchain in which *trustworthiness*, *explainability*, and *efficiency* are not afterthoughts but architectural invariants.

Bibliography

[1] Shan Ren, Yingfeng Zhang, Yang Liu, Tomohiko Sakao, Donald Huisingh, and Cecilia M V B Almeida. "A comprehensive review of big data analytics throughout product lifecycle to support sustainable smart manufacturing: A framework, challenges and future research directions". en. In: *J. Clean. Prod.* 210 (Feb. 2019), pp. 1343–1365.

[2] Junping Wang, Wensheng Zhang, Youkang Shi, Shihui Duan, and Jin Liu. "Industrial big data analytics: Challenges, methodologies, and applications". In: *arXiv [cs.DB]* (July 3, 2018).

[3] J Krauß, T Hülsmann, L Leyendecker, and R H Schmitt. "Application areas, use cases, and data sets for machine learning and artificial intelligence in production". en. In: *Lecture Notes in Production Engineering*. Cham: Springer International Publishing, 2023, pp. 504–513. (Visited on 09/05/2025).

[4] Eduardo A Hinojosa-Palafox, Oscar M Rodríguez-Elías, José A Hoyo-Montaño, Jesús H Pacheco-Ramírez, and José M Nieto-Jalil. "An analytics environment architecture for industrial cyber-physical systems Big Data solutions". en. In: *Sensors (Basel)* 21 (13 June 23, 2021), p. 4282.

[5] Mohd Javaid, Abid Haleem, Ravi Pratap Singh, and Rajiv Suman. "An integrated outlook of Cyber–Physical Systems for Industry 4.0: Topical practices, architecture, and applications". en. In: *Green Technologies and Sustainability* 1 (1 Jan. 2023), p. 100001.

[6] Adib Bin Rashid and M D Ashfakul Karim Kausik. "AI revolutionizing industries worldwide: A comprehensive overview of its diverse applications". en. In: *Hybrid Adv.* 7 (100277 Dec. 1, 2024), p. 100277. (Visited on 09/02/2025).

[7] Elias Dritsas and Maria Trigka. "Exploring the intersection of machine learning and big data: A survey". en. In: *Mach. Learn. Knowl. Extr.* 7 (1 Feb. 7, 2025), p. 13.

[8] Shiva Nejati, Stefano Di Alesio, Mehrdad Sabetzadeh, and Lionel Briand. “Modeling and analysis of CPU usage in safety-critical embedded systems to support stress testing”. In: *Model Driven Engineering Languages and Systems*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 759–775.

[9] John Regehr and Usit Duongsaa. “Preventing interrupt overload”. en. In: *SIGPLAN Not.* 40 (7 July 12, 2005), pp. 50–58.

[10] Derek Bruening, Qin Zhao, and Saman Amarasinghe. “Transparent dynamic instrumentation”. In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*. VEE ’12: ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (London England, UK). New York, NY, USA: ACM, Mar. 3, 2012.

[11] Kevin Gurney. *An Introduction to Neural Networks*. London, England: CRC Press, Apr. 21, 2014. 234 pp.

[12] Vanessa Buhrmester, David Münch, and Michael Arens. “Analysis of explainers of black box Deep Neural Networks for Computer Vision: A survey”. en. In: *Mach. Learn. Knowl. Extr.* 3 (4 Dec. 8, 2021), pp. 966–989.

[13] Francesco Leofante, Nina Narodytska, Luca Pulina, and Armando Tacchella. “Automated verification of neural networks: Advances, challenges and perspectives”. In: *arXiv [cs.AI]* (May 24, 2018).

[14] Haoze Wu, Omri Isac, Aleksandar Zeljić, Teruhiro Tagomori, Matthew Daggitt, Wen Kokke, Idan Refaeli, Guy Amir, Kyle Julian, Shahaf Bassan, Pei Huang, Ori Lahav, Min Wu, Min Zhang, Ekaterina Komendantskaya, Guy Katz, and Clark Barrett. “Marabou 2.0: A versatile formal analyzer of neural networks”. In: *arXiv [cs.AI]* (Jan. 25, 2024).

[15] Yizhak Yisrael Elboher, Justin Gottschlich, and Guy Katz. “An abstraction-based framework for neural network verification”. en. In: *Computer Aided Verification*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 43–65. (Visited on 09/05/2025).

[16] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. “Counterexample-guided abstraction refinement”. In: *Lecture Notes in Computer Science*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 154–169.

[17] Peter Backeman, Edin Jelačić, Cristina Seceleanu, Ning Xiong, and Tiberiu Seceleanu. “Abstraction-based reduction of input size for neural networks”. In: *Lecture Notes in Computer Science vol. 15250*. AISoLA 2023 (Greece). 2023.

[18] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*. Vol. 1. MIT press Cambridge, 2016.

[19] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. en. In: *Neural Netw.* 2 (5 Jan. 1, 1989), pp. 359–366. (Visited on 04/06/2026).

[20] S Hochreiter and J Schmidhuber. “Long short-term memory”. en. In: *Neural Comput.* 9 (8 Nov. 15, 1997), pp. 1735–1780.

[21] Alex Sherstinsky. “Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network”. en. In: *Physica D* 404 (132306 Mar. 2020), p. 132306.

[22] Vincent Froese and Christoph Hertrich. “Training neural networks is NP-hard in fixed dimension”. In: *arXiv [cs.CC]* (Mar. 29, 2023).

[23] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. “The limitations of deep learning in adversarial settings”. en. In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2016 IEEE European Symposium on Security and Privacy (EuroS&P) (Saarbrucken). IEEE, Mar. 2016, pp. 372–387. (Visited on 04/07/2026).

[24] Wei Emma Zhang, Quan Z Sheng, Ahoud Alhazmi, and Chenliang Li. “Adversarial attacks on deep learning models in natural language processing: A survey”. In: *arXiv [cs.CL]* (Jan. 21, 2019). (Visited on 04/07/2026).

[25] Aws Albarghouthi. “Introduction to neural network verification”. In: *arXiv [cs.LG]* (Sept. 21, 2021).

[26] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. “Safety verification of deep neural networks”. en. In: *Computer Aided Verification*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 3–29. (Visited on 04/07/2026).

[27] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. "Robust Physical-World Attacks on Deep Learning Visual Classification". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 1625–1634. (Visited on 04/07/2026).

[28] Kyle D Julian, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer. "Policy compression for aircraft collision avoidance systems". In: *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC) (Sacramento, CA, USA). IEEE, Sept. 2016.

[29] Changliu Liu, Tomer Arnon, Chris Lazarus, Christopher Strong, Clark Barrett, and Mykel J Kochenderfer. "Algorithms for verifying deep neural networks". en. In: *Found. trends® optim.* 4 (3-4 Feb. 11, 2021), pp. 244–404.

[30] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. "Weight uncertainty in neural networks". In: *arXiv [stat.ML]* (May 20, 2015). Ed. by Francis Bach and David Blei, pp. 1613–1622.

[31] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. "Simple and scalable predictive uncertainty estimation using deep ensembles". In: *arXiv [stat.ML]* (Dec. 5, 2016).

[32] Yarin Gal and Zoubin Ghahramani. "Dropout as a Bayesian approximation: Representing model uncertainty in deep learning". In: *arXiv [stat.ML]* (June 6, 2015). Ed. by Maria Florina Balcan and Kilian Q Weinberger, pp. 1050–1059.

[33] Anastasios N Angelopoulos and Stephen Bates. "A gentle introduction to conformal prediction and distribution-free uncertainty quantification". In: *arXiv [cs.LG]* (July 15, 2021).

[34] Vladimir Vovk. "Conditional validity of inductive conformal predictors". In: *Asian conference on machine learning*. 2012, pp. 475–490.

[35] Eyal Winter. "Chapter 53 The shapley value". In: *Handbook of Game Theory with Economic Applications*. Vol. 3. Handbook of Game Theory and Economic Applications. Elsevier, 2002, pp. 2025–2054.

[36] Scott Lundberg and Su-In Lee. “A unified approach to interpreting model predictions”. In: *arXiv [cs.AI]* (May 22, 2017). Ed. by I Guyon, U V Luxburg, S Bengio, H Wallach, R Fergus, S Vishwanathan, and R Garnett, pp. 4765–4774. (Visited on 09/18/2025).

[37] Xiaotie Deng and Christos H Papadimitriou. “On the complexity of cooperative solution concepts”. en. In: *Math. Oper. Res.* 19 (2 May 1994), pp. 257–266.

[38] Leslie G Valiant. “The complexity of enumeration and reliability problems”. en. In: *SIAM J. Comput.* 8 (3 Aug. 1979), pp. 410–421.

[39] Scott M Lundberg, Gabriel G Erion, and Su-In Lee. “Consistent individualized feature attribution for tree ensembles”. In: *arXiv [cs.LG]* (Feb. 11, 2018).

[40] J L Hennessy and D A Patterson. *Computer Architecture: A Quantitative Approach*. San Francisco, CA: Morgan Kaufmann, 2011.

[41] I Intel. “IA-32 architectures software developer’s manual”. In: *Volume 3A: System Programming Guide, Part, 64* (Mar. 2026).

[42] Wm A Wulf and Sally A McKee. “Hitting the memory wall: implications of the obvious”. en. In: *Comput. Archit. News* 23 (1 Mar. 1995), pp. 20–24.

[43] Carlos Carvalho. “The gap between processor and memory speeds”. In: 5000 (2002), p. 15000.

[44] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D Hill, and David A Wood. “The gem5 simulator”. en. In: *Comput. Archit. News* 39 (2 May 31, 2011), pp. 1–7.

[45] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’07: ACM SIGPLAN Conference on Programming Language Design and Implementation (San Diego California USA). PLDI ’07. New York, NY, USA: ACM, June 10, 2007, pp. 89–100.

[46] Walther Wachenfeld and Hermann Winner. “The release of autonomous vehicles”. In: *Autonomous Driving*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 425–449.

[47] Christian Neurohr, Lukas Westhofen, Tabea Henning, Thies de Graaff, Eike Mohlmann, and Eckard Bode. "Fundamental considerations around scenario-based testing for automated driving". In: *2020 IEEE Intelligent Vehicles Symposium (IV)*. 2020 IEEE Intelligent Vehicles Symposium (IV) (Las Vegas, NV, USA). IEEE, Oct. 19, 2020, pp. 121–127.

[48] Stefan Riedmaier, Thomas Ponn, Dieter Ludwig, Bernhard Schick, and Frank Diermeyer. "Survey on scenario-based safety assessment of automated vehicles". In: *IEEE Access* 8 (2020), pp. 87456–87477.

[49] Hermann Winner, Karsten Lemmer, Thomas Form, and Jens Mazzega. "PEGASUS—first steps for the safe introduction of automated driving". en. In: *Lecture Notes in Mobility*. Cham: Springer International Publishing, 2019, pp. 185–195. (Visited on 02/11/2026).

[50] Till Menzel, Gerrit Bagschik, and Markus Maurer. "Scenarios for development, test and validation of automated vehicles". In: *arXiv [cs.SE]* (Jan. 5, 2018).

[51] Roland Galbas, Marcus Nolte, Ulrich Eberle, Hardi Hungar, Henning Mosebach, Nayel Fabian Salem, Helmut Schittenhelm, Jan Reich, Thomas Kirschbaum, and Lukas Westhofen. *VV methods safety assurance position paper*. en. Research rep. 2024.

[52] Gerrit Bagschik, Till Menzel, and Markus Maurer. "Ontology based scene creation for the development of automated vehicles". In: *2018 IEEE Intelligent Vehicles Symposium (IV)*. 2018 IEEE Intelligent Vehicles Symposium (IV) (Changshu). IEEE, June 2018, pp. 1813–1820.

[53] Yuan Gao, Mattia Piccinini, Korbinian Moller, Amr Alanwar, and Johannes Betz. "From Words to Collisions: LLM-guided evaluation and adversarial generation of safety-critical driving scenarios". In: *arXiv [cs.AI]* (July 18, 2025).

[54] ASAM e.V. *ASAM OpenSCENARIO: Dynamic Content in Open Driving Simulations*. Research rep. ASAM e.V., 2022.

[55] Sudhi Sinha and Young M Lee. "Challenges with developing and deploying AI models and applications in industrial systems". en. In: *Discov. Artif. Intell.* 4 (1 Aug. 16, 2024), pp. 1–19. (Visited on 09/25/2025).

[56] Marc Schmitt. "Deep learning in business analytics: A clash of expectations and reality". en. In: *International Journal of Information Management Data Insights* 3 (1 Apr. 2023), p. 100146. (Visited on 09/25/2025).

[57] Kaveh Shahedi, Matthew Khouzam, Heng Li, Maxime Lamothe, and Foutse Khomh. “From technical excellence to practical adoption: Lessons learned building an ML-enhanced trace analysis tool”. In: *arXiv [cs.SE]* (Aug. 2, 2025). (Visited on 09/25/2025).

[58] Johannes Lederer. “Statistical guarantees for sparse deep learning”. en. In: *Adv. Stat. Anal.* 108 (2 June 2024), pp. 231–258. (Visited on 09/18/2025).

[59] Xiaofan Zhou, Baiting Chen, Yu Gui, and Lu Cheng. “Conformal prediction: A data perspective”. en. In: *ACM Comput. Surv.* 58 (2 Jan. 31, 2026), pp. 1–37.

[60] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. “Pin: building customized program analysis tools with dynamic instrumentation”. en. In: *SIGPLAN Not.* 40 (6 June 12, 2005), pp. 190–200.

[61] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. en. In: *SIGPLAN Not.* 42 (6 June 10, 2007), pp. 89–100.

[62] Fabrice Bellard. “QEMU, a fast and portable dynamic translator”. In: *USENIX annual technical conference, FREENIX Track* (Apr. 10, 2005), pp. 41–46.

[63] Lingda Li, Thomas Flynn, and Adolfy Hoisie. “Learning generalizable program and architecture representations for performance modeling”. en. In: *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. SC24: International Conference for High Performance Computing, Networking, Storage and Analysis (Atlanta, GA, USA). IEEE, Nov. 17, 2024, pp. 1–15. (Visited on 09/10/2025).

[64] Lingda Li, Santosh Pandey, Thomas Flynn, Hang Liu, Noel Wheeler, and Adolfy Hoisie. “SimNet: Accurate and high-performance computer architecture simulation using deep learning”. en. In: *Proc. ACM Meas. Anal. Comput. Syst.* 6 (2 May 26, 2022), pp. 1–24. (Visited on 09/10/2025).

[65] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. “Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks”. In: *arXiv [cs.DC]* (Aug. 20, 2018).

[66] Santosh Pandey, Amir Yazdanbakhsh, and Hang Liu. “TAO: Re-thinking DL-based microarchitecture simulation”. en. In: *Proc. ACM Meas. Anal. Comput. Syst.* 8 (2 May 21, 2024), pp. 1–25.

[67] Alex Renda, Yishen Chen, Charith Mendis, and Michael Carbin. “DiffTune: Optimizing CPU simulator parameters with learned differentiable surrogates”. In: *arXiv [cs.LG]* (Oct. 8, 2020). (Visited on 09/10/2025).

[68] Pulei Xiong, Scott Buffett, Shahrear Iqbal, Philippe Lamontagne, Mohammad Mamun, and Heather Molyneaux. “Towards a robust and trustworthy machine learning system development: An engineering perspective”. en. In: *J. Inf. Secur. Appl.* 65 (103121 Mar. 2022), p. 103121. (Visited on 09/18/2025).

[69] Jakob Gawlikowski, Cedrique Rovile Njieutcheu Tassi, Mohsin Ali, Jongseok Lee, Matthias Humt, Jianxiang Feng, Anna Kruspe, Rudolph Triebel, Peter Jung, Ribana Roscher, Muhammad Shahzad, Wen Yang, Richard Bamler, and Xiao Xiang Zhu. “A survey of uncertainty in deep neural networks”. en. In: *Artif. Intell. Rev.* 56 (S1 Oct. 29, 2023), pp. 1513–1589. (Visited on 09/18/2025).

[70] Rob Ashmore, Radu Calinescu, and Colin Paterson. “Assuring the machine learning lifecycle: Desiderata, methods, and challenges”. In: *arXiv [cs.LG]* (May 10, 2019). (Visited on 09/18/2025).

[71] Nijat Mehdiyev, Maxim Majlatow, and Peter Fettke. “Quantifying and explaining machine learning uncertainty in predictive process monitoring: an operations research perspective”. en. In: *Ann. Oper. Res.* 347 (2 Apr. 2025), pp. 991–1030. (Visited on 09/18/2025).

[72] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. “Code as policies: Language model programs for embodied control”. In: *arXiv [cs.RO]* (Sept. 16, 2022). (Visited on 09/10/2025).

[73] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. “Language models as zero-shot planners: Extracting actionable knowledge for embodied agents”. In: *arXiv [cs.LG]* (Jan. 18, 2022). (Visited on 09/10/2025).

[74] Ali Nouri, Johan Andersson, Kailash De Jesus Hornig, Zhennan Fei, Emil Knabe, Hakan Sivencrona, Beatriz Cabrero-Daniel, and Christian Berger. “On simulation-guided LLM-based code generation for safe autonomous driving software”. en. In: *arXiv [cs.SE]* (Apr. 2, 2025). (Visited on 09/10/2025).

[75] Yicheng Xiao, Yangyang Sun, and Yicheng Lin. “ML-SceGen: A Multi-level Scenario Generation Framework”. en. In: *arXiv [cs.AI]* (Jan. 18, 2025). (Visited on 09/10/2025).

[76] Roel J Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. en. 2014th ed. Berlin, Germany: Springer, Nov. 20, 2014. 332 pp. (Visited on 09/24/2025).

[77] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. "Design science in Information Systems Research1". en. In: *Manag. Inf. Syst. Q.* 28 (1 Mar. 1, 2004), pp. 75–106. (Visited on 09/24/2025).

[78] J W Creswell and J D Creswell. "Research design: Qualitative, quantitative, and mixed methods approaches". In: (2017).

[79] Hilary J Holz, Anne Applin, Bruria Haberman, Donald Joyce, Helen Purchase, and Catherine Reed. "Research methods in computing: what are they, and how should we teach them?" en. In: *SIGCSE Bull.* 38 (4 Dec. 26, 2006), pp. 96–114. (Visited on 09/26/2025).

[80] Glenn Shafer and Vladimir Vovk. "A tutorial on conformal prediction". In: *arXiv [cs.LG]* (June 21, 2007).

[81] Ken Brown. "Generation and search methods in design: Discussion". en. In: *Advances in Formal Design Methods for CAD*. Boston, MA: Springer US, 1996, pp. 97–103. (Visited on 09/24/2025).

[82] Joan E van Aken. "Design science: Valid knowledge for Socio-technical system design". en. In: *Design Science: Perspectives from Europe*. Communications in Computer and Information Science. Cham: Springer International Publishing, 2013, pp. 1–13. (Visited on 09/24/2025).

[83] Ali Nouri, Beatriz Cabrero-Daniel, Zhennan Fei, Krishna Ronanki, Håkan Sivencrona, and Christian Berger. "Large Language Models in code co-generation for safe autonomous vehicles". In: *arXiv [cs.SE]* (May 26, 2025).

[84] esmini. *esmini: a basic OpenSCENARIO player*. en. Comp. software. 2025. (Visited on 03/25/2026).

[85] Yizhak Yisrael Elboher, Elazar Cohen, and Guy Katz. "Neural Network Verification Using Residual Reasoning". In: *Software Engineering and Formal Methods*. Ed. by Bernd-Holger Schlingloff and Ming Chai. Cham: Springer International Publishing, 2022, pp. 173–189.

[86] Sumathi Gokulanathan, Alexander Feldsher, Adi Malca, Clark Barrett, and Guy Katz. "Simplifying neural networks using formal verification". In: *Lecture Notes in Computer Science*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 85–93.

[87] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. "Beta-CROWN: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network robustness verification". In: *arXiv [cs.LG]* (Mar. 11, 2021).

[88] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. "Efficient neural network robustness certification with general activation functions". In: *arXiv [cs.LG]* (Nov. 2, 2018).

[89] Yizhak Yisrael Elboher, Omri Isac, Guy Katz, Tobias Ladner, and Haoze Wu. "Abstraction-based proof production in formal verification of neural networks". In: *arXiv [cs.LO]* (June 11, 2025). (Visited on 09/17/2025).

[90] Ian T Jolliffe and Jorge Cadima. "Principal component analysis: a review and recent developments". en. In: *Philos. Trans. A Math. Phys. Eng. Sci.* 374 (2065 Apr. 13, 2016), p. 20150202.

[91] Vipin Kumar. "Feature Selection: A literature Review". In: *Thesmart Comput. Rev.* 4 (3 June 30, 2014), pp. 211–229.

[92] Rina Foygel Barber, Emmanuel J Candes, Aaditya Ramdas, and Ryan J Tibshirani. "Predictive inference with the jackknife+". In: *arXiv [stat.ME]* (May 8, 2019).

[93] Salim I Amoukou and Nicolas J B Brunel. "Adaptive Conformal Prediction by reweighting nonconformity score". In: *arXiv [stat.ML]* (Mar. 22, 2023).

[94] Ciaran O'Connor, Mohamed Bahloul, Roberto Rossi, Steven Prestwich, and Andrea Visentin. "Conformal Prediction for electricity price forecasting in the day-ahead and real-time balancing market". en. In: *Energy and AI* 21 (100571 Sept. 1, 2025), p. 100571. (Visited on 09/18/2025).

[95] Jorge De la Torre, Leticia R Rodriguez, Francisco E L Monteagudo, Leonel R Arredondo, and José B Enriquez. "Electricity price forecast in wholesale markets using conformal prediction: Case study in Mexico". en. In: *Energy Sci. Eng.* 12 (3 Mar. 2024), pp. 524–540.

[96] Matthias Wess, Daniel Schnöll, Dominik Dallinger, Matthias Bittner, and Axel Jantsch. "Conformal prediction based confidence for latency estimation of DNN accelerators: A black-box approach". In: *IEEE Access* 12 (2024), pp. 109847–109860.

[97] Rishikesh Jha, Arjun Karuvally, Saket Tiwari, and J Eliot B Moss. "Cache Miss Rate Predictability via Neural Networks". In: (2018).

[98] Yuan Zeng and Xiaochen Guo. "Long short term memory based hardware prefetcher: a case study". In: *Proceedings of the International Symposium on Memory Systems*. MEMSYS 2017: The International Symposium on Memory Systems, 2017 (Alexandria Virginia). New York, NY, USA: ACM, Oct. 2, 2017, pp. 305–311.

[99] Yao Deng, Jiaohong Yao, Zhi Tu, Xi Zheng, Mengshi Zhang, and Tianyi Zhang. "TARGET: Automated scenario generation from traffic rules for testing autonomous vehicles via validated LLM-guided knowledge extraction". In: *arXiv [cs.SE]* (May 10, 2023). (Visited on 07/25/2025).

[100] Yongqi Zhao, Wenbo Xiao, Tomislav Mihalj, Jia Hu, and Arno Eichberger. "Chat2Scenario: Scenario extraction from dataset through utilization of large language model". In: *2024 IEEE Intelligent Vehicles Symposium (IV)*. 2024 IEEE Intelligent Vehicle Symposium (IV) (Jeju Island, Korea, Republic of). IEEE, June 2, 2024. (Visited on 08/27/2025).

[101] Karim Elmaaroufi, Devan Shanker, Ana Cismaru, Marcell Vazquez-Chanlatte, Alberto Sangiovanni-Vincentelli, Matei Zaharia, and Sanjit A Seshia. "ScenicNL: Generating probabilistic scenario programs from natural language". In: *arXiv [cs.SE]* (May 3, 2024).

[102] Vladimir Vovk. "Conditional validity of inductive conformal predictors". en. In: *Mach. Learn.* 92 (2-3 Sept. 2013), pp. 349–376.

Part II

Included Papers

Chapter 8

Paper A: Abstraction-Based Reduction of Input Size for Neural Networks

Peter Backeman, Edin Jelačić, Cristina Seceleanu, Ning Xiong, Tiberiu Seceleanu

Published in: First International Conference on Bridging the Gap between AI and Reality (AISoLA 2023)

Note: This paper has been reformatted to comply with the thesis layout. The content is unchanged from the published version.

Abstract

Machine learning is an increasingly popular method for modeling complex systems, to make predictions or recognize data patterns. A common machine learning model is the neural network, which can be trained to represent complicated functions to a high accuracy. While neural networks often grow large and complex, recent work is looking in how to abstract networks to yield simpler representations, while retaining some property of the original network. For instance, for every input, the abstracted network's output should be at least as large as the original. In this work, we build on previous ideas and extend them to allow for removing inputs, obtaining an under/over-approximating network instead. Further, we show how to combine these approximating networks to identify inputs which have a low impact on the final output.

8.1 Introduction

The advent of machine learning algorithms has led to their application for classification, decision making, and data analytics of complex systems, in various fields. By presenting a machine learning (ML) algorithm with a set of training patterns (relating inputs to outputs), one can obtain a model that can predict the output for an arbitrary, unseen input (in the input domain). *Neural networks* (NN), as a subset of ML approaches, can be trained to represent complicated functions to a high accuracy, however they often grow large and complex. Recent work has been studying ways of abstracting neural networks to yield simpler representations, without breaching desirable properties of the original network, e.g., for every input the abstracted network's output is at least as large as the original.

In some cases the input vector has a large size, yet only a few of the elements are significant in the computation of the output. To reduce the size of the input vector, one can for example apply principal component analysis (PCA) to obtain a smaller representation while retaining the most important information [1]. In this paper, we show how a combination of an over-approximating and an under-approximating network can be used to identify insignificant input elements (i.e., not affecting the output by more than a relatively small *delta*). While the PCA method generates a reduced dimensional data representation, our approach focuses on identifying a subset of elements to represent the data. This approach imposes tighter constraints but facilitates a more direct connection between the new input domain and the original one, as it remains a subset, enabling a form of feature selection [2]. Furthermore, it allows us to obtain information regarding individual variables. Understanding the insignificance of certain inputs can be valuable in itself, by its potential to reveal properties about the original problem.

In this work, we build on previous ideas [3], and extend them to also consider the input layer of NN. We study how input nodes can be eliminated from a network in such a manner that an over/under-approximating network is obtained. Such an abstracted network would not be as accurate, but can be used for analysis, as well as for gaining an understanding of the network model. It should be noted that this method works on the final trained model, which means it is applicable in scenarios where the original training data may not be available.

The contributions of this paper are as follows:

- A novel method for abstracting away selected inputs from a neural network, obtaining an over/under-approximation of the latter,

- A method for obtaining a worst-case measure of the impact of a particular input to a neural network,
- A small experimental evaluation demonstrating how the proposed methods can be applied.

We begin by presenting background in Section 11.2, that is, a brief presentation of neural networks, including verification and abstraction techniques. In Section 8.3, we show our method of how inputs can be removed to obtain an over/under-approximating network with smaller dimension, followed in Section 8.4 by our proposed method of identifying insignificant inputs. We perform a small experimental evaluation in Section 8.5. Finally, we present our conclusions and future work in Section 8.7.

8.2 Background

We start by introducing some notation regarding vectors. We use $\mathbf{x} = \{x_1, \dots, x_n\}$ to denote *vectors* from domain $\bar{X}$, where $\mathbf{x}[i] = x_i$. We denote substitution of the i-th element by c as $\mathbf{x}[x_i = c] = \{x_1, \dots, x_{i-1}, c, x_{i+1}, \dots, x_n\}$. In this paper all vectors are over real numbers. Given a vector $\mathbf{x} = \{x_1, \dots, x_i, \dots, x_n\}$, let $\mathbf{x}^{+i} = \{x_1, \dots, x_{i-1}, x_i, x_i, x_{i+1} \dots, x_n\}$, that is the vector $\mathbf{x}$ with the i-th element repeated once, and $\mathbf{x}^{-i} = \{x_1, \dots, x_{i-1}, x_{i+1} \dots, x_n\}$, the vector $\mathbf{x}$ with x_i removed. We let X_i^{max} and X_i^{min} represent the maximum and minimum values of the domain of x_i, respectively.

8.2.1 Neural Networks

Neural networks (NN) are a widely used form of machine learning [4]. They consist of interconnected neurons trained on input-output pairs to learn functions and make predictions of their output values. Each neuron - denoted by capital letters - in a layer, is connected to each neuron in the previous layer via an edge with a *weight*. When computing the output value of a node, an activation function is applied to the sum consisting of a node bias and each weight of every incoming edge multiplied by the output value of its corresponding node. In this paper, we focus on networks that contain the input layer (nodes $I_1, I_2, \dots$), a number (L) of hidden layers (nodes $H_1^1, H_2^1, \dots,\ H_1^2, H_2^2 \dots,\ \dots,\ H_1^L, H_2^L,\ \dots$) and an output layer with a single output node (O). We will use $\forall H_i^\ell \in H^\ell$ to express statements about all nodes in layer H^ℓ.

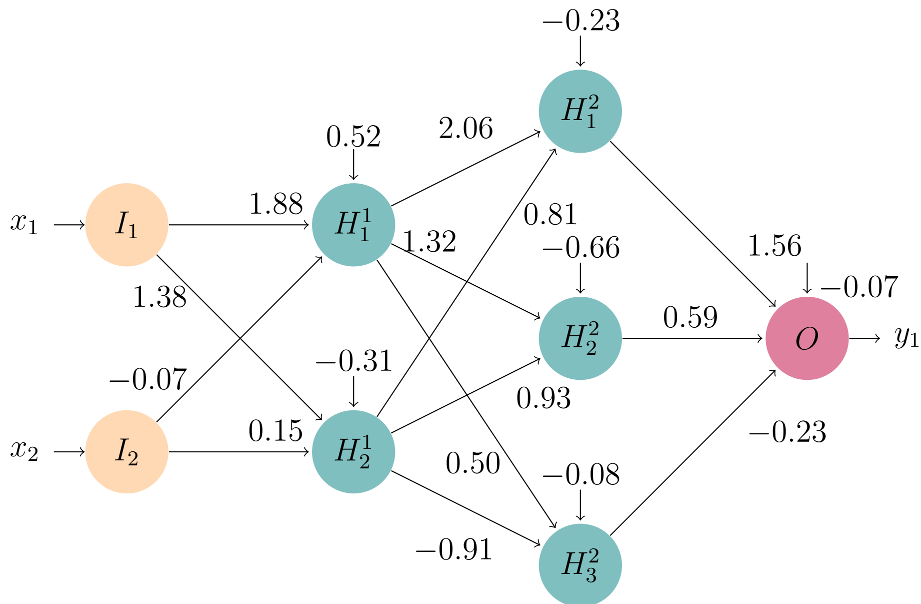


Figure 8.1: Simple neural network.

The input and output layer has no activation (i.e., the identity function), while the hidden layers use the ReLU activation function, i.e.,:

$$
\begin{aligned}
H_i^1 &= \mathrm{ReLU}(\mathbf{W^1}[i]\mathbf{I} + \mathbf{B^1}[i]) \\
H_i^\ell &= \mathrm{ReLU}(\mathbf{W}^\ell[i]\mathbf{H}^{\ell-\mathbf{1}} + \mathbf{B}^\ell[i]) \qquad \text{for} \quad 1 < \ell \leq L \\
O &= \mathbf{W^O}[i]\mathbf{H^L} + \mathbf{B^O}
\end{aligned}
$$

where $\mathbf{I}$ is the output of the input layer, $\mathbf{W}^\ell[i], \mathbf{B}^\ell[i]$ are the weights and the bias of node H_i^ℓ, $\mathbf{W^O}[i]$ and $\mathbf{B^O}$ are weights and the bias for the output node, and ReLU is the nonlinear activation function. Let $e(I_i, H_j^1)$ be the weight of the edge connecting input node I_i with hidden layer node H_j^1, and $e(H_i^\ell, H_j^{\ell+1})$ the weight of the edge connecting hidden layer node H_i^ℓ and $H_j^{\ell+1}$ (and similarly for $e(H_i^\ell, O_1)$. We use $NN(\mathbf{x})$ to denote the output of the neural network NN with input $\mathbf{x}$.

Example 4. *In Figure 8.1 a simple neural network NN is presented. It has two input nodes I_1, I_2, two layers of hidden nodes $H^1 = \{H_1^1, H_2^1\}$ and $H^2 = \{H_1^2, H_2^2, H_3^2\}$, and one output node O. There are also weighted edges, e.g. $e(I_1, H_2^1) = 1.38$, $e(H_2^1, H_1^2) = 0.81$ and*

$e(H_2^2, O_1) = 0.59$. *Consider feeding the input vector* $\mathbf{x} = (10, 10)$ *into the network, then*[1]*:*

$$H_1^1 = \text{ReLU}(1.88x_1 - 0.07x_2 + 0.52) = 18.62$$
$$H_2^1 = \text{ReLU}(1.38x_1 + 0.15x_2 + -0.31) = 14.99$$
$$H_1^2 = \text{ReLU}(2.06H_1^1 + 0.81H_2^1 - 0.23) = 50.27$$
$$\ldots$$
$$O = 1.56H_1^2 + 0.59H_2^2 - 0.23H_3^2 - 0.07 \approx 100,7$$

Thus $NN(\mathbf{x}) \approx 100.7$

8.2.2 Verification of NNs

Recently, significant efforts have been put into the verification of NNs [5], and tools for checking various aspects of NNs have arisen in the past several years [6]. For example, the verification tool `Marabou` [7] can prove a linear bound on the output under given constraint on the input in a neural network with piece-wise linear activation functions, of which ReLU is one. We use it by asking for an assignment to a network yielding an output violating the bound, and if none is found the bound is proven.

Example 5. *Consider the neural network presented in Fig. 8.1. Let's say that when we restrict the real-valued inputs* x_1 *and* x_2 *to the range* $[0, 10]$*, it appears that the output is strictly less than* 101*. To rigorously establish this, we can employ* `Marabou` *by setting input constraints as follows:* $x_1, x_2 \in [0, 10]$*, and defining the output bound as* $NN(\mathbf{x}) > 100$ *(note that, in practice, we formulate a bound on the negated output:* $-NN(\mathbf{x}) \leq -100$*, since* `Marabou` *requires the constraint to be in the form of a less-than-or-equal comparison).* `Marabou` *will in this case return that the given constraints are unsatisfiable, verifying that* 101 *is indeed an upper bound for the output of the network (when* $x_1, x_2 \in [0, 10]$*).*

8.2.3 Abstracting NNs

In this section, we present a short summary of the work by Elboher et al. [3], as our work is significantly based upon it. In the mentioned work, the authors present a methodology for abstracting and refining neural networks. The motivation is to create, for a given neural network NN, a simpler network (i.e., with fewer nodes) NN' such that $NN(\mathbf{x}) \leq NN'(\mathbf{x})$. This

[1]For convenience, we will overload and use the notation N_i to refer both to the node N_i, and its output value.

network can then be used for verification of upper bounds; since the abstract network is an over-approximation, any upper bound for NN' is also an upper bound for NN.

Coloring.

The core idea of the methodology of Elboher et al. is to classify all nodes into categories, depending on how each node is contributing to the final output. The nodes are grouped as *positive/negative*, as well as *increasing/decreasing*. The former group contains nodes that have all outgoing edges with positive/negative values, respectively, while the latter has nodes such that increasing the output of the node would increase/decrease the output of the network. In this paper, we only consider the second category, i.e., we only care if nodes are increasing or decreasing, and use the following change of terminology: we *color* a node N *green* if increasing the input value to N results in the final network output increasing. Analogously, if increasing the input value to a node N would lead to the total output of the network decreasing, we color the node *red*. It can be the case that a node cannot be colored either *green* or *red*, then we call the node *colorless*.

We can iteratively color all the nodes in a network. The process begins with the output node (we assume networks have exactly one output node in the final layer). The output node is trivially *green*. To color each node H_i^L in the second to last layer (the final hidden layer H^L), we can use the following formula:

$$\text{color}(H_i^L) = \begin{cases} green, & \text{if } e(H_i^L, O) \geq 0 \\ red, & \text{if } e(H_i^L, O) < 0 \end{cases}$$

Intuitively, if a node is connected to the output node with a positive weight, increasing its output will increase the output of the network, thus it is *green* (and v.v.). Note that if for a node H_i^ℓ, $e(H_i^\ell, O) = 0$, the node H_i^ℓ has no non-zero outgoing weight and can be ignored as it can not affect the output of the network. In the remainder of the paper, we frequently write $green(N)$ as a short-hand for $color(N) = green$ (and analogously for *red*).

The situation is a bit more complicated for the preceding hidden layers. Assume that layer $H^{\ell+1}$ has been colored (i.e., all nodes $H_i^{\ell+1}$ are either *green* or *red*). Now, each node that is a node in layer H^ℓ can be colored according to the following formula:

$$\text{color}(H_i^\ell) = \begin{cases} green, & \text{if } \forall H_k^{\ell+1} \in H^{\ell+1} \quad (e(H_i^\ell, H_k^{l+1}) \geq 0 \wedge green(H_k^{\ell+1})) \vee \\ & (e(H_i^\ell, H_k^{\ell+1}) \leq 0 \wedge red(H_k^{\ell+1})) \\ red, & \text{if } \forall H_k^{\ell+1} \in H^{\ell+1} \quad (e(H_i^\ell, H_k^{\ell+1}) \geq 0 \wedge red(H_k^{\ell+1})) \vee \\ & (e(H_i^\ell, H_k^{\ell+1}) \leq 0 \wedge green(H_k^{\ell+1})) \end{cases}$$

Intuitively, this means that if a node is connected to only *green* nodes with positive weights or *red* nodes with negative weights, it will increase the total output of the network, and is thus *green*. On the other hand, if a node is connected to only *green* nodes with negative weights or *red* nodes with positive weights, it will decrease the total output of the network, and is thus *red*. In a similar way as for the last hidden layer, if a node has no non-zero outgoing weights it can be ignored.

Example 6. *Consider the network in Fig. 8.1, if we increase the output of node H_1^2, the input of O_1 will increase, thus increasing the output of the network. Therefore, node H_1^2 is* green *(and likewise for H_2^2). On the other hand, increasing the output of node H_3^2 actually decreases the input of O (since it is multiplied by the weight -0.23) and therefore H_3^2 is* red. *In the preceding layer, increasing the output of H_2^1, will increase the input to both H_1^2 and H_2^2 (and since they are green, increasing the total output) and decrease the input to H_3^2 (which is red, thus also increasing total output). Thus, H_2^1 can be colored* green. *However, increasing the output of H_1^1 both increases the input to H_1^2 and H_3^2 which has a mixed effect on the total output of the network. Thus H_1^1 remains* colorless. *We show the (partially) colored network in Fig. 8.2.*

Splitting node.

Coloring nodes in a network might be impossible right away, as none of the two cases might apply. Then we proceed with a *split*. Splitting a node means replacing it by two nodes with the same incoming weights but each outgoing edge of the original node is only mapped to exactly one of the new nodes. This ensures that the two new nodes have the same output (as they have identical inputs), as well as the sum of the output of the two new nodes to each node in the next layer is identical to the output of the split node in the original network. Thus the resulting network computes the exact same function as before splitting, but has one extra node.

We can see that since we assign each edge of the original node to one of the new nodes, we can choose freely which edge to assign to which node. We let all positive edges to green nodes

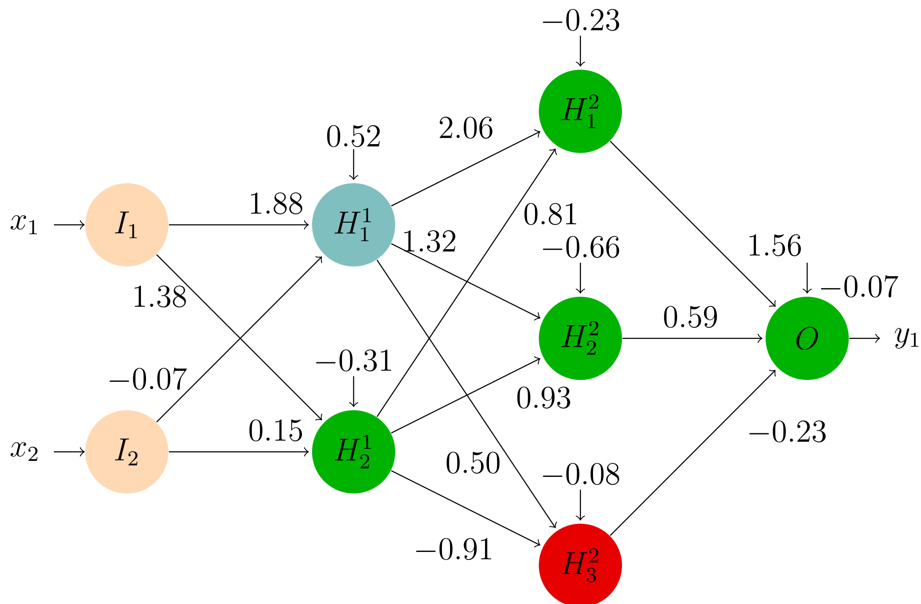


Figure 8.2: Colored neural network.

and negative edges to red nodes be assigned to the first node, and the remaining edges to the second node. Following this, it is easy to see that one of the new nodes will be colored *green*, and the other *red*. Formally, if we split node H_i^ℓ into two new nodes H_{i+}^ℓ and H_{i-}^ℓ, we assign edges such that in the new network the following holds for all nodes $H_j^{\ell+1}$ in the next layer (we let $h_i = H_i^\ell$ and $h_j = H_j^{\ell+1}$ for clarity):

$$e(H_{i+}^\ell, h_j) =$$

$$\begin{array}{ll} e(h_i, h_j) & \textbf{if}(e(h_i,h_j) \geq 0 \land \mathit{green}(h_j)) \lor (e(h_i,h_j) \leq 0 \land \mathit{red}(h_j)) \\ 0 & \textbf{if}(e(h_i,h_j) \geq 0 \land \mathit{red}(h_j)) \lor (e(h_i,h_j) \leq 0 \land \mathit{green}(h_j)) \end{array}$$

$$e(H_{i-}^\ell, h_j) =$$

$$\begin{array}{ll} 0 & \textbf{if}(e(h_i,h_j) \geq 0 \land \mathit{green}(h_j)) \lor (e(h_i,h_j) \leq 0 \land \mathit{red}(h_j)) \\ e(h_i, h_j) & \textbf{if}(e(h_i,h_j) \geq 0 \land \mathit{red}(h_j)) \lor (e(h_i,h_j) \leq 0 \land \mathit{green}(h_j)) \end{array}$$

Moreover, the incoming edges to H_{i+}^ℓ and H_{i-}^ℓ are identical to the edges to the node H_i^ℓ before splitting:

$$\forall H_j^{\ell-1} \in H^{\ell-1} \quad e(H_j^{\ell-1}, H_{i+}^\ell) = e(H_j^{\ell-1}, H_i^\ell) = e(H_j^{\ell-1}, H_{i-}^\ell)$$

We omit the special case when the preceding layer is the input layer.

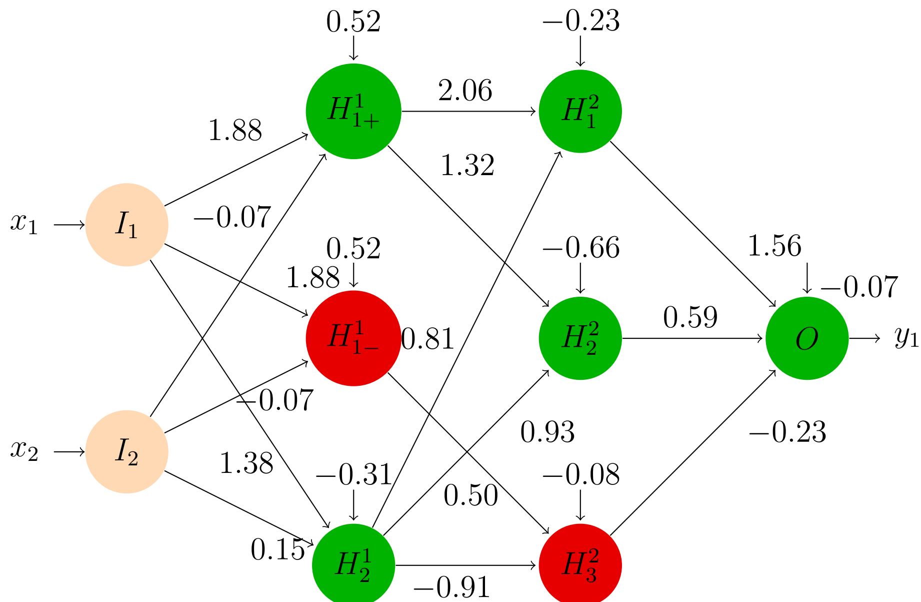


Figure 8.3: Colored split neural network. Edges with weight zero are omitted.

Example 7. *Once again, we consider the neural network we saw earlier. It could not be completely colored as H^1_1 remained* colorless. *Therefore, we split H^1_1 into two nodes H^1_{1+} and H^1_{1-} accordingly. The new network can be colored and is shown in Fig. 8.3. It should be noted that for any input vector, the networks in Fig. 8.2 and Fig. 8.3 compute exactly the same output value.*

8.3 Removing Inputs by Over/Under-estimation

In their work, Elboher et al. define an abstraction operator, to reduce the total number of nodes in the hidden layers of the network. However, in our work, we utilize the coloring in a different manner. In this section, we present how we can remove an input node x_i from a neural network NN creating an over-estimating network NN^+_i, such that:[2]

$$\forall \mathbf{x} : NN(\mathbf{x}) \leq NN^+_i(\mathbf{x}^{-i}). \tag{8.1}$$

Assume that the NN is already colored (with potentially split nodes) according to the methodology described in Sec. 8.2.3. To remove input x_i, we begin by splitting the node I_i (into two new nodes I^+_i and I^-_i, which are green and red respectively). For the resulting network NN'_i, we have $NN(\mathbf{x}) = NN'_i(\mathbf{x}^{+i})$, that is, if we duplicate the i:th element of the input

[2] $\mathbf{x}^{-i}$ refers to the vector $\mathbf{x}$ with the i:th input removed.

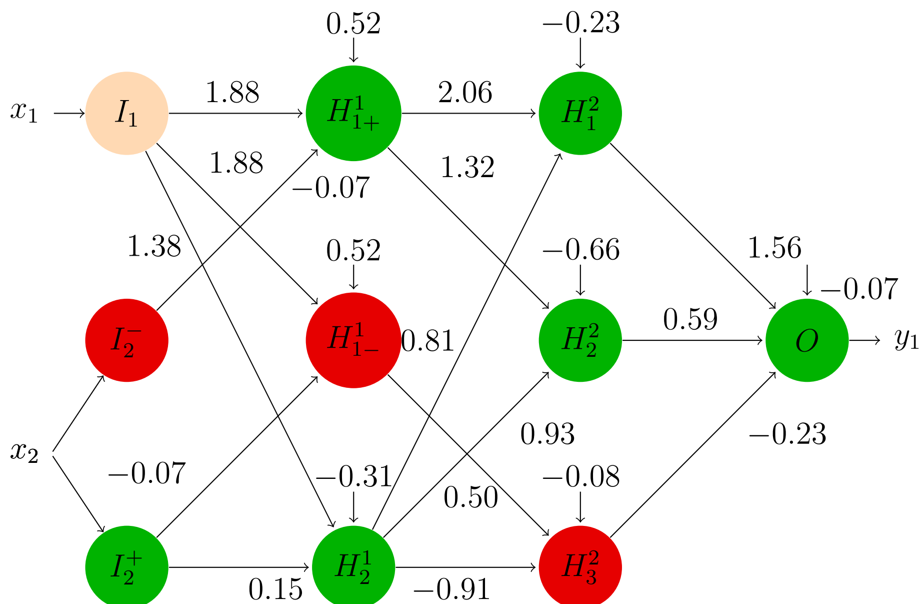


Figure 8.4: Neural network after the second input node has been split. Edges with weight zero are omitted.

vector and feed it into the new network, we have the same result as the original vector for the original network.[3].

Example 8. *We revisit the example neural network. If we split the second input node I_2, the resulting network is as shown in Fig. 8.4.*

Now, for any input vector to NN'_i, we know that increasing the input value of I_i^+ will increase the output of the network (as the node is green). In particular, if we change the input to be the maximum value, with X_i^{max} the network output can only grow:

$$NN'_i(\mathbf{x}^{i+}[x_i = X_I^{max}]) \geq NN'_i(\mathbf{x}^{i+})$$

Note that the i:th input corresponds to the green node I_i^+ (and $i+1$:th to the red node I_i^-). In similar vein, if we replace the value for input node I_i^- with X_i^{min} the output also grows:

$$NN'_i(\mathbf{x}^{i+}[x_{i+1} = X_I^{min}]) \geq NN'_i(\mathbf{x}^{i+})$$

With this in mind, we create a new network NN_i^+ by replacing the input nodes I_i^+ and I_i^- with constant inputs X_i^{max} and X_i^{min}, respectively. To achieve this, the bias for each node in the first hidden layer is increased with it's incoming weight from I_i^+ (I_i^-) multiplied with X_i^{max}

[3] $\mathbf{x}^{+i}$ refers to the vector $\mathbf{x}$ with the i:th input duplicated.

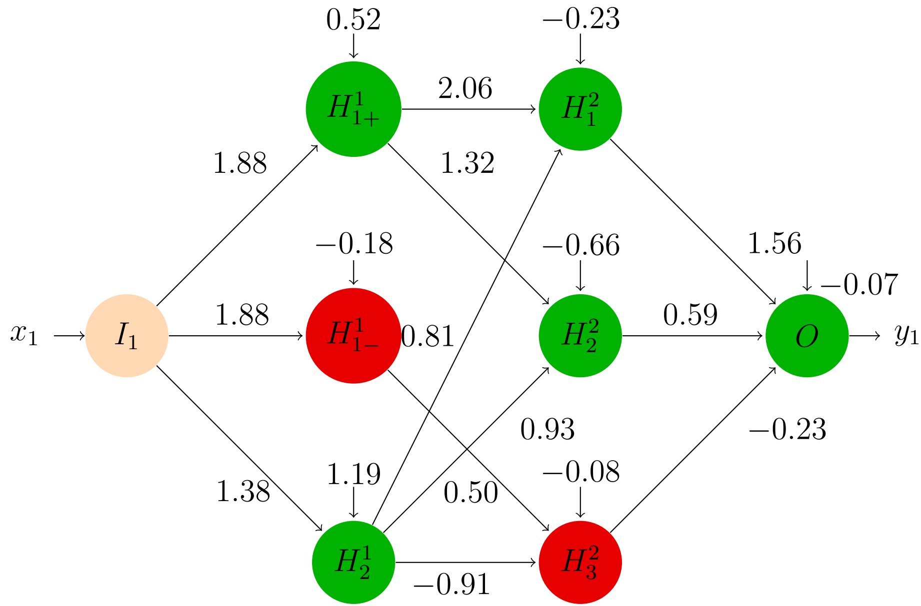


Figure 8.5: Neural network after the second input has been removed. Edges with weight zero are omitted.

(X_i^{min}). The procedure for doing this is outlined in Alg. 1. The new network can be fed input vectors, with the i:th input removed, and will compute an over-approximation of the result.

Algorithm 1: Algorithm for creating NN_i^+.

Input: Neural network NN and input node I_i

Output: Output Neural network NN_i^+

Color network NN;

Split I_i into I_i^+ and I_i^- s.t. I_i^+ can be colored green and I_i^- red;

for *each* $e(I_i^+, H_j^1) \neq 0$ **do**

$\quad \mathbf{B_{H^1}}[j] \leftarrow \mathbf{B_{H^1}}[j] + e(I_i^+, H_j^1) * X_i^{max}$;

for *each* $e(I_i^-, H_j^1) \neq 0$ **do**

$\quad \mathbf{B_{H^1}}[j] \leftarrow \mathbf{B_{H^1}}[j] + e(I_i^-, H_j^1) * X_i^{min}$;

Example 9. *Fig. 8.5 shows the resulting network of replacing I_2^+ and I_2^- with $X_2^{max} = 10$ and $X_2^{min} = 0$, and then propagating to the bias accordingly to the procedure shown in Alg. 1. Note that for any input vector, if we remove the second element and feed it into the network of Fig. 8.5, the result will always be equal or greater to feeding the full vector into any of the networks before I_2 has been removed.*

Theorem 1.

$$\forall \mathbf{x} : NN(\mathbf{x}) \leq NN_i^+(\mathbf{x}^{-i})$$

This theorem tells us that the neural network NN_i^+ is an over-approximating network, disregarding the ith input. The proof follows from the construction of the network (i.e., splitting and coloring).

We can in a symmetric way define a network NN_i^- underestimating the resulting value. The construction is similar to Alg. 1, except that X_i^{max} and X_i^{min} are swapped, i.e., for each green (red) node we replace with the minimum (maximum) value to minimize the output. Given a NN_i^- constructed as such, a symmetric theorem will hold.

Theorem 2.

$$\forall \mathbf{x} : NN(\mathbf{x}) \geq NN_i^-(\mathbf{x}^{-i})$$

8.4 Identifying Insignificant Inputs

One of the challenges in using neural networks, especially when we lack knowledge about the network's internal structure, is dealing with the potentially high dimensionality of input data. Input vectors can comprise hundreds, or even more, values. In some cases, only a subset of these inputs may significantly influence the output. In this section, we demonstrate an approach that can help to identify inputs with a low impact on the output of the network. We use the following to define insignificant inputs:

Definition 1. *For a NN with input range $\bar{X}$ and a particular input x_i, we call x_i* insignificant *(w.r.t. some $T \in \mathbb{Z}$) if*

$$\forall \mathbf{x} = \{x_0, \ldots, x_n\} \in \bar{X} : \left(\max_{c \in X_i} NN(\mathbf{x}[x_i = c]) - \min_{c \in X_i} NN(\mathbf{x}[x_i = c]) \right) \leq T$$

Intuitively, this means that for any input vector, varying x_i will not change the resulting output by more than a threshold T. For a given neural network, it can be interesting to identify the insignificant inputs, as for low enough T, these could be disregarded as their impact is negligible.

Example 10. *Consider a neural network designed to estimate the total load on a computer system, where each input x_i is either zero or one, indicating whether a function f_i of the system is activated. The output should be a prediction of the CPU load of the system as a percentage.*

If a function f_i can be identified as insignificant w.r.t. to $L = 1$, it can be deduced to not affect the final load by more than 1%.

Computing the maximum and minimum values as required in Def. 1 by enumeration can take a prohibitively long time due to large domains. Instead, we propose to use the `Marabou` verification tool to establish if a value is insignificant or not. We begin by presenting how we can construct a *difference neural network*.

Definition 2. *We define a difference neural network for an input node x_i as NN_i^{diff}, such that,*

$$NN_i^{diff}(\mathbf{x}) = NN_i^{+}(\mathbf{x}) - NN_i^{-}(\mathbf{x})$$

Intuitively, a difference network NN_i^{diff} bounds for an input vector $\mathbf{x}^{-i}$ how much the i:th input affects the total output. It works by taking the difference between the over-approximation and under-approximation for the input vector, thus measuring a maximum change between the the highest and lowest possible values. We present a high-level algorithm for constructing a difference network in Alg. 2 for a particular input x_i. The resulting network can then be analyzed using `Marabou` to establish if it is the case that the output is less than the limit L. If this is the case, this fulfills the condition of Def. 1, hence x_i can be deemed insignificant.

Algorithm 2: Algorithm for creating a difference network.

Input: Neural network NN, Input node I_i

Output: Output result Difference neural network NN_i^{diff}

Create NN_i^{+}, NN_i^{-} as described in Alg. 1.;

Merge the two input layers;

Create an extra layer, computing the difference between the two output nodes;

Example 11. *Consider the neural networks in Fig. 8.6. The three steps of the methodology are shown. First a neural network is obtained after training on a data set generated by the function $f(x_1, x_2) = 100x_1 + x_2$. Next we remove the first input I_1 to create an over-approximating network (the under-approximating network is not shown). Finally, we apply Alg. 2 to obtain the bottom network shown in Fig. 8.6. Note that the upper and lower half of the network are identical except for the biases on the hidden nodes, as well as the weight from the subtraction layer to the output layer (one for top half, negative one for bottom half). If we apply* `Marabou` *and verify, the results indicates that the first input is significant, as expected.*

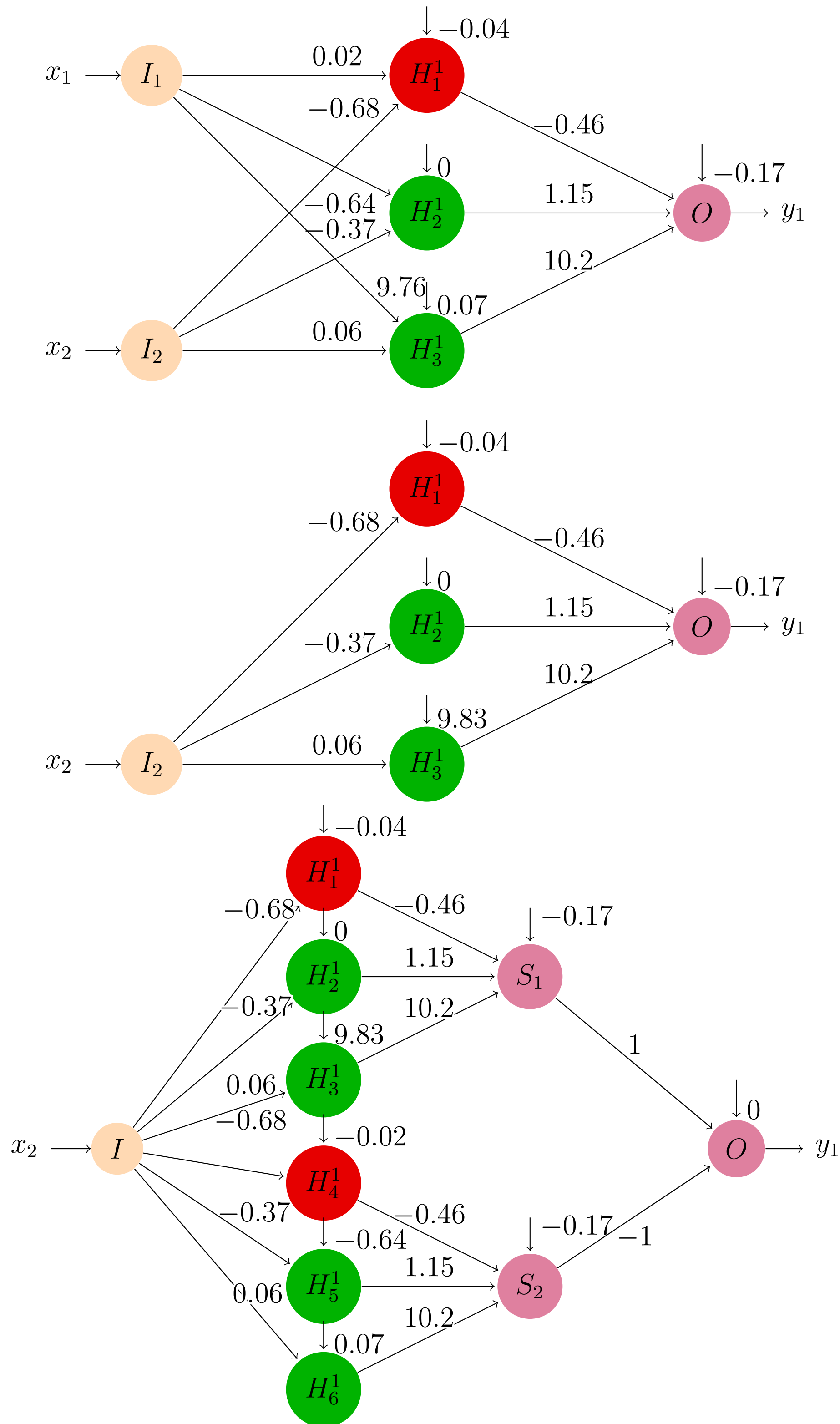


Figure 8.6: (Top) A neural network trained to compute $f(x_1, x_2) = 100x_1 + x_2$. (Middle) The network after I_1 is removed. (Bottom) The difference network NN_1^{diff}. We omit all edges with weight zero.

8.4.1 Estimating Impact of Inputs

In this section, we explore the idea of not only finding insignificant inputs, but estimating the *maximum impact of a particular input*. A difference network tells us for an input node x_i an upper bound on the change in the output for a particular assignment to the remaining inputs. Thus, if we can establish an upper bound on the output of a difference network, *we can establish an upper bound on the impact for the particular input x_i.* `Marabou` can verify if the output of a network is limited by some constant b. By applying this verification repeatedly, we can establish a bound on the difference network. We use a *binary-split* approach shown in Alg. 3 (note that the query $\forall \mathbf{x} NN_i^{diff}(\mathbf{x}) > bound$ is answered by one query to `Marabou`).

The binary approach considers the inputs one-by-one: it takes a lower bound (assumed to be zero) and an upper bound and checks if the middle point is a bound. If this is the case, the true bound lies somewhere in the lower half of the interval, otherwise in the upper half. In this way the search interval is repeatedly split by two until it is sufficiently small. Note, if the upper bound is too small, an erroneous bound will be reported (as the search will not investigate bounds above the original upper bound). This can be alleviated by initially checking if the upper bound is a true upper bound.

Algorithm 3: Algorithm for the binary approach.

Input: Neural network NN and input node I_i

Input: Upper bound UB for search

Output: An upper bound of the impact of input node I_i

Create difference network NN_i^{diff};

$lower \leftarrow 0$;

$upper \leftarrow UB$;

$bound \leftarrow UB/2$;

while $upper > lower + 1$ **do**

    if $\forall \mathbf{x} NN_i^{diff}(\mathbf{x}) > bound$ **then**

        $lower \leftarrow bound$;

    else

        $upper \leftarrow bound$;

return $bound$;

8.5 Experimental Evaluation

In this section, we perform two small studies to evaluate the approach presented in this paper. The implementation of the algorithms presented above can be found at `https://github.com/ptrbman/neuralnetworkabstraction`.

8.5.1 Polynomial Coefficient Estimation

Consider a polynomial of the form $f(\bar{x}) = c_1x_1 + c_2x_2 + \cdots + c_nx_n$. If we restrict $x_i \in \{0, 1\}$ it is clear that the maximum impact of any variable x_i is equal to c_i, as when changing x_i from zero to one (or v.v.) will affect the final sum by at most c_i. We generate ten random polynomials of this form with $n = 10$ and $0 < c_i \leq 10$ and the resulting estimated impact of each input is shown in Table 8.1. It is noteworthy that the estimated coefficient is very often quite close to the original, except in the third case where many of the estimations are far off. This is probably due to a poorly trained network with a high validation error. The results demonstrates that the approach is capable of extracting information from a trained neural network. Each binary search requires around 60 queries from `Marabou`, where each query takes around 1 millisecond. The total time is about 16 seconds (with overhead coming from reloading the model from disk for each query).

8.5.2 Identifying Demanding Functions

We exemplify our approach on an industrial example: a certain embedded device has multiple interconnected software components that are operated by switching them or their subcomponents “on" or “off". Every set of these values (concretely, 1 for “on" and 0 for “off"), is deemed a configuration of the device. Every configuration loaded onto the device and activated exerts a certain load on the device’s CPU. We create a neural network that predicts the expected CPU load of the embedded device under a certain configuration. Afterwards, we can apply the methodology presented in this paper to estimate the impact of each input, using the maximum (1) and minimum (0) values (the respective function being enabled and disabled, respectively). The final results are shown in Table 8.2. These values provide the knowledge that, under the assumption that the neural network is accurate, the functions $0, 1$ and 3, never affect the CPU load by more than roughly one percent, while function 11 has by far the greatest impact on the CPU load (but turning it off does not save more than roughly ten percent). The `Marabou`

c_i	6	3	3	4	10	6	2	8	5	7
est.	6.640	4.296	3.515	4.296	11.328	6.640	1.953	8.984	5.078	7.421
c_i	5	4	7	8	10	6	10	1	2	4
est.	5.078	4.296	7.421	8.203	10.546	6.640	10.546	1.171	2.734	4.296
c_i	1	1	6	4	3	1	4	9	6	9
est.	4.296	6.640	8.203	6.640	5.078	3.515	5.859	8.984	5.859	8.984
c_i	4	5	4	5	7	3	6	4	2	5
est.	4.296	5.078	4.296	5.078	7.421	2.734	5.859	4.296	1.953	5.078
c_i	9	4	4	7	9	5	4	5	10	9
est.	8.984	5.859	6.640	8.984	8.984	8.203	8.984	5.078	9.765	8.984
c_i	7	5	3	3	1	6	7	4	1	1
est.	8.203	5.078	2.734	4.296	3.515	5.859	8.203	4.296	2.734	3.515
c_i	3	3	4	4	2	5	10	9	6	6
est.	3.515	3.515	4.296	4.296	1.953	5.078	10.546	9.765	6.640	6.640
c_i	10	4	10	10	3	6	10	3	4	3
est.	9.765	5.078	9.765	9.765	5.078	8.203	9.765	3.515	8.203	3.515
c_i	1	2	5	8	5	3	10	8	4	7
est.	1.171	3.515	5.859	8.984	5.859	3.515	11.328	8.984	4.296	8.203
c_i	6	7	10	8	9	3	6	10	1	6
est.	5.859	6.640	10.546	8.203	8.984	8.203	5.859	9.765	6.640	5.859

Table 8.1: Estimated coefficients of linear polynomial. The top value corresponds to the actual coefficient and the bottom value is the estimated.

Function	0	1	2	3	4	5	6	7	8	9	10	11	12
Est. impact	1.17	1.17	1.95	1.17	1.95	1.95	1.95	2.73	3.51	1.95	2.73	9.76	2.73

Table 8.2: The maximum impact on CPU load different functions in industrial example.

solver was queried 52 times, with each query taking between 1 and 500 milliseconds. The total time for estimating the bounds was about 26 seconds (once again with overhead coming from reloading the model from disk for each query).

8.5.3 Repeated Experiments

Repeated experiments on neural network training are complicated as the outcome depends on the initial (randomized) weights. Thus, the result may vary between experiments. Noteworthy is the fact that verification of a network depends on its contained weights, that is, two networks with the same structure (e.g., nodes, layers) but different weights might have significant variation in the time that it takes to verify a certain property. It has been encountered during our experiments that certain networks take a long time to verify. Most often, this seems to be related to a high validation loss for the trained network. As our methodology does not really value any output from such a network (if the validation loss is high, the predicted accuracy of insignificant inputs is very low), we would abort such a task and restart the training with the goal of ensuring a more accurate model before continuing with the rest of the process.

8.6 Related Work

This work is mainly based on previous research found in the literature [3], and there have been several extensions or related work closely resembling the topic of the mentioned paper. One extension looks into how to retain information between different verification queries on similar networks [8]. This could also be possible for our case, perhaps the removal of two different insignificant input nodes requires much of the same work. Another example identifies nodes in a network that always produce an output almost zero (thus enabling the removal of those nodes) [9]. In another related work [10], the authors reformulate a Deep Neural Network (DNN) minimization problem as a DNN verification problem, and construct a provably minimal network (that is still sufficiently close to the original).

Moreover, there are different approaches towards abstracting neural networks, e.g., [11]. It

would be interesting to study if these methods could also be extended to the input layer, as they give different kinds of guarantees on the abstraction, potentially enabling different use cases.

Identifying insignificant inputs can be seen as a form of *feature selection*, as studied in the literature [2]. However, since we aim for an approximated result, our method is allowed a bit more leeway when discarding input dimensions. At the moment, we have not investigated how this premise could affect other popular feature selection methods.

8.7 Conclusions and Future Work

In this paper, we propose a new methodology for removing inputs of neural networks, by creating an over/under-approximating network. We then show how such networks can be combined into a difference network that is capable of estimating the impact of various inputs, and we therefore combine it into a methodology by which we can identify insignificant inputs. All steps work on a trained neural network where one does not have access to the training data.

8.7.1 Future Work

The solution presented in this paper has been run on small networks with an unoptimized prototype implementation. We intend to improve the implementation and detect insignificant inputs in more complicated examples to investigate the scalability of the approach. Furthermore, we have seen great variations in the speed of the underlying solver `Marabou`, and it would be beneficial to gain more understanding when the verification becomes hard. Finally, it is also interesting to see if extra goals during training (e.g., minimizing weights from input nodes) can affect the usability of the approach by tweaking the final model to try and isolate significant inputs.

Acknowledgements

We acknowledge the support of the Swedish Knowledge Foundation via PerFlex - Performant and Flexible digital Systems through Verifiable Artificial Intelligence project, grant nr. 20220033, and ACICS – Assured Cloud Platforms for Industrial Cyber-Physical Systems, grant nr. 20190038.

Bibliography

[1] Ian T Jolliffe and Jorge Cadima. “Principal component analysis: a review and recent developments”. en. In: *Philos. Trans. A Math. Phys. Eng. Sci.* 374 (2065 Apr. 13, 2016), p. 20150202.

[2] Vipin Kumar. “Feature Selection: A literature Review”. In: *Thesmart Comput. Rev.* 4 (3 June 30, 2014), pp. 211–229.

[3] Yizhak Yisrael Elboher, Justin Gottschlich, and Guy Katz. “An abstraction-based framework for neural network verification”. In: *Computer Aided Verification*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 43–65.

[4] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*. Vol. 1. MIT press Cambridge, 2016.

[5] Francesco Leofante, Nina Narodytska, Luca Pulina, and Armando Tacchella. “Automated verification of neural networks: Advances, challenges and perspectives”. In: *arXiv [cs.AI]* (May 24, 2018).

[6] Changliu Liu, Tomer Arnon, Chris Lazarus, Christopher Strong, Clark Barrett, and Mykel J Kochenderfer. “Algorithms for verifying deep neural networks”. en. In: *Found. trends® optim.* 4 (3-4 Feb. 11, 2021), pp. 244–404.

[7] Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, David L Dill, Mykel J Kochenderfer, and Clark Barrett. “The marabou framework for verification and analysis of deep neural networks”. In: *Computer Aided Verification*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 443–452.

[8] Yizhak Yisrael Elboher, Elazar Cohen, and Guy Katz. “Neural Network Verification Using Residual Reasoning”. In: *Software Engineering and Formal Methods*. Ed. by

Bernd-Holger Schlingloff and Ming Chai. Cham: Springer International Publishing, 2022, pp. 173–189.

[9] Sumathi Gokulanathan, Alexander Feldsher, Adi Malca, Clark Barrett, and Guy Katz. “Simplifying Neural Networks Using Formal Verification”. In: *NASA Formal Methods*. Ed. by Ritchie Lee, Susmit Jha, Anastasia Mavridou, and Dimitra Giannakopoulou. Cham: Springer International Publishing, 2020, pp. 85–93.

[10] Ben Goldberger, Guy Katz, Yossi Adi, and Joseph Keshet. “Minimal modifications of deep neural networks using verification”. In: *LPAR*. LPAR23. LPAR-23: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning. Vol. 2020. EasyChair, 2020, 23rd.

[11] Fateh Boudardara, Abderraouf Boussif, Pierre-Jean Meyer, and Mohamed Ghazel. “A review of abstraction methods toward verifying neural networks”. en. In: *ACM Trans. Embed. Comput. Syst.* 23 (4 July 31, 2024), pp. 1–19.

Chapter 9

Paper B: A Conformal Prediction-Based Framework for CPU Load Forecasting: A Black-Box Approach

Edin Jelačić, Cristina Seceleanu, Peter Backeman, Ning Xiong, Tiberiu Seceleanu, Axel Jantsch

Published in: 49th IEEE International Conference on Computers, Software, and Applications (COMPSAC 2025)

Note: This paper has been reformatted to comply with the thesis layout. The content is unchanged from the published version.

Abstract

To address safety concerns in industrial systems, we propose a framework for forecasting CPU load with respect to a predetermined threshold, allowing customers to add tasks from a pre-defined library. Existing tools, akin to Windows Task Manager, provide limited insights due to their aggregate nature and high computational overhead. Our approach uses conformal prediction for rapid uncertainty-aware forecasts and Shapley value analysis to quantify individual task contributions to the CPU load. This proof-of-concept framework improves system safety assessment by addressing key research questions in load prediction and validation, paving the way for refined measurement methodologies in industrial applications.

9.1 Introduction

Industrial hardware systems power critical applications in fields like cybersecurity, telecommunications, power distribution and automatic control. These are devices such as programmable logic controllers (PLCs), embedded systems, and electronic control units (ECUs). Operating in real-time, these devices demand reliability and specialized expertise. Manufacturers typically deliver these systems pre-configured with essential tasks such as control algorithms, measurements, signal processing routines, and kernel-level processes. The systems are designed to run immediately upon delivery. However, when customers need to add or modify tasks to suit evolving operational conditions, the increasing complexity can overload the device's processing unit(s) and compromise real-time performance [1, 2].

To avoid such undesirable situations, we propose a user-friendly machine-learning-based framework that forecasts the processor load before deploying any new task configuration. Our approach is inspired by monitoring tools like the *Windows Task Manager* and UNIX's *top/htop*, tools that directly measure the proportion of CPU time used by tasks versus idle time [3, 4]. However, unlike these tools that provide online measurements of load, we utilize statistical methods to provide offline load forecasts and an understanding of individual task impacts. We employ *conformal prediction* [5] for reliable uncertainty quantification and *Shapley value analysis* [6] to explain each model prediction. Although industrial use-case forecasting has received attention in the literature [7, 8, 9], the mentioned combination of techniques has not been applied to CPU load forecasting before. Our framework provides detailed load information for individual cores in multi-core systems and quantifies the impact that each task has on overall computing load. Due to practical constraints on the procedure of extracting the data from the real device, such as excessive time and labor required, our experiments focus on a limited set of configurations. This limitation motivates our framework's design, which offers continuous, non-intrusive analysis without the need for deep architectural knowledge.

9.1.1 Objectives and Contributions

Our goal is to create a statistical framework for uncertainty-aware explainable black-box processor load prediction for an embedded system processing unit (or units, generally in the multi-core case). This research goal is rooted in an industrial problem important to a large partner company, with the target of processor load forecasting with a degree of confidence that quantifies

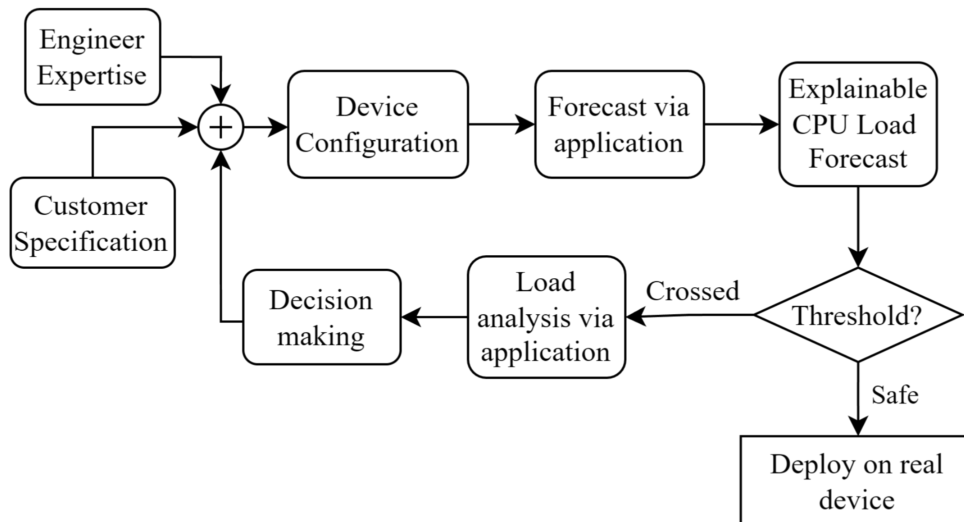


Figure 9.1: Workflow of the framework, from expertise and specification to deployment

the certainty of a device setup to not cause the total load crossing a certain load threshold. Based on this need, we extract prominent research topics that we tackle in the paper. Our integrated framework combines conformal prediction with Shapley values to provide statistically sound prediction intervals alongside clear, interpretable insights into feature contributions. Additionally, we have implemented a user-friendly GUI, which enhances the practical application of our framework by simplifying its deployment and use in operational settings, thus addressing the customer-side device programmability queries. The applicable workflow of the framework that we propose is presented in fig. 9.1.

9.2 Our Methodology

9.2.1 Data Extraction and Preprocessing

To test our framework, we utilize an experimentally attained processor load dataset from a unit manufactured and pre-programmed by our industrial partner. The device in question is a dual-core real-time measurement instrument with a task set extensible by proprietary software provided by the manufacturer. We loaded the instrument onto a test environment that mimicks how it operates in customer facilities. For our experiments, we selected a limited subset of 27 tasks from the full range of tasks running on the device, which is in the hundreds. There also exist many other tasks running on the instrument as part of its operating system and some of them interact with some of the ones we have selected, which may present unobserved confound-

ing features for our statistical approach. Formally, letting $\mathbb{B} = \{0, 1\}$, we represent our dataset as $\mathbb{B}^{27}$, where each component corresponds to one of the 27 tasks. Each vector $\vec{f}_{27\times 1} \in \mathbb{B}^{27}$ represents a unique configuration of tasks being active or inactive during testing. Note that, due to practical constraints, our experiments are conducted on this finite set of configurations, toggled via a script that sequentially activates and deactivates individual functions using the manufacturer's software, as well as performs logging and raw data pickup.

The device is equipped with a basic internal CPU load measurement system. The measurement system produces relatively little overhead on the device, around 1% as reported by the manufacturer, and does not impact the load measurement process. Its margin of error is $\pm 2\%$, with an updating frequency of 0.5 Hz, and it reports three components independently:

- Overall CPU (Dual Core) load,
- Processor 0 (Core 0) load,
- Processor 1 (Core 1) load.

An example of the measurement system's logs is as per single row:

```
0533394390 14-06:12:04.038558
279: CPU load: 45.48 %
CPU0: 36.01 %   CPU1: 54.96 %
free: 109.03 % of UP
```

This row states that the overall processor load, as measured by the device at 6 hours, 12 seconds and approximately 4 seconds, is 45.48%, the processor 0 load is 36.01%, processor 1 load is 54.96% The raw data was in need of several preprocessing steps. CPU load logs are filtered to remove configuration-switching spikes, reordered to reflect the actual configuration sequence, and aggregated — each configuration is sampled for about one minute at 0.5 Hz —to yield the metrics in table 9.1, with the resulting data stored in CSV format. Moreover, we must stress that, due to the way in which the experimental setup works, the datapoints we acquire are **independent** and **identically distributed** (IID), which is an otherwise difficult to make assumption on data, opening the door to later conformal treatment. In this work, we focus on the **mean** load of each configuration, i.e., for our dataset with N entries, for each $f_i, i \in [1, N]$, we focus on $\bar{L}_0(f_i)$, $\bar{L}_1(f_i)$ and $\bar{L}_{dual}(f_i)$ and their numerical relationships, as well as the relationships of their confidence intervals to the threshold load values.

Table 9.1: Data attained from instrument log processing

Processor	Avg. load $\bar{L}$	Std. deviation σ	Minimum load L_{min}	Maximum load L_{max}
0	$\bar{L}_0$	σ_0	$L_{0,min}$	$L_{0,max}$
1	$\bar{L}_1$	σ_1	$L_{1,min}$	$L_{1,max}$
Dual	$\bar{L}_{dual}$	σ_{dual}	$L_{dual,min}$	$L_{dual,max}$

9.2.2 Conformal Prediction Framework

Conformal prediction (CP) is a statistical method used in machine learning, to predict the probability of correctly classifying new observations [10, 11, 12]. This method is a statistically sound approach to creating rigorous uncertainty sets (in the case of classification tasks) or intervals (in the case of regression tasks, which we concern ourselves with in this work). These intervals are statistically guaranteed to contain the true value of the sample with a predetermined probability, which is generally chosen to be 90%. Furthermore, CP takes a black-box approach, that is, the intervals generated by this approach are valid without having any prior knowledge of the model trained to model the distribution of the underlying data, as long as the scoring function utilized by CP models appropriately how well the prediction of a sample "conforms" to the training data. This means that we can apply CP to many types of regression models successfully, gradient-boosted trees, random forests and neural networks alike, which gives us breadth in terms of adequate model selection and explainability, as well as provide quantification of uncertainty in situations such as ours, where one must be able to decide how "safe" a certain task configuration is with regards to CPU overloading. This gives the engineer a statistically sound insight into what can be put into production. This is even more significant since the procedure of testing is tedious, laborious and time-consuming, so it is to be avoided if possible. For instance, let us assume that we have 5 task configurations to predict. In fig. 9.2 we see a direct comparison between mere point forecasting and CP interval prediction. We see all the possibilities with respect to the relationship between the predicted load values and the threshold, which is 70 in this case. The *safe* CPU load zone is marked with green, the *unsafe* zone with red. In the point forecast case, configuration 1 appears clearly safe, whereas configuration 3 appears clearly unsafe. However, the situation is not so clear for configurations 2, 4 and 5. Without a prediction interval, it is impossible to include a quantification of uncertainty into

the decision process of whether to allow these configurations on the machine. However, once CP is applied, we can clearly estimate how “certainly" safe or unsafe a particular configuration is. We see that configuration 2 is deemed likely safe, whereas configuration 4, despite seeming unsafe at first, may likely produce a non-overloading load. On the other hand, configuration 5 is deceptively safe, given that CP produces bounds that are mostly in the *unsafe* zone. This is the way in which CP helps inform point forecasting, however the benefits go the other way as well. Point forecasts inform the user of the direction of the interval to put more emphasis on, when deciding whether to implement a task configuration on the device.

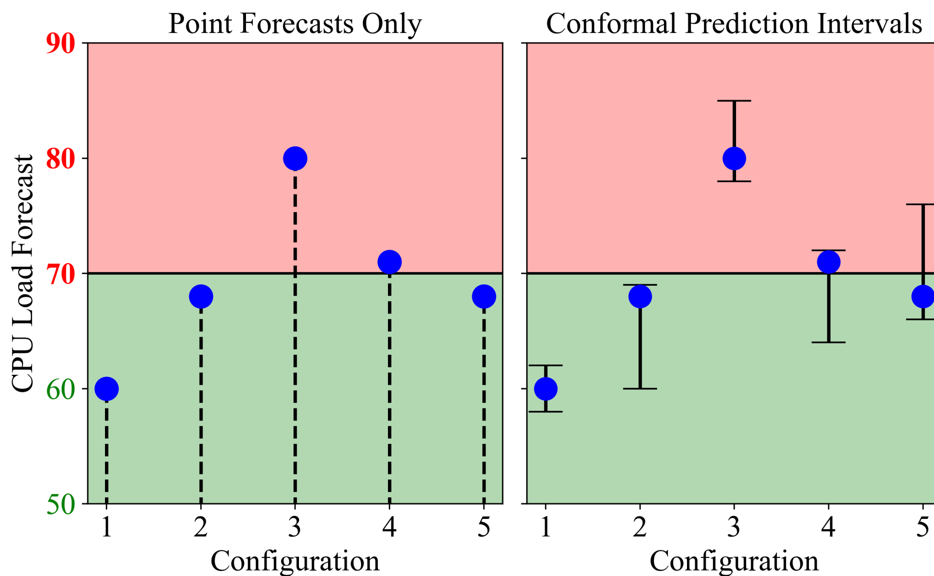


Figure 9.2: Point forecasting of CPU load versus intervals around predictions generated by conformal prediction

Normally the paradigm of regression learning requires the splitting of the dataset that models our underlying distribution, into a *training* dataset and a *test* dataset. However, to be able to create the CP intervals, the *training* dataset is once again split, and from the training data we reserve a moderately sized representative sample that we term the *calibration* dataset. Upper and lower quantile regressors (QRs) are trained on the training samples so that, by definition, there is a 5% probability that the true label values fall below the lower quantile regressor (0.05), and a 5% probability that they exceed the upper quantile regressor (0.95). QRs are used as a base, and are in subsequent steps conformalized to statistical certainty. Although in implementation this step differs slightly between models, the statistical guarantee remains identical between them. In general, for the miscoverage error, we denote and set $\alpha = 0.1$, so that the coverage rate is

$\mathbb{P} = 1 - \alpha = 0.9$.

We denote the features $X \in \mathbb{B}^{27}$, $X_i = \vec{f_i}$, $i = 1, ..., N$ and $y = \{y_1, y_2, \ldots, y_N\}$ of vectors in $\mathbb{R}^3$ such that $y_i = (\bar{L}_{0,i}, \bar{L}_{1,i}, \bar{L}_{dual,i})$ is the vector of mean loads for the i-th observation. We take N_{cal} samples from the training dataset for calibration. Let $X_{train} \subset \mathbb{B}^{27}$ and $y_{train} \subset y$ have length $N_{train} = |X_{train}|$, and $X_{test} = X \setminus X_{train}$ and $y_{test} = y \setminus y_{train}$ have length $N_{test} = N - N_{train} - N_{cal}$.

Finally, let $X_{cal} = \mathbb{B}^{27} \setminus (X_{train} \cup X_{test})$ and $y_{cal} = y \setminus (y_{train} \cup y_{test})$ have length N_{cal}. Our conformal prediction framework proceeds as follows:

Step 1: Train the Base Point Forecast Model M We begin by training a neural network model $\mathbf{M}$ on the training data (X_{train}, y_{train}), so that for each observation X_i the model outputs the prediction $\mathbf{M}(X_i) = \hat{y}_i \approx y_i$. Training is performed over multiple epochs using mean squared error (MSE) as the loss function.

Step 2: Fit Upper and Lower Quantile Regressors

For each output component (i.e. for $\bar{L}_0, \bar{L}_1, \bar{L}_{dual}$), we train two quantile regressors on the calibration set:

- A **lower QR** to estimate the $\alpha/2$ quantile of the residuals,
- An **upper QR** to estimate the $1 - \alpha/2$ quantile.

We denote these regressors QR_{lower} and QR_{upper}. They are fitted using the training features (X_{train}, y_{train}) and the corresponding residuals for each output on the asymmetric pinball loss (or quantile loss) metric, whose specific graph can be seen in fig. 9.3. Although they are normally usable for the purposes of estimating confidence intervals, in their present state they do not provide coverage guarantees and hence we will conformalize them.

Step 3: Compute Residuals Following the guidelines for split conformal prediction outlined in [10], our conformal score $s(x, y)$ is based on the difference between the predicted label value and the nearest quantile value of the calibration samples, as per equation:

$$s(x, y) = \max\left(QR_{lower}(x) - y, y - QR_{upper}(x)\right),$$

with $x \in X_{cal}, y \in y_{cal}$. Once the conformal scores are calculated, the corresponding quantiles for the core 0, core 1 and dual core predictions are calculated as:

$\hat{q} = \text{Quantile}\left(s_1, s_2, ...s_n; \frac{\lceil (n+1)(1-\alpha) \rceil}{n}\right)$[1].

[1] In [10] the authors suggest adding an empirical correction term $1/(n+1)$ to α (n being the size of the

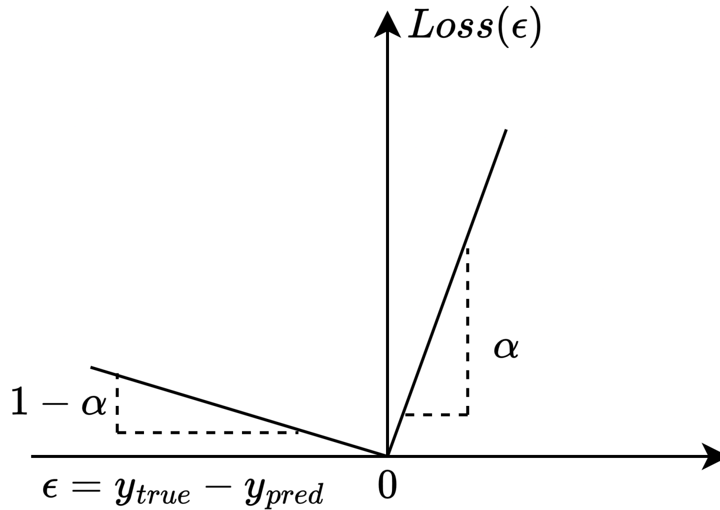


Figure 9.3: Pinball loss (Quantile loss) function

Step 4: Construct Prediction Intervals

For a new input x, the model produces a point forecast $\hat{y} = \mathbf{M}(x)$. From the quantile regressors we then get the non-conformalized intervals $[QR_{lower}, QR_{upper}]$, which we then conformalize by adding the corresponding quantiles $\hat{q}$ to form valid prediction intervals according to:

$$\text{Lower bound} = QR_{lower}(x) - \hat{q}, \tag{9.1}$$

$$\text{Upper bound} = QR_{upper}(x) + \hat{q}. \tag{9.2}$$

In other words, we expand the quantile regression to achieve the desired coverage.

These intervals, by design, satisfy the CP guarantees, ensuring that the true output is contained within the interval $[\text{Lower bound}, \text{Upper bound}]$ with probability at least $1 - \alpha$.

For example, assume that our CP model has been fully trained and calibrated and that we are presented with 100 fresh samples to predict. The samples are binary vectors $\vec{f} \in \mathbb{B}^{27}$. The model first generates a point forecast of CPU load for each sample using the underlying distribution approximator (i.e. neural network). Concurrently, the pre-trained quantile regressors compute the lower and upper residual adjustments which are added to the point forecasts to form statistically valid prediction intervals. If our miscoverage rate α was set to 0.9, our predic-

calibration dataset) to correct for the fact that the CP procedure entails a finite sample. Namely, since the quantiles of the score function (in our case the residuals) are estimated from a finite sample, the empirical quantile selection is discrete rather than, ideally, continuous. To properly adjust for this discreteness, instead of using the theoretical $(1 - \alpha)$ quantile directly, we use $\lceil (n + 1)(1 - \alpha) \rceil / n$.

tion intervals are constructed to achieve a 90% coverage rate. Statistically, this means that out of our 100 fresh samples, approximately 90 will have their true CPU load values fall within the predicted intervals.

9.2.3 Shapley Value Integration

Given the systematic nature of our framework in acquiring the dataset—through experimental control and clear separation of individual inputs—a natural question arises: can we measure individual function contributions solely based on the system's inputs and outputs, without delving into the model's internal details? For notational simplicity, let us denote the $\mathbf{M}$ model's output as a 3×3 matrix, where the each column is vector of $\mathrm{Upper\ bound}$, prediction and $\mathrm{Lower\ bound}$ values for core 0, core 1 and dual core as

$$\begin{aligned}\mathbf{M}(\vec{f}) &= \left[\bar{L}_{0\downarrow}^{\uparrow}, \bar{L}_{1\downarrow}^{\uparrow}, \bar{L}_{dual\downarrow}^{\uparrow}\right] \\ &= \begin{bmatrix} \bar{L}_{0}{}^{\uparrow} & \bar{L}_{1}{}^{\uparrow} & \bar{L}_{dual}{}^{\uparrow} \\ \hat{\bar{L}}_{0} & \hat{\bar{L}}_{1} & \hat{\bar{L}}_{dual} \\ \bar{L}_{0\downarrow} & \bar{L}_{1\downarrow} & \bar{L}_{dual\downarrow} \end{bmatrix}.\end{aligned} \tag{9.3}$$

To attain estimates for individual task contribution to the load intervals and predicted load values, we integrate Shapley values into our framework in the form of the SHAP library [13]. Our model diagram then becomes as per fig. 9.4.

Shapley values are an approach to explaining outputs of machine learning models rooted in cooperative game theory. Each of the model inputs, that is, each one of the tasks (with binary value 0 or 1) gets its own Shapley value that it relates to each of the output components from the matrix in eq. (9.3). Therefore, the total output of our framework amounts to

$$\overbrace{27}^{\text{\# of inputs}} \times \underbrace{3}_{\text{core 0, 1, dual}} \times \overbrace{3}^{\text{upper, lower, prediction}} = 243$$

individual values. The way Shapley values work is by providing a principled way to quantify the contribution of each parameter by averaging its marginal impact over all possible feature subsets. In our context, this means that for each prediction, we compute the Shapley value ϕ_i for every feature i, which reflects how much the presence of that feature alters the forecasted CPU load and the interval.

For a set of features Φ and a prediction function γ, the Shapley value for feature $i \in \Phi$ is calculated as the weighted average of the increase of the output value when i is active with the

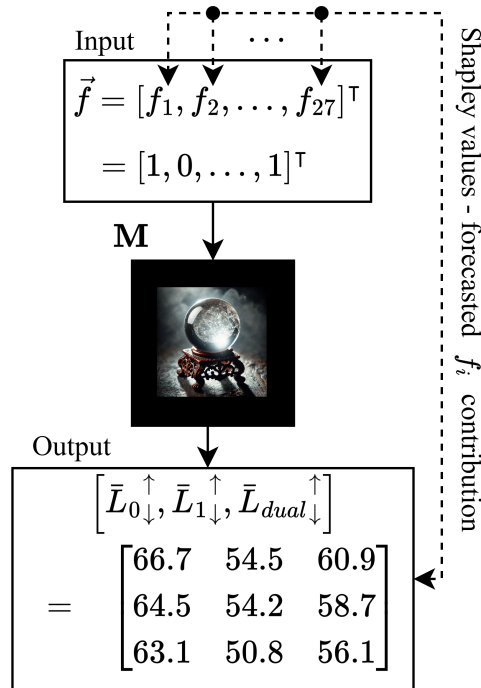


Figure 9.4: Framework outline

group of features $S \subseteq \Phi$ versus when i is inactive with group S. This averaging is done over all possible subsets S where i is not in, and formally the definition is:

$$\overbrace{\phi_i}^{\text{i's Shapley}} = \sum_{S \subseteq \Phi \setminus \{i\}} \underbrace{\frac{|S|!(|\Phi| - |S| - 1)!}{|\Phi|!}}_{\text{S's weight}} \overbrace{\Big[\gamma(S \cup \{i\}) - \gamma(S)\Big]}^{\text{i's marginal contribution}}, \tag{9.4}$$

where $\gamma(S)$ is the model output when only the features in subset S are active [14]. Although calculating this sum exactly is computationally expensive (in fact, calculating Shapley values in general is an NP-hard problem [15]), efficient approximations such as KernelSHAP or TreeSHAP from the SHAP library allow us to estimate these values in practice.

In our framework, the conformal prediction method guarantees that the overall prediction interval has a positive length. By design, the upper bound is always higher than the lower bound. However, when we decompose the interval using Shapley values, we calculate the marginal contribution of each feature to the lower and upper bounds independently. This means that a given feature may have a stronger impact on one bound than the other. For instance, if for a specific feature f_1 we obtain

$$\phi_{1\downarrow}(\vec{f}) = 5.0 \ \& \ \phi_1{}^{\uparrow}(\vec{f}) = 4.0,$$

it indicates that f_1 contributes more to the lower bound than to the upper bound. In practical

terms, f_1 is pushing the lower bound upward, i.e., reducing the safety margin more significantly than it is elevating the upper bound. This distinction is important because it offers deeper insight into the model's behavior: even though the overall interval remains valid and of positive length, the individual contributions reveal that certain features may disproportionately influence the conservative (lower) aspect of the interval. Such insights are valuable in safety-critical applications, where understanding the source of prediction conservatism can guide more informed decisions.

By applying these methods, we obtain local explanations that detail the influence of each parameter on the CPU load forecast. This not only aids in understanding model behavior but also highlights which features are most critical to system performance, thereby supporting more informed decision-making in industrial applications.

9.2.4 System Integration & GUI Application

For the integration of this system our target was a portable and platform-independent, easy-to-use mechanism with which experienced device engineers can estimate CPU loads and function contributions at the click of a few buttons with minimal time spent learning a new methodology of interfacing with the underlying algorithms. The application is composed in the Electron front-end framework and runs a Python Flask server on the back-end. Communication between the two is achieved via HTTP requests. The application stack is shown in fig. 9.5.

The entire application stack is in this stage kept to a minimum, as the core of the development is focused on the underlying algorithms. This also keeps development overhead low and strikes a moderate balance between modularity (extensibility) and code maintainability. Upon opening the application, the user is welcomed with the screen shown in fig. 9.6. The user can choose whether to train a new model with customized hyperparameters or load an existing model from the application storage. If the user chooses to train a new model, they can choose from the implemented model types. The core idea of the GUI for our framework is to interactively provide actionable insights into the device under development, while abstracting away low-level ML-related design decisions (and still providing the optional functionality to perform them).

The selection of the model opens the modal window wherein the user specifies hyperparameters. An example of such a modal for the neural network model is in fig. 9.7. The model training is tracked throughout the console through information and warning lines presented to

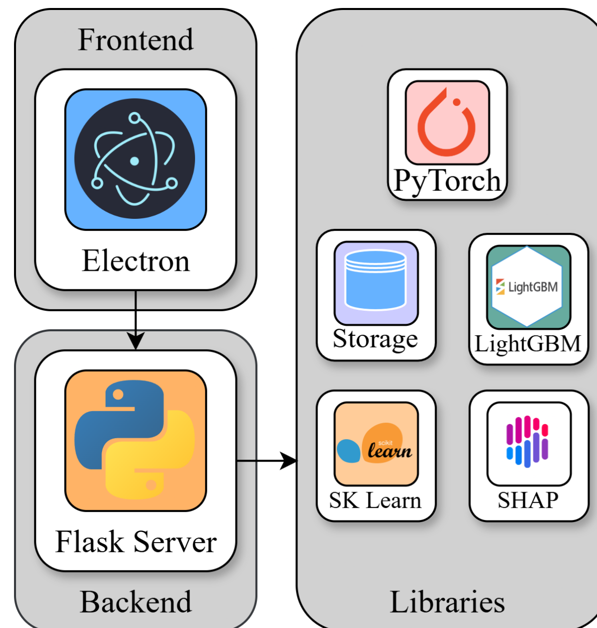


Figure 9.5: Utilized application stack

the user. Once the model is trained, it is stored to the application database, along with the metadata containing information about the machine used for training, the duration of the training, the hyperparameters used and any model comments left by the user. If the user chooses to load a model, the model selection is sent to the backend for server-side model verification, and if it is successful, the user is presented with console output showing the details of the loaded model's structure.

The user can switch the available tasks on/off in the right-hand side of the screen and execute predictions. Each prediction, along with its metadata, is stored in the application storage. The user can choose to view the graphs for the predictions, whereupon he is presented with three screens. An example of a screen for Core 0 with 5 measurements may be seen at fig. 9.8, with analogues for Core 1 and Dual Core. Each screen represents predictions for Core 0, Core 1 and Dual Core respectively with mean load intervals and predictions, as well as individual task Shapley values for the lower contribution, prediction contribution and upper contribution.

Shapley value charts allow users to understand which tasks have significant impact on the predicted loads. For example:

- A task with a **high positive contribution** to the upper bound indicates that its presence increases the estimated maximum load, such as *FUNC_12* in the first prediction in fig. 9.8,
- Conversely, a task with a **low or negative contribution** suggests it has less influence or even

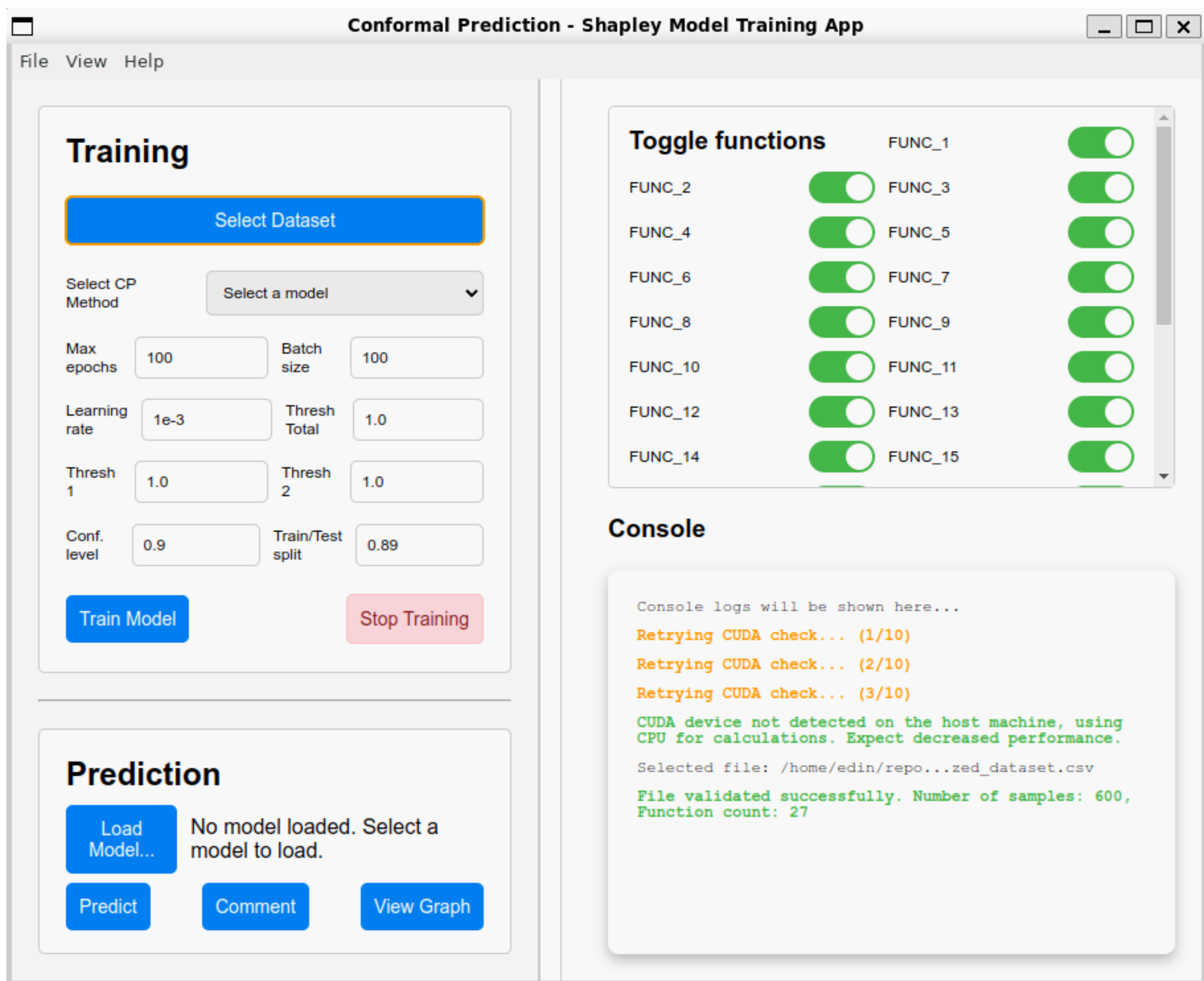


Figure 9.6: Initial GUI screen - The left hand side is divided into Training and Prediction sections. The Training section uses user input for various general model training parameters, the Prediction section controls the loading of a trained model, annotations and graph window. The right hand side is divided into the Toggle section where tasks are toggled On/Off for prediction and the Console section where various information, warnings and errors are shown to the user.

reduces the predicted load, such as *FUNC_20* in fig. 9.8.

By analyzing these contributions across all 27 features, we can identify which tasks are critical in shaping the model's predictions. This information is particularly valuable for understanding why certain prediction intervals are wider or narrower and how specific tasks contribute to the overall safety margins (lower bounds) or performance expectations (upper bounds).

The matrix in eq. (9.3) provides a structured view of these contributions, where each task's Shapley value reflects its marginal impact on the predicted values. For instance, if a task has unequal contributions to the lower and upper bounds, it highlights an asymmetric effect indicating that the task influences one aspect of the prediction more strongly than the other.

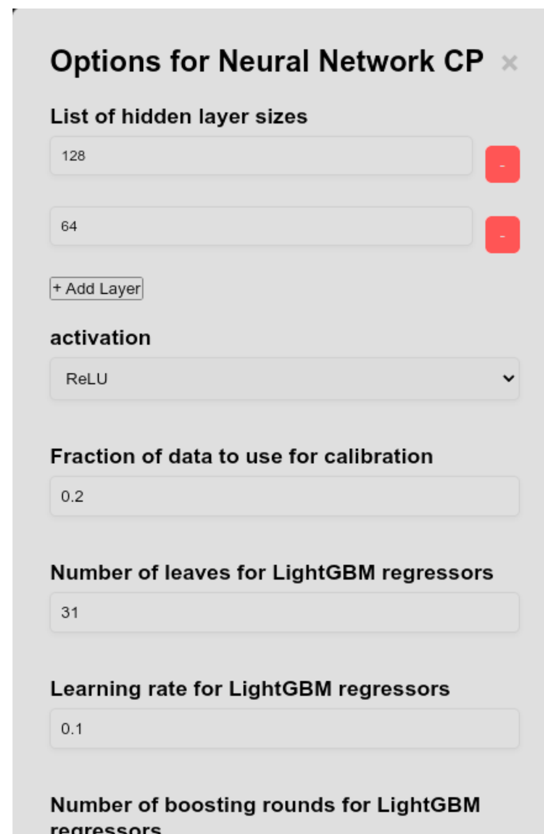


Figure 9.7: Modal that opens inside the GUI once a model type is selected. Offers model-specific hyperparameters for input. Default values are always set to what has been (through our experiments) deemed satisfactory, but setting these values has been left to the user.

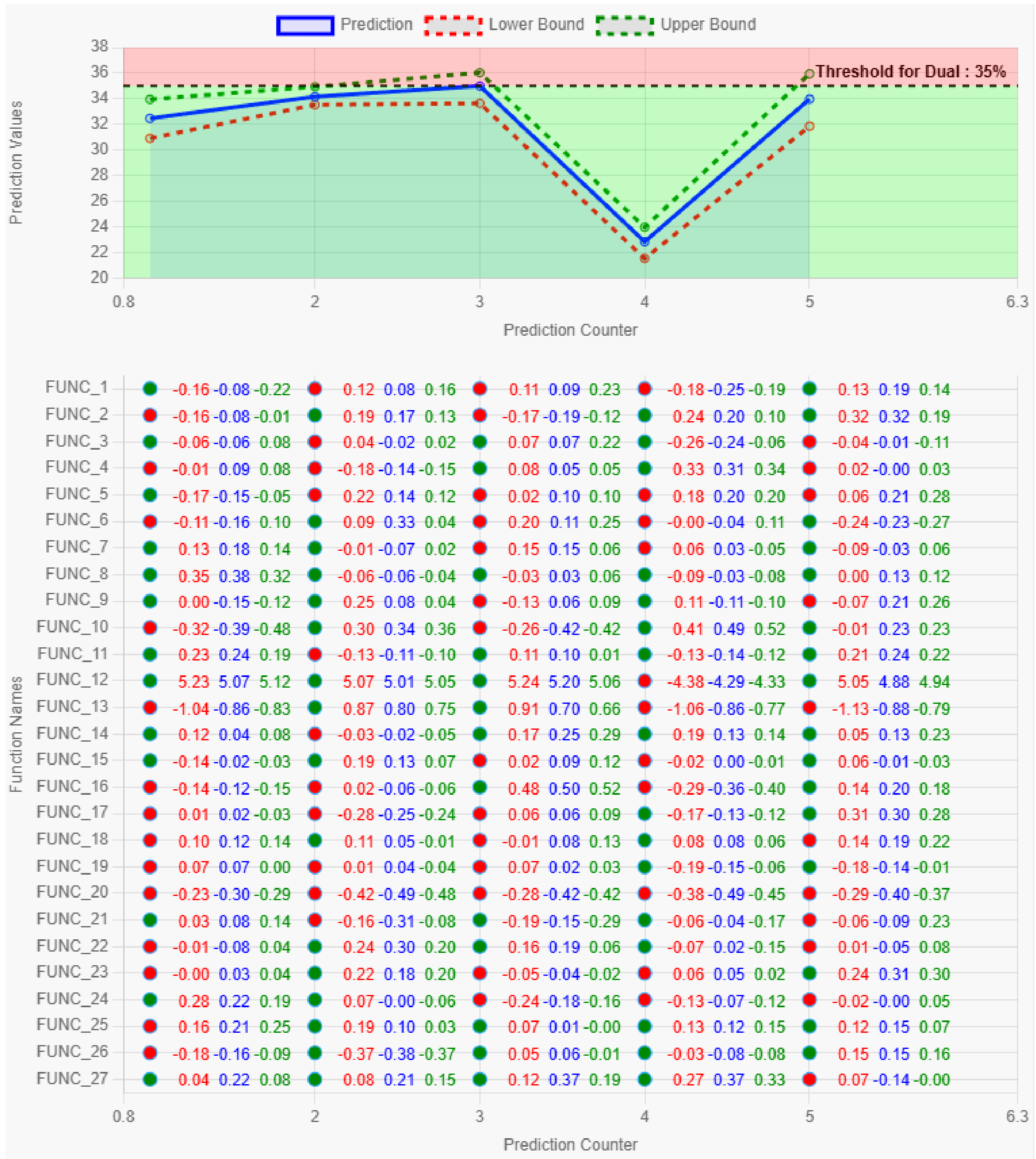


Figure 9.8: Graph window for the predicted CPU loads for Core 0. Displayed with 5 predictions and a set threshold of 25%. Active tasks are colored green, while inactive ones are colored red. Individual task contributions are shown as a triplet: contribution to lower bound, contribution to predicted value and contribution to the upper bound.

Ultimately, this integration enables users to interpret not only what the model predicts but also why specific tasks are driving those predictions. This transparency is crucial for making informed decisions in safety-critical applications where understanding the root causes of prediction intervals is essential. For example, assume that we must implement functionality achievable by either loading task A or task B onto the device and the set is already predicted to be operating close to the CPU threshold. Both tasks may take the predicted load over the threshold, perhaps similarly so, but if task A has a lower contribution to the upper bound of the prediction than task B, we can quantify via Shapley values the inherent risk of choosing to implement the functionality with task B over task A w.r.t. the threshold. In this case, despite similar predicted load, task B is deemed more dangerous than task A, and we would proceed with implementation using task A.

9.3 Experimental Setup

9.3.1 Dataset Description

In this study, we utilize a dataset collected from the aforementioned industrial hardware system. The dataset consists of 27 binary input features representing task configurations, as outlined previously in the paper, along with corresponding CPU load measurements. The dataset contains 7129 samples, wherein 25% or 1784 samples are left out for testing the approach, 37.5% or 2673 utilized as the calibration dataset for the split conformal prediction algorithm and the remaining 37.5% (2673) for the predictor training. For the corresponding task configurations we record the NN's outputs: point predictions, lower bounds, and upper bounds. These measurements allow us to assess both the performance of our predictive framework and the interpretability of feature contributions.

9.3.2 Experimental Design

Our evaluation framework is two-fold:

- **Coverage, Interval length and Prediction Error:** We assess the predictive performance of our conformal prediction model using standard metrics:
  - **Coverage Rate:** The percentage of true CPU load values that fall within the predicted intervals, ideally meeting the specified confidence level (e.g., 90%).

  - **Interval Length:** The size of the intervals themselves, as the smaller they are while retaining coverage guarantees, the higher the model's certainty about the outcome.
  - **Prediction Error:** Metrics such as Mean Squared Error (MSE) are used to quantify the accuracy of the point predictions.

- **Shapley Value Evaluation:** Evaluating the quality of Shapley explanations is challenging. In our experiments, we use the following strategies to validate and interpret the Shapley values computed for each function:

  - **Additivity Check:** For each prediction, we verify that the sum of the baseline prediction and the computed Shapley values closely approximates the model's output. This confirms the internal consistency of the Shapley decomposition.
  - **Sensitivity Analysis:** Features with high Shapley values should cause larger changes when perturbed, thereby supporting the validity of the feature attributions. This is particularly significant given that we realized, both from our experimental testing and discussions with the partner experts, tasks *FUNC_12* and *FUNC_13* are some of the most load impacting on the device. The observation of this peculiarity must be maintained throughout testing.
  - **Ablation Study:** By selectively masking tasks one by one (removing their variance from the dataset) and analyzing the impact on the model's prediction, we compare the observed prediction shifts to the corresponding Shapley values. If our Shapley value analysis is correct, these value shifts should not be so significant as to impact the contributions of non-masked tasks.

9.4 Results

We first establish a baseline using the unmodified testing dataset. Our framework predicts lower, median, and upper CPU loads for Core 0, Core 1, and Dual Core configurations, and computes Shapley values for each of the 27 tasks for each setup. Next, we perform 27 separate ablations by masking each task in the training, calibration, and testing data one at a time. These ablations simulate scenarios in which some functions present on the device are excluded from the analysis. If our framework continues to operate robustly under these ablations, it implies resilience for the real system when additional functions are implemented. Accordingly, all performance metrics are compared between the baseline and the ablated models.

During discussions with industry experts, it was suggested that *FUNC_12* should serve as a benchmark, as it is anticipated to be the most impactful task. Capturing its influence accurately is crucial for validating our approach, and conversely, if capturing its influence on the system failed to materialize, it would serve as a signal that the framework is flawed.

9.4.1 Coverage, Interval Length, and Prediction Error

Table 9.2 summarizes the key metrics for both the baseline and ablated models. Overall, the coverage rates remain close to the nominal level of 90% $(1 - \alpha)$ across Core 0, Core 1, and Dual Core measurements, confirming that our conformal prediction (CP) framework is valid for device utilization. Notably, ablating *FUNC_12* does not substantially affect coverage, though it does lead to a significant increase in interval length. This suggests that *FUNC_12* is critical for constraining predictive uncertainty—a result that aligns with our Shapley analysis, which identifies it as the most impactful function.

The relatively small mean interval lengths (except in the case of ablated *FUNC_12* and, to a lesser degree, ablated *FUNC_13*) and consistent MSE values indicate that removing any single function causes only minor deviations compared to the full baseline. This robustness is promising, particularly for the overall device prediction system, as it implies that even if functions not encompassed in our dataset were omitted, the system would not break down dramatically. The device measurement system, therefore, demonstrates resilience to individual task losses, ensuring reliable performance and actionable insights for end users.

In summary, while some functions (notably *FUNC_12*) have a more pronounced effect on model performance, the overall results confirm that the model is robust to the removal of single tasks, with only modest changes in predictive accuracy and uncertainty estimates under ablation.

9.4.2 Forecasting Performance

An example of our mechanism's forecasting performance is shown in fig. 9.9 for Core 0, with similar behavior observed for Core 1 and Dual Core. In this figure, the value of CP intervals is evident. The intervals offer a more informative prediction range than mere point estimates. For example, at prediction indices 5, 6, and 9, the point forecast falls outside the CP interval, yet the interval itself successfully captures the true value. Comprehensive results, including coverage

Table 9.2: Coverage, Mean Interval Length and Prediction Error for the model M, C0 - Core 0, C1 - Core 1, Dual - Dual Core

	Coverages			**Mean Interval Lengths**			**MSE**		
Ablated Function	**C0**	**C1**	**Dual**	**C0**	**C1**	**Dual**	**C0**	**C1**	**Dual**
1	0.89	0.89	0.89	1.32	7.94	4.66	0.05	0.47	0.17
2	0.89	0.89	0.89	1.32	7.93	4.66	0.04	0.45	0.16
3	0.89	0.89	0.89	1.27	7.74	4.58	0.04	0.46	0.16
4	0.89	0.89	0.89	1.32	7.92	4.65	0.04	0.45	0.16
5	0.89	0.89	0.90	1.32	7.93	4.66	0.06	0.48	0.17
6	0.89	0.88	0.89	1.32	7.93	4.66	0.05	0.44	0.16
7	0.89	0.89	0.89	1.27	7.94	4.67	0.06	0.47	0.17
8	0.89	0.89	0.89	1.32	7.93	4.65	0.05	0.46	0.17
9	0.89	0.88	0.89	1.32	7.93	4.67	0.04	0.47	0.17
10	0.89	0.89	0.89	1.33	7.98	4.69	0.05	0.53	0.19
11	0.89	0.89	0.89	1.32	7.92	4.65	0.04	0.47	0.17
12	0.89	0.89	0.90	2.60	17.94	10.26	2.44	70.85	23.76
13	0.88	0.89	0.89	1.48	8.07	4.75	0.26	2.66	1.06
14	0.89	0.89	0.89	1.34	7.93	4.64	0.05	0.45	0.16
15	0.89	0.89	0.90	1.33	7.93	4.66	0.04	0.47	0.16
16	0.88	0.89	0.88	1.32	7.91	4.64	0.06	0.48	0.17
17	0.89	0.88	0.89	1.32	7.90	4.63	0.05	0.49	0.17
18	0.89	0.89	0.89	1.32	7.92	4.66	0.04	0.46	0.16
19	0.89	0.89	0.89	1.32	7.92	4.66	0.04	0.45	0.16
20	0.89	0.88	0.89	1.33	8.00	4.73	0.08	0.75	0.28
21	0.89	0.88	0.89	1.32	7.93	4.66	0.06	0.47	0.16
22	0.89	0.89	0.89	1.33	7.93	4.67	0.06	0.46	0.16
23	0.89	0.88	0.89	1.34	7.93	4.66	0.07	0.48	0.18
24	0.89	0.89	0.89	1.33	7.93	4.66	0.05	0.47	0.16
25	0.89	0.89	0.89	1.35	7.99	4.71	0.06	0.47	0.16
26	0.88	0.88	0.89	1.35	8.01	4.73	0.06	0.46	0.16
27	0.89	0.88	0.89	1.32	7.93	4.67	0.04	0.46	0.16
None	0.89	0.89	0.89	1.32	7.92	4.66	0.06	0.46	0.16

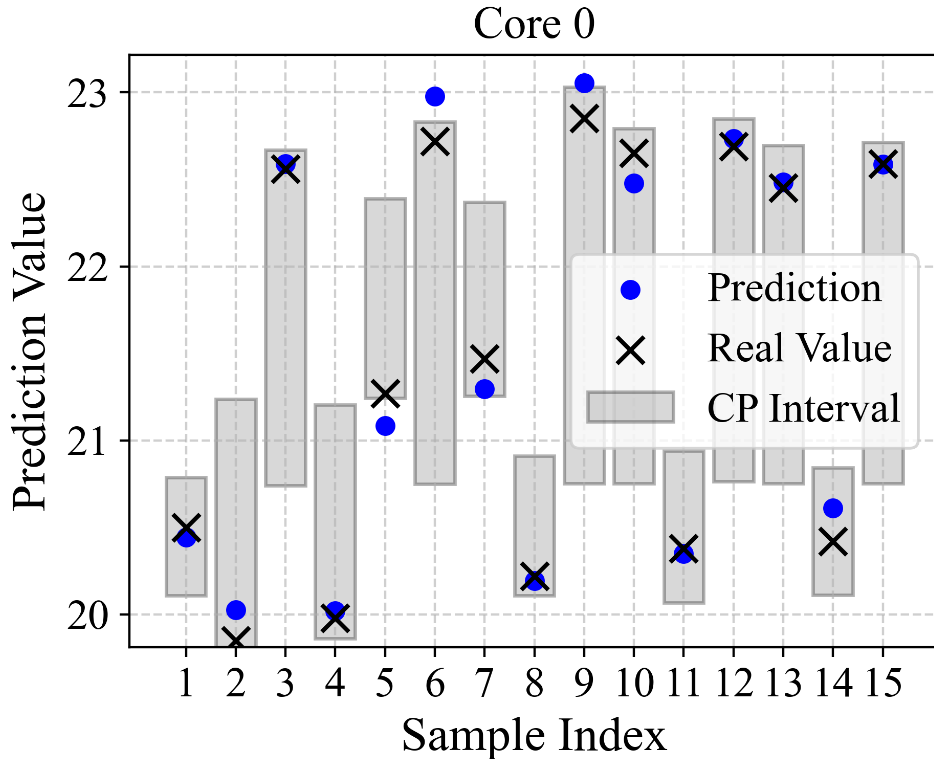


Figure 9.9: Outputs of M, i.e. the point forecasts and intervals for Core 0, with the real value shown for demonstration

metrics, mean interval lengths, and prediction errors over the test dataset and its ablations are summarized in table 9.2.

9.4.3 Additivity Check

For our model, we generate a total of 9 SHAP library explainers, a total which results from combining the 3 Lower, Prediction and Upper outputs with the 3 Core 0, Core 1 and Dual Core setups. For each of these Shapley explainers, it must hold that the model's prediction of the given target value is approximately equal to the sum of the base value (implicit in the Shapley decomposition) plus the individual Shapley contributions towards the target value. The effective base value is computed as

$$\mathrm{Base}_{est} = \mathbf{M}(x) - \sum_{i=1}^{27} \phi_i(x)$$

and if the additivity property holds, the base estimate calculated like this should be nearly constant across individual samples. In other words, the difference between the mean $\overline{\mathrm{Base}_{est}}$ and the actual Base_{est} calculated from individual samples should be 0. This is tested in table 9.3.

We see that most ablations yield zero or near-zero mean additivity errors, indicating that the Shapley decomposition remains consistent across tasks. However, ablating *FUNC_12* or *FUNC_13* increases errors, especially for the upper bounds. Ablating *FUNC_12* or *FUNC_13* forces the model to redistribute their relatively large contributions among other tasks, which disrupts the otherwise tight match between the summed Shapley values and the model's output. As a result, the additivity error rises when either of those tasks is removed, which implies their outsized role in the model's learned representation.

9.4.4 Sensitivity Analysis over Ablations

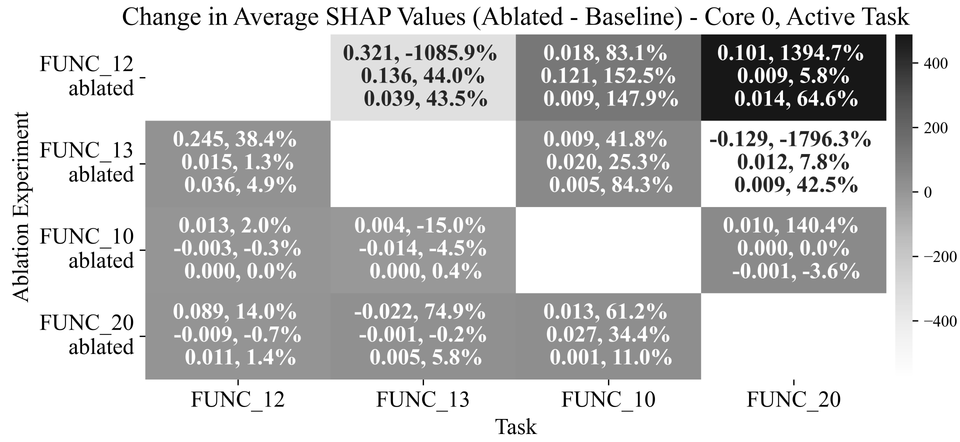
Upon reviewing the frameworks outputs over the baseline and the ablations, we may conclude that *FUNC_12*, *FUNC_13*, *FUNC_20* and *FUNC_10* are by far the most impactful for the CPU loads in the machine, which may be seen in fig. 9.10. We will focus primarily on these four tasks in the Shapley value analysis, more specifically, on these tasks' contributions when active. The results may be observed in figs. 9.11a to 9.11c.

Figure 9.10: Shapley value summary for predictions on Core 0, un-ablated baseline test dataset, 1784 testing samples.

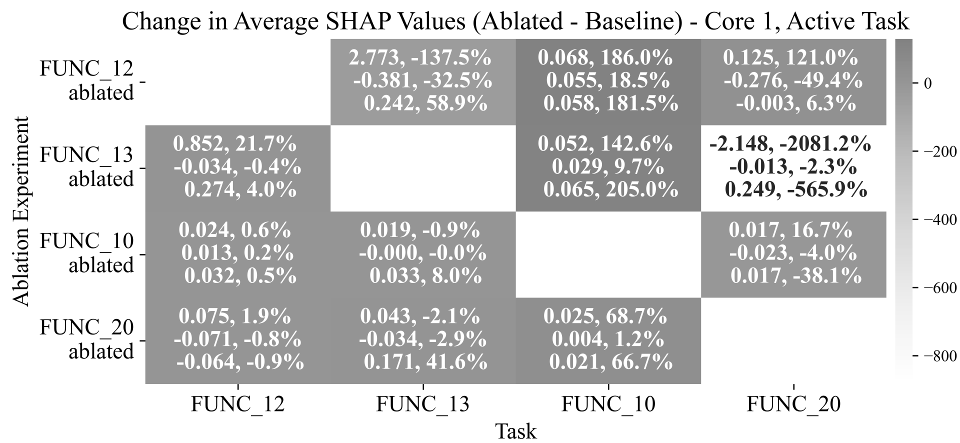
Observing the percentage changes as compared to baseline gives us insights into the effect

Table 9.3: Mean Absolute Additivity Error table for Shapley values across ablations and baseline $\overline{|\overline{\text{Base}_{est}} - \text{Base}_{est}|}$

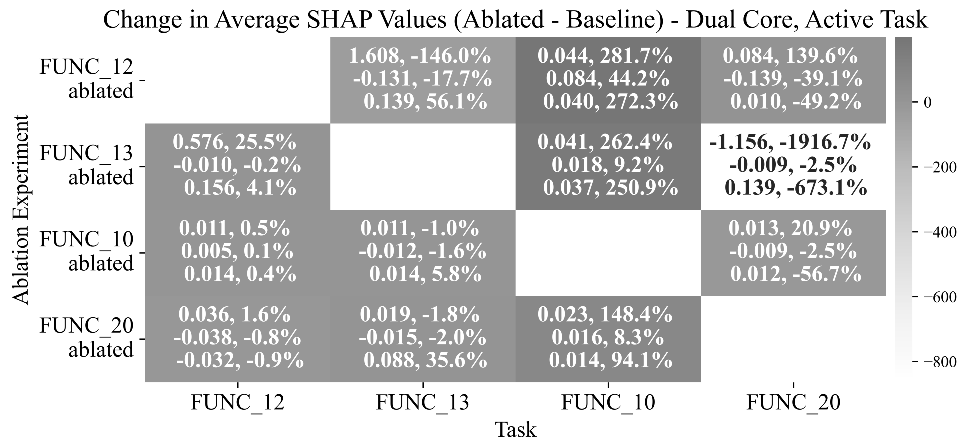
	Core 0			**Core 1**			**Dual Core**		
Ablated Function	**Lower**	**Upper**	**Pred.**	**Lower**	**Upper**	**Pred.**	**Lower**	**Upper**	**Pred.**
None	0	0	0.17	0	0	0.17	0	0	0.14
1	0	0	0.12	0	0	0.16	0	0	0.13
2	0	0	0.16	0	0	0.16	0	0	0.13
3	0	0	0.12	0	0	0.15	0	0	0.13
4	0	0	0.19	0	0	0.19	0	0	0.14
5	0	0	0.15	0	0	0.19	0	0	0.14
6	0	0	0.13	0	0	0.18	0	0	0.14
7	0	0	0.19	0	0	0.19	0	0	0.14
8	0	0	0.13	0	0	0.18	0	0	0.14
9	0	0	0.16	0	0	0.18	0	0	0.14
10	0	0	0.19	0	0	0.22	0	0	0.17
11	0	0	0.16	0	0	0.2	0	0	0.16
12	0	0	0.62	0	0	1.1	0	0	0.86
13	0	0	0.24	0	0	0.33	0	0	0.27
14	0	0	0.13	0	0	0.16	0	0	0.14
15	0	0	0.18	0	0	0.17	0	0	0.14
16	0	0	0.12	0	0	0.18	0	0	0.13
17	0	0	0.18	0	0	0.19	0	0	0.16
18	0	0	0.15	0	0	0.17	0	0	0.13
19	0	0	0.18	0	0	0.18	0	0	0.14
20	0	0	0.18	0	0	0.21	0	0	0.17
21	0	0	0.16	0	0	0.16	0	0	0.13
22	0	0	0.16	0	0	0.2	0	0	0.15
23	0	0	0.12	0	0	0.16	0	0	0.13
24	0	0	0.15	0	0	0.14	0	0	0.11
25	0	0	0.13	0	0	0.16	0	0	0.12
26	0	0	0.14	0	0	0.2	0	0	0.15
27	0	0	0.16	0	0	0.18	0	0	0.13



(a) Core 0 measurements.



(b) Core 1 measurements.



(c) Dual Core measurements.

Figure 9.11: Change in Shapley values of most significant tasks in ablated vs. baseline cases for the task's contribution when active. Values on the bottom represent the lower bound contribution, values in the middle are the prediction contribution, and values on top are the upper bound contribution.

of ablating individual measurement features. Since the baseline contribution for *FUNC_20* is relatively small, when *FUNC_12* is ablated, even a modest absolute increase in *FUNC_20*'s contribution (0.101) translates into an outsized percentage change. Additionally, if *FUNC_12* and *FUNC_20* are correlated or interact in the model, ablating *FUNC_12* might force the model to compensate by relying much more on *FUNC_20*, leading to a dramatic increase in its attributed importance. This reallocation can cause the computed percentage change to spike, such as the 1394% observed for the upper bound contribution in fig. 9.11a, even though the absolute change is only 0.101. We see a similar pattern for all tasks except *FUNC_12*, such as the -1085% change in *FUNC_13*'s Upper bound contribution to Core 0 when *FUNC_12* is ablated, as well as *FUNC_20*'s upper bound contribution for the Dual Core and Core 1 cases at almost 20 times the baseline contribution.

9.5 Discussion

From the perspective of modeling the CPU usage in an embedded device with respect to the tasks loaded onto the device, we can state that our framework and application present a usable alternative to manual testing and "rule-of-thumb" estimates of CPU load. The synergic composition of point forecasting along with conformal prediction gives us uncertainty-bounded load estimates usable in the real world as guidelines for device deployment. Furthermore, provided that we continuously utilize the device data storing and model refinement, it is an approach with potential to provide more insights with more data and more device parameters forwarded to it. Presently the application supports training a new model based on refreshed data, but we intend to implement support for continuous model refinement. Our industrial partners have found the application highly useful, providing invaluable feedback for further development. They intend to utilize a specialized version running on their premises. Based on table 9.2, we can see that we approach 90% coverage on Core 0, Core 1 and Dual Core measurements. The mean interval length for Core 0 is within the measurement system's margin of error, whereas Core 1 and Dual Core measurements, though informative, leave some place for improvement. For example, while we do rely on the essential algorithm of split conformal regression on I.I.D. data for our estimates, there exist more informed adaptive approaches (such as [16, 17]) that could tighten these intervals further while retaining coverage. For the point forecast predictor, we can state that it is highly accurate based on the minimal MSE scores recorded on the test dataset.

Though the contribution shifts in ablated datasets in absolute terms are within the measurement system's margin of error, they cannot be ignored in the sense of formulating a robust and pervasive CPU load measurement framework. They underline the significance of getting good data from the devices that the application users are trying to model, in terms of capturing as many tasks running on the device as possible. In line with the general notion of machine learning-based approaches becoming more useful the richer the underlying datasets become, our approach would greatly benefit from deeper analysis of the device.

9.6 Related Work

One of the big motivations for this work was the black-box latency estimation approach in [18], where the authors implement a conformal prediction estimator for embedded hardware platforms running machine-learning inference. Though we did not utilize a similar embedded device or measurement framework, and targeted overall load instead of latency, the approach was inspirational in the application of conformal prediction to the task at hand. Though there have been a plethora of papers published in the field of conformal prediction in recent years [19], the application of Shapley values to conformal predictors is a relatively unexplored one. In [20] the authors apply a Shapley value calculation algorithm augmented by split conformal prediction to achieve explainable feature attributions. Though the authors here focus primarily on quantifying uncertainty of the feature attributions via a more efficient Shapley value estimate, it is in fact a target for future improvement of our own work. In [21] the authors focus on the interpretation of Shapley values for tree models in particular. This contains insights we intend to apply to our own future model implementations, since our framework leaves space for modular model improvements. There is significant room for improvement in the very approach to explainable feature attributions, as is evidenced by work on the very topic of replacing Shapley values as explanation mechanisms with *regional explanations* [22] so as to capture global, non-specific-prediction-specific significances of the features.

9.7 Conclusion and Future Work

In this work, we have developed an integrated framework for CPU load forecasting that combines deep neural networks with conformal prediction and Shapley value-based interpretability.

Our approach yields robust uncertainty quantification while providing clear, actionable insights into the contributions of individual tasks. Experimental results demonstrate that our model reliably predicts CPU load across a range of configurations, and that the Shapley values effectively capture the importance of each task, even under ablation conditions, underlining the critical role of certain functions in maintaining system performance. Our work awaits more extensive validation in different domains. Building on these results, as future work, we intend to pursue several avenues for enhancement. First, we plan to expand our model library by incorporating alternative methods such as Gradient Boosting Machine, Random Forest, and Quantile Random Forest based conformal prediction models, which may offer improved performance or robustness in different settings. Additionally, we intend to augment our dataset with supplementary system-level metrics, including background processes, interrupts, memory management, I/O operations, context switching, and peripheral interactions to capture a more holistic view of the factors influencing CPU load. From a model development perspective, we will explore incremental learning strategies (e.g., elastic weight consolidation) to enable our models to incorporate new data without requiring complete retraining, thereby preserving learned knowledge over time. We also envision enhancing the application interface to support intuitive, drag-and-drop task management and to facilitate real-time model validation and automatic refinement via direct interfacing with the embedded device. Finally, recognizing that our current conformal prediction implementation guarantees only marginal coverage, we will investigate training-conditional coverage methods [23, 24] to provide more specific, per-observation uncertainty guarantees. These enhancements are expected to further strengthen the safety and robustness of our forecasting framework in real-world industrial deployments.

Acknowledgements

The authors are partially supported by the Swedish Knowledge Foundation via the projects PerFlex (*Performant and Flexible digital Systems through Verifiable Artificial Intelligence*), grant nr. 20220033, and IGP-IDS (*International guest professor of Intelligent Distributed Systems*), grant nr. 20230147.

Bibliography

[1] Shiva Nejati, Stefano Di Alesio, Mehrdad Sabetzadeh, and Lionel Briand. “Modeling and analysis of CPU usage in safety-critical embedded systems to support stress testing”. In: *Model Driven Engineering Languages and Systems*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 759–775.

[2] John Regehr and Usit Duongsaa. “Preventing interrupt overload”. en. In: *SIGPLAN Not.* 40 (7 July 12, 2005), pp. 50–58.

[3] Eduardo Ciliendo, Takechika Kunimasa, and Byron Braswell. *Linux performance and tuning guidelines*. IBM, International Technical Support Organization, 2007.

[4] Pavel Yosifovich, Mark E Russinovich, Alex Ionescu, and David A Solomon. *Windows Internals: System architecture, processes, threads, memory management, and more, Part 1*. Redmond, WA: Microsoft Press, 2017.

[5] Glenn Shafer and Vladimir Vovk. “A tutorial on conformal prediction”. In: *arXiv [cs.LG]* (June 21, 2007).

[6] Eyal Winter. “Chapter 53 The shapley value”. In: *Handbook of Game Theory with Economic Applications*. Vol. 3. Handbook of Game Theory and Economic Applications. Elsevier, 2002, pp. 2025–2054.

[7] Abbas Khosravi, Saeid Nahavandi, and Doug Creighton. “Construction of optimal prediction intervals for load forecasting problems”. In: *IEEE Trans. Power Syst.* 25 (3 Aug. 2010), pp. 1496–1503.

[8] Jorge De la Torre, Leticia R Rodriguez, Francisco E L Monteagudo, Leonel R Arredondo, and José B Enriquez. “Electricity price forecast in wholesale markets using conformal prediction: Case study in Mexico”. en. In: *Energy Sci. Eng.* 12 (3 Mar. 2024), pp. 524–540.

[9] Ciaran O'Connor, Mohamed Bahloul, Roberto Rossi, Steven Prestwich, and Andrea Visentin. "Conformal Prediction for electricity price forecasting in the day-ahead and real-time balancing market". en. In: *Energy and AI* 21 (100571 Sept. 1, 2025), p. 100571. (Visited on 09/18/2025).

[10] Anastasios N Angelopoulos and Stephen Bates. "A gentle introduction to conformal prediction and distribution-free uncertainty quantification". In: *arXiv [cs.LG]* (July 15, 2021).

[11] Nicolas Dewolf, Bernard De Baets, and Willem Waegeman. "Valid prediction intervals for regression problems". en. In: *Artif. Intell. Rev.* 56 (1 Jan. 2023), pp. 577–613.

[12] Econometric Society. *Econometrica: journal of the Econometric Society*. Vol. 23. Econometric Society, the University of Chicago, 1955.

[13] Scott Lundberg and Su-In Lee. "A unified approach to interpreting model predictions". In: *arXiv [cs.AI]* (May 22, 2017). Ed. by I Guyon, U V Luxburg, S Bengio, H Wallach, R Fergus, S Vishwanathan, and R Garnett, pp. 4765–4774. (Visited on 09/18/2025).

[14] Hugh Chen, Ian C Covert, Scott M Lundberg, and Su-In Lee. "Algorithms to estimate Shapley value feature attributions". en. In: *Nat. Mach. Intell.* 5 (6 May 22, 2023), pp. 590–601.

[15] Xiaotie Deng and Christos H Papadimitriou. "On the complexity of cooperative solution concepts". en. In: *Math. Oper. Res.* 19 (2 May 1994), pp. 257–266.

[16] Nicolas Deutschmann, Mattia Rigotti, and Maria Rodriguez Martinez. "Adaptive conformal regression with Jackknife+ rescaled scores". In: *arXiv [cs.LG]* (May 31, 2023).

[17] Salim I Amoukou and Nicolas J B Brunel. "Adaptive Conformal Prediction by reweighting nonconformity score". In: *arXiv [stat.ML]* (Mar. 22, 2023).

[18] Matthias Wess, Daniel Schnöll, Dominik Dallinger, Matthias Bittner, and Axel Jantsch. "Conformal prediction based confidence for latency estimation of DNN accelerators: A black-box approach". In: *IEEE Access* 12 (2024), pp. 109847–109860.

[19] Valery Manokhin. *Awesome Conformal Prediction*. Comp. software. Version v1.0.0. Apr. 2022.

[20] David S Watson, Joshua O'Hara, Niek Tax, Richard Mudd, and Ido Guy. "Explaining predictive uncertainty with information theoretic Shapley values". In: *arXiv [stat.ML]* (June 9, 2023).

[21] Fan Xu, Zhi-Jian Zhou, Jie Ni, and Wei Gao. "Interpretation with baseline shapley value for feature groups on tree models". en. In: *Front. Comput. Sci.* 19 (5 May 2025), p. 195316.

[22] Salim Ibrahim Amoukou. "Trustworthy machine learning: explainability and distribution-free uncertainty quantification". PhD thesis. Université Paris-Saclay, 2023.

[23] Vladimir Vovk. "Conditional validity of inductive conformal predictors". In: *Asian conference on machine learning*. 2012, pp. 475–490.

[24] Michael Bian and Rina Foygel Barber. "Training-conditional coverage for distribution-free predictive inference". In: *Electron. J. Stat.* 17 (2 Jan. 1, 2023), pp. 2044–2066.

Chapter 10

Paper C: Machine Learning-Based Cache Miss Prediction

Edin Jelačić, Cristina Seceleanu, Ning Xiong, Peter Backeman, Sharifeh Yaghoobi, Tiberiu Seceleanu

Published in: International Journal on Software Tools for Technology Transfer (STTT), April 2025

Note: This paper has been reformatted to comply with the thesis layout. The content is unchanged from the published version.

Abstract

Integrating machine learning into computer architecture simulation offers a new approach to performance analysis, moving away from traditional algorithmic methods. While existing simulators accurately replicate hardware, they often suffer from slow execution, complex documentation, and require deep CPU knowledge, limiting their usability for quick insights. This paper presents a deep learning-based approach for simulating a key CPU component, cache memory. Our model “learns” cache characteristics by observing cache miss distributions, without needing detailed manual modeling. This method accelerates simulations and adapts to different program needs, demonstrating accuracy comparable to traditional simulators. Tested on Sysbench and image processing algorithms, it shows promise for faster, scalable, and hardware-independent simulations.

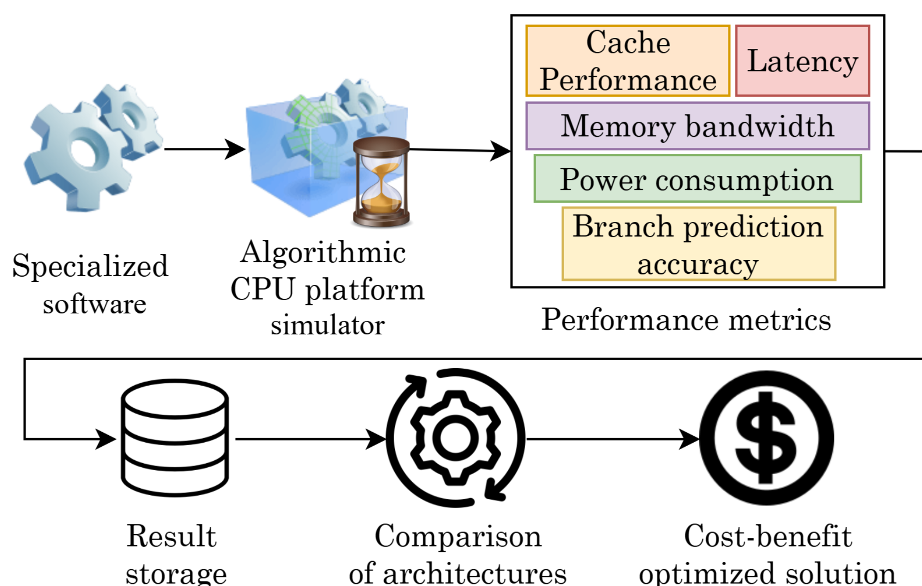


Figure 10.1: Commonly utilized industrial software simulation paradigm

10.1 Introduction

As hardware costs increase, computer architecture simulation has become essential for optimizing and estimating performance, identifying bottlenecks, and providing a cost & time-effective solution without the explicit need for physical hardware. A commonly used way to perform this optimization is the analysis of large amounts of data obtained from such simulators, as illustrated by Figure 10.1. Several algorithmic simulators and instrumentation tools are available for dynamic memory, security analysis, and performance monitoring, including DynamoRIO Cachesim [1], Valgrind [2], Intel PIN [3], and QEMU [4]. These tools serve specific purposes and use cases, but they share the drawback of requiring either algorithmic processing of each individual instruction, or instrumentation of execution on an existing CPU, which introduces a significant performance overhead in terms of computation time [5]. This overhead may lead to unreliable performance measurements and is therefore preferably avoidable.

In this work, we focus on the prediction of cache misses based on a novel statistical machine learning design. Traditional algorithmic simulators provide detailed insight into program execution, but face limitations due to high overhead and the need for detailed architectural knowledge, especially in complex systems. Two main challenges arise in instruction-accurate simulation of x86 CISC architectures: the construction of complex, error-prone models for each architecture [6], and significant performance issues, with simulators running much slower than native execution [7]. Recent advances like Ithemal [6], SimNet [7], and the latest TAO [8] address these

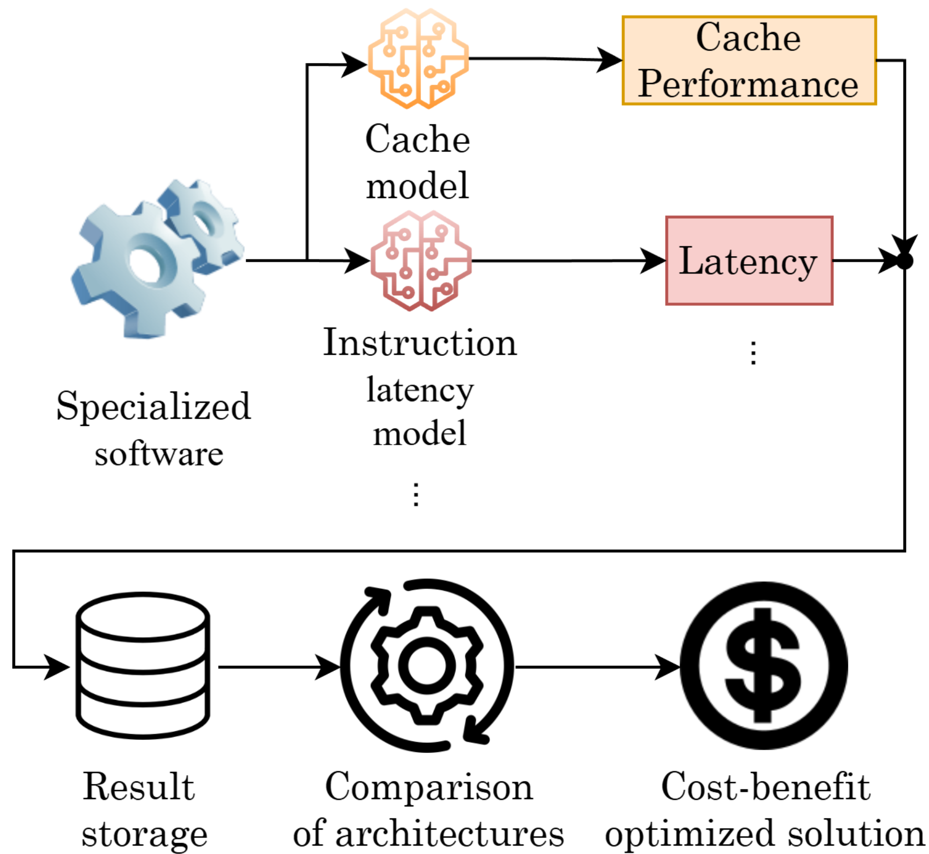


Figure 10.2: Proposed industrial software simulation paradigm

challenges using deep learning to predict instruction latencies and other performance metrics in specific architectures without extensive manual system definitions.

In modern computing, optimizing the speed of memory access is critical to improving performance and energy efficiency. Cache memory plays a vital role in this by providing rapid data retrieval to the processor. Consequently, the optimization of a hardware platform concerns the organization of the cache memory utilized by the platform. Each distinct application demands a varied utilization of cache memory throughout its runtime. Dimensioning of memory resources with respect to an executing program strongly impacts cache miss occurrences that lead to potentially significant delays in processing. Therefore, system designers, particularly in high-cost industrial environments, are keen to cater to specific software needs when deciding on prospective platform investment.

However, managing cache size and structure remains a challenge due to trade-offs between cost, performance, energy consumption, and cache misses, which lead to increased memory access times and degraded system performance. While existing simulators model cache be-

havior, faster and more adaptable solutions are needed to better address these inefficiencies. A promising, yet underexplored approach is the use of machine learning for modeling application-specific cache behavior.

This work focuses on the design and implementation of a neural network-based model with the purpose of predicting cache performance within an existing cache simulation framework, aiming to enhance and improve it for specific use cases, without assuming a fixed CPU architecture. Complementary to some existing work, our research here aims ultimately: (i) to replace a traditional simulator for cache prediction with a holistic machine learning model; (ii) for this model to function and generate performance metrics across various program executions using a hardware-agnostic approach; (iii) for this model to focus on the comparison of as many hardware combinations as possible while targeting a limited set of programs.

We introduce a *long-short-term memory* (LSTM) [9] deep learning approach for predictive cache miss modeling on program execution traces on multiple prospective architectures and demonstrate this on benchmark traces for testing. Our choice is justified by the fact that LSTMs are variants of recurrent neural networks, capable of adequately handling long-term sequential dependencies. To our knowledge, it is the only work that addresses the issue of simulating cache performance on various architectures in this manner. Our ML-based method offers the potential for faster predictions and greater flexibility to adapt to varying program and architecture characteristics without the need for detailed full-system architectural simulation. This ultimate goal framework, along with its utility, is illustrated

In summary, the key contributions of this paper are:

- A novel method for modeling different cache levels using a recurrent multivariate regression model;
- Pioneering the use of LSTM networks in modeling cache miss distribution, independent of the architecture;
- The ability of our method to observe the impact of core/cache configurations on cache miss distribution per individual core and core count for the corresponding first-level caches and shared unified caches.

The validation of this approach comes by testing it on widely used benchmark software and an image processing algorithm across various cache sizes and core counts, and the evaluation of the efficiency of the method is performed by comparison to an algorithmic cache simulator.

The remainder of the paper is structured as follows. Section 11.2 provides essential background on memory, cache, neural networks, and the DynamoRio Cachesim simulator. Section 10.3 details our method to predict the cache failure distribution throughout the execution of the program. We follow with Section 10.4 where we dive into the data, the instruction features and the process of training and deploying our model. Section 10.5 presents the results of our model's performance, and Section 11.7 discusses the successes and challenges of our approach, while Section 10.7 explores alternative approaches and state-of-the-art techniques, comparing them with our objectives. We conclude with final thoughts and suggestions for future work in Section 11.8.

10.2 Background

In contemporary microarchitectures, *cache memory* embodies a hardware implementation of small, yet high-speed memory situated in close proximity to the processor. In modern processors, it serves as a hardware-implemented memory layer positioned between individual registers and the main memory blocks. The latter are commonly identified today based on the main implementation technology, e.g. *DRAM* (Dynamic Random-Access Memory).

10.2.1 Memory hierarchy

Processor registers are compact memory locations, usually 32 or 64 bits, and serve as architectural components of the processor instruction set. The processor operates on these registers directly, through various instructions. Registers are used to store instructions, counters, memory addresses, operands, and the immediate results of instruction execution.

The main memory constitutes volatile read/write storage, where programs are loaded directly during run-time. It is notably large, and it is common for contemporary desktop platforms to be equipped with more than 32 gigabytes, due to the necessities of modern operating systems and increasingly complex software. However, accessing DRAM for data whenever necessary would be prohibitively slow, as DRAM lies architecturally and physically outside the processor. This would represent a bottleneck with the processor being starved waiting for incoming data, particularly given that a program's execution often relies on reusing data.

10.2.2 Cache memory

Cache memory is a specialized component integrated into the processor to reduce latency by storing recently accessed data. It is structured into multiple levels, including L1, L2, and L3, each with varying capacities and speeds. L1 is the fastest but the smallest, while L3 offers greater capacity but higher access latency. The processor utilizes the cache in a hierarchical manner to minimize reliance on slower main memory, significantly reducing program execution delays. In multicore processors, usually each core has at least one cache level (L1), with other levels usually shared between cores.

Throughout program execution, the processor employs cache memory to store frequently used data and instructions. When the processor is instructed to access a specific address by the instruction code, it initiates the process by checking the L1 cache. If the data is not found in L1, it proceeds to check L2, and so forth. If the data is not found in any of the caches, it is retrieved from the main memory and stored in one of the cache levels [10], and until data is retrieved, the execution is paused. This event is known as a *cache miss*, and various advances in modern computer design aim to minimize these occurrences and mitigate the corresponding latency penalties. Strategies include continuous increases in on-die cache memory sizes throughout history [11], cache memory restructuring, the incorporation of various optimizations [12], and the introduction of mechanisms, such as pre-fetchers, which anticipate the most likely instructions or data that the processor will need and fetch them before they are needed [13].

In addition to the above, the evolution of processors, exemplified by advancements like multithreading, can be viewed as a step towards minimizing the time-delay associated with a cache miss by implementing thread-level parallelism [14]. Predicting cache misses, implementing strategies for their avoidance or scalable mitigation techniques for compulsory cache misses is of great significance, especially in a modern computing landscape, where memory speed lags behind that of contemporary processor designs [15].

10.2.3 Neural Networks

Neural networks (NN) have proven to be useful as universal function approximators [16, 17]. They have been used, in various forms, for improvements or predictions in computer architectures [18]. The basic neural network is the *feed-forward* NN, called so due to its structure of a mesh of neurons with one or multiple inputs, one or multiple outputs and one or more hidden layers representing linear functions paired with nonlinear activation functions.

An NN feedforward estimates a function f, with data flowing from the input $\mathbf{x}$, processing in the hidden layer(s) defined by a parameter set $\mathbf{\Theta}$, the output being a vector of real-valued results, commonly denoted by the *prediction* $\mathbf{y}$. More precisely, NN represent statistical mappings $p(\mathbf{y}|\mathbf{x};\mathbf{\Theta})$ in which the parameters $\mathbf{\Theta}$ are tuned, by backpropagation, in such a way as to locally minimize some cost function that maps the weights to the outputs of the objective cost function. Feed-forward NNs do not have feedback loops that map past quantities to internal memory when making predictions. This makes them only appropriate for calculating predictions in which the predictions have no temporal relationship with historical data.

In those problems where the dataset on which we are trying to calculate the predictions is sequential in nature, sequential NN or RNN (recurrent neural networks) are commonly used. By sequential nature, we mean that the sequence of inputs is a factor in predicting the desired distribution, as $p(\mathbf{y}|(\mathbf{x},\mathbf{x}_{-1},\mathbf{x}_{-2},\ ...,\mathbf{x}_{-k});\mathbf{\Theta})$ where k is the number of time steps in the sequence.

The most prominent sequential architecture is the LSTM NN, composed of a memory cell and multiple non-linear gates that regulate the flow of data, represent current, and selectively memorize prior states. This minimizes the vanishing / exploding gradient problem, typically present in RNN backpropagation [9, 19]. Such an architecture is shown in Figure 10.3. Due to its prominence in the field of sequential timeseries multivariate forecasting, we opted for this model as well.

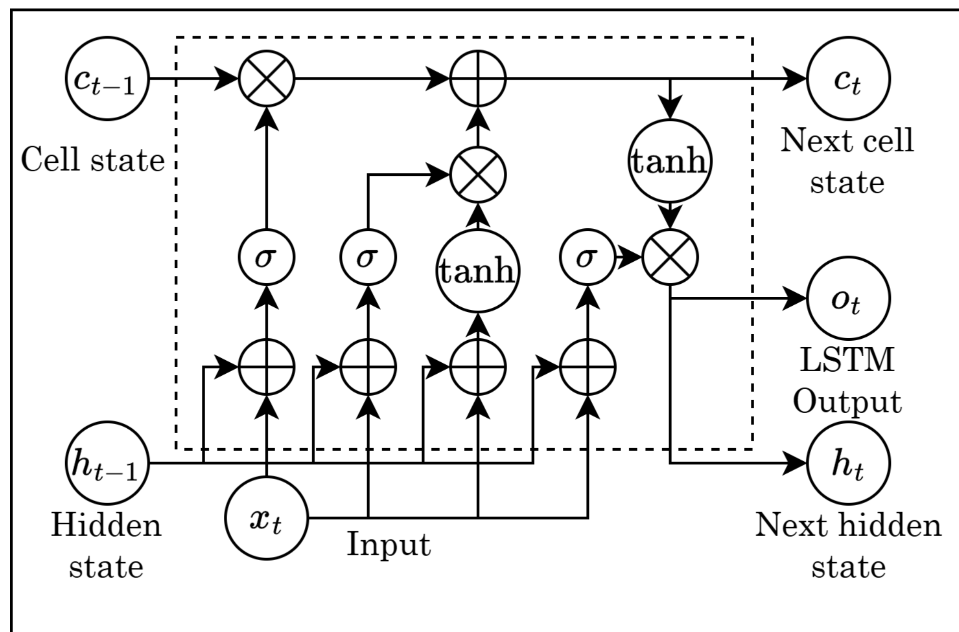


Figure 10.3: LSTM Cell architecture

10.2.4 DynamoRIO Cachesim

DynamoRIO Cachesim is a simulator tool designed to analyze the cache performance of applications by simulating the impact of various cache configurations on memory access patterns. It functions by dynamically instrumenting running applications to collect detailed memory access data, which is then utilized to simulate cache behavior.

The simulator models L1 and L2 cache levels by default and can be extended to support more complex configurations. It supports varying sizes, cache associativities, and block sizes to provide insights into cache performance metrics. The simulator operates on an executable by injecting instrumentation code into the application's binary at runtime, capturing each memory access that the application performs. These captures are stored in a format that Cachesim and other components in the DynamoRIO suite can use to simulate desired cache configurations and compute metrics such as cache hit rates, miss rates, performance across different cache levels, register states, and latency impacts. DynamoRIO itself is extensible and well-documented, offering the ability for users to build their own client tools for performance analysis within the DynamoRIO framework or build upon existing ones.

10.3 The Approach

In this section, we present our approach that relies on equating predictive modeling to the statistical estimation of a cache miss distribution across a program's execution through running aggregates of cache misses. Specifically, given a program of length K and a sequence length of N ($N < K$), to estimate such a distribution we generate $\lceil K/N \rceil$ data points, with each data point at most of value N (observe Figure 10.4 for an example with a sequence length of 9 instructions).

Our model utilizes deep learning to take a program trace and specified cache characteristics as input, returning the cache miss distribution as output. By program traces, here we refer to x86 assembly traces of executed programs. When access to the original source code and compilation procedure is unavailable, these traces can be obtained using instrumentation tools, such as Intel PIN [3], Valgrind [2], or other methods like debuggers and disassemblers. Hence, we turn the task of predictive modeling of cache behavior into one of multivariate sequential regression. Program traces are represented by the x86 assembly instructions (the instructions themselves, with the corresponding operands, such as constants and registers), which the execution of a

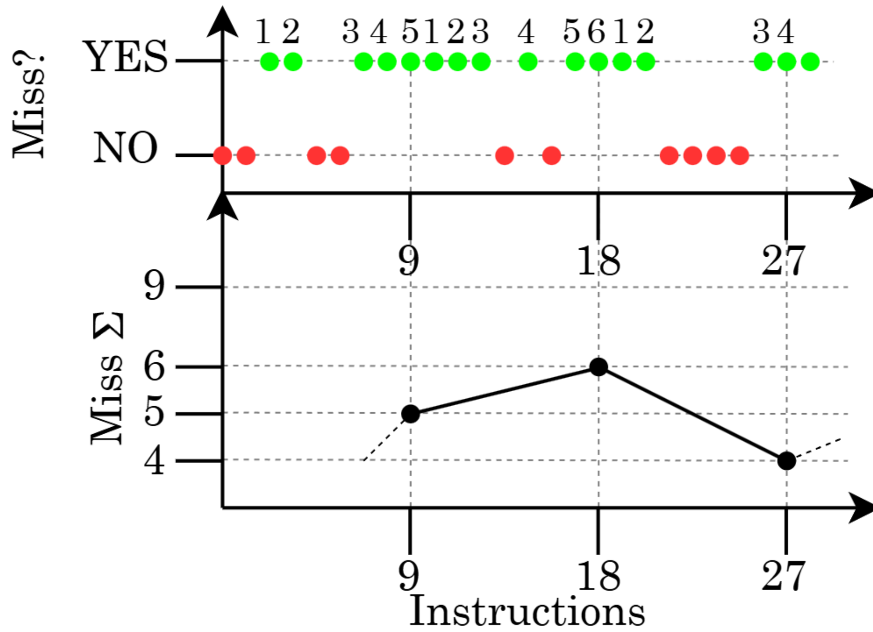


Figure 10.4: Miss sub-sequence summing

program consists of, as exemplified below:

```
...
cmp     %rax $0x0000000000000022
jbe     $0x00007f9290f3e0c9
mov     %rdi -> %rsi
sub     %rax %rsi -> %rsi
cmp     %rsi $0x000000000000000f
...
```

In the above, each instruction is supplemented with additional features procured by the cache simulator that we employ for initial data gathering, that is, DynamoRIO Cachesim [1].

10.4 Implementation

For gathering our traces, we use Cachesim, with modifications enabling writing features of a trace to a suitable SQLite database. Namely, we implement a mechanism for writing out the results of Cachesim simulation, for each execution, to a dedicated SQLite file, in a format that could be utilized later for machine learning with reasonable overhead by the pandas [20] library for Python.

Table 10.1: Table of instruction row features.

Feature Name	Values	Explanation
Disassembly String	String	The instruction itself (e.g., `sub %rax %rsi → %rsi`)
Instruction Number	Positive integers	The ordinal number of the instruction during the execution of the program.
Access Address Delta	Integers	The positive or negative offset in the memory of the address of requested data w.r.t. the previous instruction, initialized from zero and resetting at each thread switch.
PC Address Delta	Integers	The positive or negative offset of the Program Counter (PC) w.r.t. the previous instruction's PC, initialized from 0 and resetting at each thread switch.
Instruction Type	Categorical	The type of the recorded program event. Can be an Instruction Fetch (ifetch), Memory Store (write), Memory Load (read), etc. [1]
Byte Count	Positive integers	Number of bytes loaded or read.
Core	Positive integers	The current simulated core that the thread is running on.
Thread Switch	Boolean	True (1) if a thread switch occurred for this entry, false (0) if it has not.
Core Switch	Boolean	True (1) if a core switch occurred for this entry, false (0.0) if it has not.
L1D size	Positive integers	The size (in bytes) of the L1 Data cache, constant throughout a single execution.
L1I size	Positive integers	The size (in bytes) of the L1 Instruction cache, constant throughout a single execution.
LL size	Positive integers	The size (in bytes) of the unified Last Level cache, constant throughout a single execution.

10.4.1 DynamoRIO Instruction Features

DynamoRIO has the ability to supplement the executed instructions with various features, of which we keep those that we deem most important for the task at hand. The features for our deep learning algorithm are shown in Table 10.1. The reasoning for the delta encoding is the reduction of the range of possible values by way of storing merely the relative difference between sequential addresses rather than the absolute addresses. More importantly, programs generally exhibit spatio-temporal locality [12], which the delta encoding exploits by improving overall storage efficiency and reduces the overall computational cost of the encoding as opposed to computing for 64-bit addresses.

10.4.2 Tokenization

The key aspect for applying deep learning in our approach is the tokenization process, a fundamental preprocessing step that involves converting raw text data into a format interpretable by machine learning models. This is the process of transforming the input text of the disassembled instruction string into a sequence of discrete elements, or tokens, which represent the building

[1]The full list may be observed as part of the trace_entry.h header file, which is part of the DynamoRIO repository at `https://github.com/DynamoRIO/dynamorio`.

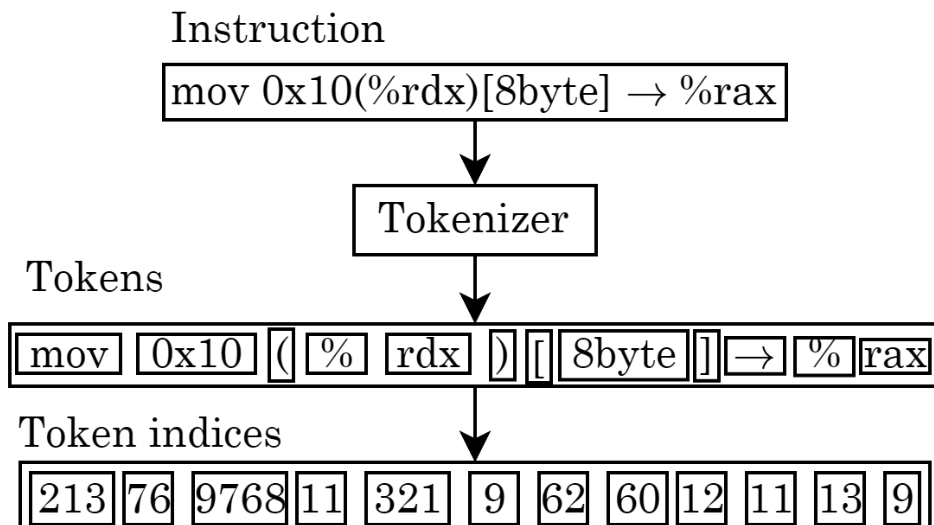


Figure 10.5: x86 Instruction tokenization example (the numbers for token indices are purely exemplary)

blocks of the text. In our case, these elements are the assembly instructions, their operands and operators such as brackets and assignments (an example is shown in Figure 10.5).

For this procedure, we opt for the "Byte Level" tokenizer and the "Whitespace" preprocessor by HuggingFace [21]. The tools available from HuggingFace are widely used and highly regarded in the field of natural language processing [22], albeit we only use a small tool subset in our work.

The tokenizer is initially trained on the set of instructions present in the corresponding program traces, upon which a dictionary of tokens and corresponding token indices are generated and stored. After the tokenizer is prepared (trained, stored or loaded), each instruction is broken down into individual tokens and then the individual tokens are converted into token indices, numerical representations of tokens that a deep learning model can process. The outlined tokenization procedure is supplemented in the deep learning model with a dedicated trainable embedding layer, which contextualizes each token's ID in the form of a specialized lookup table with preset maximum embedding length, as per Fig. 10.6. In our case, the maximum embedding length was 15, up until which a shorter instruction string was to be padded with [PAD] tokens, and after which longer instruction strings were truncated.

This ensures that each instruction's components, the operands, and the location registers are given numerical values that the model can mathematically operate on. For the desired model output, we take the cache misses for the three simulated cache levels, for which traces are exe-

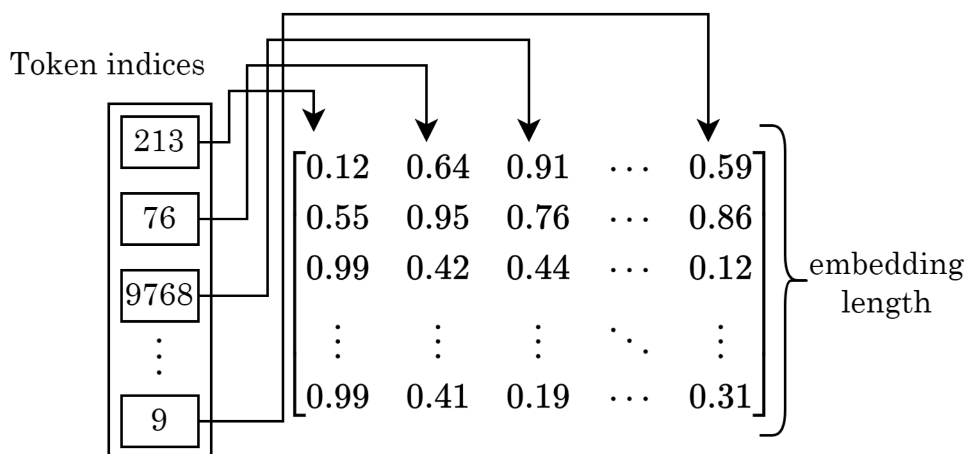


Figure 10.6: Token indices lookup table, generated by the embedding layer, whereby each token is assigned a corresponding vector of a-priori fixed length (15 here)

cuted and stored: L1 data, L1 instruction and shared LL (last-level) cache, summed up across sub-sequences. In this way, we introduce profiling of the trace without sliding the prediction window one-by-one instruction-wise, but rather sub-sequence by sub-sequence, without overlapping, which enables a significant speedup both in training and inference.

10.4.3 Model and Training Procedure

Our model is composed of three components: (i) a **token processing embedding layer**, (ii) an **LSTM layer** for the instructions and other features, and (iii) **an affine, fully connected dimension-reduction layer** that takes the final output of the major LSTM layer as input. The model form is described in Figure 10.7 and is implemented in the PyTorch framework [23].

When a certain program is selected (typically one with parameters, such as a benchmark with variable time and stress loading limits or targets), we gather the dataset database initially by running a simulation through a custom tool within the DynamoRIO framework, with various cache parameter combinations, within a certain reasonable span of these parameters. This dataset is termed the “training/validation dataset of program X”, as it is the one in which all training will take place. In this work we opt for a training / validation ratio of 0.9.

For “testing dataset of program X”, we generate data on the same program family but with cache parameters previously unseen by the model. By *program family*, henceforth, we will refer to a set of programs with broadly similar purposes and methods, making specific cache configuration performance modeling reasonable. The models themselves are based on vari-

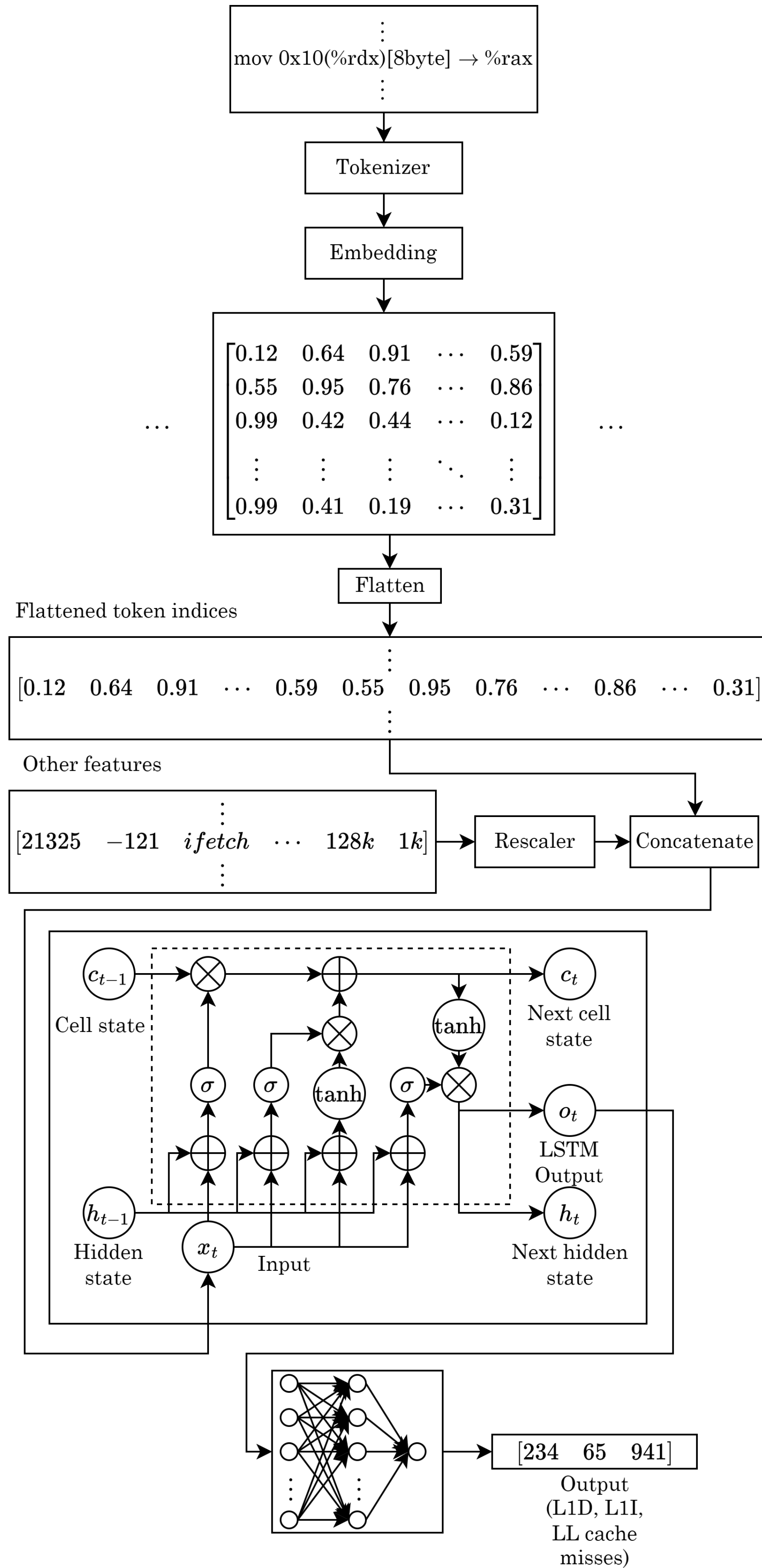


Figure 10.7: Utilized LSTM model outline

ous hyperparameters, such as **batch size** (the number of samples that is fed into our model at each iteration of the training process), **sub-sequence length**, **LSTM hidden dimension width**, **depth**, **learning rate** and **dropout** between neighboring LSTM layers.

We also propose hyperparameter optimization for each program family, since different program types have different cache access patterns in different parts of the execution. Hyperparameter optimization is taken care of by Optuna framework [24], chosen because of the easily modifiable *Trial* class provided. The loss function that we use is MSE and we limit training to 10 epochs and the hyperparameter optimization to 5 epochs on a greatly reduced dataset. The reason for such limits is the relatively long training time on our hardware, reaching up to 2 hours for a single epoch on the full training dataset.

We gear the optimization towards minimizing the validation loss within the initial 5 epochs, but with an emphasis on compute time limitations. Namely, an optimization trial of 5 epochs that takes longer than a given preset amount of time is automatically considered a failure, along with its hyperparameters. This leads to a potential exclusion of otherwise satisfactory parameters due to the constraints on execution time. Hyperparameter optimization is performed on a highly reduced data set, comprising only two databases (runs). It should be noted that several runs with improper batch size / sequence length had to be discarded during the hyperparameter optimization, as they showed to retrieve too much data for our compute machine's RAM to handle. This was also taken care of automatically by the framework with some minor tweaks to the code.

In total, the final parameters utilized were rounded to 330 for the hidden layer dimension size, 2 LSTM layers, 0.05 for the dropout between the LSTM layers, 60 for the batch size and $N = 200$ for the sequence length.

10.5 Results

The experimental evaluation is carried out on an i7-13700HX, 32 GB RAM, Nvidia RTX A1000 (6 GB VRAM, 2048 CUDA cores) laptop platform. To attain a solid base for testing this approach out, a program family has to be selected such that it consists of a practical set of programs that stress specific platform components, with a broad array of algorithms. For this purpose, Sysbench [25] is chosen, along with three of its utility benchmarks: CPU, Memory, and FileIO. The CPU benchmark tests computation-intensive tasks, as efficient cache utiliza-

tion, particularly L1D and L1I caches, is crucial for reducing latency. FileIO simulates disk read/write operations, with performance sensitive to cache sizes, especially the LL cache, as effective caching reduces disk access times. The Memory benchmark assesses the system's ability to handle memory operations, with larger caches improving data retrieval speed from main memory.

Another valuable use case that we used is a highly specific image processing algorithm termed "Good Features to Track" or GFTT, exposed as a memory-intensive application [26]. Such algorithms represent a range of workloads, providing a comprehensive analysis of how cache configurations impact system performance. We introduce a wrapper utility for DynamoRIO's cache simulator (drcachesim), to gather simulation data in the form of serverless SQlite3 databases, where one database corresponds to one program execution. The wrapper is available on GitHub (`https://github.com/ptrbman/dynamorio-missing-instructions`).

10.5.1 Performance Metrics

In our work, by *model performance* we refer to the ability of the model to capture the distribution of cache misses from the original trace. We measure this ability by the values of the mean squared error and the model's R^2, both of which are defined in further text.

In the following, let

$$\mathbf{y_{L1D}} = [y_{L1D_0}, y_{L1D_1}, \ldots, y_{L1D_j}, \ldots, y_{L1D_K}]^\intercal \tag{10.1}$$

be the actual mini-sums of L1D cache misses across K sub-sequences, where $K = program\ length\ /\ sub-sequence\ length$ and $y_{L1D_j}, j \in [0, K]$ represents the number of cache misses in the L1 data cache in the j-th sub-sequence. Further, let

$$\mathbf{\hat{y}_{L1D}} = [\hat{y}_{L1D_0}, \hat{y}_{L1D_1}, \ldots, \hat{y}_{L1D_K}]^\intercal \tag{10.2}$$

be the mini-sums of L1D cache misses across K sub-sequences as predicted outputs of model inference. Similarly, we define $\mathbf{y_{L1I}}$, $\mathbf{y_{L1I}}$, $\mathbf{\hat{y}_{L1I}}$, $\mathbf{y_{LL}}$, and $\mathbf{\hat{y}_{LL}}$.

The mean squared error E_{MSE}, for each of the cache types, respectively, is defined as per eq. (10.3).

$$E_{MSE} = \frac{1}{K}\sum_{i=1}^{K}(y_i - \hat{y}_i)^2 = \frac{1}{K}(\mathbf{y} - \mathbf{\hat{y}})^\intercal(\mathbf{y} - \mathbf{\hat{y}}) \tag{10.3}$$

We use E_{MSE} for both training the model and as a measurement of the model's performance on the training and test datasets, since for our approach the distribution of errors throughout an

execution is the most significant observation and we ideally desire to penalize the model for large outliers while retaining a smooth gradient for smaller deviations of predicted values as compared to the real values [27]. The E_{MSE} and its root square counterpart, the root mean squared error $E_{RMSE} = \sqrt{E_{MSE}}$ tell us how well the model performs with cache miss predictions throughout the execution of a program, which is one aspect of the prediction that we are interested in.

Another component of the prediction that we are interested in is the model's ability to predict total cache miss numbers, which we attain by summing up the outputs throughout the model's execution on a single program. For this measurement, we define the relative error percentage metric, based on the L1 norm, relative to the actual values, that is, E_{REP}. We let the total sum of actual cache misses be $T = \sum^K \mathbf{y}$ and the total sum of predicted cache misses be $\hat{T} = \sum^K \hat{\mathbf{y}}$, then the E_{REP} is defined as per eq. (10.4).

$$E_{REP} = \frac{|\sum^K \mathbf{y} - \sum^K \hat{\mathbf{y}}|}{\sum^K \mathbf{y}} \cdot 100 = \frac{|T - \hat{T}|}{T} \cdot 100 \tag{10.4}$$

The equation eq. (10.4) that we utilize here is similar to the equation for the *mean absolute error percentage*, more commonly abbreviated as MAPE. However, while MAPE concerns the relative distribution of errors across an entire execution, E_{REP} concerns the relative discrepancy of the aggregate sums. This is of importance because the total number of cache misses is critical and a significant concern is how well the total predicted cache misses align with the total actual cache misses. For example, a E_{REP} of 0 is ideal, whereas an E_{REP} of 50 means that predictions overshoot or undershoot the real values by 50%.

The two focal values to be observed are the **mean squared error** (paired with the R^2 and E_{RMSE}), demonstrating how well the forecast of cache misses along a program execution "tracks" the actual cache misses on average, as well as the **total numbers of cache misses** at the end of executions, where each execution regards a specific cache size / core combination.

The *determination coefficient*, R^2, is a statistical measure that indicates how well the independent variables in a regression model explain the variability of the dependent variable. It is calculated by taking the ratio of the sum of the squared differences between the predicted values and the mean of the dependent variable to the sum of the squared differences between the actual values and the mean of the dependent variable. Mathematically, $R^2 = 1-(SS_{res}/SS_{tot})$, where SS_{res} is the sum of squared residuals (the differences between observed and predicted values), and SS_{tot} is the total sum of squares (the differences between observed values and their mean). However, it can be negative in the case that the model is highly misleading.

10.5.2 Execution

Training runs on several cache size, core count and program combinations:

- L1 Data cache sizes: 512 bytes, 8 kilobytes, 32 kilobytes,
- L1 Instruction cache sizes: 512 bytes, 8 kilobytes, 32 kilobytes,
- LL unified cache sizes: 1 kilobyte, 8 kilobyte, 32 kilobytes,
- Core counts: 1, 2, 4,
- Benchmarks: Sysbench CPU, Sysbench Memory, Sysbench FileIO, GFTT.

Note that we have 3 possible sizes for each L1D, L1I, LL, 3 core counts, and four benchmarks in total, giving us $3 \times 3 \times 3 \times 3 \times 4 = 324$ traces to train on.

To limit the amount of training time, due to the sheer volume of data, a limit is set for the maximum amount of instructions recorded for each execution by the DynamoRIO wrapper to $K = 25 \times 10^6$. The training datasets consist of 90% of all datasets, while 5% of the dataset are set aside for validation during training and 5% for any residual testing.

For the full model testing, a brand new dataset is collected in a similar manner, but with different options:

- L1 Data cache sizes: 1 kilobyte, 2 kilobytes, 64 kilobytes,
- L1 Instruction cache sizes: 1 kilobyte, 2 kilobytes, 64 kilobytes,
- LL unified cache sizes: 2 kilobytes, 4 kilobyte, 16 kilobytes,
- Core counts: 1, 2, 4,
- Benchmarks: Sysbench CPU, Sysbench Memory, Sysbench FileIO, GFTT.

Of particular interest here are the 64 kB cache sizes, as they are twice the size of the maximum cache sizes from training, which potentially may produce significant issues for the model. The general executions are all of the form observed in figures 10.8 and 10.9, for the benchmark CPU L1D errors.

Figure 10.8 shows an example of our predictions on one sample of executions of L1D misses of the CPU benchmark; Figure 10.9 presents the first 200 sub-sequences. The trace ends at

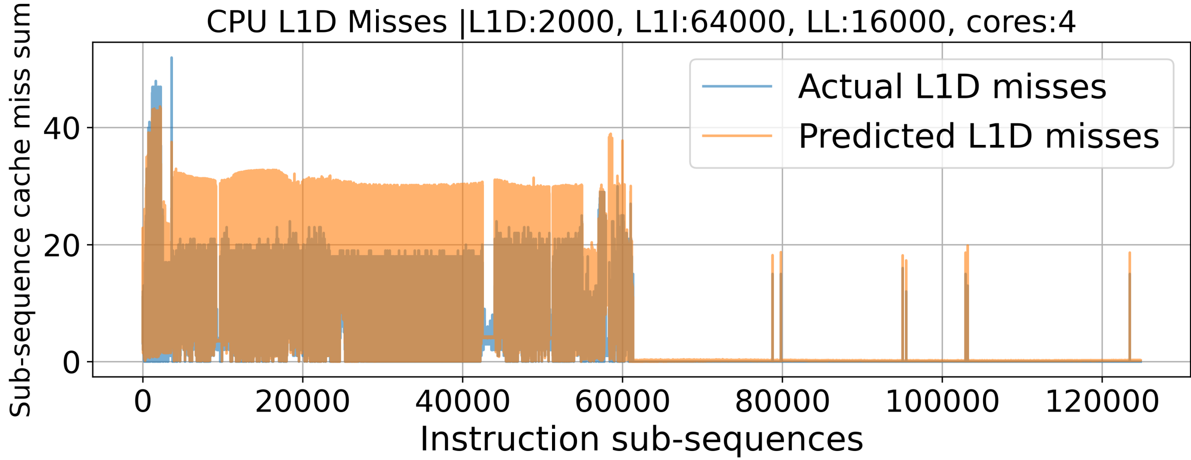


Figure 10.8: Example of execution, L1D cache, Sysbench CPU Benchmark.

125000 on the x-axis, due to the fact that we have $K = 25 \times 10^6$ and $N = 200$, which then corresponds to $K/N = 125 \times 10^3$.

In this example, we can observe that the model has a tendency to exceed the number of cache misses fairly regularly. For this particular run, the E_{MSE} was 39.1 and E_{REP} was 67.4%, meaning that although the overall distribution of cache predictions is satisfactory with the E_{RMSE} around 6, the total aggregate counts leave room for improvement. The values for all E_{MSE} cache / core combinations for the tested programs, as well as E_{REP} can be observed in Figures 10.10 to 10.13. Overall error confidence intervals with respect to core count for E_{MSE} and E_{REP} can be observed in Figures 10.14 to 10.17. For example, for the L1I cache performance for the Memory benchmark, for the combination with L1D size 64k, 2 cores, L1I at 1k, LL at 2k, the E_{MSE} is 9.9 and the E_{REP} is 31.0, meaning that the model is 31% off the mark in terms of aggregate instructions when compared to the real value but that the overall distributed mean squared error is relatively small.

Overall, the model generally achieves satisfactory results for the distribution of cache misses, with a few notable outliers, which is to be expected given that the training procedure has been optimized to minimize MSE loss. However, when it comes to the total aggregate count of cache misses, the model consistently overestimates, particularly in the case of the GFTT program. For instance, in certain runs, especially with the 64k L1I - 16k LL cache combination, the model predicts up to 10 times more total cache misses than observed in the actual data, as shown in fig. 10.18. This overestimation may be due to insufficient training or the model's difficulty fully capturing the complexity of the executing algorithm, especially for edge-case cache sizes that fall well outside the range of training data. These extreme inputs could be pushing the model to

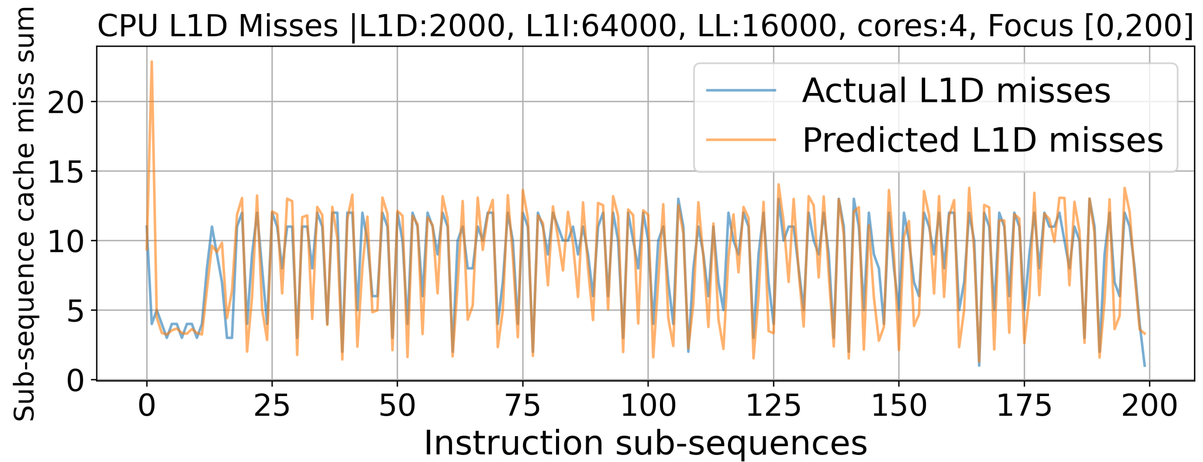


Figure 10.9: Example of execution, L1D cache, Sysbench CPU Benchmark, first 200 sub-sequences

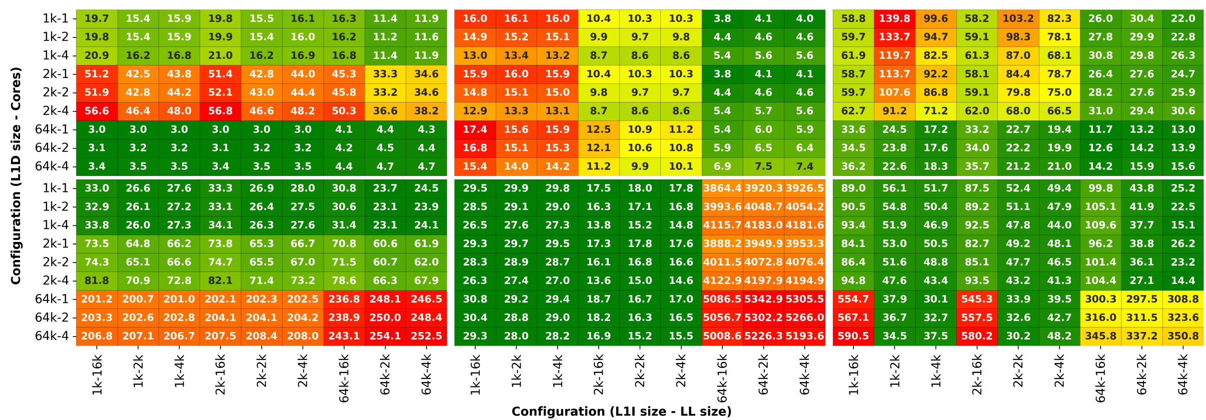


Figure 10.10: CPU Benchmark prediction measurements; top in left to right order: L1D E_{MSE}, L1I E_{MSE}, LL E_{MSE}; bottom in left-to-right order: L1D E_{REP}, L1I E_{REP}, LL E_{REP}, green is better, red is worse

output values that significantly deviate from reality.

Regarding the analysis of output values per core count, the relatively small differences between metrics across various core counts suggest that the model performs consistently, regardless of core count. While this limits our ability to infer a strong relationship between core count and model performance, it could also indicate that the model is resilient across different core configurations. Additionally, this observation may point to potential opportunities for further enhancing the underlying DynamoRIO cache simulation to better capture the complexities of multi-core environments.

The performance of the model (in terms of E_{RMSE}) when comparing the different programs

Figure 10.11: Memory Benchmark prediction measurements; top in left to right order: L1D E_{MSE}, L1I E_{MSE}, LL E_{MSE}; bottom in left-to-right order: L1D E_{REP}, L1I E_{REP}, LL E_{REP}, green is better, red is worse

Figure 10.12: FileIO Benchmark prediction measurements; top in left to right order: L1D E_{MSE}, L1I E_{MSE}, LL E_{MSE}; bottom in left-to-right order: L1D E_{REP}, L1I E_{REP}, LL E_{REP}, green is better, red is worse

chosen may be observed at Figure 10.19. The E_{RMSE} drops off significantly when raising the L1D size and L1I size, which is due to the overall fewer misses, naturally due to the cache having to retrieve data from the main memory less often. The R^2 values across different cache sizes indicate how well the model explains the variability in the data, with higher R^2 values (close to 1) being desirable as they indicate better model performance. In this case, CPU and FileIO show mostly positive R^2 values, indicating reasonable model performance.

The GFTT L1I shows a negative R^2 at cache sizes of 2k and 64k, suggesting that the model's performance in these specific cases may require further refinement - a topic for future work. Importantly, the CPU benchmark's cache behavior remains the most reliably well-modeled among

L1D E_{MSE}

Configuration (L1D size - Cores) \ Configuration (L1I size - LL size)	1k-16k	1k-2k	1k-4k	2k-16k	2k-2k	2k-4k	64k-16k	64k-2k	64k-4k
1k-1	29.7	53.7	52.4	28.0	48.4	47.8	27.7	31.0	29.6
1k-2	33.9	57.2	57.3	30.2	51.6	52.6	27.7	33.8	32.3
1k-4	44.2	62.8	64.2	37.6	57.2	59.0	27.9	40.6	38.5
2k-1	46.1	51.5	51.5	45.3	51.2	51.0	52.5	41.3	42.0
2k-2	49.7	54.2	54.6	47.9	53.5	53.6	53.0	43.9	43.9
2k-4	54.6	54.9	54.7	51.1	54.6	54.1	56.8	46.9	46.3
64k-1	1.1	1.1	1.1	1.1	1.2	1.1	1.1	1.2	1.2
64k-2	1.1	1.1	1.1	1.1	1.2	1.2	1.1	1.2	1.2
64k-4	1.1	1.1	1.1	1.1	1.2	1.2	1.2	1.2	1.2

L1I E_{MSE}

Configuration (L1D size - Cores) \ Configuration (L1I size - LL size)	1k-16k	1k-2k	1k-4k	2k-16k	2k-2k	2k-4k	64k-16k	64k-2k	64k-4k
1k-1	6.6	6.3	6.4	20.4	23.4	21.4	0.2	0.1	0.1
1k-2	6.5	6.3	6.3	18.9	22.6	20.5	0.2	0.2	0.2
1k-4	6.3	6.2	6.4	16.0	20.8	19.2	0.3	0.4	0.3
2k-1	6.6	6.3	6.4	20.3	23.9	22.0	0.2	0.1	0.1
2k-2	6.5	6.2	6.3	18.7	22.7	20.9	0.2	0.2	0.2
2k-4	6.3	6.2	6.4	15.6	20.5	19.2	0.3	0.4	0.4
64k-1	8.1	8.2	8.2	16.3	16.1	16.1	0.2	0.2	0.2
64k-2	7.9	8.0	8.0	15.9	15.8	15.8	0.2	0.2	0.2
64k-4	7.6	7.8	7.7	15.5	15.4	15.3	0.2	0.2	0.2

LL E_{MSE}

Configuration (L1D size - Cores) \ Configuration (L1I size - LL size)	1k-16k	1k-2k	1k-4k	2k-16k	2k-2k	2k-4k	64k-16k	64k-2k	64k-4k
1k-1	38.2	71.0	115.5	34.7	87.5	104.2	12.6	22.0	40.0
1k-2	37.2	76.0	121.0	34.3	90.7	110.4	12.7	24.5	40.2
1k-4	35.0	86.2	124.0	32.7	95.9	116.9	12.7	31.7	41.6
2k-1	35.3	63.4	132.6	33.8	88.6	120.8	12.6	29.6	34.0
2k-2	34.7	66.2	131.9	33.5	88.4	123.3	12.7	32.3	34.3
2k-4	33.0	68.0	113.5	32.1	86.0	119.1	12.7	35.3	35.2
64k-1	2.9	9.0	49.9	2.7	11.8	58.6	1.4	1.5	1.5
64k-2	2.8	9.0	49.6	2.6	11.7	58.5	1.5	1.5	1.5
64k-4	2.8	9.0	49.0	2.6	11.4	57.8	1.5	1.6	1.6

L1D E_{REP}

Configuration (L1D size - Cores) \ Configuration (L1I size - LL size)	1k-16k	1k-2k	1k-4k	2k-16k	2k-2k	2k-4k	64k-16k	64k-2k	64k-4k
1k-1	10.8	15.4	12.9	13.5	9.1	7.9	27.2	8.4	11.9
1k-2	7.9	17.1	15.2	11.8	10.4	10.0	27.0	6.1	9.5
1k-4	1.5	19.6	18.5	7.1	13.3	13.1	27.7	0.7	4.3
2k-1	18.7	13.8	12.9	25.8	5.4	5.4	54.7	19.3	24.2
2k-2	14.8	14.2	14.3	23.3	6.0	6.5	55.2	15.7	20.8
2k-4	10.0	12.5	13.5	20.5	5.5	5.9	60.0	10.8	15.5
64k-1	3.1	1.3	0.3	0.5	3.4	4.1	18.4	9.4	10.7
64k-2	3.4	1.5	0.8	0.8	2.8	3.2	21.2	11.7	13.0
64k-4	4.3	2.3	2.0	1.9	1.1	1.2	26.1	15.9	17.5

L1I E_{REP}

Configuration (L1D size - Cores) \ Configuration (L1I size - LL size)	1k-16k	1k-2k	1k-4k	2k-16k	2k-2k	2k-4k	64k-16k	64k-2k	64k-4k
1k-1	6.0	5.0	5.6	13.5	10.1	14.2	150.9	200.6	238.0
1k-2	6.0	4.9	5.4	16.2	10.8	14.4	151.3	220.3	252.5
1k-4	5.9	4.6	4.8	20.3	11.6	13.8	165.8	263.7	284.8
2k-1	6.2	5.2	5.7	13.6	9.4	13.0	152.7	214.2	247.2
2k-2	6.2	5.1	5.6	16.4	10.6	13.6	153.4	225.2	254.0
2k-4	5.9	4.9	5.1	20.6	12.1	13.8	167.3	252.6	271.5
64k-1	8.5	8.6	8.6	30.6	30.6	30.5	28.1	24.8	25.4
64k-2	8.2	8.2	8.2	30.1	30.0	29.9	29.1	25.7	26.3
64k-4	7.7	7.6	7.5	29.4	29.2	29.1	28.4	24.2	25.2

LL E_{REP}

Configuration (L1D size - Cores) \ Configuration (L1I size - LL size)	1k-16k	1k-2k	1k-4k	2k-16k	2k-2k	2k-4k	64k-16k	64k-2k	64k-4k
1k-1	95.5	9.1	18.6	84.4	4.5	14.2	70.3	11.0	54.9
1k-2	89.2	10.0	17.5	81.5	5.2	13.0	66.3	8.9	52.9
1k-4	75.2	11.6	14.9	71.5	7.1	10.7	55.7	3.9	50.6
2k-1	78.5	10.5	25.3	75.9	7.2	20.7	67.9	11.7	41.8
2k-2	73.5	10.6	23.6	73.3	7.1	19.1	63.6	9.3	39.0
2k-4	65.4	9.9	17.4	67.9	6.8	15.0	51.7	7.9	35.1
64k-1	20.9	9.3	118.7	18.3	16.1	144.8	15.0	7.4	6.5
64k-2	20.6	9.1	118.4	18.1	15.8	145.1	17.1	8.8	8.1
64k-4	20.1	8.7	117.2	17.8	15.2	144.1	21.0	12.1	11.5

Figure 10.13: GFTT prediction measurements; top in left to right order: L1D E_{MSE}, L1I E_{MSE}, LL E_{MSE}; bottom in left-to-right order: L1D E_{REP}, L1I E_{REP}, LL E_{REP}, green is better, red is worse

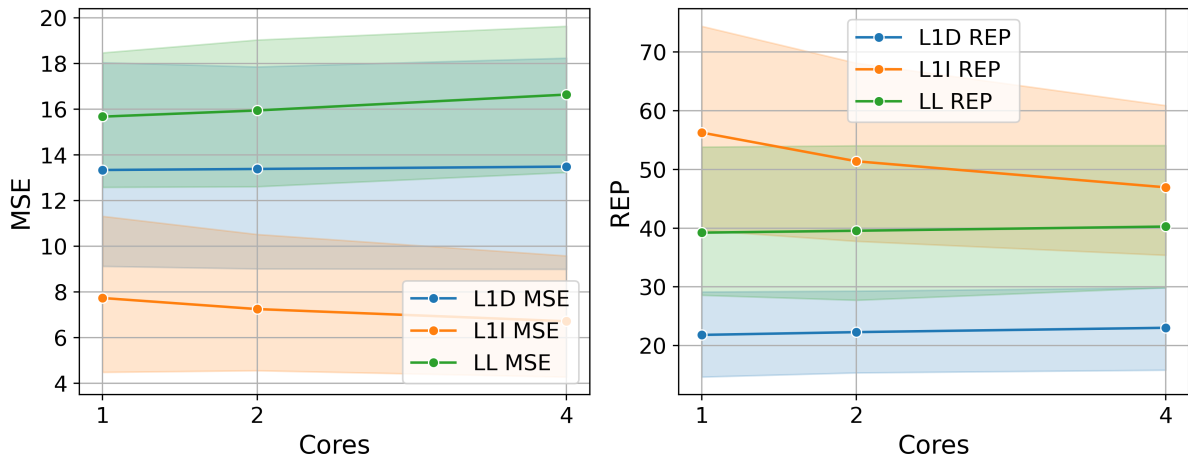


Figure 10.14: Sysbench CPU Benchmark Error plot per core count

our tests. Although FileIO and Memory benchmarks exhibit significant deviations at 2k for the L1D cache size and at 4k for the LL cache size, these findings highlight areas where the model could be improved - some improvements are suggested in section 10.8. This may indicate that our current model, in its present form, does not fully capture the cache dynamics for these particular types of programs. Alternatively, there may be unique characteristics in these workloads sensitive to parameters beyond those explored in our study (Table 10.1) that the model has yet to account for.

All of the final numerical results of the executions may be observed for review purposes in the appendix, Tables 10.2 to 10.5.

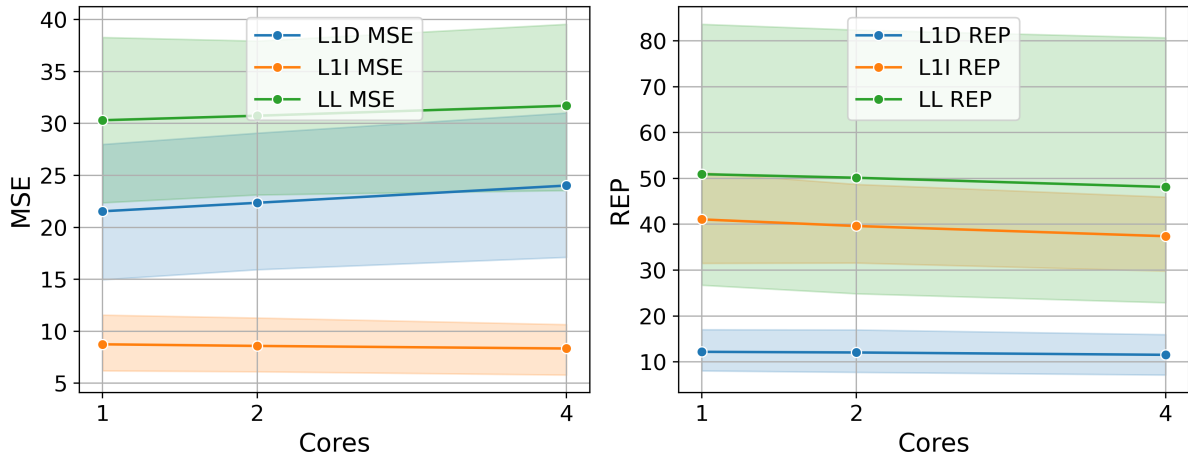


Figure 10.15: Sysbench Memory Benchmark Error plot per core count

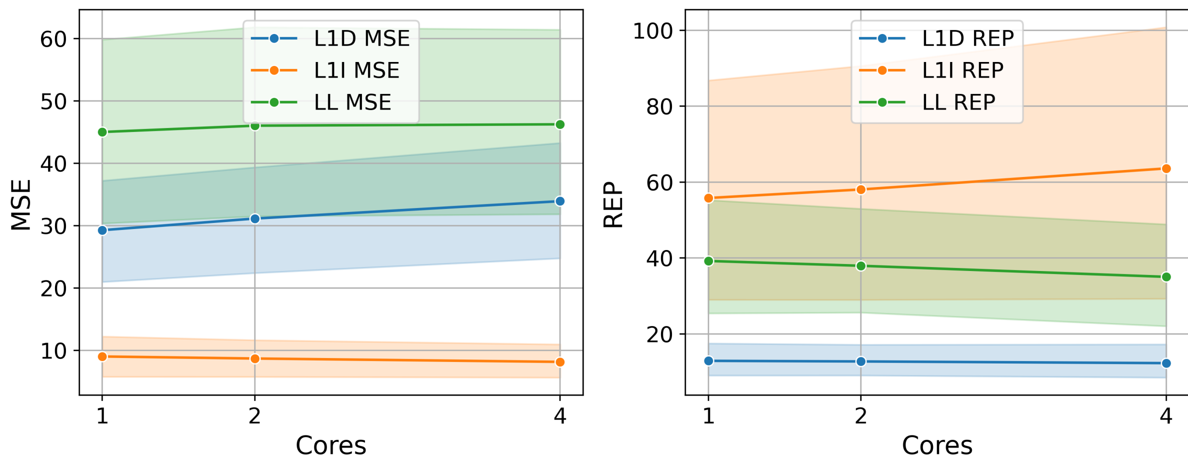


Figure 10.16: Sysbench FileIO Benchmark Error plot per core count

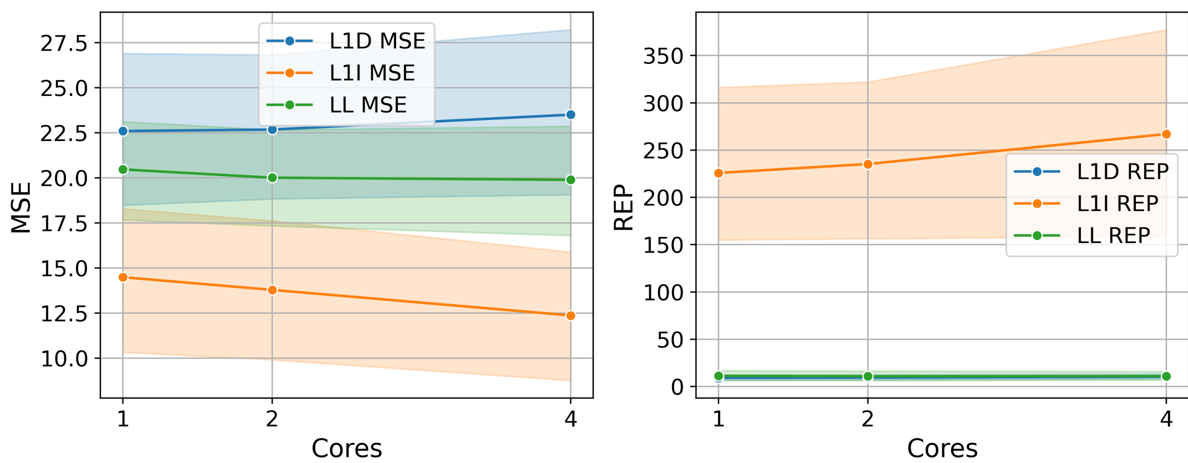


Figure 10.17: GFTT Error plot per core count

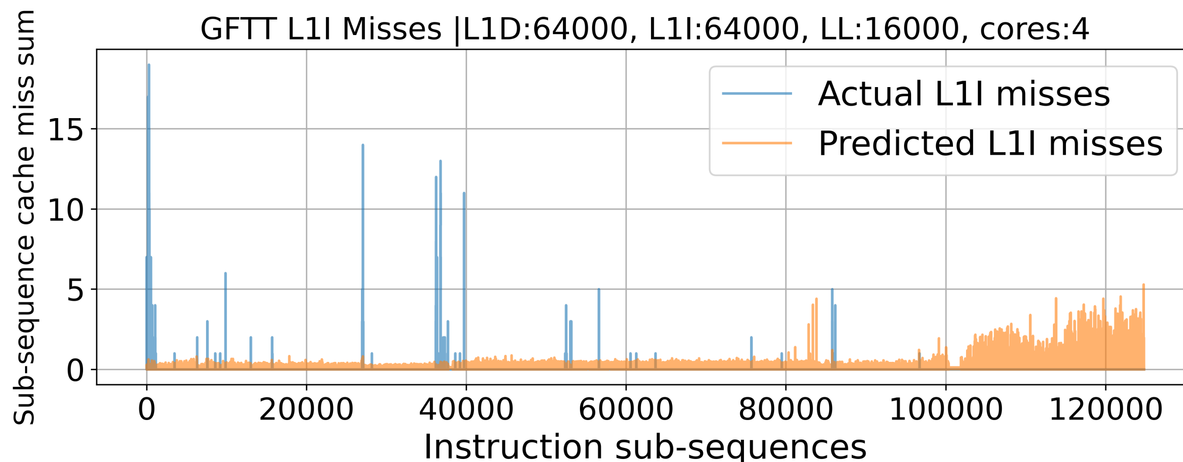


Figure 10.18: GFTT, worst execution, L1I cache

10.6 Discussion

Our machine learning-based cache simulation model demonstrated consistent performance across the CPU, Memory, FileIO and GFTT benchmarks, with each completing 25 million instructions in approximately 45 seconds, whereas the duration was approximately 22 seconds on the i7-13700HX platform using DynamoRio. However, our model provides the premises for a critical advantage: the ability to execute in parallel multiple sequential subsets of a program trace. This capability allows for distributed processing across multiple platforms, with the potential of significantly reducing the overall simulation time compared to DynamoRIO.

Our assumption above is based on the following. Each execution trace was divided into eight parts, and the model was run and timed on each part, with the goal of analyzing the predictive power of the model on subsections of the program trace running on multiple GPUs. On average, the execution time of each section was 17.35 seconds with $\sigma = 2.14$ seconds, producing, on average, a **21% time reduction** to our model when compared to running the simulation through Cachesim.

While our model excels in scenarios with predictable data access patterns, particularly in CPU workloads, it encounters challenges with more complex I/O operations, as reflected in the negative R^2 values observed in the FileIO benchmark at smaller L1D cache sizes. This may be due to several contributing factors, including synchronous disk operations, which are known to increase memory latency and cause cache inefficiencies, leading to more frequent cache misses. Background I/O processes, such as fsync [2], can further exacerbate cache pressure, resulting

[2]fsync is a system call that forces a file's in-memory data to be written to disk, ensuring data integrity in

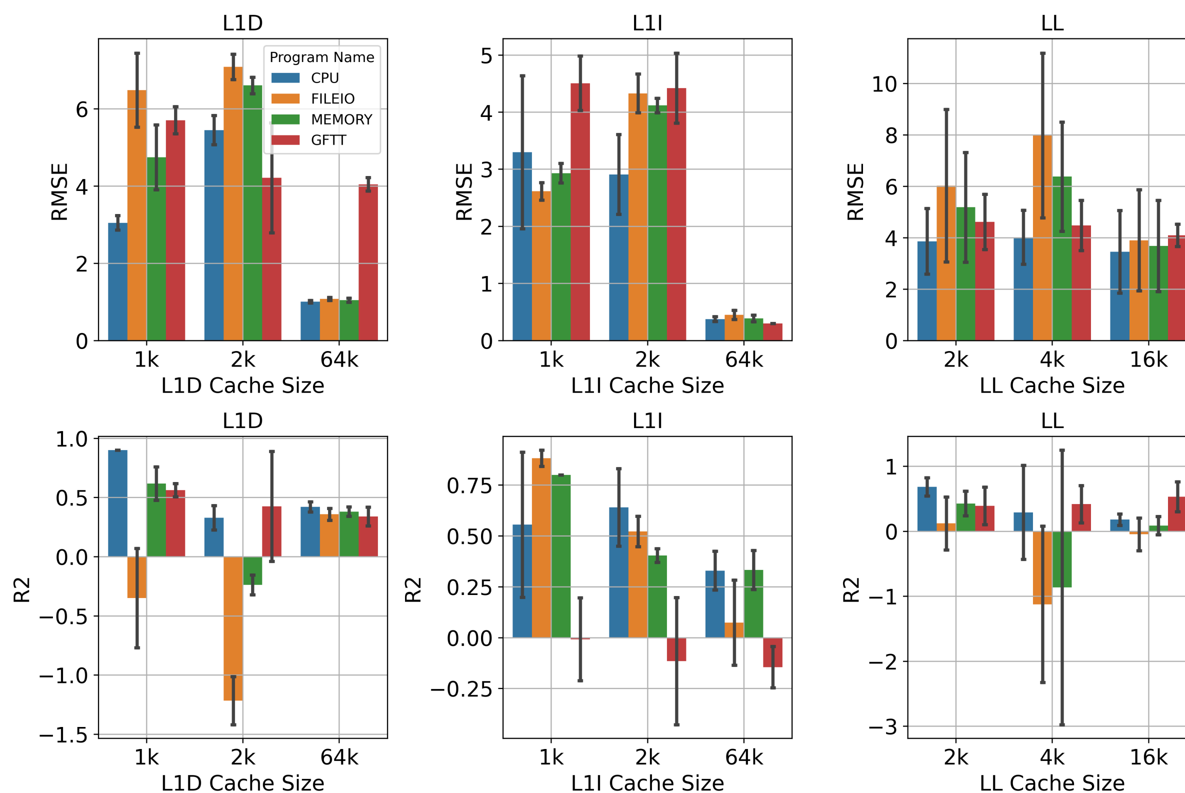


Figure 10.19: E_{RMSE}/R^2 analysis of the programs used in the model testing, cache E_{RMSE}/R^2 against respective cache sizes (an R^2 of 1.0 is ideal), standard deviation for the confidence intervals

in additional misses. Frequent and diverse file accesses can lead to sub-optimal data access patterns, increasing cache misses and making the modeling process more complex [29].

Despite these challenges, the flexibility of the model and the potential for parallelization represent significant advantages. Unlike DynamoRIO, which requires running the entire program for effective simulation, our approach allows for faster, more scalable simulations. This work represents a proof-of-concept and a first step towards a more ambitious model. Future work should focus on enhancing the model's accuracy across varied and complex workloads, particularly by improving its handling of edge cases and its generalization capabilities. These improvements could make the model a more robust tool for simulated cache performance across a wider range of applications.

case of a system crash. However, this operation can introduce significant performance overhead, especially in I/O-intensive workloads, due to the additional time required for disk synchronization [28].

10.7 Related Work

The predictive capabilities of LSTM-based models for cache miss occurrences on instrumentation traces have previously been investigated, as demonstrated by R. Jha et al. [30], validating the feasibility of employing LSTMs to anticipate cache miss patterns derived from program behaviors. This investigation underpins our own predictive modeling endeavors, yet our approach uniquely emphasizes the variability of cache sizes alongside the programs being analyzed. Various successful integrations of LSTM architectures into prefetchers aimed at mitigating cache misses - rather than simply modeling their distribution - have been documented, exemplified by J. Rogers [31] and Y. Zeng et al. [32]. For simulation of program performance using sequential deep learning, the Ithemal tool [6] epitomizes a sophisticated LSTM-driven technique to forecast instruction block latency; it operates, though, under the presumption of negligible cache misses within the program trace, contrasting with the focus of our method.

An alternative approach to latency prediction, predicated on parallelization rather than sequential modeling, is detailed by Li et al. [7] with the SimNet temporal convolutional network framework. Further work by Li et al. [33] showcases a higher-level simulation framework called PerfVec that is demonstrated in an instruction latency reduction objective against different combinations of cache sizes.

The work of S. Pandey et al. [34] suggests optimizations of SimNet and Ithemal to reduce the costs of moving datasets from the CPU to the GPU for overhead reduction purposes, and the most very recent work by S. Pandey et al. [8] showcases an effort in the direction of a microarchitecture simulator and dataset gathering approach that can be transferred between architectures differing by certain cache size and CPU pipeline characteristics. However, this tool focuses on the forecasting of metrics including cache misses on the granularity level of a single instruction, whereas our tool treats instruction sub-sequences in a statistical manner, which gives the user the choice between more granularity (fewer instructions per sub-sequence, but slower) or more speed (more instructions per sub-sequence, but less focus on individual instructions).

Additionally, our approach gives the user flexibility in terms of providing estimates for hypothetical but possible cache and multicore/single-core architectures, whereas the availability of combinations in the referenced work is constrained by the microarchitecture loaded in gem5. Moreover, our work also implements the ability of introducing higher core counts and the corresponding L1D/L1I caches, as well as the corresponding analysis that can be attained from

running the model.

The robust and frequently updated software instrumentation tool, DynamoRIO, is a preferred solution among contemporary computer science researchers, as evidenced by [1, 35]. We utilize DynamoRIO not solely for its prominence and ongoing advancements, but also for its open-source status, user-friendliness, and seamless integration with our tool for capturing program execution traces into its infrastructure.

10.8 Conclusion and Future work

This work introduces a new method for modeling cache performance tailored to specific applications using a distributable deep learning model.

We have shown that our approach not only mimics traditional simulators but also replicates results across multiple architectures. Our model handles complex Sysbench benchmarks and GFTT algorithm implementations with remarkable accuracy, even when faced with unfamiliar cache sizes and core counts. Although it tends to predict higher cache miss rates, it is excellent at pinpointing high-activity sections of program execution.

Though DynamoRIO outpaces our model on a single machine, our model has the potential to outperform it through parallelization: just three to four concurrent GPU iterations would suffice.

While relying on tools that provide hardware-specific traces, our approach itself is hardware-agnostic.

We have also provided the initial steps towards extending our approach on individual cores of multicore architectures.

Future work. We are set on fine-tuning our model on the present architecture, such as employing positional encoding for the tokens. We will also focus on minimizing cache-miss discrepancies during training by honing in on specific application benchmarks rather than broad benchmark sets. In addition, we plan to enhance the model architecture with batch normalization and regularization techniques to push performance further.

Extracting more features from Cachesim and breaking down instructions more granularly are on the agenda, as is expanding our approach using advanced simulators like gem5 and ZSim for richer data, for example extracting used registers, individual instruction latencies, branch predictor types and branch predictions and mispredictions. Richer trace data may prove

to be a key component in tackling intense workloads that depend heavily on additional information about the executed instructions and the underlying access patterns, such as the FileIO benchmark where our approach faced a significant challenge, as was shown in section 11.7. In addition to this, we will explore features of independent cores with regard to cache modeling.

We are also exploring different recurrent architectures, such as temporal convolutional networks and ga ted recurrent units, to boost predictive accuracy and efficiency. Crucially, our model is non-fixed; by feeding architectures as features into the model, we can effectively compare various CPU characteristics. Our ultimate goal? Develop a flexible ML simulation framework that holistically models hardware for specialized software, allowing direct comparisons across diverse CPU configurations. The challenges are significant, but so are the opportunities, and we are committed to making substantial contributions now and in the future.

Acknowledgement

The author's work was partially supported by the Swedish Knowledge Foundation via the project PerFlex - *Performant and Flexible digital Systems through Verifiable Artificial Intelligence*, grant nr. 20220033.

Bibliography

[1] Derek Bruening, Qin Zhao, and Saman Amarasinghe. "Transparent dynamic instrumentation". In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*. VEE '12: ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (London England, UK). New York, NY, USA: ACM, Mar. 3, 2012.

[2] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07: ACM SIGPLAN Conference on Programming Language Design and Implementation (San Diego California USA). PLDI '07. New York, NY, USA: ACM, June 10, 2007, pp. 89–100.

[3] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation". en. In: *SIGPLAN Not.* 40 (6 June 12, 2005), pp. 190–200.

[4] Fabrice Bellard. "QEMU, a fast and portable dynamic translator". In: ATEC '05 (Apr. 10, 2005), pp. 41–46.

[5] Andreas Sandberg, Nikos Nikoleris, Trevor E Carlson, Erik Hagersten, Stefanos Kaxiras, and David Black-Schaffer. "Full speed ahead: Detailed architectural simulation at near-native speed". In: *2015 IEEE International Symposium on Workload Characterization*. 2015 IEEE International Symposium on Workload Characterization (IISWC) (Atlanta, GA, USA). IEEE, Oct. 2015, pp. 183–192.

[6] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks". In: *arXiv [cs.DC]* (Aug. 20, 2018), pp. 4505–4515.

[7] Lingda Li, Santosh Pandey, Thomas Flynn, Hang Liu, Noel Wheeler, and Adolfy Hoisie. “SimNet: Accurate and high-performance computer architecture simulation using deep learning”. en. In: *Proc. ACM Meas. Anal. Comput. Syst.* 6 (2 May 26, 2022), pp. 1–24. (Visited on 09/10/2025).

[8] Santosh Pandey, Amir Yazdanbakhsh, and Hang Liu. “TAO: Re-thinking DL-based microarchitecture simulation”. en. In: *Proc. ACM Meas. Anal. Comput. Syst.* 8 (2 May 21, 2024), pp. 1–25.

[9] S Hochreiter and J Schmidhuber. “Long short-term memory”. en. In: *Neural Comput.* 9 (8 Nov. 15, 1997), pp. 1735–1780.

[10] Swadhesh Kumar and P K Singh. “An overview of modern cache memory and performance analysis of replacement policies”. In: *2016 IEEE International Conference on Engineering and Technology (ICETECH)*. 2016 IEEE International Conference on Engineering and Technology (ICETECH) (Coimbatore, India). Vol. 17. IEEE, Mar. 2016, pp. 210–214.

[11] Daniel Etiemble. “45-year CPU evolution: one law and two equations”. In: *arXiv [cs.AR]* (Mar. 1, 2018).

[12] J L Hennessy and D A Patterson. *Computer Architecture: A Quantitative Approach*. San Francisco, CA: Morgan Kaufmann, 2011.

[13] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. “When prefetching works, when it doesn’t, and why”. en. In: *ACM Trans. Archit. Code Optim.* 9 (1 Mar. 2012), pp. 1–29.

[14] H Kwak, B Lee, A R Hurson, Suk-Han Yoon, and Woo-Jong Hahn. “Effects of multithreading on cache performance”. In: *IEEE Trans. Comput.* 48 (2 1999), pp. 176–184.

[15] Carlos Carvalho. “The gap between processor and memory speeds”. In: 5000 (2002), p. 15000.

[16] Anastasis Kratsios. “The universal approximation property: Characterization, construction, representation, and existence”. en. In: *Ann. Math. Artif. Intell.* 89 (5-6 June 2021), pp. 435–469.

[17] Takato Nishijima. “Universal Approximation Theorem for Neural Networks”. In: *arXiv [cs.LG]* (Feb. 19, 2021).

[18] Laith Alzubaidi, Jinglan Zhang, Amjad J Humaidi, Ayad Al-Dujaili, Ye Duan, Omran Al-Shamma, J Santamaría, Mohammed A Fadhel, Muthana Al-Amidie, and Laith Farhan. "Review of deep learning: concepts, CNN architectures, challenges, applications, future directions". en. In: *J. Big Data* 8 (1 Mar. 31, 2021), p. 53.

[19] Alex Sherstinsky. "Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network". en. In: *Physica D* 404 (132306 Mar. 2020), p. 132306.

[20] The pandas development team. *pandas-dev/pandas: Pandas*. Comp. software. Version latest. 2026.

[21] *Tokenization algorithms · Hugging Face*. (Visited on 03/25/2026).

[22] Joel Castaño, Silverio Martínez-Fernández, Xavier Franch, and Justus Bogner. "Analyzing the evolution and maintenance of ML models on hugging face". In: *Proceedings of the 21st International Conference on Mining Software Repositories*. MSR '24: 21st International Conference on Mining Software Repositories (Lisbon Portugal). Vol. 94. MSR '24. New York, NY, USA: ACM, Apr. 15, 2024, pp. 607–618.

[23] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An imperative style, high-performance deep learning library". In: *arXiv [cs.LG]* (Dec. 3, 2019).

[24] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. "Optuna: A Next-generation Hyperparameter Optimization Framework". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019.

[25] Alexey Kopytov. "Sysbench: a system performance benchmark". In: *http://sysbench.sourceforge. net/* (2004).

[26] Sharifeh Yaghoobi. "LEVERAGING MACHINE LEARNING FOR FAST PERFORMANCE PREDICTION FOR INDUSTRIAL SYSTEMS : Data-Driven Cache Simulator". MA thesis. Mälardalen University, School of Innovation, Design and Engineering, 2024.

[27] Aryan Jadon, Avinash Patil, and Shruti Jadon. “A comprehensive survey of regression based loss functions for time Series Forecasting”. In: *arXiv [cs.LG]* (Nov. 5, 2022).

[28] Chang-Gyu Lee, Hyunki Byun, Sunghyun Noh, Hyeongu Kang, and Youngjae Kim. “Write optimization of log-structured flash file system for parallel I/O on manycore servers”. In: *Proceedings of the 12th ACM International Conference on Systems and Storage*. SYSTOR ’19: The 12th ACM International Systems and Storage Conference (Haifa Israel). SYSTOR ’19. New York, NY, USA: ACM, May 22, 2019, pp. 21–32.

[29] Jeongha Lee and Hyokyung Bahn. “File access characteristics of deep learning workloads and cache-friendly data management”. In: *2023 10th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*. 2023 10th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI) (Palembang, Indonesia). IEEE, Sept. 20, 2023, pp. 328–331.

[30] Rishikesh Jha, Arjun Karuvally, Saket Tiwari, and J Eliot B Moss. “Cache Miss Rate Predictability via Neural Networks”. In: (2018).

[31] Joseph Rogers. “Effects of an LSTM Composite Prefetcher”. In: (2019).

[32] Yuan Zeng and Xiaochen Guo. “Long short term memory based hardware prefetcher: a case study”. In: *Proceedings of the International Symposium on Memory Systems*. MEMSYS 2017: The International Symposium on Memory Systems, 2017 (Alexandria Virginia). New York, NY, USA: ACM, Oct. 2, 2017, pp. 305–311.

[33] Lingda Li, Thomas Flynn, and Adolfy Hoisie. “Learning generalizable program and architecture representations for performance modeling”. en. In: *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. SC24: International Conference for High Performance Computing, Networking, Storage and Analysis (Atlanta, GA, USA). IEEE, Nov. 17, 2024, pp. 1–15. (Visited on 09/10/2025).

[34] Santosh Pandey, Lingda Li, Thomas Flynn, Adolfy Hoisie, and Hang Liu. “Scalable Deep Learning-Based Microarchitecture Simulation on GPUs”. In: *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (Dallas, TX, USA). IEEE, Nov. 2022, pp. 1–15.

[35] Yixiao Yang, Chen Gao, Zhiqi Li, Yifan Wang, and Rui Wang. “Binary level concolic execution on windows with rich instrumentation based taint analysis”. en. In: *Lecture Notes in Computer Science*. Lecture Notes in Computer Science. Singapore: Springer Nature Singapore, 2024, pp. 351–367.

Appendix - PyTorch model code

```
class CombinedLSTMModel(nn.Module):
    def __init__(
        self,
        token_vocab_size, # characteristic of the tokenizer
        token_embedding_dim, # number of tokens per instruction
        access_feature_size, # number of trace features
        hidden_dim, # matrix size of the hidden dimension of
         ↪  the LSTM layer
        output_dim, # number of outputs of our model
        num_layers, # number of consecutive LSTM layers
        dropout, # dropout rate between the LSTM layers
    ):
        super(CombinedLSTMModel, self).__init__()
        self.token_embedding = nn.Embedding(token_vocab_size,
         ↪  token_embedding_dim)
        # Input size must be the sum of flattened token
         ↪  embedding size
        # and access feature size
        self.lstm = nn.LSTM(
            input_size=token_embedding_dim**2 +
             ↪  access_feature_size,
            hidden_size=hidden_dim,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout,
        )
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, token_features, access_features):
        # Embed the token features
        embedded_tokens = self.token_embedding(token_features)
```

```
# Assume access_features is already of the appropriate
↪  size and
# requires no embedding
# LSTM can handle variable sequence lengths if
↪  necessary, but here we
# focus on batch size flexibility
elongated_embeddings = embedded_tokens.view(
    token_features.size(0), token_features.size(1), -1
)
combined_features = torch.cat(
    (elongated_embeddings, access_features), dim=2
)  # Concatenate along the feature dimension
lstm_out, _ = self.lstm(combined_features)
# Using the last time step's output
output = self.fc(lstm_out[:, -1, :])
return output
```

Appendix - Execution Results

Table 10.2: Table of results for Sysbench CPU Benchmark Model executions

				L1D		L1I		LL		L1D				L1I				LL				Execution time per block							
L1D Size (bytes)	L1I Size (bytes)	LL Size (bytes)	Cores	Total Actual	Total Predicted	Total Actual	Total Predicted	Total Actual	Total Predicted	MSE	RMSE	R2	Abs. Err. Percent	MSE	RMSE	R2	Abs. Err. Percent	MSE	RMSE	R2	Abs. Err. Percent	t_1 s	t_2 s	t_3 s	t_4 s	t_5 s	t_6 s	t_7 s	t_8 s
1k	1k	2k	1	929,857	970,716	343,013	428,490	1,056,247	970,716	10.5	3.2	0.9	4.4	5.6	2.4	0.8	24.9	25.4	5.0	0.8	20.0	17	17	15	14	16	17	19	16
1k	1k	2k	2	929,842	981,834	343,007	426,989	1,056,225	981,834	10.1	3.2	0.9	5.6	5.5	2.4	0.8	24.5	26.1	5.1	0.8	21.3	18	16	16	18	17	16	17	18
1k	1k	4k	1	929,857	945,486	343,013	429,432	952,906	945,486	12.0	3.5	0.9	1.7	5.6	2.4	0.8	25.2	18.9	4.3	0.8	4.4	18	16	14	19	18	16	18	16
1k	1k	4k	2	929,842	955,566	343,007	427,865	952,871	955,566	11.6	3.4	0.9	2.8	5.6	2.4	0.8	24.7	19.7	4.4	0.8	8.1	17	16	16	18	18	17	17	18
1k	1k	2k	4	929,794	999,360	343,014	424,442	1,056,170	999,360	9.5	3.1	0.9	7.5	5.4	2.3	0.8	23.7	27.0	5.2	0.8	23.1	18	14	15	17	18	17	18	16
1k	1k	4k	4	929,794	974,292	343,014	424,945	952,791	974,292	11.1	3.3	0.9	4.8	5.4	2.3	0.8	23.9	22.8	4.8	0.8	17.1	17	13	13	16	16	14	18	18
1k	1k	16k	1	929,857	1,058,499	343,013	435,221	386,988	1,058,499	8.0	2.8	0.9	13.8	5.9	2.4	0.8	26.9	25.3	5.0	0.1	55.7	17	16	14	16	16	17	17	17
1k	1k	16k	2	929,842	1,061,302	343,007	433,769	386,985	1,061,302	8.1	2.8	0.9	14.1	5.9	2.4	0.8	26.5	25.5	5.0	0.1	55.0	17	14	14	16	17	16	16	18
1k	1k	16k	4	929,794	1,063,067	343,014	431,174	386,969	1,063,067	8.4	2.9	0.9	14.3	5.8	2.4	0.8	25.7	25.7	5.1	0.1	52.7	18	14	17	16	19	18	18	16
1k	2k	2k	1	929,857	1,006,300	314,724	402,809	1,027,114	1,006,300	8.9	3.0	0.9	8.2	5.6	2.4	0.8	28.0	24.4	4.9	0.8	23.7	17	16	18	18	14	17	18	16
1k	2k	2k	2	929,842	1,013,684	314,737	399,673	1,027,116	1,013,684	8.7	3.0	0.9	9.0	5.6	2.4	0.8	27.0	24.6	5.0	0.8	24.5	16	13	16	16	17	18	18	16
1k	2k	4k	1	929,857	980,388	314,724	402,579	932,916	980,388	10.7	3.3	0.9	5.4	5.8	2.4	0.8	27.9	17.0	4.1	0.8	6.9	16	17	17	15	16	14	18	17
1k	2k	4k	2	929,842	987,532	314,737	399,427	932,909	987,532	10.5	3.2	0.9	6.2	5.7	2.4	0.8	26.9	18.1	4.3	0.8	10.3	16	17	16	14	17	19	18	18
1k	2k	16k	1	929,857	1,072,163	314,724	409,211	382,912	1,072,163	8.0	2.8	0.9	15.3	6.3	2.5	0.7	30.0	24.5	4.9	0.1	54.4	16	15	17	17	16	14	18	20
1k	2k	16k	2	929,842	1,073,120	314,737	406,698	382,911	1,073,120	8.1	2.8	0.9	15.4	6.2	2.5	0.7	29.2	24.4	4.9	0.1	53.1	16	16	17	17	18	19	18	18
1k	2k	2k	4	929,794	1,024,456	314,774	390,779	1,027,090	1,024,456	8.5	2.9	0.9	10.2	5.6	2.4	0.8	24.1	25.0	5.0	0.8	25.1	16	17	13	17	17	17	15	17
1k	2k	4k	4	929,794	1,001,137	314,774	391,069	932,855	1,001,137	10.2	3.2	0.9	7.7	5.7	2.4	0.8	24.2	21.7	4.7	0.8	18.2	16	14	16	19	17	18	18	18
1k	2k	16k	4	929,794	1,072,981	314,774	400,051	382,891	1,072,981	8.3	2.9	0.9	15.4	6.2	2.5	0.7	27.1	24.1	4.9	0.1	49.4	18	15	16	16	14	18	16	18
1k	64k	2k	1	929,857	1,033,081	2,911	4,557	666,006	1,033,081	8.9	3.0	0.9	11.1	0.1	0.3	0.4	56.5	16.4	4.0	0.6	31.0	16	14	15	16	19	16	17	19
1k	64k	2k	2	929,842	1,029,859	3,167	4,700	666,269	1,029,859	9.0	3.0	0.9	10.8	0.1	0.4	0.4	48.4	16.5	4.1	0.6	31.1	16	13	14	14	16	17	18	19
1k	64k	4k	1	929,857	1,050,885	2,911	5,009	572,400	1,050,885	8.9	3.0	0.9	13.0	0.1	0.3	0.4	72.1	19.6	4.4	0.5	49.2	18	16	14	17	19	19	19	19
1k	64k	4k	2	929,842	1,047,054	3,167	5,089	572,675	1,047,054	9.0	3.0	0.9	12.6	0.1	0.4	0.4	60.7	20.0	4.5	0.5	49.9	16	15	13	17	15	17	19	17
1k	64k	16k	1	929,857	1,129,502	2,911	5,718	230,767	1,129,502	8.9	3.0	0.9	21.5	0.1	0.3	0.4	96.4	11.7	3.4	0.2	60.8	18	14	14	18	20	17	15	19
1k	64k	16k	2	929,842	1,128,150	3,167	5,889	231,036	1,128,150	8.9	3.0	0.9	21.3	0.1	0.4	0.3	85.9	11.8	3.4	0.2	57.8	17	13	14	15	18	16	18	19
1k	64k	2k	4	929,794	1,021,430	3,534	5,174	666,598	1,021,430	9.4	3.1	0.9	9.9	0.2	0.4	0.4	46.4	16.7	4.1	0.6	30.8	18	16	13	18	16	17	16	17

Table 10.2 continued from previous page

1k	64k	4k	4	929,794	1,037,912	3,534	5,502	573,007	1,037,912	9.3	3.0	0.9	11.6	0.2	0.4	0.4	55.7	20.3	4.5	0.5	50.4	16	15	15	17	16	18	16	18
1k	64k	16k	4	929,794	1,125,346	3,534	6,401	231,279	1,125,346	9.1	3.0	0.9	21.0	0.2	0.4	0.4	81.1	12.0	3.5	0.2	51.4	17	16	14	19	16	18	17	17
2k	1k	2k	1	653,093	859,704	343,013	427,986	989,786	859,704	25.8	5.1	0.4	31.6	5.6	2.4	0.8	24.8	22.3	4.7	0.8	18.7	18	16	15	16	17	18	18	18
2k	1k	2k	2	653,059	874,546	343,007	426,834	989,742	874,546	26.4	5.1	0.4	33.9	5.5	2.4	0.8	24.4	23.4	4.8	0.8	20.4	19	19	15	16	16	17	14	15
2k	1k	4k	1	653,093	825,456	343,013	428,950	936,508	825,456	24.2	4.9	0.5	26.4	5.6	2.4	0.8	25.1	19.0	4.4	0.8	0.4	19	13	17	16	17	18	15	18
2k	1k	4k	2	653,059	838,541	343,007	427,779	936,463	838,541	24.6	5.0	0.5	28.4	5.6	2.4	0.8	24.7	19.2	4.4	0.8	4.3	17	19	13	17	18	16	18	18
2k	1k	16k	1	653,093	978,540	343,013	436,372	393,565	978,540	31.0	5.6	0.3	49.8	5.9	2.4	0.8	27.2	25.6	5.1	0.1	52.9	16	15	13	17	17	16	18	16
2k	1k	16k	2	653,059	985,513	343,007	435,292	393,554	985,513	31.1	5.6	0.3	50.9	5.9	2.4	0.8	26.9	26.0	5.1	0.1	52.9	19	15	15	16	18	16	15	18
2k	2k	2k	1	653,093	907,625	314,724	404,139	964,751	907,625	26.8	5.2	0.4	39.0	5.8	2.4	0.8	28.4	23.4	4.8	0.8	23.6	13	11	10	11	11	12	16	17
2k	2k	2k	2	653,059	918,673	314,737	401,728	964,728	918,673	27.3	5.2	0.4	40.7	5.7	2.4	0.8	27.6	24.2	4.9	0.7	24.8	12	10	10	11	11	12	12	12
2k	2k	4k	1	653,093	872,860	314,724	404,308	921,435	872,860	25.2	5.0	0.4	33.7	6.0	2.4	0.8	28.5	17.7	4.2	0.8	4.2	19	15	13	16	17	17	15	18
2k	2k	4k	2	653,059	885,091	314,737	401,539	921,412	885,091	25.9	5.1	0.4	35.5	5.9	2.4	0.8	27.6	18.3	4.3	0.8	7.8	15	16	16	16	17	15	17	15
2k	2k	16k	1	653,093	1,004,574	314,724	411,536	389,470	1,004,574	32.2	5.7	0.3	53.8	6.3	2.5	0.7	30.8	25.1	5.0	0.1	53.1	21	18	14	15	14	17	14	17
2k	2k	16k	2	653,059	1,008,908	314,737	409,720	389,464	1,008,908	32.3	5.7	0.3	54.5	6.3	2.5	0.7	30.2	25.2	5.0	0.1	52.5	20	18	16	16	14	17	17	15
2k	64k	2k	1	653,093	958,709	2,911	4,296	640,176	958,709	30.2	5.5	0.3	46.8	0.1	0.3	0.4	47.6	15.3	3.9	0.7	27.6	17	15	13	16	17	17	18	17
2k	64k	2k	2	653,059	953,539	3,167	4,416	640,406	953,539	29.9	5.5	0.3	46.0	0.1	0.4	0.4	39.4	15.3	3.9	0.7	27.3	20	13	16	16	16	15	17	18
2k	64k	4k	1	653,093	984,560	2,911	4,816	593,580	984,560	31.9	5.7	0.3	50.8	0.1	0.3	0.4	65.4	16.8	4.1	0.6	36.2	16	14	15	16	17	17	18	18
2k	64k	4k	2	653,059	978,740	3,167	4,893	593,832	978,740	31.6	5.6	0.3	49.9	0.1	0.4	0.4	54.5	16.8	4.1	0.6	36.4	16	14	16	17	17	17	15	18
2k	64k	16k	1	653,093	1,098,346	2,911	5,778	234,682	1,098,346	38.9	6.2	0.1	68.2	0.1	0.3	0.4	98.5	11.7	3.4	0.2	55.8	16	14	16	17	17	17	17	17
2k	64k	16k	2	653,059	1,096,890	3,167	5,934	234,953	1,096,890	39.0	6.2	0.1	68.0	0.1	0.4	0.4	87.4	11.8	3.4	0.2	52.8	16	16	16	17	18	19	18	17
2k	1k	2k	4	653,011	900,456	343,014	425,028	989,682	900,456	27.3	5.2	0.4	37.9	5.5	2.3	0.8	23.9	25.3	5.0	0.7	23.1	18	18	16	16	14	16	17	16
2k	1k	4k	4	653,011	868,687	343,014	425,663	936,333	868,687	25.9	5.1	0.4	33.0	5.5	2.3	0.8	24.1	21.6	4.6	0.8	13.6	17	16	13	17	19	16	18	18
2k	1k	16k	4	653,011	994,444	343,014	433,108	393,524	994,444	31.4	5.6	0.3	52.3	5.8	2.4	0.8	26.3	26.4	5.1	0.1	51.7	18	18	18	17	16	14	14	18
2k	2k	2k	4	653,011	936,983	314,774	394,140	964,697	936,983	28.0	5.3	0.4	43.5	5.8	2.4	0.8	25.2	25.4	5.0	0.7	26.1	12	11	11	11	12	12	12	12
2k	2k	4k	4	653,011	909,891	314,774	394,228	921,307	909,891	27.0	5.2	0.4	39.3	5.9	2.4	0.8	25.2	21.2	4.6	0.8	15.9	17	15	14	19	14	17	18	18
2k	2k	16k	4	653,011	1,011,652	314,774	404,442	389,432	1,011,652	32.4	5.7	0.3	54.9	6.3	2.5	0.7	28.5	25.1	5.0	0.1	49.7	18	16	15	16	18	17	15	16
2k	64k	2k	4	653,011	940,205	3,534	4,920	640,711	940,205	29.3	5.4	0.3	44.0	0.1	0.4	0.4	39.2	15.3	3.9	0.7	26.3	18	18	15	16	17	18	18	19
2k	64k	4k	4	653,011	962,839	3,534	5,285	594,115	962,839	30.7	5.5	0.3	47.4	0.2	0.4	0.4	49.5	16.7	4.1	0.6	35.7	17	16	16	17	14	15	17	18
2k	64k	16k	4	653,011	1,093,036	3,534	6,432	235,184	1,093,036	39.1	6.3	0.1	67.4	0.2	0.4	0.4	82.0	12.0	3.5	0.2	45.8	17	13	17	16	14	16	18	19

Table 10.2 continued from previous page

64k	1k	2k	1	75,338	72,780	343,013	864,998	399,109	72,780	1.0	1.0	0.4	3.4	30.3	5.5	-0.1	152.2	10.6	3.3	0.6	53.0	19	15	15	19	15	18	18	19
64k	1k	2k	2	75,448	72,227	343,007	825,070	399,217	72,227	1.0	1.0	0.4	4.3	27.9	5.3	0.0	140.5	10.5	3.2	0.6	50.8	16	14	16	15	16	16	16	18
64k	1k	4k	1	75,338	73,601	343,013	842,304	191,035	73,601	1.0	1.0	0.5	2.3	28.9	5.4	0.0	145.6	19.2	4.4	-1.0	145.3	17	13	14	16	16	15	17	20
64k	1k	4k	2	75,448	73,258	343,007	800,436	191,161	73,258	1.0	1.0	0.4	2.9	26.3	5.1	0.1	133.4	19.6	4.4	-1.1	149.0	18	14	13	16	16	16	19	18
64k	1k	16k	1	75,338	71,651	343,013	841,378	90,556	71,651	0.9	1.0	0.5	4.9	28.6	5.4	0.0	145.3	2.5	1.6	0.2	26.1	17	14	14	16	18	15	18	17
64k	1k	16k	2	75,448	71,401	343,007	797,472	90,703	71,401	1.0	1.0	0.5	5.4	25.8	5.1	0.1	132.5	2.5	1.6	0.2	24.9	20	13	14	15	15	18	16	19
64k	2k	2k	1	75,338	74,126	314,724	646,284	385,200	74,126	1.0	1.0	0.4	1.6	17.5	4.2	0.3	105.3	6.6	2.6	0.8	32.3	19	13	16	16	19	15	16	19
64k	2k	2k	2	75,448	74,230	314,737	609,430	385,357	74,230	1.0	1.0	0.4	1.6	15.7	4.0	0.4	93.6	6.6	2.6	0.8	30.6	16	13	13	14	16	17	18	19
64k	2k	4k	1	75,338	74,811	314,724	624,193	190,097	74,811	1.0	1.0	0.4	0.7	16.4	4.0	0.3	98.3	18.1	4.3	-0.9	131.9	19	14	16	17	16	15	16	19
64k	2k	4k	2	75,448	75,087	314,737	590,308	190,228	75,087	1.0	1.0	0.4	0.5	14.7	3.8	0.4	87.6	18.1	4.3	-0.9	131.7	17	13	14	17	15	18	17	18
64k	2k	16k	1	75,338	73,604	314,724	618,119	90,644	73,604	1.0	1.0	0.5	2.3	16.2	4.0	0.3	96.4	2.3	1.5	0.3	22.3	16	13	16	17	14	17	18	16
64k	2k	16k	2	75,448	73,635	314,737	589,930	90,788	73,635	1.0	1.0	0.5	2.4	14.7	3.8	0.4	87.4	2.3	1.5	0.3	21.1	17	13	13	16	16	16	17	19
64k	64k	2k	1	75,338	57,844	2,911	2,921	77,144	57,844	1.1	1.0	0.4	23.2	0.1	0.4	0.2	0.4	1.3	1.1	0.4	21.7	17	17	14	16	16	16	20	20
64k	64k	2k	2	75,448	58,246	3,167	3,130	77,474	58,246	1.1	1.0	0.4	22.8	0.2	0.4	0.2	1.2	1.4	1.2	0.4	21.3	17	14	16	16	18	15	16	20
64k	64k	4k	1	75,338	56,796	2,911	2,870	74,068	56,796	1.1	1.0	0.4	24.6	0.1	0.4	0.2	1.4	1.3	1.1	0.4	20.7	17	14	16	16	15	18	15	16
64k	64k	4k	2	75,448	57,059	3,167	3,071	74,395	57,059	1.1	1.0	0.4	24.4	0.2	0.4	0.2	3.0	1.4	1.2	0.3	20.5	17	14	16	20	17	17	16	19
64k	64k	16k	1	75,338	51,295	2,911	2,625	69,030	51,295	1.1	1.0	0.4	31.9	0.1	0.4	0.2	9.8	1.3	1.1	0.4	27.3	17	16	14	15	17	16	17	19
64k	64k	16k	2	75,448	51,105	3,167	2,804	69,319	51,105	1.1	1.0	0.4	32.3	0.2	0.4	0.2	11.5	1.3	1.1	0.3	27.8	17	16	14	18	16	16	19	17
64k	1k	2k	4	75,561	71,770	343,014	767,735	399,338	71,770	1.0	1.0	0.4	5.0	24.6	5.0	0.1	123.8	10.5	3.2	0.6	47.5	17	14	17	16	15	15	18	16
64k	1k	4k	4	75,561	72,657	343,014	744,153	191,332	72,657	1.0	1.0	0.4	3.8	22.9	4.8	0.2	116.9	20.7	4.6	-1.2	159.4	19	13	15	17	15	19	19	17
64k	1k	16k	4	75,561	70,834	343,014	742,269	90,830	70,834	1.0	1.0	0.5	6.3	22.5	4.7	0.2	116.4	2.5	1.6	0.2	22.7	18	14	14	15	15	16	16	18
64k	2k	2k	4	75,561	74,496	314,774	580,450	385,492	74,496	1.0	1.0	0.4	1.4	14.2	3.8	0.4	84.4	6.7	2.6	0.8	29.6	16	16	14	17	14	19	17	16
64k	2k	4k	4	75,561	75,086	314,774	564,912	190,382	75,086	1.0	1.0	0.4	0.6	13.3	3.6	0.5	79.5	18.3	4.3	-0.9	133.7	17	13	14	16	15	17	19	18
64k	2k	16k	4	75,561	73,031	314,774	566,406	90,913	73,031	1.0	1.0	0.4	3.3	13.2	3.6	0.5	79.9	2.3	1.5	0.3	18.5	18	15	14	15	16	15	17	21
64k	64k	2k	4	75,561	58,703	3,534	3,645	77,942	58,703	1.1	1.1	0.4	22.3	0.2	0.4	0.2	3.1	1.4	1.2	0.3	20.7	17	14	14	17	15	16	18	19
64k	64k	4k	4	75,561	57,393	3,534	3,585	74,739	57,393	1.1	1.1	0.4	24.0	0.2	0.4	0.2	1.4	1.4	1.2	0.3	20.1	17	14	15	18	18	15	18	16
64k	64k	16k	4	75,561	50,418	3,534	3,316	69,582	50,418	1.1	1.1	0.4	33.3	0.2	0.4	0.2	6.2	1.3	1.2	0.3	28.9	17	16	15	17	16	18	18	19

Table 10.3: Table of results for Sysbench FileIO Benchmark Model executions

				L1D		L1I		LL		L1D				L1I				LL				Execution time per block							
L1D Size (bytes)	L1I Size (bytes)	LL Size (bytes)	Cores	Total Actual	Total Predicted	Total Actual	Total Predicted	Total Actual	Total Predicted	MSE	RMSE	R2	Abs. Err. Percent	MSE	RMSE	R2	Abs. Err. Percent	MSE	RMSE	R2	Abs. Err. Percent	t_1 s	t_2 s	t_3 s	t_4 s	t_5 s	t_6 s	t_7 s	t_8 s
1k	1k	2k	1	1,654,670	1,399,186	1,351,265	1,418,858	2,886,788	1,399,186	53.70	7.30	-0.60	15.40	6.30	2.50	0.90	5.00	71.00	8.40	0.20	9.10	19	16	18	17	17	18	18	19
1k	1k	2k	2	1,649,213	1,366,599	1,351,290	1,417,640	2,880,669	1,366,599	57.20	7.60	-0.80	17.10	6.30	2.50	0.90	4.90	76.00	8.70	0.20	10.00	19	16	18	17	17	17	18	17
1k	1k	4k	1	1,654,670	1,441,105	1,351,265	1,427,518	2,517,207	1,441,105	52.40	7.20	-0.60	12.90	6.40	2.50	0.90	5.60	115.50	10.70	-0.60	18.60	19	21	20	21	17	18	17	17
1k	1k	4k	2	1,649,213	1,399,176	1,351,290	1,424,298	2,509,735	1,399,176	57.30	7.60	-0.80	15.20	6.30	2.50	0.90	5.40	121.00	11.00	-0.70	17.50	20	16	18	20	17	17	18	19
1k	1k	2k	4	1,624,166	1,306,244	1,351,163	1,413,247	2,854,762	1,306,244	62.80	7.90	-1.10	19.60	6.20	2.50	0.90	4.60	86.20	9.30	0.00	11.60	19	17	17	18	17	18	18	19
1k	1k	4k	4	1,624,166	1,323,401	1,351,163	1,416,690	2,471,721	1,323,401	64.20	8.00	-1.10	18.50	6.40	2.50	0.90	4.80	124.00	11.10	-0.80	14.90	19	17	18	17	17	18	18	19
1k	1k	16k	1	1,654,670	1,833,876	1,351,265	1,431,797	465,345	1,833,876	29.70	5.40	0.10	10.80	6.60	2.60	0.90	6.00	38.20	6.20	-0.40	95.50	19	17	18	16	17	18	17	17
1k	1k	16k	2	1,649,213	1,778,815	1,351,290	1,432,972	466,087	1,778,815	33.90	5.80	0.00	7.90	6.50	2.50	0.90	6.00	37.20	6.10	-0.40	89.20	19	16	17	16	17	17	17	19
1k	1k	16k	4	1,624,166	1,648,160	1,351,163	1,430,321	467,819	1,648,160	44.20	6.70	-0.50	1.50	6.30	2.50	0.90	5.90	35.00	5.90	-0.30	75.20	19	17	17	17	16	16	17	17
1k	2k	2k	1	1,654,670	1,504,889	1,098,937	1,210,159	2,618,505	1,504,889	48.40	7.00	-0.50	9.10	23.40	4.80	0.40	10.10	87.50	9.40	-0.10	4.50	19	14	16	16	16	17	17	17
1k	2k	2k	2	1,649,213	1,477,612	1,099,776	1,218,090	2,613,402	1,477,612	51.60	7.20	-0.60	10.40	22.60	4.80	0.40	10.80	90.70	9.50	-0.20	5.20	18	15	17	17	17	16	17	18
1k	2k	4k	1	1,654,670	1,524,201	1,098,937	1,254,832	2,356,975	1,524,201	47.80	6.90	-0.50	7.90	21.40	4.60	0.50	14.20	104.20	10.20	-0.60	14.20	19	18	16	16	16	17	18	17
1k	2k	4k	2	1,649,213	1,485,032	1,099,776	1,257,981	2,347,743	1,485,032	52.60	7.30	-0.60	10.00	20.50	4.50	0.50	14.40	110.40	10.50	-0.70	13.00	19	17	16	17	18	17	17	18
1k	2k	16k	1	1,654,670	1,878,589	1,098,937	1,246,866	460,524	1,878,589	28.00	5.30	0.10	13.50	20.40	4.50	0.50	13.50	34.70	5.90	-0.40	84.40	20	14	17	17	18	17	17	22
1k	2k	16k	2	1,649,213	1,843,033	1,099,776	1,278,425	461,020	1,843,033	30.20	5.50	0.10	11.80	18.90	4.40	0.50	16.20	34.30	5.90	-0.30	81.50	19	16	18	17	17	18	18	17
1k	2k	2k	4	1,624,166	1,408,820	1,100,469	1,227,855	2,588,553	1,408,820	57.20	7.60	-0.90	13.30	20.80	4.60	0.50	11.60	95.90	9.80	-0.30	7.10	19	18	17	16	16	17	18	18
1k	2k	4k	4	1,624,166	1,412,142	1,100,469	1,252,833	2,305,122	1,412,142	59.00	7.70	-0.90	13.10	19.20	4.40	0.50	13.80	116.90	10.80	-1.00	10.70	19	15	18	17	16	18	18	18
1k	2k	16k	4	1,624,166	1,738,869	1,100,469	1,324,401	462,690	1,738,869	37.60	6.10	-0.20	7.10	16.00	4.00	0.60	20.30	32.70	5.70	-0.30	71.50	20	14	17	17	19	17	17	19
1k	64k	2k	1	1,654,670	1,794,144	3,199	9,616	1,410,778	1,794,144	31.00	5.60	0.10	8.40	0.10	0.40	0.30	200.60	22.00	4.70	0.10	11.00	19	17	18	17	17	19	17	17
1k	64k	2k	2	1,649,213	1,749,370	3,430	10,987	1,406,335	1,749,370	33.80	5.80	0.00	6.10	0.20	0.40	0.10	220.30	24.50	5.00	0.00	8.90	19	17	18	17	17	17	18	19
1k	64k	4k	1	1,654,670	1,851,791	3,199	10,814	1,012,708	1,851,791	29.60	5.40	0.10	11.90	0.10	0.40	0.30	238.00	40.00	6.30	-0.50	54.90	19	17	18	18	16	17	17	18
1k	64k	4k	2	1,649,213	1,805,836	3,430	12,091	1,006,750	1,805,836	32.30	5.70	0.00	9.50	0.20	0.40	0.20	252.50	40.20	6.30	-0.50	52.90	19	16	17	17	17	17	18	17
1k	64k	16k	1	1,654,670	2,104,130	3,199	8,026	257,728	2,104,130	27.70	5.30	0.20	27.20	0.20	0.40	0.20	150.90	12.60	3.50	0.10	70.30	19	16	18	17	17	17	17	18
1k	64k	16k	2	1,649,213	2,094,499	3,430	8,619	258,188	2,094,499	27.70	5.30	0.10	27.00	0.20	0.40	0.20	151.30	12.70	3.60	0.10	66.30	19	16	18	18	18	17	18	17
1k	64k	2k	4	1,624,166	1,634,794	3,882	14,119	1,385,018	1,634,794	40.60	6.40	-0.30	0.70	0.40	0.60	-0.40	263.70	31.70	5.60	-0.30	3.90	19	16	18	17	17	16	17	17

Table 10.3 continued from previous page

1k	64k	4k	4	1,624,166	1,693,946	3,882	14,937	969,195	1,693,946	38.50	6.20	-0.30	4.30	0.30	0.60	-0.30	284.80	41.60	6.40	-0.60	50.60	19	16	18	17	16	17	18	18
1k	64k	16k	4	1,624,166	2,073,697	3,882	10,320	262,969	2,073,697	27.90	5.30	0.10	27.70	0.30	0.60	-0.10	165.80	12.70	3.60	0.10	55.70	20	17	17	17	17	18	17	17
2k	1k	2k	1	1,322,828	1,139,818	1,351,265	1,421,099	2,661,216	1,139,818	51.50	7.20	-1.30	13.80	6.30	2.50	0.90	5.20	63.40	8.00	0.20	10.50	18	16	19	17	16	16	17	17
2k	1k	2k	2	1,310,469	1,124,203	1,351,290	1,420,521	2,647,139	1,124,203	54.20	7.40	-1.40	14.20	6.20	2.50	0.90	5.10	66.20	8.10	0.10	10.60	18	16	17	18	16	16	17	17
2k	1k	4k	1	1,322,828	1,151,576	1,351,265	1,428,865	2,414,616	1,151,576	51.50	7.20	-1.30	12.90	6.40	2.50	0.90	5.70	132.60	11.50	-0.90	25.30	18	16	18	17	16	16	16	17
2k	1k	4k	2	1,310,469	1,123,641	1,351,290	1,426,319	2,388,914	1,123,641	54.60	7.40	-1.40	14.30	6.30	2.50	0.90	5.60	131.90	11.50	-0.90	23.60	18	16	17	17	17	16	18	18
2k	1k	16k	1	1,322,828	1,570,214	1,351,265	1,434,600	468,409	1,570,214	46.10	6.80	-1.00	18.70	6.60	2.60	0.90	6.20	35.30	5.90	-0.30	78.50	20	16	18	17	17	17	18	17
2k	1k	16k	2	1,310,469	1,504,578	1,351,290	1,434,870	467,197	1,504,578	49.70	7.10	-1.20	14.80	6.50	2.50	0.90	6.20	34.70	5.90	-0.30	73.50	18	15	17	17	17	17	17	17
2k	2k	2k	1	1,322,828	1,251,750	1,098,937	1,202,260	2,416,573	1,251,750	51.20	7.20	-1.30	5.40	23.90	4.90	0.40	9.40	88.60	9.40	-0.40	7.20	18	16	17	19	18	18	18	19
2k	2k	2k	2	1,310,469	1,231,764	1,099,776	1,216,327	2,403,720	1,231,764	53.50	7.30	-1.40	6.00	22.70	4.80	0.40	10.60	88.40	9.40	-0.40	7.10	18	17	16	16	19	17	19	18
2k	2k	4k	1	1,322,828	1,251,153	1,098,937	1,241,884	2,260,930	1,251,153	51.00	7.10	-1.30	5.40	22.00	4.70	0.40	13.00	120.80	11.00	-1.00	20.70	18	15	17	16	16	17	18	17
2k	2k	4k	2	1,310,469	1,225,761	1,099,776	1,249,885	2,243,335	1,225,761	53.60	7.30	-1.40	6.50	20.90	4.60	0.50	13.60	123.30	11.10	-1.00	19.10	19	14	16	16	16	17	16	18
2k	2k	16k	1	1,322,828	1,664,623	1,098,937	1,248,036	463,809	1,664,623	45.30	6.70	-1.00	25.80	20.30	4.50	0.50	13.60	33.80	5.80	-0.30	75.90	19	15	17	17	16	16	17	18
2k	2k	16k	2	1,310,469	1,616,386	1,099,776	1,280,080	462,495	1,616,386	47.90	6.90	-1.10	23.30	18.70	4.30	0.50	16.40	33.50	5.80	-0.30	73.30	18	16	18	16	16	16	17	17
2k	64k	2k	1	1,322,828	1,577,630	3,199	10,052	1,244,121	1,577,630	41.30	6.40	-0.80	19.30	0.10	0.40	0.30	214.20	29.60	5.40	-0.20	11.70	19	17	17	16	16	16	17	17
2k	64k	2k	2	1,310,469	1,516,702	3,430	11,155	1,225,755	1,516,702	43.90	6.60	-0.90	15.70	0.20	0.40	0.10	225.20	32.30	5.70	-0.30	9.30	19	14	17	17	16	17	17	17
2k	64k	4k	1	1,322,828	1,642,835	3,199	11,108	988,309	1,642,835	42.00	6.50	-0.90	24.20	0.10	0.40	0.30	247.20	34.00	5.80	-0.30	41.80	19	16	18	18	18	16	17	18
2k	64k	4k	2	1,310,469	1,582,519	3,430	12,142	976,708	1,582,519	43.90	6.60	-0.90	20.80	0.20	0.40	0.10	254.00	34.30	5.90	-0.30	39.00	19	16	17	16	16	17	17	17
2k	64k	16k	1	1,322,828	2,046,465	3,199	8,084	256,443	2,046,465	52.50	7.20	-1.30	54.70	0.20	0.40	0.30	152.70	12.60	3.50	0.10	67.90	19	16	16	17	17	17	18	19
2k	64k	16k	2	1,310,469	2,033,553	3,430	8,691	256,855	2,033,553	53.00	7.30	-1.30	55.20	0.20	0.40	0.20	153.40	12.70	3.60	0.10	63.60	18	17	17	16	17	17	18	16
2k	1k	2k	4	1,248,824	1,093,122	1,351,163	1,416,780	2,581,011	1,093,122	54.90	7.40	-1.40	12.50	6.20	2.50	0.90	4.90	68.00	8.20	0.00	9.90	19	16	18	16	16	17	18	17
2k	1k	4k	4	1,248,824	1,080,042	1,351,163	1,419,892	2,274,658	1,080,042	54.70	7.40	-1.40	13.50	6.40	2.50	0.90	5.10	113.50	10.70	-0.80	17.40	18	15	18	17	17	17	17	19
2k	1k	16k	4	1,248,824	1,373,943	1,351,163	1,431,353	459,020	1,373,943	54.60	7.40	-1.40	10.00	6.30	2.50	0.90	5.90	33.00	5.70	-0.20	65.40	18	16	16	18	16	18	17	17
2k	2k	2k	4	1,248,824	1,180,458	1,100,469	1,233,963	2,341,336	1,180,458	54.60	7.40	-1.40	5.50	20.50	4.50	0.50	12.10	86.00	9.30	-0.40	6.80	19	17	18	17	18	19	18	19
2k	2k	4k	4	1,248,824	1,174,920	1,100,469	1,252,067	2,156,179	1,174,920	54.10	7.40	-1.30	5.90	19.20	4.40	0.50	13.80	119.10	10.90	-1.00	15.00	18	16	15	16	16	16	17	17
2k	2k	16k	4	1,248,824	1,505,025	1,100,469	1,327,444	454,454	1,505,025	51.10	7.10	-1.20	20.50	15.60	3.90	0.60	20.60	32.10	5.70	-0.20	67.90	18	15	15	16	16	17	17	17
2k	64k	2k	4	1,248,824	1,383,731	3,882	13,689	1,136,180	1,383,731	46.90	6.80	-1.00	10.80	0.40	0.60	-0.40	252.60	35.30	5.90	-0.30	7.90	19	17	19	16	16	18	17	18
2k	64k	4k	4	1,248,824	1,442,470	3,882	14,420	923,174	1,442,470	46.30	6.80	-1.00	15.50	0.40	0.60	-0.30	271.50	35.20	5.90	-0.20	35.10	18	15	17	17	17	17	18	17
2k	64k	16k	4	1,248,824	1,997,835	3,882	10,376	263,977	1,997,835	56.80	7.50	-1.50	60.00	0.30	0.50	-0.10	167.30	12.70	3.60	0.10	51.70	18	16	17	17	17	18	16	17

Table 10.3 continued from previous page

64k	1k	2k	1	86,625	85,525	1,351,265	1,468,141	1,219,563	85,525	1.10	1.10	0.40	1.30	8.20	2.90	0.80	8.60	9.00	3.00	0.80	9.30	20	16	19	17	16	17	18	17
64k	1k	2k	2	86,691	85,354	1,351,290	1,462,752	1,218,428	85,354	1.10	1.10	0.40	1.50	8.00	2.80	0.80	8.20	9.00	3.00	0.80	9.10	19	16	17	17	17	17	17	17
64k	1k	4k	1	86,625	86,377	1,351,265	1,467,031	580,371	86,377	1.10	1.00	0.40	0.30	8.20	2.90	0.80	8.60	49.90	7.10	-2.40	118.70	20	17	18	17	16	17	17	18
64k	1k	4k	2	86,691	85,961	1,351,290	1,461,447	579,852	85,961	1.10	1.10	0.40	0.80	8.00	2.80	0.80	8.20	49.60	7.00	-2.40	118.40	20	17	18	18	18	18	17	18
64k	1k	16k	1	86,625	83,948	1,351,265	1,466,795	103,209	83,948	1.10	1.00	0.40	3.10	8.10	2.80	0.80	8.50	2.90	1.70	0.10	20.90	19	16	19	18	17	17	18	18
64k	1k	16k	2	86,691	83,774	1,351,290	1,461,825	103,091	83,774	1.10	1.00	0.40	3.40	7.90	2.80	0.90	8.20	2.80	1.70	0.10	20.60	19	16	19	18	16	17	17	17
64k	2k	2k	1	86,625	89,585	1,098,937	1,435,525	1,125,835	89,585	1.20	1.10	0.30	3.40	16.10	4.00	0.60	30.60	11.80	3.40	0.70	16.10	19	16	18	18	16	17	18	19
64k	2k	2k	2	86,691	89,100	1,099,776	1,430,127	1,124,576	89,100	1.20	1.10	0.30	2.80	15.80	4.00	0.60	30.00	11.70	3.40	0.70	15.80	18	16	19	17	18	18	18	18
64k	2k	4k	1	86,625	90,148	1,098,937	1,434,461	505,939	90,148	1.10	1.10	0.40	4.10	16.10	4.00	0.60	30.50	58.60	7.70	-3.90	144.80	19	17	18	18	17	17	17	18
64k	2k	4k	2	86,691	89,457	1,099,776	1,428,972	503,894	89,457	1.20	1.10	0.40	3.20	15.80	4.00	0.60	29.90	58.50	7.70	-3.90	145.10	18	16	18	17	17	18	18	17
64k	2k	16k	1	86,625	86,168	1,098,937	1,435,211	103,367	86,168	1.10	1.00	0.40	0.50	16.30	4.00	0.60	30.60	2.70	1.60	0.20	18.30	20	16	18	18	17	17	17	18
64k	2k	16k	2	86,691	85,955	1,099,776	1,430,509	103,241	85,955	1.10	1.10	0.40	0.80	15.90	4.00	0.60	30.10	2.60	1.60	0.20	18.10	19	17	19	17	16	18	18	17
64k	64k	2k	1	86,625	78,465	3,199	2,406	88,587	78,465	1.20	1.10	0.30	9.40	0.20	0.40	0.10	24.80	1.50	1.20	0.30	7.40	20	16	18	19	17	17	18	18
64k	64k	2k	2	86,691	76,578	3,430	2,550	88,750	76,578	1.20	1.10	0.30	11.70	0.20	0.40	0.10	25.70	1.50	1.20	0.30	8.80	19	18	17	18	17	17	17	18
64k	64k	4k	1	86,625	77,396	3,199	2,387	85,404	77,396	1.20	1.10	0.30	10.70	0.20	0.40	0.10	25.40	1.50	1.20	0.30	6.50	20	16	18	18	17	18	20	19
64k	64k	4k	2	86,691	75,393	3,430	2,530	85,536	75,393	1.20	1.10	0.30	13.00	0.20	0.40	0.10	26.30	1.50	1.20	0.30	8.10	20	17	19	18	17	17	17	19
64k	64k	16k	1	86,625	70,652	3,199	2,299	80,072	70,652	1.10	1.10	0.40	18.40	0.20	0.40	0.10	28.10	1.40	1.20	0.30	15.00	20	17	18	18	17	18	18	18
64k	64k	16k	2	86,691	68,295	3,430	2,431	80,123	68,295	1.10	1.10	0.40	21.20	0.20	0.40	0.10	29.10	1.50	1.20	0.30	17.10	20	17	19	18	17	17	17	19
64k	1k	2k	4	86,868	84,828	1,351,163	1,454,100	1,217,782	84,828	1.10	1.10	0.40	2.30	7.80	2.80	0.90	7.60	9.00	3.00	0.80	8.70	19	16	18	18	18	17	18	18
64k	1k	4k	4	86,868	85,143	1,351,163	1,452,716	579,493	85,143	1.10	1.10	0.40	2.00	7.70	2.80	0.90	7.50	49.00	7.00	-2.30	117.20	19	17	20	18	17	17	17	19
64k	1k	16k	4	86,868	83,124	1,351,163	1,454,987	103,032	83,124	1.10	1.00	0.40	4.30	7.60	2.80	0.90	7.70	2.80	1.70	0.20	20.10	20	16	19	17	17	17	18	18
64k	2k	2k	4	86,868	87,781	1,100,469	1,421,823	1,123,772	87,781	1.20	1.10	0.30	1.10	15.40	3.90	0.60	29.20	11.40	3.40	0.70	15.20	19	17	18	18	17	17	18	18
64k	2k	4k	4	86,868	87,890	1,100,469	1,420,883	502,203	87,890	1.20	1.10	0.30	1.20	15.30	3.90	0.60	29.10	57.80	7.60	-3.80	144.10	19	16	19	17	17	17	17	18
64k	2k	16k	4	86,868	85,190	1,100,469	1,423,897	103,176	85,190	1.10	1.10	0.40	1.90	15.50	3.90	0.60	29.40	2.60	1.60	0.20	17.80	19	16	18	18	17	18	17	17
64k	64k	2k	4	86,868	73,015	3,882	2,942	89,194	73,015	1.20	1.10	0.30	15.90	0.20	0.50	0.20	24.20	1.60	1.30	0.30	12.10	18	16	18	18	17	17	18	19
64k	64k	4k	4	86,868	71,639	3,882	2,904	85,846	71,639	1.20	1.10	0.30	17.50	0.20	0.50	0.10	25.20	1.60	1.30	0.20	11.50	20	17	20	17	17	17	17	19
64k	64k	16k	4	86,868	64,205	3,882	2,780	80,214	64,205	1.20	1.10	0.30	26.10	0.20	0.50	0.10	28.40	1.50	1.20	0.30	21.00	20	17	18	19	17	17	17	18

Table 10.4: Table of results for Sysbench Memory Benchmark Model executions

				L1D		L1I		LL		L1D				L1I				LL				Execution time per block							
L1D Size (bytes)	L1I Size (bytes)	LL Size (bytes)	Cores	Total Actual	Total Predicted	Total Actual	Total Predicted	Total Actual	Total Predicted	MSE	RMSE	R2	Abs. Err. Percent	MSE	RMSE	R2	Abs. Err. Percent	MSE	RMSE	R2	Abs. Err. Percent	t_1 s	t_2 s	t_3 s	t_4 s	t_5 s	t_6 s	t_7 s	t_8 s
1k	1k	2k	1	1,375,523	1,157,767	726,633	872,551	1,879,632	1,157,767	31.9	5.6	0.5	15.8	7.9	2.8	0.8	20.1	42.9	6.5	0.6	0.3	17	14	14	16	17	16	19	18
1k	1k	2k	2	1,375,355	1,145,833	726,948	872,259	1,879,087	1,145,833	34.2	5.8	0.4	16.7	7.7	2.8	0.8	20.0	45.6	6.8	0.6	0.9	17	14	14	15	16	17	17	17
1k	1k	4k	1	1,375,523	1,186,618	726,633	882,332	1,628,029	1,186,618	29.3	5.4	0.5	13.7	8.0	2.8	0.8	21.4	53.6	7.3	0.4	5.7	17	14	14	16	15	18	17	18
1k	1k	4k	2	1,375,355	1,178,483	726,948	881,037	1,627,343	1,178,483	31.0	5.6	0.5	14.3	7.9	2.8	0.8	21.2	54.5	7.4	0.4	3.6	17	14	15	15	16	16	16	17
1k	1k	2k	4	1,374,963	1,130,585	727,131	868,038	1,878,337	1,130,585	37.8	6.1	0.4	17.8	7.5	2.7	0.8	19.4	50.9	7.1	0.6	2.0	17	14	15	16	16	16	17	17
1k	1k	4k	4	1,374,963	1,160,957	727,131	874,718	1,626,862	1,160,957	34.7	5.9	0.4	15.6	7.7	2.8	0.8	20.3	57.5	7.6	0.4	0.2	17	14	14	16	16	16	17	17
1k	1k	16k	1	1,375,523	1,420,896	726,633	872,110	393,838	1,420,896	16.5	4.1	0.7	3.3	8.4	2.9	0.8	20.0	31.9	5.6	-0.1	86.2	18	14	14	16	16	16	17	17
1k	1k	16k	2	1,375,355	1,406,617	726,948	875,580	393,871	1,406,617	18.4	4.3	0.7	2.3	8.3	2.9	0.8	20.4	31.8	5.6	-0.1	84.9	18	14	14	16	16	17	17	17
1k	1k	16k	4	1,374,963	1,373,452	727,131	875,837	393,932	1,373,452	22.3	4.7	0.6	0.1	8.2	2.9	0.8	20.5	31.4	5.6	-0.1	80.6	18	14	14	16	15	17	17	17
1k	2k	2k	1	1,375,523	1,228,521	576,357	700,589	1,702,896	1,228,521	28.2	5.3	0.5	10.7	16.7	4.1	0.4	21.6	55.8	7.5	0.4	4.6	17	14	15	16	15	16	16	17
1k	2k	2k	2	1,375,355	1,220,455	577,390	698,105	1,703,300	1,220,455	29.8	5.5	0.5	11.3	16.5	4.1	0.4	20.9	57.4	7.6	0.4	4.0	17	14	14	16	15	16	18	17
1k	2k	4k	1	1,375,523	1,229,360	576,357	717,414	1,508,862	1,229,360	27.8	5.3	0.5	10.6	17.1	4.1	0.4	24.5	54.7	7.4	0.3	1.5	17	14	14	16	16	16	16	16
1k	2k	4k	2	1,375,355	1,229,114	577,390	717,382	1,508,841	1,229,114	28.8	5.4	0.5	10.6	16.8	4.1	0.4	24.2	56.7	7.5	0.3	0.6	17	14	14	17	15	16	16	17
1k	2k	16k	1	1,375,523	1,418,073	576,357	671,881	389,519	1,418,073	17.7	4.2	0.7	3.1	16.4	4.1	0.4	16.6	28.3	5.3	0.0	74.6	17	14	14	15	17	16	16	17
1k	2k	16k	2	1,375,355	1,413,127	577,390	693,758	389,549	1,413,127	18.6	4.3	0.7	2.7	16.1	4.0	0.4	20.2	28.3	5.3	0.0	74.5	18	14	14	16	16	16	17	17
1k	2k	2k	4	1,374,963	1,201,551	580,038	685,609	1,705,315	1,201,551	33.3	5.8	0.5	12.6	16.1	4.0	0.4	18.2	60.8	7.8	0.3	1.9	17	13	14	16	15	16	17	17
1k	2k	4k	4	1,374,963	1,220,017	580,038	703,048	1,510,533	1,220,017	31.6	5.6	0.5	11.3	16.1	4.0	0.4	21.2	62.0	7.9	0.2	4.1	17	14	14	16	15	16	16	16
1k	2k	16k	4	1,374,963	1,394,899	580,038	724,734	389,556	1,394,899	21.1	4.6	0.7	1.4	15.5	3.9	0.5	24.9	28.4	5.3	0.0	73.0	17	14	16	16	16	16	16	17
1k	64k	2k	1	1,375,523	1,450,931	2,998	5,307	1,066,028	1,450,931	14.6	3.8	0.8	5.5	0.1	0.3	0.4	77.0	20.9	4.6	0.4	18.0	18	14	15	16	16	16	17	17
1k	64k	2k	2	1,375,355	1,440,978	3,204	5,499	1,065,844	1,440,978	15.7	4.0	0.7	4.8	0.1	0.4	0.4	71.6	21.9	4.7	0.4	17.4	18	14	14	16	15	16	16	17
1k	64k	4k	1	1,375,523	1,478,505	2,998	5,826	821,613	1,478,505	14.0	3.7	0.8	7.5	0.1	0.3	0.4	94.3	30.0	5.5	0.0	51.0	17	14	14	16	16	19	17	17
1k	64k	4k	2	1,375,355	1,466,891	3,204	5,971	821,739	1,466,891	15.0	3.9	0.8	6.7	0.1	0.4	0.4	86.4	30.2	5.5	0.0	50.2	18	14	15	16	17	16	17	17
1k	64k	16k	1	1,375,523	1,614,702	2,998	5,635	238,609	1,614,702	13.0	3.6	0.8	17.4	0.1	0.3	0.4	88.0	12.0	3.5	0.1	67.1	18	14	17	16	16	16	17	18
1k	64k	16k	2	1,375,355	1,614,995	3,204	5,795	238,806	1,614,995	13.1	3.6	0.8	17.4	0.1	0.4	0.4	80.9	12.1	3.5	0.1	64.2	18	15	15	17	16	18	17	17
1k	64k	2k	4	1,374,963	1,413,231	3,611	5,954	1,065,797	1,413,231	18.8	4.3	0.7	2.8	0.2	0.4	0.4	64.9	24.6	5.0	0.3	15.2	17	14	14	16	16	16	19	17

Table 10.4 continued from previous page

1k	64k	4k	4	1,374,963	1,435,853	3,611	6,470	822,149	1,435,853	17.9	4.2	0.7	4.4	0.2	0.4	0.4	79.2	30.7	5.5	0.0	47.5	18	14	14	16	16	16	17	18
1k	64k	16k	4	1,374,963	1,613,818	3,611	6,259	239,128	1,613,818	13.3	3.6	0.8	17.4	0.2	0.4	0.4	73.3	12.0	3.5	0.1	57.9	18	14	14	16	16	16	17	17
2k	1k	2k	1	1,012,999	985,394	726,633	871,607	1,717,481	985,394	44.2	6.6	-0.2	2.7	7.8	2.8	0.8	20.0	40.3	6.3	0.6	0.6	17	14	14	17	15	17	16	20
2k	1k	2k	2	1,012,850	988,731	726,948	872,160	1,716,601	988,731	46.0	6.8	-0.3	2.4	7.6	2.8	0.8	20.0	41.9	6.5	0.6	0.5	17	14	14	16	14	16	17	16
2k	1k	4k	1	1,012,999	1,006,178	726,633	880,628	1,611,174	1,006,178	41.6	6.4	-0.2	0.7	8.0	2.8	0.8	21.2	63.4	8.0	0.3	13.2	17	14	14	15	18	16	19	17
2k	1k	4k	2	1,012,850	1,000,976	726,948	879,346	1,609,861	1,000,976	43.4	6.6	-0.2	1.2	7.8	2.8	0.8	21.0	62.4	7.9	0.3	11.4	19	14	14	16	15	15	17	17
2k	1k	16k	1	1,012,999	1,258,112	726,633	870,533	401,147	1,258,112	40.4	6.4	-0.1	24.2	8.4	2.9	0.8	19.8	30.7	5.5	-0.1	76.5	17	14	15	16	15	17	16	17
2k	1k	16k	2	1,012,850	1,245,659	726,948	874,618	401,192	1,245,659	41.7	6.5	-0.2	23.0	8.2	2.9	0.8	20.3	30.8	5.5	-0.1	75.4	18	15	14	16	18	15	17	20
2k	2k	2k	1	1,012,999	1,059,390	576,357	685,884	1,570,333	1,059,390	44.5	6.7	-0.3	4.6	16.7	4.1	0.4	19.0	56.5	7.5	0.3	2.2	19	13	14	15	16	17	19	16
2k	2k	2k	2	1,012,850	1,060,374	577,390	687,318	1,570,379	1,060,374	46.0	6.8	-0.3	4.7	16.5	4.1	0.4	19.0	57.1	7.6	0.3	2.2	19	14	16	16	15	17	19	19
2k	2k	4k	1	1,012,999	1,071,401	576,357	701,928	1,514,993	1,071,401	42.4	6.5	-0.2	5.8	17.4	4.2	0.4	21.8	63.4	8.0	0.2	9.8	16	14	16	17	18	16	19	19
2k	2k	4k	2	1,012,850	1,073,501	577,390	702,792	1,513,926	1,073,501	43.6	6.6	-0.2	6.0	17.0	4.1	0.4	21.7	63.6	8.0	0.2	7.7	17	14	17	17	15	18	16	18
2k	2k	16k	1	1,012,999	1,285,568	576,357	676,109	396,893	1,285,568	41.6	6.5	-0.2	26.9	16.1	4.0	0.4	17.3	28.5	5.3	0.0	70.8	18	14	14	18	15	17	16	19
2k	2k	16k	2	1,012,850	1,281,327	577,390	695,793	396,939	1,281,327	42.6	6.5	-0.2	26.5	15.8	4.0	0.4	20.5	28.6	5.3	0.0	70.6	17	14	15	15	15	15	16	16
2k	64k	2k	1	1,012,999	1,314,829	2,998	5,163	982,521	1,314,829	38.8	6.2	-0.1	29.8	0.1	0.3	0.4	72.2	23.3	4.8	0.4	16.5	17	14	14	15	16	15	16	20
2k	64k	2k	2	1,012,850	1,294,089	3,204	5,304	981,914	1,294,089	39.6	6.3	-0.1	27.8	0.1	0.4	0.4	65.5	24.4	4.9	0.3	14.9	17	13	16	16	20	16	18	17
2k	64k	4k	1	1,012,999	1,357,122	2,998	5,690	887,243	1,357,122	40.0	6.3	-0.1	34.0	0.1	0.3	0.4	89.8	23.6	4.9	0.3	28.9	17	14	14	16	15	19	16	18
2k	64k	4k	2	1,012,850	1,334,332	3,204	5,844	886,744	1,334,332	40.8	6.4	-0.2	31.7	0.1	0.4	0.4	82.4	24.3	4.9	0.2	27.2	17	14	14	16	15	17	17	19
2k	64k	16k	1	1,012,999	1,563,732	2,998	5,630	242,397	1,563,732	45.6	6.8	-0.3	54.4	0.1	0.3	0.4	87.8	12.0	3.5	0.2	62.7	17	14	15	16	15	16	16	16
2k	64k	16k	2	1,012,850	1,562,214	3,204	5,802	242,591	1,562,214	45.6	6.8	-0.3	54.2	0.1	0.4	0.4	81.1	12.0	3.5	0.2	59.8	17	14	14	16	16	16	18	18
2k	1k	2k	4	1,012,240	998,042	727,131	869,211	1,715,345	998,042	48.6	7.0	-0.4	1.4	7.4	2.7	0.8	19.5	45.4	6.7	0.6	0.2	16	14	14	16	16	16	17	17
2k	1k	4k	4	1,012,240	997,827	727,131	873,941	1,608,532	997,827	46.1	6.8	-0.3	1.4	7.6	2.8	0.8	20.2	60.4	7.8	0.3	7.2	17	14	14	17	15	16	16	17
2k	1k	16k	4	1,012,240	1,216,142	727,131	875,802	401,277	1,216,142	44.7	6.7	-0.3	20.1	8.1	2.9	0.8	20.4	30.5	5.5	-0.1	72.2	17	14	14	16	15	17	16	20
2k	2k	2k	4	1,012,240	1,058,619	580,038	681,070	1,571,729	1,058,619	48.3	7.0	-0.4	4.6	16.0	4.0	0.4	17.4	58.9	7.7	0.3	1.2	17	14	15	17	14	17	18	21
2k	2k	4k	4	1,012,240	1,066,920	580,038	691,316	1,514,548	1,066,920	46.1	6.8	-0.3	5.4	16.1	4.0	0.4	19.2	65.0	8.1	0.2	4.6	23	14	16	18	15	18	18	18
2k	2k	16k	4	1,012,240	1,265,878	580,038	725,787	397,046	1,265,878	44.5	6.7	-0.3	25.1	15.2	3.9	0.5	25.1	28.7	5.4	0.0	69.2	17	14	14	15	16	16	16	17
2k	64k	2k	4	1,012,240	1,244,733	3,611	5,782	981,018	1,244,733	41.3	6.4	-0.2	23.0	0.2	0.4	0.4	60.1	26.6	5.2	0.3	10.7	17	14	14	15	16	16	18	17
2k	64k	4k	4	1,012,240	1,281,504	3,611	6,274	886,714	1,281,504	42.4	6.5	-0.2	26.6	0.2	0.4	0.4	73.7	25.5	5.1	0.2	22.8	17	14	14	16	17	17	17	16
2k	64k	16k	4	1,012,240	1,555,893	3,611	6,248	242,938	1,555,893	45.7	6.8	-0.3	53.7	0.2	0.4	0.4	73.0	12.0	3.5	0.2	53.1	17	14	14	16	16	18	17	16

Table 10.4 continued from previous page

64k	1k	2k	1	83,523	83,118	726,633	956,454	728,023	83,118	1.1	1.0	0.4	0.5	10.1	3.2	0.8	31.6	6.6	2.6	0.8	19.0	18	15	16	18	17	17	17	18
64k	1k	2k	2	83,631	82,971	726,948	952,510	727,479	82,971	1.1	1.0	0.4	0.8	9.9	3.2	0.8	31.0	6.6	2.6	0.8	18.7	18	14	16	16	16	17	17	17
64k	1k	4k	1	83,523	83,626	726,633	954,093	200,532	83,626	1.0	1.0	0.4	0.1	10.0	3.2	0.8	31.3	55.7	7.5	-4.9	314.8	19	14	14	16	15	16	16	17
64k	1k	4k	2	83,631	83,448	726,948	950,014	200,644	83,448	1.1	1.0	0.4	0.2	9.9	3.1	0.8	30.7	55.5	7.4	-4.8	313.9	19	16	15	17	16	17	17	17
64k	1k	16k	1	83,523	82,514	726,633	958,158	98,516	82,514	1.0	1.0	0.4	1.2	10.3	3.2	0.8	31.9	2.6	1.6	0.2	26.0	21	14	17	17	17	17	18	17
64k	1k	16k	2	83,631	82,198	726,948	954,959	98,509	82,198	1.0	1.0	0.4	1.7	10.1	3.2	0.8	31.4	2.6	1.6	0.2	25.4	18	15	17	18	18	17	18	18
64k	2k	2k	1	83,523	86,365	576,357	921,405	523,851	86,365	1.1	1.1	0.4	3.4	18.5	4.3	0.4	59.9	21.2	4.6	0.2	61.0	18	14	16	17	16	17	17	17
64k	2k	2k	2	83,631	86,050	577,390	917,704	522,607	86,050	1.1	1.1	0.4	2.9	18.2	4.3	0.4	58.9	21.3	4.6	0.2	60.9	18	15	14	16	16	17	18	18
64k	2k	4k	1	83,523	86,735	576,357	918,659	199,589	86,735	1.1	1.0	0.4	3.8	18.4	4.3	0.4	59.4	53.9	7.3	-4.7	305.0	18	14	15	16	15	16	16	17
64k	2k	4k	2	83,631	86,402	577,390	914,935	199,674	86,402	1.1	1.1	0.4	3.3	18.1	4.3	0.4	58.5	53.7	7.3	-4.7	304.2	18	14	15	17	16	17	17	18
64k	2k	16k	1	83,523	84,192	576,357	923,641	98,603	84,192	1.1	1.0	0.4	0.8	18.8	4.3	0.3	60.3	2.5	1.6	0.2	23.7	18	16	15	17	16	17	17	18
64k	2k	16k	2	83,631	83,888	577,390	920,215	98,597	83,888	1.1	1.0	0.4	0.3	18.4	4.3	0.4	59.4	2.5	1.6	0.2	23.2	18	14	16	18	17	17	18	17
64k	64k	2k	1	83,523	72,831	2,998	2,231	84,904	72,831	1.1	1.1	0.4	12.8	0.2	0.4	0.2	25.6	1.4	1.2	0.3	9.5	18	16	16	19	17	19	17	19
64k	64k	2k	2	83,631	72,139	3,204	2,384	85,122	72,139	1.2	1.1	0.3	13.7	0.2	0.4	0.2	25.6	1.4	1.2	0.3	10.2	20	15	17	18	18	18	18	18
64k	64k	4k	1	83,523	71,724	2,998	2,207	81,822	71,724	1.1	1.1	0.4	14.1	0.2	0.4	0.2	26.4	1.4	1.2	0.3	8.7	19	16	15	18	18	18	19	18
64k	64k	4k	2	83,631	71,011	3,204	2,357	82,032	71,011	1.2	1.1	0.3	15.1	0.2	0.4	0.2	26.4	1.5	1.2	0.3	9.4	18	14	15	16	16	17	17	17
64k	64k	16k	1	83,523	65,442	2,998	2,103	76,695	65,442	1.1	1.0	0.4	21.6	0.2	0.4	0.2	29.8	1.3	1.2	0.3	17.0	18	15	17	17	18	16	20	19
64k	64k	16k	2	83,631	64,557	3,204	2,226	76,811	64,557	1.1	1.1	0.4	22.8	0.2	0.4	0.2	30.5	1.4	1.2	0.3	17.9	19	15	15	16	16	18	17	18
64k	1k	2k	4	83,824	82,454	727,131	945,433	727,585	82,454	1.1	1.0	0.4	1.6	9.7	3.1	0.8	30.0	6.6	2.6	0.8	17.9	18	14	16	16	16	17	17	17
64k	1k	4k	4	83,824	82,806	727,131	942,816	200,943	82,806	1.1	1.0	0.4	1.2	9.6	3.1	0.8	29.7	54.8	7.4	-4.7	310.2	18	15	15	18	16	19	18	17
64k	1k	16k	4	83,824	81,633	727,131	950,155	98,644	81,633	1.1	1.0	0.4	2.6	9.8	3.1	0.8	30.7	2.6	1.6	0.2	24.2	19	14	16	16	17	18	17	19
64k	2k	2k	4	83,824	85,023	580,038	911,297	523,691	85,023	1.1	1.1	0.4	1.4	17.6	4.2	0.4	57.1	20.9	4.6	0.2	59.2	18	14	14	16	18	16	17	17
64k	2k	4k	4	83,824	85,251	580,038	908,672	199,913	85,251	1.1	1.1	0.4	1.7	17.5	4.2	0.4	56.7	53.0	7.3	-4.6	300.1	18	14	15	16	16	16	16	17
64k	2k	16k	4	83,824	83,173	580,038	914,607	98,723	83,173	1.1	1.0	0.4	0.8	17.8	4.2	0.4	57.7	2.4	1.6	0.2	22.2	18	14	14	17	16	16	17	17
64k	64k	2k	4	83,824	70,903	3,611	2,772	85,640	70,903	1.2	1.1	0.3	15.4	0.2	0.5	0.2	23.2	1.5	1.2	0.3	11.6	20	15	16	17	17	18	17	19
64k	64k	4k	4	83,824	69,693	3,611	2,733	82,536	69,693	1.2	1.1	0.3	16.9	0.2	0.5	0.2	24.3	1.6	1.3	0.3	10.8	19	15	14	16	16	17	17	17
64k	64k	16k	4	83,824	62,909	3,611	2,536	77,073	62,909	1.2	1.1	0.3	25.0	0.2	0.5	0.2	29.8	1.4	1.2	0.3	19.7	18	16	14	16	17	18	16	17

Table 10.5: Table of results for the GFTT Model executions

				L1D		L1I		LL		L1D				L1I				LL				Execution time per block							
L1D Size (bytes)	L1I Size (bytes)	LL Size (bytes)	Cores	Total Actual	Total Predicted	Total Actual	Total Predicted	Total Actual	Total Predicted	MSE	RMSE	R2	Abs. Err. Percent	MSE	RMSE	R2	Abs. Err. Percent	MSE	RMSE	R2	Abs. Err. Percent	t_1 s	t_2 s	t_3 s	t_4 s	t_5 s	t_6 s	t_7 s	t_8 s
1k	1k	2k	1	2,023,632	1,606,586	314,146	534,562	1,819,176	1,606,586	38.0	6.2	0.5	20.6	17.8	4.2	0.1	70.2	27.9	5.3	0.5	3.4	15	14	14	15	15	20	20	20
1k	1k	2k	2	2,023,632	1,634,488	314,146	531,671	1,819,176	1,634,488	36.8	6.1	0.5	19.2	16.8	4.1	0.2	69.2	28.3	5.3	0.5	5.3	16	14	14	14	14	15	15	15
1k	1k	4k	1	2,023,632	1,629,368	314,146	547,014	1,691,447	1,629,368	36.8	6.1	0.5	19.5	18.7	4.3	0.1	74.1	21.9	4.7	0.5	0.6	21	21	20	21	18	19	21	21
1k	1k	4k	2	2,023,632	1,664,880	314,146	544,691	1,691,447	1,664,880	35.2	5.9	0.5	17.7	17.7	4.2	0.1	73.4	22.4	4.7	0.5	1.4	21	23	18	19	17	20	20	20
1k	1k	2k	4	2,023,632	1,683,170	314,146	518,085	1,819,176	1,683,170	35.0	5.9	0.5	16.8	14.7	3.8	0.3	64.9	31.5	5.6	0.4	9.4	11	13	13	13	13	14	15	15
1k	1k	4k	4	2,023,632	1,706,171	314,146	531,413	1,691,447	1,706,171	33.7	5.8	0.6	15.7	15.5	3.9	0.2	69.2	24.9	5.0	0.5	5.6	20	21	18	19	19	19	20	19
1k	1k	16k	1	2,023,632	1,748,195	314,146	580,855	1,317,132	1,748,195	30.5	5.5	0.6	13.6	20.9	4.6	0.0	84.9	18.6	4.3	0.6	4.4	25	25	18	19	17	20	19	21
1k	1k	16k	2	2,023,632	1,756,234	314,146	578,944	1,317,132	1,756,234	30.5	5.5	0.6	13.2	20.0	4.5	0.0	84.3	17.1	4.1	0.7	2.8	32	20	17	19	19	20	20	20
1k	1k	16k	4	2,023,632	1,742,038	314,146	568,377	1,317,132	1,742,038	32.4	5.7	0.6	13.9	17.8	4.2	0.1	80.9	15.2	3.9	0.7	0.3	20	24	19	19	17	19	21	19
1k	2k	2k	1	2,023,632	1,595,868	273,047	425,870	1,779,048	1,595,868	39.0	6.2	0.5	21.1	17.1	4.1	0.0	56.0	30.7	5.5	0.4	5.9	24	15	16	16	16	17	18	18
1k	2k	2k	2	2,023,632	1,621,258	273,047	415,058	1,779,048	1,621,258	37.8	6.2	0.5	19.9	15.5	3.9	0.1	52.0	30.6	5.5	0.4	7.8	21	22	19	19	17	19	18	20
1k	2k	4k	1	2,023,632	1,615,371	273,047	444,989	1,662,461	1,615,371	38.1	6.2	0.5	20.2	18.1	4.3	0.0	63.0	23.6	4.9	0.5	1.9	21	16	14	15	20	18	20	20
1k	2k	4k	2	2,023,632	1,648,167	273,047	433,853	1,662,461	1,648,167	36.6	6.0	0.5	18.6	16.3	4.0	0.1	58.9	23.7	4.9	0.5	3.9	24	18	16	17	17	16	17	16
1k	2k	16k	1	2,023,632	1,745,551	273,047	499,265	1,307,282	1,745,551	31.6	5.6	0.6	13.7	20.5	4.5	-0.1	82.8	16.7	4.1	0.7	3.1	23	18	18	17	19	20	21	21
1k	2k	16k	2	2,023,632	1,757,274	273,047	488,108	1,307,282	1,757,274	31.4	5.6	0.6	13.2	18.8	4.3	0.0	78.8	15.6	3.9	0.7	1.5	20	19	21	19	18	20	21	19
1k	2k	2k	4	2,023,632	1,668,115	273,047	390,707	1,779,048	1,668,115	36.0	6.0	0.5	17.6	12.5	3.5	0.3	43.1	32.0	5.7	0.4	11.5	22	19	18	18	17	20	20	20
1k	2k	4k	4	2,023,632	1,691,980	273,047	407,725	1,662,461	1,691,980	34.7	5.9	0.5	16.4	13.1	3.6	0.3	49.3	24.9	5.0	0.4	7.7	21	19	16	14	18	21	18	20
1k	2k	16k	4	2,023,632	1,749,309	273,047	461,196	1,307,282	1,749,309	32.6	5.7	0.6	13.6	15.1	3.9	0.2	68.9	14.3	3.8	0.7	1.0	19	25	20	21	19	20	18	20
1k	64k	2k	1	2,023,632	1,836,278	1,056	5,546	1,468,709	1,836,278	25.0	5.0	0.7	9.3	0.1	0.3	-0.1	425.2	9.6	3.1	0.7	10.5	20	22	20	22	20	20	17	19
1k	64k	2k	2	2,023,632	1,824,667	1,056	5,771	1,468,709	1,824,667	27.0	5.2	0.6	9.8	0.1	0.3	-0.1	446.4	8.8	3.0	0.7	7.1	20	20	21	19	17	20	24	20
1k	64k	4k	1	2,023,632	1,831,490	1,056	5,212	1,355,125	1,831,490	27.2	5.2	0.6	9.5	0.1	0.3	-0.1	393.6	10.9	3.3	0.7	14.8	20	20	18	19	18	20	21	20
1k	64k	4k	2	2,023,632	1,838,692	1,056	5,343	1,355,125	1,838,692	28.6	5.3	0.6	9.1	0.1	0.3	-0.1	405.9	9.3	3.1	0.7	10.5	20	17	17	19	19	20	21	20
1k	64k	16k	1	2,023,632	2,045,826	1,056	4,832	1,162,057	2,045,826	30.6	5.5	0.6	1.1	0.1	0.3	0.0	357.6	13.5	3.7	0.7	0.4	20	19	18	19	23	16	18	20
1k	64k	16k	2	2,023,632	2,096,027	1,056	5,175	1,162,057	2,096,027	29.5	5.4	0.6	3.6	0.1	0.3	0.0	390.0	12.6	3.5	0.7	3.6	19	18	22	21	19	20	21	21
1k	64k	2k	4	2,023,632	1,836,476	1,056	5,948	1,468,709	1,836,476	29.6	5.4	0.6	9.2	0.1	0.3	-0.1	463.2	7.3	2.7	0.8	2.0	23	21	23	24	23	19	22	25

Table 10.5 continued from previous page

1k	64k	4k	4	2,023,632	1,886,448	1,056	5,633	1,355,125	1,886,448	30.6	5.5	0.6	6.8	0.1	0.3	-0.1	433.4	7.0	2.7	0.8	0.5	20	19	18	19	19	20	21	19
1k	64k	16k	4	2,023,632	2,145,800	1,056	6,300	1,162,057	2,145,800	29.9	5.5	0.6	6.0	0.1	0.3	-0.1	496.6	11.7	3.4	0.7	8.5	21	25	18	19	20	20	22	20
2k	1k	2k	1	1,462,803	1,501,582	314,146	536,595	1,765,780	1,501,582	10.2	3.2	0.7	2.7	18.0	4.2	0.1	70.8	24.7	5.0	0.5	1.7	20	19	18	17	16	17	15	17
2k	1k	2k	2	1,462,803	1,519,541	314,146	533,491	1,765,780	1,519,541	10.2	3.2	0.7	3.9	17.0	4.1	0.2	69.8	25.1	5.0	0.5	3.4	19	18	17	18	25	21	22	21
2k	1k	4k	1	1,462,803	1,510,327	314,146	548,358	1,721,743	1,510,327	10.6	3.3	0.7	3.2	18.8	4.3	0.1	74.6	23.0	4.8	0.5	2.1	20	18	17	18	20	19	20	21
2k	1k	4k	2	1,462,803	1,530,552	314,146	546,055	1,721,743	1,530,552	10.7	3.3	0.7	4.6	17.9	4.2	0.1	73.8	23.7	4.9	0.5	3.8	20	19	21	19	19	20	27	21
2k	1k	16k	1	1,462,803	1,576,213	314,146	582,882	1,325,555	1,576,213	13.7	3.7	0.6	7.8	21.0	4.6	0.0	85.5	18.7	4.3	0.6	4.1	20	18	18	26	18	20	21	21
2k	1k	16k	2	1,462,803	1,587,632	314,146	581,073	1,325,555	1,587,632	13.7	3.7	0.6	8.5	20.1	4.5	0.0	85.0	17.4	4.2	0.7	2.6	20	17	19	21	20	22	20	23
2k	2k	2k	1	1,462,803	1,497,188	273,047	430,568	1,730,394	1,497,188	10.4	3.2	0.7	2.4	17.3	4.2	0.0	57.7	27.1	5.2	0.4	4.2	22	17	20	19	18	20	19	21
2k	2k	2k	2	1,462,803	1,511,886	273,047	419,732	1,730,394	1,511,886	10.2	3.2	0.7	3.4	15.6	4.0	0.1	53.7	27.0	5.2	0.4	6.0	22	18	19	18	18	21	21	22
2k	2k	4k	1	1,462,803	1,503,892	273,047	448,695	1,693,860	1,503,892	10.7	3.3	0.7	2.8	18.3	4.3	0.0	64.3	24.9	5.0	0.5	4.5	19	17	19	18	19	20	23	20
2k	2k	4k	2	1,462,803	1,521,894	273,047	437,777	1,693,860	1,521,894	10.7	3.3	0.7	4.0	16.5	4.1	0.1	60.3	25.2	5.0	0.5	6.2	21	20	19	19	19	20	22	19
2k	2k	16k	1	1,462,803	1,569,702	273,047	502,588	1,315,805	1,569,702	13.7	3.7	0.6	7.3	20.7	4.5	-0.2	84.1	16.9	4.1	0.7	2.9	23	17	18	19	19	21	22	19
2k	2k	16k	2	1,462,803	1,582,855	273,047	491,592	1,315,805	1,582,855	13.9	3.7	0.6	8.2	19.0	4.4	-0.1	80.0	15.9	4.0	0.7	1.5	20	20	20	19	18	21	22	21
2k	64k	2k	1	1,462,803	1,690,298	1,056	5,510	1,449,749	1,690,298	20.2	4.5	0.4	15.6	0.1	0.3	-0.1	421.8	8.1	2.8	0.8	6.8	23	17	19	18	20	22	20	21
2k	64k	2k	2	1,462,803	1,673,469	1,056	5,877	1,449,749	1,673,469	20.9	4.6	0.4	14.4	0.1	0.3	-0.1	456.6	7.7	2.8	0.8	3.9	20	19	17	19	19	22	22	22
2k	64k	4k	1	1,462,803	1,683,601	1,056	5,478	1,392,395	1,683,601	22.3	4.7	0.3	15.1	0.1	0.3	-0.1	418.8	8.2	2.9	0.8	7.3	21	18	18	18	20	20	20	23
2k	64k	4k	2	1,462,803	1,677,347	1,056	5,836	1,392,395	1,677,347	24.0	4.9	0.3	14.7	0.1	0.3	-0.1	452.6	7.5	2.7	0.8	3.8	21	19	18	18	20	22	19	19
2k	64k	16k	1	1,462,803	1,873,174	1,056	6,516	1,170,425	1,873,174	54.6	7.4	-0.6	28.1	0.1	0.3	-0.1	517.0	13.0	3.6	0.7	1.2	21	20	20	18	20	21	21	21
2k	64k	16k	2	1,462,803	1,924,095	1,056	6,494	1,170,425	1,924,095	58.4	7.6	-0.7	31.5	0.1	0.3	-0.1	515.0	12.4	3.5	0.7	3.9	20	19	18	18	20	21	21	23
2k	1k	2k	4	1,462,803	1,550,036	314,146	521,220	1,765,780	1,550,036	10.7	3.3	0.7	6.0	14.8	3.8	0.3	65.9	27.4	5.2	0.4	7.2	20	18	17	18	17	22	22	21
2k	1k	4k	4	1,462,803	1,560,273	314,146	534,841	1,721,743	1,560,273	11.2	3.4	0.7	6.7	15.7	4.0	0.2	70.3	26.4	5.1	0.5	7.6	16	14	15	18	16	17	18	21
2k	1k	16k	4	1,462,803	1,591,357	314,146	570,694	1,325,555	1,591,357	13.5	3.7	0.6	8.8	17.9	4.2	0.1	81.7	15.8	4.0	0.7	0.3	21	19	19	24	18	20	19	21
2k	2k	2k	4	1,462,803	1,539,410	273,047	394,649	1,730,394	1,539,410	10.8	3.3	0.7	5.2	12.7	3.6	0.3	44.5	27.9	5.3	0.4	9.5	25	18	17	18	21	22	19	21
2k	2k	4k	4	1,462,803	1,549,745	273,047	411,889	1,693,860	1,549,745	11.3	3.4	0.7	5.9	13.3	3.6	0.3	50.8	26.7	5.2	0.4	9.7	21	17	17	18	18	21	21	23
2k	2k	16k	4	1,462,803	1,588,866	273,047	465,042	1,315,805	1,588,866	13.6	3.7	0.6	8.6	15.3	3.9	0.1	70.3	14.8	3.9	0.7	1.0	21	18	18	19	18	22	23	21
2k	64k	2k	4	1,462,803	1,656,982	1,056	6,708	1,449,749	1,656,982	23.7	4.9	0.3	13.3	0.1	0.3	-0.1	535.3	7.1	2.7	0.8	3.2	26	18	17	19	20	21	22	21
2k	64k	4k	4	1,462,803	1,698,397	1,056	6,830	1,392,395	1,698,397	30.4	5.5	0.1	16.1	0.1	0.3	-0.1	546.8	6.7	2.6	0.8	3.7	20	18	20	18	19	20	19	24
2k	64k	16k	4	1,462,803	2,009,624	1,056	6,993	1,170,425	2,009,624	67.0	8.2	-1.0	37.4	0.1	0.3	-0.1	562.2	11.9	3.5	0.7	9.1	20	19	19	19	18	23	22	21

Table 10.5 continued from previous page

64k	1k	2k	1	739,204	787,815	314,146	684,905	1,033,108	787,815	15.1	3.9	0.4	6.6	26.8	5.2	-0.3	118.0	29.7	5.5	-0.1	29.8	20	18	18	18	19	24	23	21
64k	1k	2k	2	739,204	792,815	314,146	684,835	1,033,108	792,815	14.9	3.9	0.4	7.3	26.4	5.1	-0.3	118.0	29.3	5.4	-0.1	28.7	21	19	18	18	18	20	21	19
64k	1k	4k	1	739,204	782,650	314,146	684,375	913,889	782,650	15.2	3.9	0.4	5.9	26.7	5.2	-0.3	117.9	31.6	5.6	-0.1	43.3	22	18	17	19	19	19	20	21
64k	1k	4k	2	739,204	787,743	314,146	684,817	913,889	787,743	15.0	3.9	0.4	6.6	26.3	5.1	-0.3	118.0	30.5	5.5	-0.1	42.2	22	18	18	18	18	20	20	24
64k	1k	16k	1	739,204	744,781	314,146	679,309	835,901	744,781	15.9	4.0	0.3	0.8	26.2	5.1	-0.3	116.2	24.2	4.9	0.1	34.0	21	17	18	19	19	20	23	22
64k	1k	16k	2	739,204	747,844	314,146	680,935	835,901	747,844	15.6	4.0	0.4	1.2	25.9	5.1	-0.3	116.8	23.3	4.8	0.2	33.1	22	17	18	19	20	21	20	20
64k	2k	2k	1	739,204	793,012	273,047	632,685	1,011,131	793,012	15.1	3.9	0.4	7.3	28.0	5.3	-0.6	131.7	29.1	5.4	-0.1	29.9	20	17	17	19	21	21	24	23
64k	2k	2k	2	739,204	797,793	273,047	629,005	1,011,131	797,793	15.0	3.9	0.4	7.9	27.3	5.2	-0.5	130.4	28.4	5.3	-0.1	28.6	22	18	18	17	19	18	20	21
64k	2k	4k	1	739,204	788,192	273,047	632,507	899,551	788,192	15.3	3.9	0.4	6.6	27.9	5.3	-0.6	131.6	30.1	5.5	-0.1	42.9	21	17	19	19	24	19	21	21
64k	2k	4k	2	739,204	793,077	273,047	629,302	899,551	793,077	15.1	3.9	0.4	7.3	27.3	5.2	-0.5	130.5	28.8	5.4	0.0	41.5	20	19	18	20	21	22	20	21
64k	2k	16k	1	739,204	751,842	273,047	630,605	829,418	751,842	16.0	4.0	0.3	1.7	27.5	5.2	-0.5	131.0	23.4	4.8	0.2	33.6	22	17	18	18	20	22	20	20
64k	2k	16k	2	739,204	755,298	273,047	628,974	829,418	755,298	15.7	4.0	0.4	2.2	26.9	5.2	-0.5	130.4	22.4	4.7	0.2	32.6	22	20	21	19	24	21	21	21
64k	64k	2k	1	739,204	758,090	1,056	8,161	740,252	758,090	18.0	4.2	0.3	2.6	0.1	0.3	-0.2	672.8	15.2	3.9	0.4	6.6	21	18	17	19	18	19	20	22
64k	64k	2k	2	739,204	741,935	1,056	8,438	740,252	741,935	18.1	4.3	0.3	0.4	0.1	0.3	-0.2	699.1	15.3	3.9	0.4	4.6	17	16	14	18	19	20	21	21
64k	64k	4k	1	739,204	754,878	1,056	7,887	740,213	754,878	17.9	4.2	0.3	2.1	0.1	0.3	-0.2	646.9	15.1	3.9	0.4	5.5	20	19	19	18	18	20	20	24
64k	64k	4k	2	739,204	738,388	1,056	8,210	740,213	738,388	18.1	4.2	0.3	0.1	0.1	0.3	-0.2	677.5	15.4	3.9	0.4	3.5	20	19	18	18	18	20	19	20
64k	64k	16k	1	739,204	722,813	1,056	7,718	717,581	722,813	18.4	4.3	0.2	2.2	0.1	0.3	-0.2	630.8	16.2	4.0	0.3	1.3	17	19	18	22	18	20	21	21
64k	64k	16k	2	739,204	706,138	1,056	8,731	717,581	706,138	18.7	4.3	0.2	4.5	0.1	0.3	-0.2	726.8	16.5	4.1	0.3	0.6	21	17	18	18	18	16	23	17
64k	1k	2k	4	739,204	800,714	314,146	678,952	1,033,108	800,714	14.9	3.9	0.4	8.3	25.6	5.1	-0.2	116.1	28.2	5.3	0.0	26.5	17	18	18	18	19	21	23	21
64k	1k	4k	4	739,204	795,675	314,146	679,616	913,889	795,675	15.0	3.9	0.4	7.6	25.6	5.1	-0.2	116.3	28.8	5.4	0.0	39.9	21	19	19	20	19	21	19	19
64k	1k	16k	4	739,204	747,317	314,146	677,526	835,901	747,317	15.7	4.0	0.4	1.1	25.2	5.0	-0.2	115.7	22.3	4.7	0.2	30.9	20	19	17	19	20	18	19	22
64k	2k	2k	4	739,204	806,470	273,047	617,510	1,011,131	806,470	14.9	3.9	0.4	9.1	26.2	5.1	-0.5	126.2	26.9	5.2	0.0	26.4	21	17	21	18	19	21	19	23
64k	2k	4k	4	739,204	801,390	273,047	618,340	899,551	801,390	15.1	3.9	0.4	8.4	26.2	5.1	-0.5	126.5	27.2	5.2	0.0	39.2	20	18	18	18	21	20	20	20
64k	2k	16k	4	739,204	753,537	273,047	618,131	829,418	753,537	15.8	4.0	0.4	1.9	25.8	5.1	-0.4	126.4	21.3	4.6	0.2	30.0	20	18	17	19	18	21	19	21
64k	64k	2k	4	739,204	708,299	1,056	9,436	740,252	708,299	18.4	4.3	0.2	4.2	0.1	0.3	-0.3	793.5	15.7	4.0	0.4	0.8	22	18	16	15	17	18	19	17
64k	64k	4k	4	739,204	704,701	1,056	9,502	740,213	704,701	18.6	4.3	0.2	4.7	0.1	0.3	-0.3	799.8	15.9	4.0	0.4	0.2	21	18	17	18	20	19	19	20
64k	64k	16k	4	739,204	666,141	1,056	12,596	717,581	666,141	19.5	4.4	0.2	9.9	0.1	0.3	-0.5	1092.8	17.2	4.1	0.3	5.0	21	19	19	18	18	20	20	22

Chapter 11

Paper D: HASCO: A Hybrid AI Simulation Compiler for Semantic Accident Reconstruction

Edin Jelačić, Rong Gu, Cristina Seceleanu, Ning Xiong, Peter Backeman, Tiberiu Seceleanu, Zhennan Fei, Ali Nouri

Submitted to the 30th Ada-Europe International Conference on Reliable Software Technologies (AEiC 2026)

Note: This paper has been reformatted to comply with the thesis layout. The content is unchanged from the published version.

Abstract

The validation of Automated Driving Systems (ADSs) has shifted from distance-based metrics to Scenario-Based Testing (SBT). Large Language Models (LLMs) have emerged as powerful tools with potential for generating vehicular scenarios at scale. However, generative models, used for direct simulation synthesis, produce inadequate output, therefore necessitating a more structured compilation approach. In this regard, we present **HASCO** (**H**ybrid **AI** **S**imulation **CO**mpiler), a system that translates natural-language driving scene specifications into executable simulation artifacts (XOSC/XODR files) for the esmini/OpenSCENARIO ecosystem. While LLMs excel at narrative parsing, we demonstrate that direct synthesis of simulation artifacts fails in the vast majority of cases due to hallucinated physics or schema violations. To resolve this, HASCO treats scenario creation as a **compilation task** rather than a generative one. The pipeline supports three compilation paths: direct synthesis, a Python intermediate (via `scenariogeneration`), and an ontology-guided path that grounds intent into an intermediate representation (IR) before compilation. We further evaluate a self-judging mechanism for automated repair. Across six operating modes evaluated on 40 real-world accident reports, the ontology-guided compiler and Python-based compiler achieve 95% and 90% executability rates, respectively (compared to 5% for direct synthesis). Additionally, we evaluate outputs on *semantic fidelity*, positioning HASCO as a robust tool for forensic scene reconstruction.

11.1 Introduction

Automated Driving Systems (ADSs) represent a paradigm shift in modern transportation, promising to significantly reduce traffic accidents. A recent *National Highway Traffic Safety Administration* study showed that Advanced Driver Assistance Systems (ADAS) features like automatic emergency braking reduce rear-end crashes by up to 49% [1]. While ADSs offer potential for optimized traffic flow and enhanced mobility, deploying these systems in unconstrained operational design domains (ODDs) remains a significant engineering challenge [2].

Unlike conventional software operating within bounded logical environments, ADSs must make safety-critical decisions in stochastic, open-world settings. Traditional "distance-based" validation of these mechanisms has proven practically unsuitable [3]. This realization has driven the industry toward Scenario-Based Testing (SBT), validating ADSs against libraries of concrete, critical traffic situations rather than random mileage [4]. The advent of generative

artificial intelligence (AI) pairs well with this paradigm [5].

Natural-language scene descriptions, such as police accident reports or crash news reports, are expressive but frequently underspecified for direct simulation [6]. Dynamic vehicular simulations are deterministically rigid and tend to require significantly more information for execution than is provided in the aforementioned reports. The industry standard for describing dynamic driving maneuvers is ASAM OpenSCENARIO [7], which mandates a strict XML schema to define actor behaviors and environmental states. Converting natural language reports into executable OpenSCENARIO files requires solving two distinct problems: *Semantic Parsing* (meaning of the scenario) and *Syntactic Engineering* (producing valid XML with correct coordinate geometry).

We demonstrate that due to the semantic and syntactic complexity of OpenSCENARIO, generating XML directly from prompts is fundamentally brittle. Specifically, stochastic models struggle to maintain the long-range dependencies required by the standard, where a single maneuver requires coordinated updates across position, speed, and orientation tags that must remain mathematically consistent over time. Therefore, we propose **HASCO**, an LLM-based structured pipeline that decouples semantic reasoning from syntactic generation. Based on our experience, LLMs are proficient at extracting meaning from natural language text. The pipeline effectively functions as a semantic compiler: the LLM acts as a parser that extracts high-level logic, while the deterministic pipeline handles the “assembly code” generation of the XML.

Currently, translating textual reports into simulation requires significant manual engineering effort to close the gap between narrative events and the rigid scenario reconstruction syntax.

We position HASCO not as a replacement for expert judgment, but as a Human-in-the-Loop co-generation tool versus the manual process, as can be seen in the workflow diagram at Figure 11.1. By automating the syntactic manual processing of XML engineering, HASCO allows experts to focus on the “semantic validity” of the scenario, accelerating the digitization of crash databases for later use in ADS testing and crash reconstruction.

11.1.1 The specification gap

Menzel et al. [8] categorize vehicular scenarios into three distinct levels based on their granularity and machine readability: functional, logical, and concrete, extending upon the foundational scenario definitions provided by Ulbrich et al. [9]. The hierarchy begins with **functional scenarios**, which consist of a verbal description focusing on actions and events, such as “maneu-

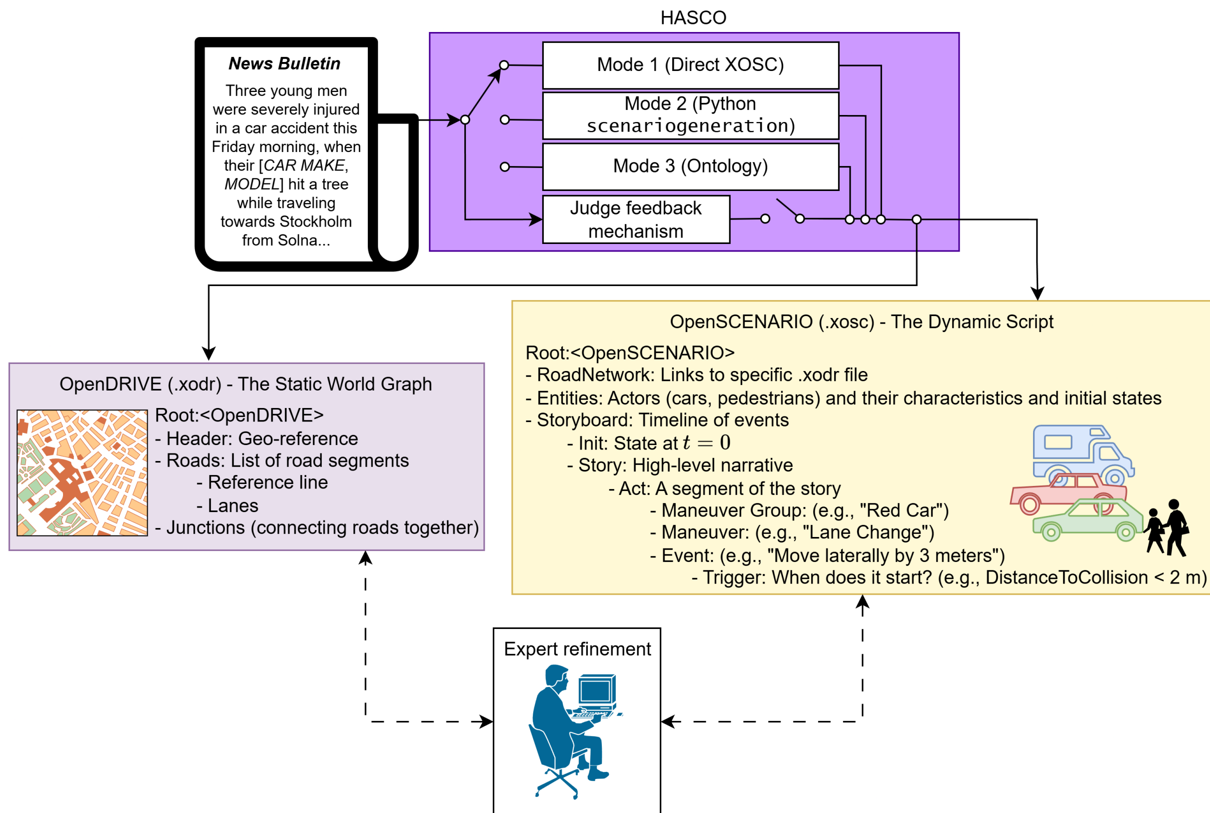


Figure 11.1: High-level architecture of the HASCO pipeline.

vers like cut-ins and following a vehicle ahead." This level corresponds directly to the nature of news reporting, where the emphasis is on semantic understanding rather than mathematical precision. As the level of detail increases, the machine readability increases. The hierarchy progresses to **logical scenarios**, defined by parameter ranges and distributions and finally to **concrete scenarios**, which are defined by exact parameter values.

The "specification gap" addressed in this work can be understood as the translational distance between these levels. News articles are rich in context but underspecified. Conversely, standards like OpenSCENARIO typically operate at the concrete layer, requiring precise coordinates, trajectories, and triggers to function. This work links these varying aspects by processing the linguistic descriptions of functional scenarios and refining them, alongside imputation, into the exact parameters required for concrete, executable simulations.

11.1.2 The need for code co-generation

While the ultimate ambition of generative AI in automotive engineering is fully automated scenario synthesis, current limitations necessitate a more guarded approach. Manual translation of

forensic reports into simulation is prohibitively slow, yet direct end-to-end generation by LLMs remains unreliable for safety-critical applications. As noted by Nouri et al. [10], while LLMs demonstrate significant potential in accelerating software development, their stochastic nature introduces distinct risks, including hallucinations and non-compliance with safety requirements. Consequently, relying on them as autonomous creators of forensic scenarios is insufficient.

To address this, we adopt the "code co-generation" paradigm, positioning HASCO as a "Human-in-the-Loop" compiler. This methodology aligns with the "Generate fast, Eliminate fast" principle proposed by Nouri et al., which emphasizes integrating rapid, automated sanity checks directly into the generation pipeline to filter out invalid outputs before they reach human review. By subjecting generated artifacts to preliminary assessment via a deterministic feedback loop (Software-in-the-Loop simulation), we mitigate the risk of generating syntactically valid but semantically illogical code.

In this framework, the role of the LLM shifts from an autonomous agent to a semantic compiler. By automating the syntactic complexity of OpenSCENARIO (the "boilerplate"), HASCO allows human experts to shift their focus from debugging XML tags to validating the semantic fidelity of crash parameters. This "Human-in-the-Loop" workflow reduces the reconstruction burden from manual coding to mere high-level overview and supervision.

To validate this paradigm, we conduct a dual-layer analysis on real-world accident reports. We evaluate performance not only on executability (the rate of valid code generation) but also on semantic fidelity, utilizing a graded qualitative rubric to measure the alignment between the generated simulation and the original forensic narrative. We demonstrate that high executability does not guarantee forensic accuracy, reinforcing the necessity of the expert-in-the-loop for final semantic validation.

To address this gap and validate the "Code Co-Generation" paradigm proposed in this work, we formulate the following research questions:

- **RQ1 (Feasibility)**: To what extent can an LLM-based tool automate the generation of syntactically valid OpenSCENARIO files from unstructured forensic text?

- **RQ2 (Fidelity)**: How does the integration of a deterministic validation mechanism (the Judge) impact the semantic fidelity of the generated scenarios compared to direct "one-shot" LLM synthesis?

11.2 Background

11.2.1 The OpenSCENARIO standard, the `scenariogeneration` library and esmini

ASAM OpenSCENARIO serves as the industry standard for describing dynamic content in driving simulations. OpenSCENARIO XML (v1.x) relies on a rigid, hierarchical Storyboard architecture to orchestrate events within a static map definition. The map definition is part of the corresponding OpenDRIVE standard. A scenario is not merely a sequence of frames, but a complex state machine where `Acts`, `Maneuvers`, and `Events` are triggered by specific run-time conditions (e.g., *"start lane change when Time-to-Collision < 2s"*).

This structure creates a high degree of syntactic coupling. A single semantic action, such as an "overtaking maneuver," requires synchronized updates across multiple nested tags, such as defining trajectories, speed profiles and trigger conditions that must remain mathematically consistent relative to the static road network. The existence of these dependencies pose a specific challenge for probabilistic models like LLMs, which are prone to losing context over long token sequences.

Manually generating machine-readable `.xosc` and `.xodr` files is highly complex. A useful intermediary framework for generating OpenSCENARIO files is the `scenariogeneration` Python package [11]. This package provides bindings between Python and the standard, enabling the generation of `.xosc` and `.xodr` files using a well-established and user-friendly syntax.

To execute these files, we utilize esmini [12], the standard open-source reference implementation. The esmini tool functions not only as a visualizer but as a deterministic constraint solver. Unlike high-fidelity simulators such as CARLA [13] or LGSVL [14], which prioritize realism and sensor rendering and require significant computational overhead, esmini focuses exclusively on logical validation and trajectory execution. This lightweight, headless architecture allows for fast validation cycles, making it the ideal simulator for our iterative feedback loop. In our pipeline, esmini serves as the ground-truth validator: if the generated XML contains logical contradictions or syntax errors, esmini provides the explicit failure signals required for the feedback loop.

11.2.2 Scenario-based testing and reconstruction

The validation of ADAS/AD systems has shifted from distance-based validation (driving billions of miles) to Scenario-Based Testing (SBT). As established by Wachenfeld et al. [3], physical test driving alone is statistically infeasible for validating safe autonomy due to the rarity of critical events. To verify safety, manufacturers must therefore test systems against rare, safety-critical events (more specifically, accidents) which are statistically scarce in naturalistic driving data [4, 15].

This methodology was standardized by the PEGASUS project [16], as part of which Menzel et al. established the aforementioned widely adopted layer model (functional, logical, concrete) for highway automation. Its successor, the VVM project [17], subsequently extended these verification frameworks to complex urban environments, cementing simulation as the cornerstone of industrial safety argumentation.

Currently, populating these test libraries relies on two primary methods:

1. **Manual Engineering:** Experts manually script scenarios using simulation environment tools (e.g., IPG CarMaker, dSPACE ASM Traffic, VIRES Virtual Test Drive). This ensures high fidelity and expert control but is unscalable and labor-intensive, as every parameter must be explicitly defined and maintained [8, 18].

2. **Sensor-Based Reconstruction:** Pipelines that convert log data (Lidar/Camera) from test vehicles into simulation files. While accurate, this method is strictly limited to events the fleet has actually encountered during operation [19].

This creates a "data scarcity" gap, as thousands of critical crashes occur daily and are documented in textual reports (police records, news), but this data remains inaccessible to simulation pipelines due to its unstructured nature. Automating the translation of these texts into executable scenarios offers a scalable third path for generating the critical edge cases required for robust ADAS validation [6, 20].

11.3 Related Work

Recent works have attempted to automate the pipeline of transforming human-readable functional scenarios into machine-readable logical scenarios using Large Language Models (LLMs). Deng et al. [21] introduced TARGET, which generates test scenarios from traffic rules. While

effective for regulatory compliance, TARGET focuses on explicit rule violations rather than the complex, causal narratives inherent in accident reconstruction. Similarly, Zhao et al. [22] proposed Chat2Scenario, utilizing LLMs to extract scenario parameters from naturalistic datasets. Crucially, however, Chat2Scenario is primarily an extraction framework; it does not address the "syntactic engineering" challenge of generating valid, complex OpenSCENARIO XML structures from scratch when no pre-existing dataset match exists.

Closer to our domain, Elmaaroufi et al. [23] introduced ScenicNL to convert police reports into simulations. Their approach targets the *Scenic* probabilistic programming language. While Scenic is concise, the automotive industry standard remains ASAM OpenSCENARIO (XML), which is significantly more verbose and prone to long-dependency syntax errors that ScenicNL does not address.

Guo et al. [6] and Ji et al. [20] utilize LLMs to parse reports into OpenSCENARIO, but treat the extraction merely as a seed for fuzzing. Txt2Sce mutates scenarios for stress-testing, and SoVAR prioritizes generalization over forensic reconstruction. Our approach differs fundamentally in intent: HASCO is a forensic reconstruction tool. We do not aim to diverge from the text via mutating instances, but to converge on it via validation.

Finally, Nouri et al. [10] highlight the stochastic risks of using LLMs for safety-critical software, arguing that LLMs should be treated as "Code Co-generators" rather than autonomous agents. We adopt this philosophy directly. While the works above address parts of the pipeline, none successfully integrate a *deterministic validation loop* (a Judge) to automatically correct the specific syntactic and semantic hallucinations common when generating verbose OpenSCENARIO XML from forensic text. More specifically, a significant gap remains in ensuring semantic fidelity of forensic reconstructions, targeting that the generated simulation adheres not only to the laws of physics but to the specific causal narrative of a news report.

11.4 Methodology: the HASCO compiler

We introduce HASCO (Hybrid AI Simulation Compiler), a command-line interface (CLI) based toolchain designed to bridge the specification gap between unstructured traffic reports and executable OpenSCENARIO files. The architecture is composed of four synchronized pipelines: the Geospatial Pipeline for static world generation, the Vehicle Extraction Module for actor dimensioning, the Semantic Pipeline for dynamic scenario synthesis, and the Validation Loop for

reaching forensic fidelity. The pipelines in this work were executed using OpenAI's **gpt-5-mini** LLM, but remain model-agnostic, and other models may be utilized. The utilization of different models may lead to varying token counts, due to varying underlying tokenizer architectures.

11.4.1 Static world generation (geospatial pipeline)

To ensure environmental exactness, HASCO first extracts location strings from the report and queries the Nominatim API for geocoordinates. It then downloads the corresponding OpenStreetMap (OSM) data for the defined bounding box. This OSM data is converted into an OpenDRIVE (`.xodr`) network using a dockerized CARLA conversion script, ensuring the static road topology matches the real-world location described in the text. Finally, the XODR map is "summarized" into a textual description that gets passed later into a combined prompt for the scenario synthesis pipeline. The pipeline may be observed at Figure 11.2.

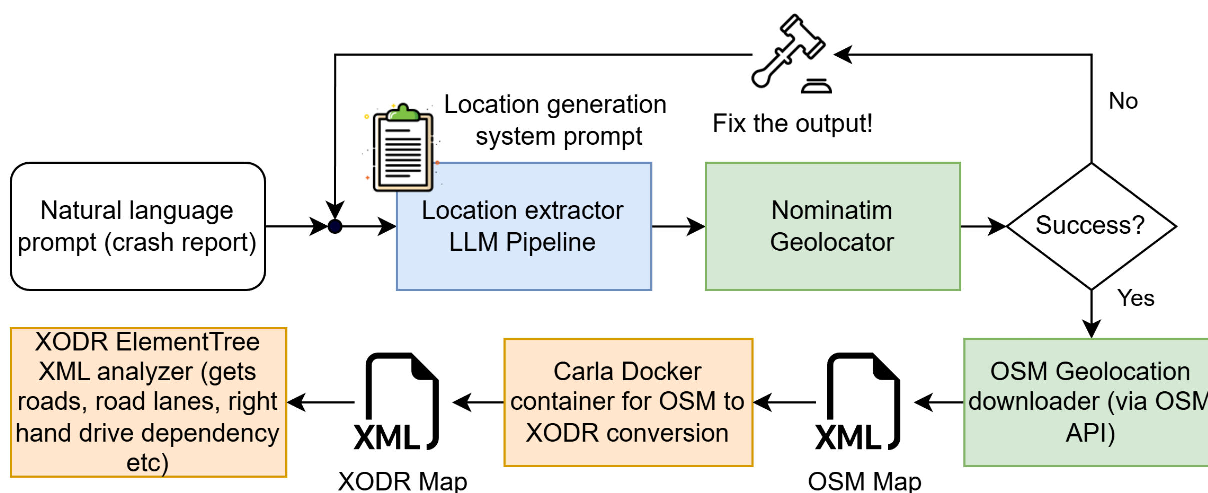


Figure 11.2: Geospatial pipeline

11.4.2 Vehicle extraction module

In parallel with this, the vehicle extraction module resolves actor dimensions. Rather than relying on a static lookup table, this module operates dynamically per scenario. It parses vehicle descriptions (e.g., "2015 Volvo V60" or "small silver hatchback") and queries an external knowledge base from Wikipedia to retrieve precise physical dimensions (length, width, wheelbase). For vague descriptions, it approximates dimensions based on the nearest real-world vehicle class, ensuring that collision physics in the simulation are grounded in realistic bounding

boxes. This output too, in the form of an OpenSCENARIO vehicle catalog, is passed to the combined prompt for later use. The pipeline may be observed in Figure 11.3.

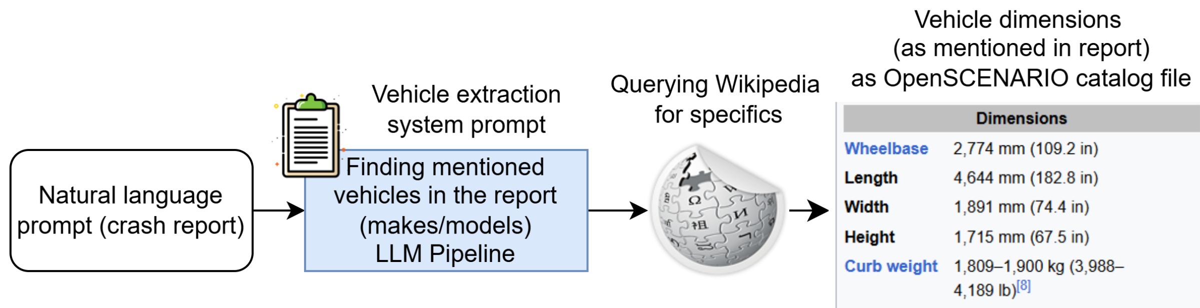


Figure 11.3: Vehicle extraction module

11.4.3 Dynamic scenario synthesis (semantic pipeline)

Simultaneously, the system synthesizes the dynamic scenario, the temporal script that dictates the behaviors of moving actors, their specific maneuvers and event triggers over time. To ensure the generated code is both syntactically valid and compliant with industry standards, the pipeline employs a hybrid Retrieval-Augmented Generation (RAG) module. RAG enhances the language model's accuracy by actively retrieving relevant technical documentation to ground its output, pulling from a composite knowledge base containing:

- The ASAM OpenSCENARIO Standard (v1.x) schema definitions,
- The esmini documentation and validator rules,
- The complete `scenariogeneration` Python library repository.

A re-ranking mechanism filters retrieved snippets to prioritize pertinent API calls (e.g., specific `ManeuverGroup` classes), which are then injected into the system prompt to guide the LLM's code generation.

11.4.4 Multi-strategy synthesis

To evaluate the trade-off between various levels of scenario representation, HASCO implements three distinct synthesis strategies. Each strategy operates on a different level of specification, employing a unique feedback loop to ensure validity before the final output reaches the Judge.

The structural overview of these different strategies is visible in Figure 11.4, and the execution in Figure 11.5.

1. **Direct syntactic synthesis:** The LLM attempts to generate raw ASAM OpenSCENARIO (`.xosc`) XML. The output is fed immediately into **esmini** in headless mode. If parsing fails (e.g., missing parameter definitions), the error log is fed back to the LLM for self-correction.

2. **Programmatic synthesis using Python:** The LLM generates a Python script utilizing the `scenariogeneration` library. This acts as an IR of a scenario. The script is executed in a sandboxed environment; if the Python interpreter throws an error (e.g., `AttributeError`), the traceback is returned to the LLM to repair the API usage.

3. **Ontology-driven synthesis:** To accommodate the difference between unstructured text and rigid syntax, we employ a semantic intermediate layer adapted from the framework by Bogdoll et al. [24], a choice made based on analyzing the ontology framework survey from Zipfl et al. [25] and selecting a framework that would match the needs of our generative pipeline without being overly detailed. We distilled a lightweight JSON-LD (JSON for linked data) profile focusing on core classes (`Actor`, `Event`, `StartPose`). This mode uses a deterministic SHACL [26] (Shapes Constraint Language) data consistency validation loop:

    - *Generation:* The LLM outputs a JSON-LD instance representing the semantic graph of the crash.
    - *Validation:* The instance is validated against a SHACL schema (`shapes.ttl`) to ensure logical consistency (e.g., verifying that all actors referenced in events exist in the actor list).
    - *Feedback:* Specific violations are injected into the next prompt. Only upon passing semantic validation is the graph passed to the heuristic translation layer for compilation.

11.4.5 The heuristic translation layer

A significant component of the ontology mode for converting from the ontology instantiation to an XOSC is the decoupling of semantic intent from syntactic implementation. Once a valid

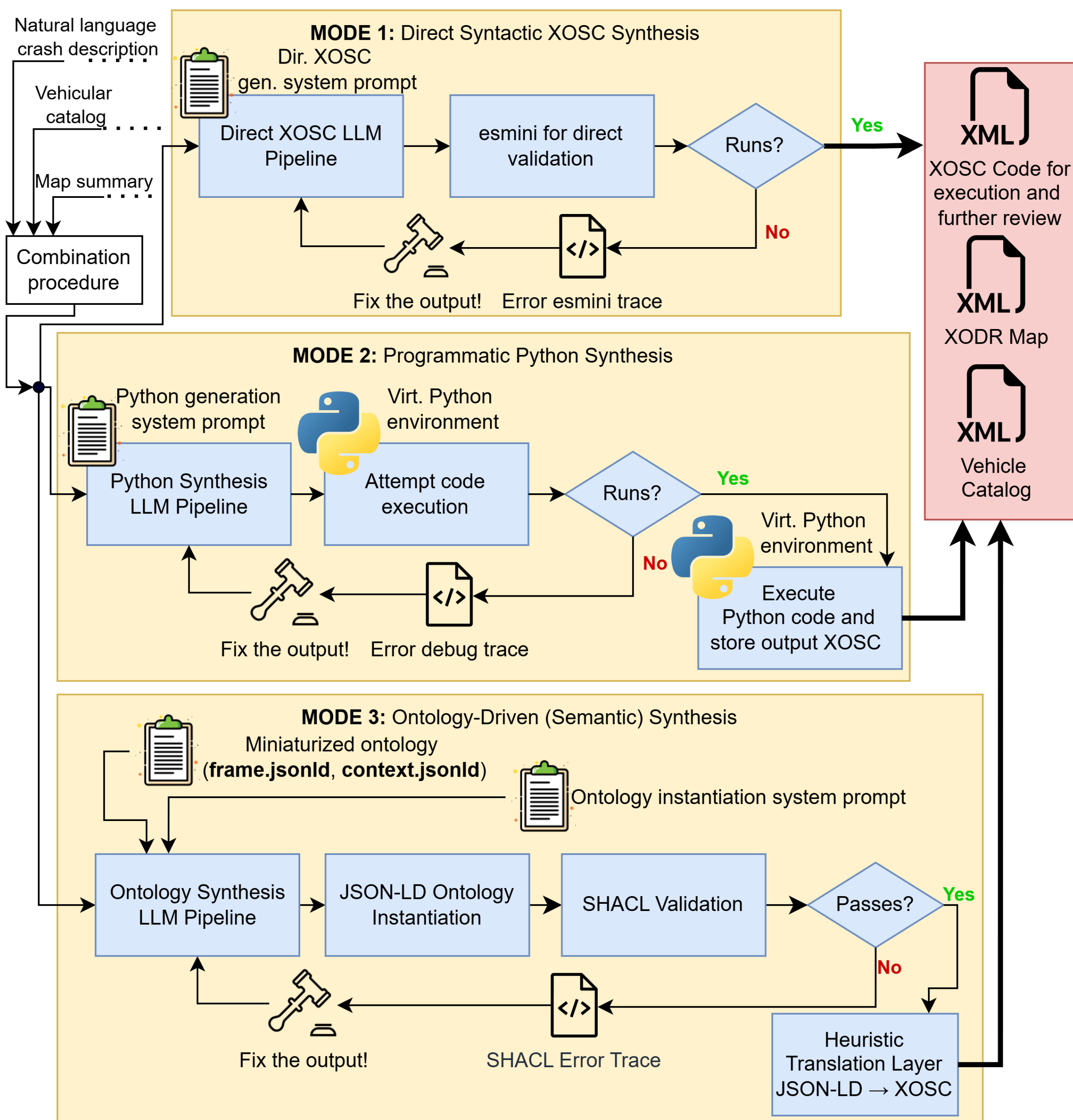


Figure 11.4: The Multi-Strategy Synthesis Architecture. HASCO accepts a unified prompt and routes it through one of three synthesis strategies. While Mode 1 (XML) and Mode 2 (Python) rely on execution-based feedback loops (via esmini and the Python interpreter, respectively), Mode 3 (Ontology) employs a semantic constraint loop (via SHACL) before compiling the intent into simulation code. All paths converge at the Judge for final forensic verification.

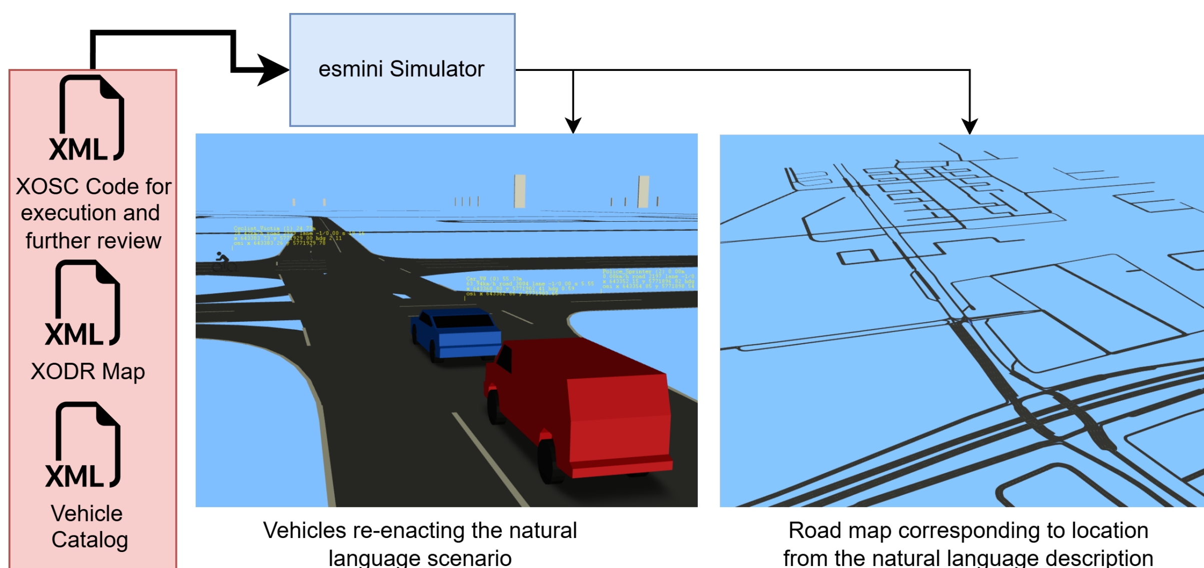


Figure 11.5: Overview of esmini being utilized to simulate a vehicular scenario. The `.xosc` is passed as a "story" of what happened in the scenario, in which way and by what actor; the `.xosc` vehicle catalog details the actors involved and the `.xodr` file represents the bounding-box map of where the incident took place, reconstructed in XML formatting.

JSON-LD instance is produced, HASCO employs a deterministic compiler to convert the high-level event graph into a concrete OpenSCENARIO XML file. This layer handles the low-level syntactic engineering that LLMs typically struggle with.

The translation layer parses the sequence of Event nodes for each actor. Unlike direct generation, which must output frame-by-frame coordinates, our compiler reconstructs a continuous polyline trajectory from sparse keyframes. It extracts the StartPose ($t = 0$) and subsequent event timestamps (t_n). If an event implies a lane change (inferred via regex matching of laneId tokens in the event summary), the compiler automatically injects intermediate waypoints to smooth the lateral transition, ensuring kinematic feasibility without requiring the LLM to calculate vector math.

To translate natural language summaries into executable triggers, the layer utilizes a keyword-based heuristic classifier:

- **Stop Logic:** Events containing tokens such as *"stop"*, *"stationary"*, or *"halt"* are automatically mapped to an OpenSCENARIO SpeedAction with a target speed of 0.0.

- **Crash Inference:** Events containing *"collision"*, *"impact"*, or *"rear-end"* (provided they lack disqualifiers like *"near-miss"*) are treated as stop triggers.

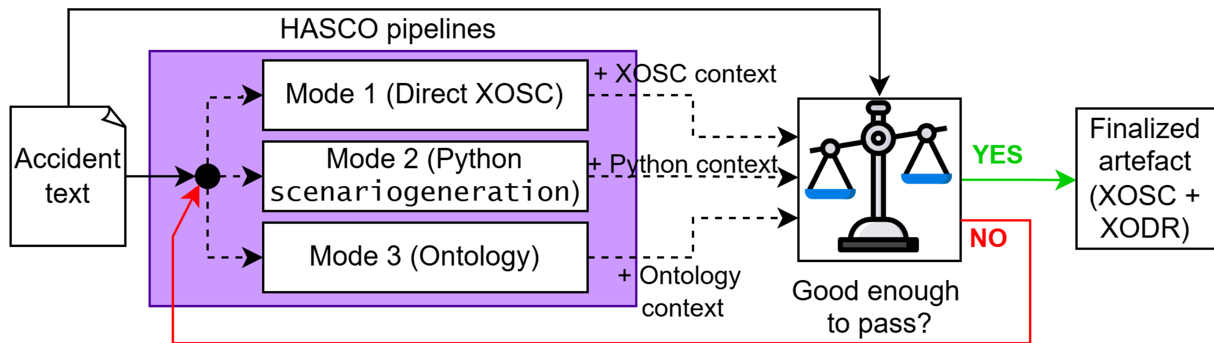


Figure 11.6: Forensic judge procedure in HASCO

This compiler ensures that the complex ConditionGroup and SimulationTimeCondition XML blocks required to synchronize these actions across multiple actors are mathematically consistent, effectively functioning as a "type system" for physical interactions.

11.4.6 The forensic judge (validation loop)

A critical innovation of HASCO is the Judge loop, which operates as a self-healing feedback mechanism to enforce both runtime robustness and forensic fidelity. The process is visualized in Figure 11.6.

Unlike "fire-and-forget" generators, this component employs a two-stage validation process:

1. **Syntax and runtime repair:** Before the scenario is semantically evaluated, HASCO attempts to compile the generated artifact (Python or Ontology) into a `.xosc` file. If the generation process throws a runtime exception or violates static guardrails (e.g., using invalid enums), the error trace is immediately fed back to the synthesis model for repair. This ensures that the system recovers from brittle syntax errors before they cascade into downstream failures.

2. **Semantic and functional alignment:** Once a valid `.xosc` artifact is produced, it is parsed into a structured event log (e.g., *"Actor A: ChangeLane at* $t = 5s$*"*). The Judge LLM compares this log against the original police report. Crucially, this step detects silent failures, scenarios where the code executes successfully but produces an empty or inert simulation (e.g., actors spawning but never moving). If the Judge detects a misalignment or a functional void, it generates a natural language critique (e.g., *"The simulation is inert; the report describes a collision, but no trigger was activated."*). This critique acts

as a semantic constraint for the next synthesis iteration, often converting a "technically valid but broken" scenario into a fully executable one.

11.5 Evaluation methodology

We evaluated HASCO on a diverse "stress test" dataset of $N = 40$ unstructured accident reports collected from open-web sources. Unlike curated datasets (e.g., NMVCCS) which provide structured metadata, our corpus consists of raw, unstructured text scraped from news articles, legal blogs, and public police logs. This dataset was specifically curated to maximize entropy and test the system's contextual imputation capabilities.

- **Linguistic Diversity:** The corpus includes reports in English, German, Swedish, Dutch, French, Japanese, Spanish and Bosnian, requiring the semantic pipeline to perform cross-lingual extraction and reasoning.
- **Information Sparsity:** Reports vary from hyper-detailed legal texts to vague news summaries (e.g., "A car hit a pedestrian near the station"). This forces the system to impute missing parameters (e.g., vehicle dimensions, exact speeds) based on context rather than explicit retrieval.

Representative examples of input reports and their corresponding generated artifacts are provided in Appendix A/B.

11.5.1 Quantitative analysis: executability

We first analyzed binary executability (pass/fail) and failure causes to quantify the validity of the overall procedure and measure the Judge's contribution. Table 11.1 presents the success rates across the three synthesis strategies. The results strongly validate the "Intermediate Representation" hypothesis.

Direct LLM generation (Mode 1) proved unreliable for complex scene engineering, suffering a 95% failure rate. The verbose nature of OpenSCENARIO XML led to frequent "long-dependency" errors, where the model failed to synchronize coordinate systems across nested `ManeuverGroup` tags. In contrast, the Ontology-guided compiler (Mode 3) achieved 95% validity. By offloading the syntactic complexity to the deterministic heuristic translation layer, the LLM only needed to produce a valid JSON graph. The remaining 5% of failures were attributed

Table 11.1: Executability Success Rates ($N = 40$).

Synthesis Strategy	No Judge (%)	Judge (%)	Judge Impact (%)
Direct XOSC (Mode 1)	5.0	7.5	**+2.5**
Intermediate Python (Mode 2)	80.0	90.0	**+10.0**
Ontology-driven (Mode 3)	92.5	95.0	**+2.5**

to external geocoding timeouts (Nominatim API) rather than generation logic errors; resolving these infrastructure issues could potentially raise this method to near 100% executability. Two notable examples of outright failures across the board are the report in Japanese (which failed to geocode in all pipelines, owing to the language barrier for the Nominatim OSM pipeline) and another report in Spanish (comprising of a complex textual report) which failed compilation across the board due to failures to extract the location, mentioned vehicles and the events that occurred.

The addition of a Judge Loop improved executability modestly across modes 1 and 3, but the effect was most pronounced in the Python mode (+10.0%). This gain is attributed to the Judge's ability to detect silent failures, scenarios where the generated Python script executes successfully (i.e., no runtime exception) but produces a functionally inert or empty OpenSCENARIO file. By critiquing these empty outputs (e.g., *"The scenario duration is 0s"*), the Judge essentially acts as a **semantic linter**, forcing the model to correct the underlying logic during the retry phase and converting invalid outputs into executable artifacts.

11.5.2 Qualitative analysis: semantic fidelity

To quantify the performance of HASCO beyond a binary pass/fail, we adopted a 4-point Likert scale (0–3) evaluating the simulation's utility. This rubric classifies scenarios based on the intervention required to make them usable:

- **Bin 0: Syntax failure (indication of compilation error).** The pipeline failed to produce a valid, runnable `.xosc` file. We categorize these failures into two distinct groups: *infrastructure errors* (e.g., geocoding API timeouts) and *generative failures*. The latter reveals specific limitations in the LLM's ability to generate rigid XML structures, which we classify as a "Failure mode catalogue" for future research:

  Structural malformation (broken XML format): The LLM fails to generate a well-

formed XML tree, often missing closing tags or root definitions due to context window limits. For example, the parser returns "Couldn't find OpenSCENARIO element," indicating a truncated file. This may be potentially fixed by enforcing constrained decoding (grammar-based sampling) or increasing token limits for verbose XML output.

Asset hallucination (catalog integration failures): The model generates syntactically valid code that references non-existent external assets, failing to ground the generation in the local file system. One example of this is a log showing "Couldn't locate catalog file: VehicleCatalog," because the model hallucinated a standard library path. A potential fix is to explicitly inject available asset paths and catalog entry names into the system prompt context.

Simulator incompatibility (semantic violations): The model generates valid standard OpenSCENARIO, but utilizes conditions or actions not supported by the specific target simulator (e.g., `esmini`). One common error of this type is "Exception: Unsupported condition: SimulationTimeCondition", occurring because the engine does not implement that specific standard feature. The model may be fine-tuned or prompt-engineered specifically on the target simulator's documentation rather than the general ASAM standard.

Convergence failure: Cases where the self-correction loop exhausted all attempts without reaching a valid state. A common instance is that the pipeline times out after the maximal number of retries, unable to resolve a complex dependency chain. One way of remediating this may be to decompose complex scenarios into smaller, modular generation steps rather than attempting single-shot generation.

- **Bin 1: Semantic failure (narrative/physics hallucination).** The scenario executes, but the core causal event is missing or fundamentally incorrect. For example, the report describes a T-bone collision at an intersection, but the simulation depicts two vehicles passing on a highway without contact.

- **Bin 2: Draft quality (necessitates a revision by a Human-in-the-Loop).** The scenario captures the correct core event, but exhibits kinematic artifacts or asset mismatches. For example, the collision occurs as described, but the target vehicle "teleports" (jerky movement) or the vehicle type is generic (e.g., a sedan instead of a bus). These files are usable but require manual "polishing", either via a GUI tool or by manually editing the generated XML.

- **Bin 3: Production quality (One-Shot).** The scenario is semantically faithful, kinematically smooth, and requires no or very little human intervention. Turns taken, events and timing align with the report; the collision physics are plausible, with potentially minor manual expert updates.

The results are visible in Table 11.2, and the distribution is visualized in Figure 11.7. These results give us key insights into the qualitative benefits of various HASCO modalities. To begin with, we may observe that Mode 1, both with Judge and without Judge, despite initially having non-zero executability, displays no possibility of achieving syntax compatibility with OpenSCENARIO. Those rare several instances where Mode 1 executes result in a static environment only determined by the `.xodr` map file within esmini. Therefore, the entire generative corpus of Mode 1 within HASCO may be discarded (bin 0), with the conclusion that recreating vehicular scenarios by way of generating direct `.xosc` files is not a task achievable without great difficulty. Modes 2 and 3 provide much more interesting nontrivial results.

Half of all the scenarios in Python mode without Judge bin as either Draft (42.5%) or Production quality (7.5%). Generation quality rises when the Judge is added into the pipeline, with Draft bin encompassing **32.5%** of cases and Production bin encompassing **30%** of total cases. Therefore, with Judge, a total of 62.5% cases are of "acceptable" Draft or Production quality. As per Figure 11.8 on the left, this indicates that adding the Judge significantly increases the quality of produced scenarios via Python intermediate representation, with a 300% relative increase in Production quality scenarios when compared to the non-Judge mode.

With regards to the Ontology mode, the non-Judge version initially bins 37.5% of all scenarios as either Draft (17.5%) or Production quality (20%), a result worse than Python, but shows a significant uptick when Judge is introduced, with acceptable case numbers going to 57.5% of either Draft (35%) or Production quality (22.5%). As visualized on Figure 11.8 on the right, the Judge reduces the amount of scenarios that bin as semantic failures by 36.3%, a significant drop.

11.5.3 Cost-benefit analysis

We tracked token consumption to determine the cost-fidelity trade-off (Figure 11.9). Here, the quality score S_{avg} is a weighted mean of the rubric bins ($S_{avg} = \frac{1}{N}\sum w_i n_i$, with weights $w \in \{0, 1, 2, 3\}$ corresponding to syntax failure, semantic failure, draft, and production quality). Figure 11.9(A) shows Ontology mode achieving the most favorable trade-off. Figure 11.9(B)

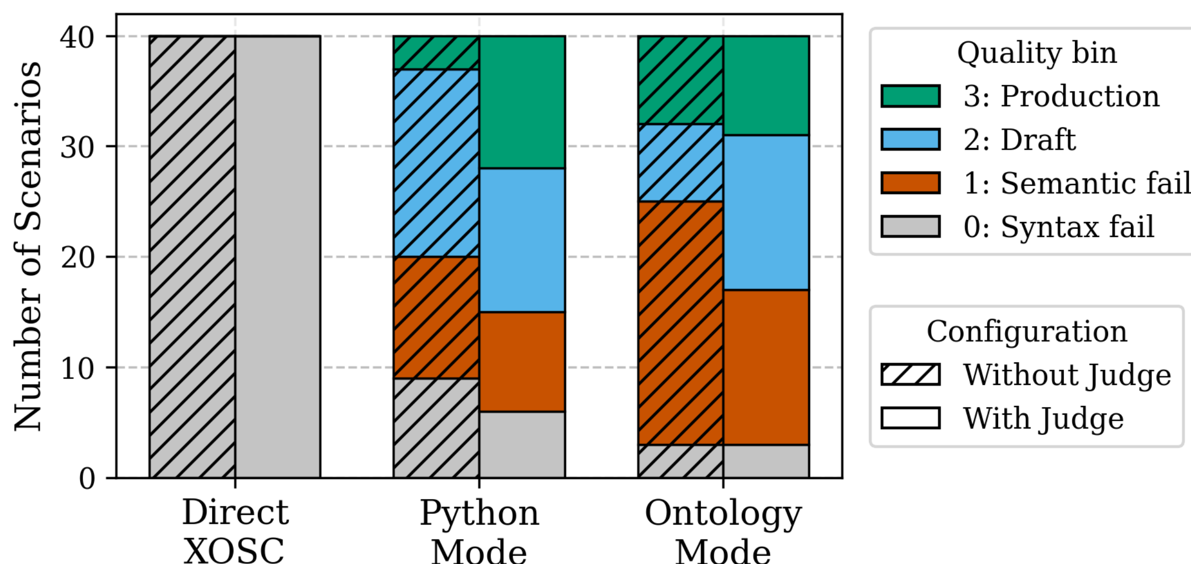


Figure 11.7: Qualitative distribution of generated scenarios ($N = 40$). In Python Mode, the Judge loop successfully converts Draft quality (Bin 2) scenarios into Production quality (Bin 3), tripling the high-fidelity yield. In Ontology Mode, the Judge reduces Semantic failures (Bin 1), shifting the distribution toward valid execution. Direct XOSC remains dominated by Syntax failures (Bin 0) regardless of validation.

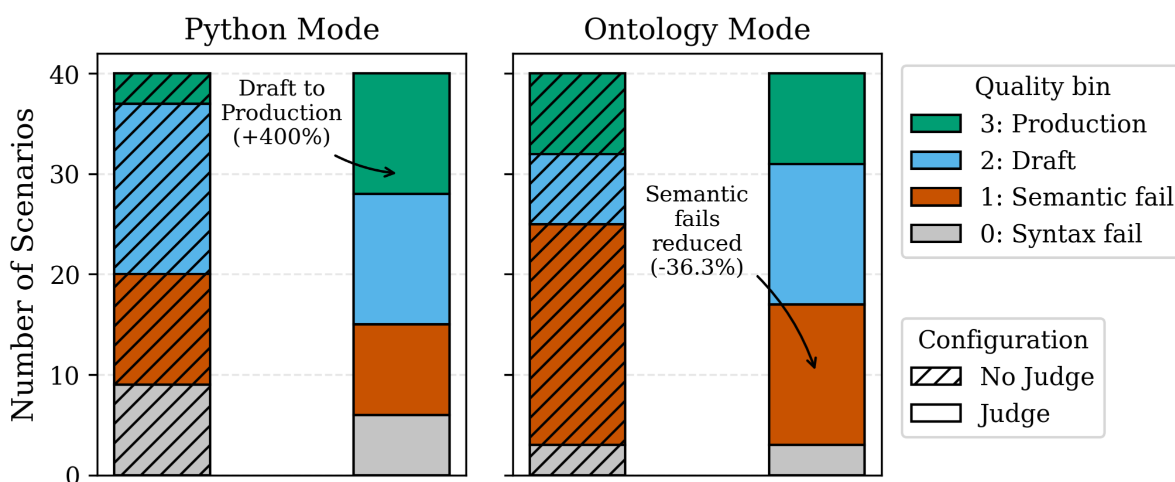


Figure 11.8: Qualitative impact of the Judge Loop on scenario fidelity ($N = 40$). The stacked bars illustrate the shift in quality distribution when the Judge loop is enabled. Left (Python mode): The Judge primarily acts as a physics refiner and converts technically functional "Draft" scenarios into "Production" quality (a 400% increase) by correcting simulation parameters. Right (Ontology mode): The Judge significantly reduces Semantic failures (-36.3%).

Table 11.2: Qualitative rubric results of HASCO across different operating modalities ($N = 40$).

Synthesis Strategy	Bin 0 (Syntax failure)	Bin 1 (Semantic failure)	Bin 2 (Draft quality)	Bin 3 (Production quality)
XOSC - Judge	40	0	0	0
XOSC - NO Judge	40	0	0	0
Python - Judge	6	9	13	12
Python - NO Judge	9	11	17	3
Ontology - Judge	3	14	14	9
Ontology - NO Judge	3	22	7	8

illustrates the executability rate, where Ontology mode again excels. The Judge loop provides relatively modest improvements for both Python and Ontology modes given the outsized $\approx 2.5\times$ token cost and $\approx 3\times$ latency increases (averaging 180 s per run vs. 60 s). Thus, for large-scale generation, Mode 3 (Ontology, No Judge) is the most cost-effective configuration, reserving the expensive Judge loop for complex edge cases (e.g., scenarios that execute but remain semantically inert).

11.6 Threats to validity

Subjectivity in evaluation. While the executability metric (bin 0) is deterministic, the semantic fidelity metric (bins 1–3) currently relies on human expert review and the "Judge" LLM's assessment. Both introduce subjectivity. The Judge LLM itself may hallucinate "alignment" (reporting OK when the simulation is flawed) or fail to catch subtle kinematic errors (e.g., impossible stopping distances). In cases of ambiguous reports, determining whether an imputed parameter (e.g., the specific lane index) is "correct" is inherently subjective. Future work will integrate deterministic safety metrics (e.g., responsibility and sensitive safety aware checks) to mathematically validate that the synthesized maneuvers align with physical crash causality, reducing reliance on qualitative grading.

Non-determinism in LLM outputs. LLMs utilize non-deterministic token sampling governed by a *temperature* parameter. Higher temperatures increase randomness, while a temperature of zero forces deterministic, greedy decoding. However, as the model we utilized did not support

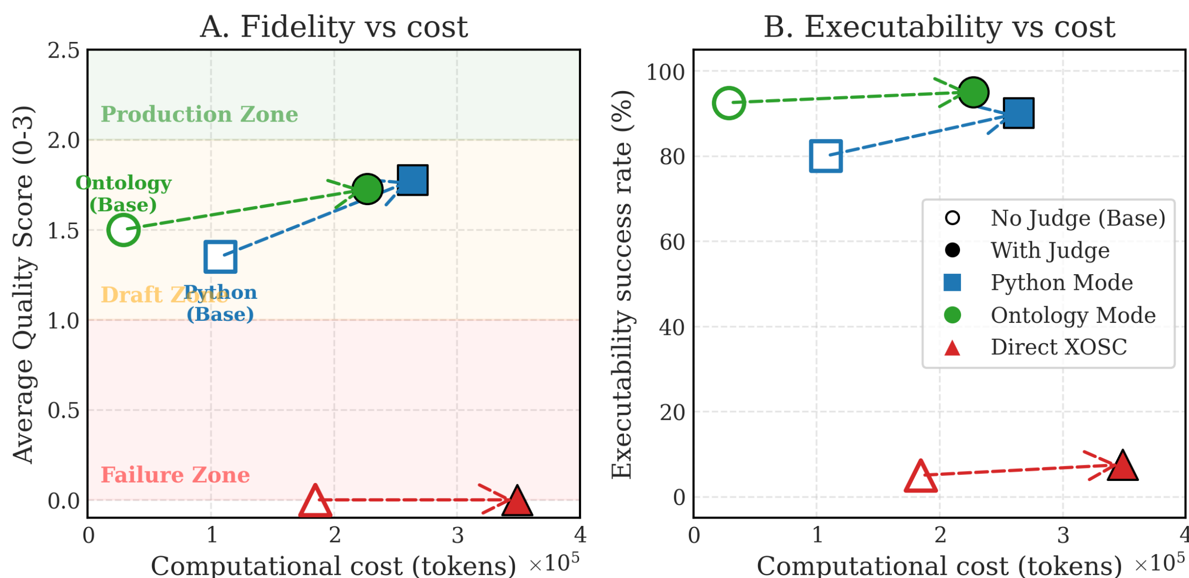


Figure 11.9: Cost-benefit analysis of synthesis strategies; the X-axis represents average token consumption. On (A) the Y-axis denotes the average quality score (S_{avg}). Ontology mode (Green) achieves the most favorable trade-off, delivering functional scenarios at minimal cost. On (B) the Y-axis denotes the executability rate.

modifying the temperature parameter, multiple runs across the same dataset will be executed as part of future research. Additionally, all experiments were conducted using a single LLM (gpt-5-mini). While the pipeline is model-agnostic by design, we did not evaluate whether performance characteristics generalize across model families (e.g., Claude, Gemini, open-weight models). A cross-model study is planned as part of future work to quantify how much of the observed executability improvement is attributable to the architectural decoupling versus the specific model's code-generation capability.

Imputation vs. accuracy trade-off. The use of unstructured data necessitates aggressive imputation. While this demonstrates the system's robustness, it creates a validity threat regarding ground truth. When HASCO encounters a report omitting specific details (e.g., "a truck" without a model year), it utilizes the vehicle extraction module to infer a "most likely" candidate (e.g., Generic Heavy Truck). While this allows for valid simulation, the resulting scenario is a probabilistic reconstruction rather than a forensic reproduction. Users must treat HASCO outputs as representative edge cases rather than exact digital twins of the specific historical event.

External dependencies. The system's fidelity is bound by the quality of external APIs. A valid scenario generation can still fail if the Nominatim geocoder fails to resolve a vague location

string, the OpenStreetMap data for a region is incomplete or lacks lane-level detail or simply the LLM generating request times out. These external failures account for the majority of the errors in our Ontology and Python modes and some of the errors in Direct XOSC mode. This underlines the need for offline fallback mechanisms in production environments.

11.7 Discussion

Our findings validate the "Code Co-Generation" paradigm by directly addressing our research questions through the data presented in Section 11.5. Regarding **RQ1**, Table 11.1 demonstrates that direct "one-shot" generation of verbose OpenSCENARIO XML is highly unreliable (95% failure rate). However, automated synthesis becomes highly feasible when utilizing an intermediate representation. By decoupling semantic intent from syntactic formatting—using intermediate Python scripts or JSON-LD ontologies, the LLM is relieved of complex XML synchronization. This allows the pipeline to achieve up to 95% executability. Therefore, we confirm that LLM-based automation is viable for unstructured text if the architectural abstraction is managed properly. Addressing **RQ2**, the integration of the Judge loop fundamentally shifts the pipeline from a fragile "fire-and-forget" system to a robust, self-healing process. As shown in Figure 11.8, the Judge significantly elevates semantic fidelity. In Ontology mode, it acted as a logical linter, reducing semantic mismatches by 36.3%. In Python mode, it refined physical parameters to promote scenarios from "Draft" to "Production" quality by 400%. Furthermore, it improved raw executability by catching and repairing "silent failures," proving that iterative feedback is essential for bridging the specification gap.

11.8 Conclusion and future work

In this work, we presented HASCO, a Human-in-the-Loop compiler designed to bridge the gap between unstructured forensic accident reports and executable OpenSCENARIO simulations. By treating scenario generation as a multi-strategy compilation task rather than a purely generative one, we demonstrated that utilizing Intermediate Representations (such as Python scripts and JSON-LD ontologies) alongside an iterative Judge feedback loop resolves the inherent brittleness of LLM-generated XML. This approach drastically reduces the manual engineering burden of Scenario-Based Testing while maintaining semantic fidelity.

Future work will focus on integrating constraint-based motion planning to enhance the kinematic realism of the heuristic translation layer, as well as introducing deterministic safety metrics to automatically quantitatively evaluate and verify the physical causality of the synthesized collisions.

Acknowledgment

We gratefully acknowledge the Swedish Knowledge Foundation via the PerFlex (*Performant and Flexible digital Systems through Verifiable Artificial Intelligence*) project, grant nr. 20220033, as well as Mälardalen University via the TSS-INSID (*In-Silico Driving Assurance Using Machine-Learning-Powered Verification and Synthesis for Safe Autonomous Vehicles*) project, of the Trusted Smart Systems initiative, which supported this research.

Bibliography

[1] Amy Aukema, Kate Berman, Travis Gaydos, Ted Sienknecht, Chou-Lin Chen, Chris Wiacek, Tim Czapp, and Schuyler St. *Real-world effectiveness of model year 2015-2020 advanced driver assistance systems*. 2023. (Visited on 04/17/2026).

[2] Hoseon Kim, Jieun Ko, Cheol Oh, and Seoungbum Kim. "Evaluation of autonomous driving safety by operational design domains (ODD) in mixed traffic". en. In: *Sustainability* 16 (22 Nov. 6, 2024), p. 9672. (Visited on 12/29/2025).

[3] Walther Wachenfeld and Hermann Winner. "The release of autonomous vehicles". In: *Autonomous Driving*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 425–449.

[4] Christian Neurohr, Lukas Westhofen, Tabea Henning, Thies de Graaff, Eike Mohlmann, and Eckard Bode. "Fundamental considerations around scenario-based testing for automated driving". In: *2020 IEEE Intelligent Vehicles Symposium (IV)*. 2020 IEEE Intelligent Vehicles Symposium (IV) (Las Vegas, NV, USA). IEEE, Oct. 19, 2020, pp. 121–127.

[5] Ji Zhou, Yongqi Zhao, Yixian Hu, Hexuan Li, Zhengguo Gu, Nan Xu, and Arno Eichberger. "Can AI generate more comprehensive test scenarios? Review on Automated Driving Systems test scenario generation methods". In: *arXiv [cs.SE]* (Dec. 17, 2025).

[6] An Guo, Yuan Zhou, Haoxiang Tian, Chunrong Fang, Yunjian Sun, Weisong Sun, Xinyu Gao, Anh Tuan Luu, Yang Liu, and Zhenyu Chen. "SoVAR: Build generalizable scenarios from accident reports for autonomous driving testing". In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE '24: 39th IEEE/ACM International Conference on Automated Software Engineering (Sacramento CA USA). New York, NY, USA: ACM, Oct. 27, 2024, pp. 268–280.

[7] ASAM e.V. *ASAM OpenSCENARIO®: Standard for Dynamic Content in Driving Simulation*. Version 1.2.0. Version 1.x (XML format). 2022. URL: https://www.asam.net/standards/detail/openscenario/.

[8] Till Menzel, Gerrit Bagschik, and Markus Maurer. "Scenarios for development, test and validation of automated vehicles". In: *arXiv [cs.SE]* (Jan. 5, 2018).

[9] Simon Ulbrich, Till Menzel, Andreas Reschka, Fabian Schuldt, and Markus Maurer. "Defining and substantiating the terms scene, situation, and scenario for automated driving". In: *2015 IEEE 18th International Conference on Intelligent Transportation Systems*. 2015 IEEE 18th International Conference on Intelligent Transportation Systems - (ITSC 2015) (Gran Canaria, Spain). IEEE, Sept. 2015, pp. 982–988.

[10] Ali Nouri, Beatriz Cabrero-Daniel, Zhennan Fei, Krishna Ronanki, Håkan Sivencrona, and Christian Berger. "Large Language Models in code co-generation for safe autonomous vehicles". In: *arXiv [cs.SE]* (May 26, 2025).

[11] pyoscx. *scenariogeneration: A Python library for generating OpenSCENARIO and OpenDRIVE files*. Version 0.12.0. Accessed: 2025-02-11. 2024. URL: https://github.com/pyoscx/scenariogeneration.

[12] esmini development team. *esmini: Basic OpenSCENARIO player and validator*. https://github.com/esmini/esmini. Accessed: 2026-01-20. 2025.

[13] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. "CARLA: An open urban driving simulator". In: *arXiv [cs.LG]* (Nov. 10, 2017). Ed. by Sergey Levine, Vincent Vanhoucke, and Ken Goldberg, pp. 1–16.

[14] Guodong Rong, Byung Hyun Shin, Hadi Tabatabaee, Qiang Lu, Steve Lemke, Mārtiņš Možeiko, Eric Boise, Geehoon Uhm, Mark Gerow, Shalin Mehta, Eugene Agafonov, Tae Hyung Kim, Eric Sterner, Keunhae Ushiroda, Michael Reyes, Dmitry Zelenkovsky, and Seonman Kim. "LGSVL simulator: A high fidelity simulator for autonomous driving". In: *arXiv [cs.RO]* (May 7, 2020).

[15] Stefan Riedmaier, Thomas Ponn, Dieter Ludwig, Bernhard Schick, and Frank Diermeyer. "Survey on scenario-based safety assessment of automated vehicles". In: *IEEE Access* 8 (2020), pp. 87456–87477.

[16] Hermann Winner, Karsten Lemmer, Thomas Form, and Jens Mazzega. “PEGASUS—first steps for the safe introduction of automated driving”. en. In: *Lecture Notes in Mobility*. Cham: Springer International Publishing, 2019, pp. 185–195. (Visited on 02/11/2026).

[17] Roland Galbas, Marcus Nolte, Ulrich Eberle, Hardi Hungar, Henning Mosebach, Nayel Fabian Salem, Helmut Schittenhelm, Jan Reich, Thomas Kirschbaum, and Lukas Westhofen. *VV methods safety assurance position paper*. en. Research rep. 2024.

[18] Gerrit Bagschik, Till Menzel, and Markus Maurer. “Ontology based scene creation for the development of automated vehicles”. In: *2018 IEEE Intelligent Vehicles Symposium (IV)*. 2018 IEEE Intelligent Vehicles Symposium (IV) (Changshu). IEEE, June 2018, pp. 1813–1820.

[19] Yuan Gao, Mattia Piccinini, Korbinian Moller, Amr Alanwar, and Johannes Betz. “From Words to Collisions: LLM-guided evaluation and adversarial generation of safety-critical driving scenarios”. In: *arXiv [cs.AI]* (July 18, 2025).

[20] Pin Ji, Yang Feng, Zongtai Li, Xiangchi Zhou, Jia Liu, Jun Sun, and Zhihong Zhao. “Txt2Sce: Scenario generation for autonomous driving system testing based on textual reports”. In: *arXiv [cs.SE]* (Sept. 2, 2025).

[21] Yao Deng, Jiaohong Yao, Zhi Tu, Xi Zheng, Mengshi Zhang, and Tianyi Zhang. “TARGET: Automated scenario generation from traffic rules for testing autonomous vehicles via validated LLM-guided knowledge extraction”. In: *arXiv [cs.SE]* (May 10, 2023). (Visited on 07/25/2025).

[22] Yongqi Zhao, Wenbo Xiao, Tomislav Mihalj, Jia Hu, and Arno Eichberger. “Chat2Scenario: Scenario extraction from dataset through utilization of large language model”. In: *2024 IEEE Intelligent Vehicles Symposium (IV)*. 2024 IEEE Intelligent Vehicle Symposium (IV) (Jeju Island, Korea, Republic of). IEEE, June 2, 2024. (Visited on 08/27/2025).

[23] Karim Elmaaroufi, Devan Shanker, Ana Cismaru, Marcell Vazquez-Chanlatte, Alberto Sangiovanni-Vincentelli, Matei Zaharia, and Sanjit A Seshia. “ScenicNL: Generating probabilistic scenario programs from natural language”. In: *arXiv [cs.SE]* (May 3, 2024).

[24] Daniel Bogdoll, Stefani Guneshka, and J Marius Zöllner. “One ontology to rule them all: Corner case scenarios for autonomous driving”. In: *Lecture Notes in Computer Science*. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023, pp. 409–425.

[25] Maximilian Zipfl, Nina Koch, and J Marius Zöllner. “A comprehensive review on ontologies for scenario-based testing in the context of autonomous driving”. In: *2023 IEEE Intelligent Vehicles Symposium (IV)*. 2023 IEEE Intelligent Vehicles Symposium (IV) (Anchorage, AK, USA). IEEE, June 4, 2023, pp. 1–7.

[26] Paolo Pareti and George Konstantinidis. “A review of SHACL: From data validation to schema reasoning for RDF graphs”. In: *Reasoning Web. Declarative Artificial Intelligence*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 115–144.

[27] Stefan Jacobs Madlen Haarbach. *Lieferwagen stand auf Radweg: Radfahrerin stirbt bei Unfall in Berlin-Friedrichshain – Mahnwache am Freitag*. [Online; accessed 2026-04-16]. May 2021. URL: `https://www.tagesspiegel.de/berlin/polizei-justiz/radfahrerin-stirbt-bei-unfall-in-berlin-friedrichshain--mahnwache-am-freitag-6174754.html`.

[28] Sebastian Pagel, Flo, Michi Scholz, Muhammed Kerem Kahraman, Pranav Ashok, victorjarlow, cxruan, Egan, Eric Xue, boettol, Emil Knabe, Hüseyin Aydın, and m0uH. *grepthat/libOpenDRIVE: v0.5.0*. Comp. software. 2023.

Appendix

11.9 Prompts and LLM Context

```
You are HASCo's ontology specialist. Convert police/incident reports
↪ plus OpenDRIVE context into detailed JSON-LD instances that conform to
↪ the provided context/frame and can later be rendered into OpenSCENARIO.

When the user prompt includes "### Reference docs ... ### End docs",
↪ treat those snippets as authoritative.

OUTPUT RULES
- Respond with EXACTLY one JSON object (no markdown fences, no prose
↪ before/after).
- Copy the @context block exactly as provided in the user prompt and
↪ keep all existing prefixes.
```

```
- Ensure the instance passes SHACL: include actors, assets, start poses,
↪ and events as required.
- Every actor must have a unique hasco: identifier, a concrete ontology
↪ class (Vehicle, Pedestrian, Bicycle, etc.), a catalog asset, and a
↪ startPose with xodrRoadId/laneId/s, headingDeg, speedMps.
- Events describe concrete interactions: include subject, target,
↪ timeSec, and eventSummary text that narrates the collision/interaction.
- Use OpenDRIVE IDs from the map summary (lane signs follow the local
↪ traffic side). Never invent roadId=0 or omit lane polarity.
- For multi-vehicle crashes, instantiate each participant separately
↪ (e.g., ego car, struck vehicle, bus, pedestrian).
- Populate eventSummary with meaningful prose so later tools can recover
↪ intent ("Tanker rear-ends slowing sedan near junction J2").
- Add additional derived actors if the report implies them (e.g., parked
↪ truck, waiting pedestrian, lead vehicle).
- Choose realistic speeds for the road type; if the report states
↪ someone stopped, set speedMps ≈ 0 and describe the reason in
↪ eventSummary.
- Prefer chronological ordering of events using timeSec.
- Never include TODOs or placeholders. If uncertain, pick the most
↪ defensible option and explain the assumption briefly in eventSummary.
```

Listing 11.1: System prompt for the ontology mode.

```
You are HASCo's OpenSCENARIO architect. Convert report text and map
↪ summaries directly into complete .xosc XML files.

OUTPUT RULES
- Respond with EXACTLY one XML document (no markdown fences, comments,
↪ or explanations).
- Use the literal map/catalogn paths provided in the user prompt without
↪ modification.
- Instantiate every actor implied in the report using
↪ <Entities><ScenarioObject> with CatalogReference entries from the
↪ provided catalog list.
- Include a rich <Storyboard> structure with Init → Story → Act →
↪ ManeuverGroup → Event(s), plus <StartTrigger> and <StopTrigger>.
- Actions must be deterministic: use SpeedAction, LaneChangeAction,
↪ FollowRelativeDistance, CollisionCondition, etc. to choreograph the
```

```
↪ described crash/interaction.
- Ground LanePosition/WorldPosition coordinates in the map context;
↪ never invent roadId=0 or impossible laneIds.
- Respect traffic-side semantics: negative laneIds for right-hand
↪ driving, positive for left-hand driving.
- Always stop or park every actor after the event concludes; include a
↪ simulation-wide StopTrigger.
- Reference catalog entries exactly; do not fabricate new vehicle
↪ definitions.
- When uncertain, choose the most believable interpretation and
↪ proceed--do not omit the participant.
```

Listing 11.2: System prompt for the direct XOSC mode.

```
You are an assistant specialized in generating OpenSCENARIO (.xosc)
↪ traffic scenarios in Python using the
scenariogeneration library (xosc/xodr).

You will receive a user request, and sometimes an inline section titled:
### Reference docs
...retrieved API snippets here...
### End docs

When present, treat the Reference docs as authoritative and prefer those
↪ APIs.

REQUIREMENTS
- Output: RETURN ONLY Python code (no markdown, no text before/after).
- Use scenariogeneration.xosc/xodr (and standard library) only.
- Load the map path provided in the user prompt (see "Map path") and do
↪ NOT build your own road graph:
    road = xosc.RoadNetwork(roadfile="<MAP PATH FROM PROMPT>",
↪ scenegraph="")
- Register the provided VehicleCatalog directory verbatim (e.g.,
↪ scenario.add_catalog("VehicleCatalog", "../catalogs")); never rewrite
↪ the path or point at the .xosc file directly.
- Include: at least one vehicle actor, an Init/setup, at least one
↪ Maneuver/Event/Action,
  and a simple Story/Act structure that reproduces the described
```

```
↪ scenario.
- When the report involves multiple vehicles, model each distinct
↪ participant (3+ if specified) instead of collapsing the crash to two
↪ actors; use CatalogReference entries for all of them.
- Avoid placeholders/dummies. Choose reasonable defaults if missing.
- Add light inline comments where helpful. When you reference an API,
↪ add a marker like <SpeedAction> or <LaneChangeAction> in a comment.
- No I/O except reading the .xodr via RoadNetwork and (optionally)
↪ writing the .xosc file.
- No external network calls, no shelling out.
- Start by inferring how every involved vehicle moves relative to the
↪ others (head-on, rear-end, crossing paths, same-direction merge, etc.)
↪ and pick roadId/laneId/orientations that physically match the report
↪ before adding Actions.
- If uncertain about specifics, make sensible assumptions and proceed
↪ (do NOT ask questions).
- If the report indicates a crash/collision/impact, YOU MUST choreograph
↪ a deterministic impact:
  * Drive the storyline through: approach (accelerate/close gap), impact
↪ (LaneChange/RelativeDistance trigger <=0.5 m or CollisionCondition),
↪ and aftermath (stop/park both entities).
  * Align geometry with the stated crash type: head-on crashes require
↪ opposing headings on the same road, rear-end crashes require a faster
↪ follower in the same lane, intersection/side impacts require
↪ perpendicular or turning approaches, etc.
  * Do not leave the crash chance-based--script the actions/timing so
↪ contact always occurs within the Story.
- Use Position objects
↪ (WorldPosition/LanePosition/RelativeObjectPosition). No tuples/strings
↪ for positions.
- Do NOT use strings for enums (use xosc.Rule.lessThan / greaterThan /
↪ equalTo).
- Do NOT subclass any ScenarioGenerator; build a plain xosc.OpenScenario.
- Do NOT redefine vehicle characteristics; load the provided vehicle
↪ catalog using the exact directory listed in the prompt's catalog
↪ header before creating Entities:
    scenario = xosc.Scenario('PoliceReportScene', '1.0', road, entities,
↪ story)
    scenario.add_catalog('VehicleCatalog', '<CATALOG DIR FROM PROMPT>')
```

```
# Valid entries are enumerated in the catalog header section of the user
↪ prompt.

...
```

Listing 11.3: System prompt for the Python mode.

```
You are HASCo's alignment judge for OpenSCENARIO and ontology outputs.

General rules:
- Your only job is to compare the incident report + map context to the
↪ candidate scene the user supplies.
- Always follow the user's immediate instructions for response format
↪ (e.g., reply exactly "OK", return critique text, emit corrected
↪ JSON-LD, or return revised OpenSCENARIO XML). Do not introduce any
↪ other format.
- Never emit Python or scenariogeneration code. You are validating
↪ completed scenes, not writing scripts.
- Keep feedback concise, factual, and actionable. When returning
↪ corrected data, output ONLY the requested JSON/XML with no markdown
↪ fences or commentary.
```

Listing 11.4: Judge system prompt.

```
    You are a natural-language geolocation assistant.

Task: Extract the most probable traffic location from a police/incident
↪ description and return a single, human-readable query string suitable
↪ for OpenStreetMap/Nominatim search.

OUTPUT RULES (STRICT)
- Return EXACTLY one line of text with no line breaks.
- Include the most specific intersection or segment if clear (e.g., "
↪ Sveavägen & Tegnérgatan, Stockholm, Sweden"), else a single street or
↪ area plus city and country.
- Include neighborhood/municipality if it disambiguates.
- No coordinates, no postal codes.
- No explanations, no metadata, no JSON/dicts, no notes.
- If you cannot identify a plausible location, return an empty string.
- If a junction of two streets is mentioned, only use one of the streets.
```

```
GOOD EXAMPLES
- Turreff Avenue, Donnington, Telford and Wrekin, England, United Kingdom
- Sveavägen, Stockholm, Sweden
- A1, Denton Burn, Newcastle, United Kingdom
- E18, Viksäng, Västerås, Sweden
```

Listing 11.5: Geospatial pipeline system prompt.

11.10 Worked example: cyclist–truck collision (German report)

This appendix illustrates HASCO end-to-end on a real German-language report describing a cyclist-truck collision near a complex multi-lane intersection. The German source [27] is passed to the pipeline without translation, and the English rendering is provided for reader convenience only.

```
Update:  Lieferwagen stand auf Radweg: Radfahrerin stirbt bei Unfall in
↪ Berlin-Friedrichshain - Mahnwache am Freitag
Eine Radlerin ist auf der Frankfurter Allee von einem Lkw tödlich
↪ verletzt worden. Sie war einem Lieferwagen ausgewichen, der auf dem
↪ Pop-up-Radweg stand. Eine Radfahrerin ist bei einem Unfall auf der
↪ Frankfurter Allee in Berlin-Friedrichshain tödlich verletzt worden.
↪ Nach vorläufigen Informationen musste die Radfahrerin einem auf dem
↪ Radstreifen geparkten Lieferwagen ausweichen. Dabei wurde sie von
↪ einem von hinten kommenden Sattelzug offenbar gerammt und tödlich
↪ verletzt.Ein Feuerwehrsprecher sagte, drei Menschen, die in der Nähe
↪ waren, hätten einen Schock erlitten. Sie würden von Rettungskräften
↪ betreut. Zu Details des Unfalls äußerten sich zunächst weder die
↪ Pressestelle der Polizei noch die Einsatzleitung am Unfallort....
```

Listing 11.6: Input (original, German).

```
Update: Delivery Van Parked on Bike Lane: Cyclist Dies in Accident in
↪ Berlin-Friedrichshain - Vigil on Friday
A cyclist was fatally injured by a truck on Frankfurter Allee. She had
↪ swerved to avoid a delivery van that was parked on the pop-up bike
↪ lane. A cyclist was fatally injured in an accident on Frankfurter
```

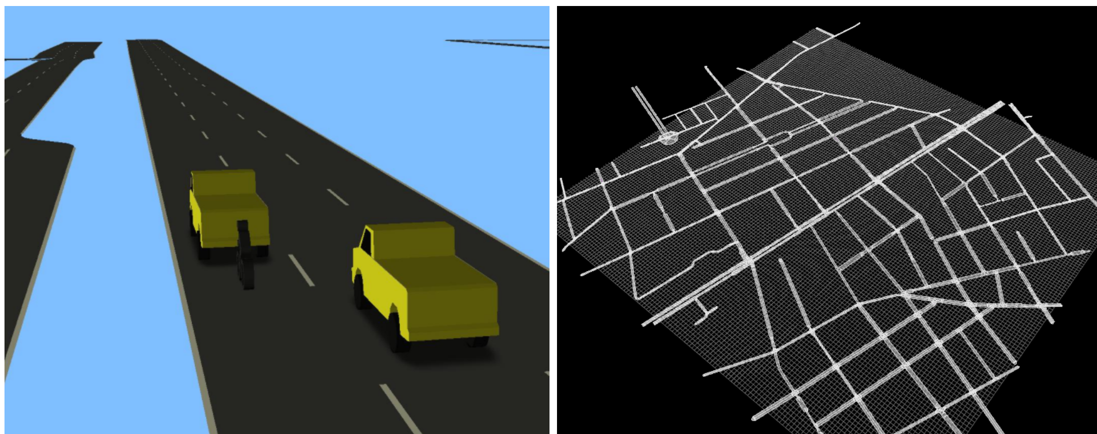
Table 11.3: HASCO outputs on the example report across all six configurations.

Mode	Judge	Bin	Reconstructed scene
Ontology	No	3	Cyclist and articulated truck on the left of a three-lane road exiting the intersection; truck clips the cyclist, advances, and stops.
Ontology	Yes	3	Cyclist on the right of a multi-lane road near the intersection, with a parked delivery van ahead and an articulated truck in the middle lane; cyclist strikes the stationary van as the truck moves up.
Python	No	2	Three-lane road with bystanders off-road; articulated truck in the middle lane, cyclist and delivery van in the rightmost lane; cyclist strikes van, truck continues briskly.
Python	Yes	2	Delivery van in the leftmost lane, cyclist behind it, articulated truck in the center; cyclist collides with van and continues forward, van halts, truck decelerates to a stop.
Direct XOSC	–	0	No artifact produced.

```
↪ Allee in Berlin-Friedrichshain. According to preliminary information,
↪ the cyclist had to swerve to avoid a delivery van parked on the bike
↪ lane. In doing so, she was apparently struck by a semi-truck
↪ approaching from behind and fatally injured. A fire department
↪ spokesperson said three people who were nearby suffered shock. They
↪ are being treated by emergency responders. Neither the police press
↪ office nor the incident command at the scene provided any details
↪ about the accident at this time...
```

Listing 11.7: Input (English translation).

Table 11.3 summarizes the bin assigned to each of the six configurations. Figure 11.10 shows the Ontology-with-Judge reconstruction and the generated OpenDRIVE map. Direct XOSC synthesis failed in both variants, consistent with its 95% corpus-wide failure rate. The Python mode introduced a delivery van absent from the source narrative and mislocated the collision target, which was an actor-hallucination failure mode undetected by the Judge because the resulting simulation is internally consistent. The Ontology mode with Judge produced the most faithful reconstruction (Bin 3): the SHACL validation step constrained the admissible actor set to those extracted from the report, preventing the hallucination observed in Python mode.



(a) esmini reconstruction (Python + Judge). (b) OpenDRIVE map generated using libOpenDRIVE [28].

Figure 11.10: Rendered output for the worked example.